

# Contents

<b>1 A program to check if a binary tree is BST or not</b>	<b>28</b>
Source . . . . .	41
<b>2 AA Trees   Set 1 (Introduction)</b>	<b>42</b>
Source . . . . .	43
<b>3 AVL Tree   Set 1 (Insertion)</b>	<b>44</b>
Source . . . . .	61
<b>4 AVL Tree   Set 2 (Deletion)</b>	<b>62</b>
Source . . . . .	83
<b>5 AVL with duplicate keys</b>	<b>84</b>
Source . . . . .	91
<b>6 Add all greater values to every node in a given BST</b>	<b>92</b>
Source . . . . .	98
<b>7 Advantages of Trie Data Structure</b>	<b>99</b>
Source . . . . .	100
<b>8 Applications of Minimum Spanning Tree Problem</b>	<b>101</b>
Source . . . . .	102
<b>9 Applications of tree data structure</b>	<b>103</b>
Source . . . . .	104
<b>10 Averages of Levels in Binary Tree</b>	<b>105</b>
Source . . . . .	108
<b>11 BFS vs DFS for Binary Tree</b>	<b>109</b>
Source . . . . .	110
<b>12 BK-Tree   Introduction &amp; Implementation</b>	<b>111</b>
Source . . . . .	119
<b>13 BST to a Tree with sum of all smaller keys</b>	<b>120</b>
Source . . . . .	125

<b>14 Binary Indexed Tree : Range Updates and Point Queries</b>	<b>126</b>
Source . . . . .	133
<b>15 Binary Tree (Array implementation)</b>	<b>134</b>
Source . . . . .	136
<b>16 Binary Tree to Binary Search Tree Conversion</b>	<b>137</b>
Source . . . . .	143
<b>17 Binary Tree to Binary Search Tree Conversion using STL set</b>	<b>144</b>
Source . . . . .	148
<b>18 Binary Tree   Set 1 (Introduction)</b>	<b>149</b>
Source . . . . .	155
<b>19 Binary Tree   Set 2 (Properties)</b>	<b>156</b>
Source . . . . .	157
<b>20 Binary Tree   Set 3 (Types of Binary Tree)</b>	<b>158</b>
Source . . . . .	160
<b>21 Binary tree to string with brackets</b>	<b>161</b>
Source . . . . .	165
<b>22 Bottom View of a Binary Tree</b>	<b>166</b>
Source . . . . .	167
<b>23 Boundary Traversal of binary tree</b>	<b>175</b>
Source . . . . .	183
<b>24 Calculate depth of a full Binary tree from Preorder</b>	<b>184</b>
Source . . . . .	188
<b>25 Calculate number of nodes in all subtrees   Using DFS</b>	<b>189</b>
Source . . . . .	194
<b>26 Change a Binary Tree so that every node stores sum of all nodes in left subtree</b>	<b>195</b>
Source . . . . .	198
<b>27 Check for Children Sum Property in a Binary Tree</b>	<b>199</b>
Source . . . . .	203
<b>28 Check for Symmetric Binary Tree (Iterative Approach)</b>	<b>204</b>
Source . . . . .	209
<b>29 Check given array of size n can represent BST of n levels or not</b>	<b>210</b>
Source . . . . .	214
<b>30 Check if a Binary Tree (not BST) has duplicate values</b>	<b>215</b>
Source . . . . .	217

<b>31 Check if a Binary Tree contains duplicate subtrees of size 2 or more</b>	<b>218</b>
Source . . . . .	221
<b>32 Check if a binary tree is sorted level-wise or not</b>	<b>222</b>
Source . . . . .	225
<b>33 Check if a binary tree is subtree of another binary tree   Set 1</b>	<b>226</b>
Source . . . . .	233
<b>34 Check if a binary tree is subtree of another binary tree   Set 2</b>	<b>234</b>
Source . . . . .	241
<b>35 Check if a given Binary Tree is Heap</b>	<b>242</b>
Source . . . . .	249
<b>36 Check if a given Binary Tree is SumTree</b>	<b>250</b>
Source . . . . .	258
<b>37 Check if a given Binary Tree is height balanced like a Red-Black Tree</b>	<b>259</b>
Source . . . . .	261
<b>38 Check if a given array can represent Preorder Traversal of Binary Search Tree</b>	<b>262</b>
Source . . . . .	267
<b>39 Check if a given graph is tree or not</b>	<b>268</b>
Source . . . . .	275
<b>40 Check if all leaves are at same level</b>	<b>276</b>
Source . . . . .	284
<b>41 Check if all levels of two trees are anagrams or not</b>	<b>285</b>
Source . . . . .	291
<b>42 Check if an array represents Inorder of Binary Search tree or not</b>	<b>292</b>
Source . . . . .	293
<b>43 Check if given Preorder, Inorder and Postorder traversals are of same tree</b>	<b>294</b>
Source . . . . .	297
<b>44 Check if leaf traversal of two Binary Trees is same?</b>	<b>298</b>
Source . . . . .	303
<b>45 Check if removing an edge can divide a Binary Tree in two halves</b>	<b>304</b>
Source . . . . .	312
<b>46 Check if the given array can represent Level Order Traversal of Binary Search Tree</b>	<b>313</b>
Source . . . . .	316

<b>47 Check if there is a root to leaf path with given sequence</b>	<b>317</b>
Source . . . . .	319
<b>48 Check if two nodes are cousins in a Binary Tree</b>	<b>320</b>
Source . . . . .	326
<b>49 Check if two nodes are cousins in a Binary Tree   Set-2</b>	<b>327</b>
Source . . . . .	331
<b>50 Check if two trees are Mirror</b>	<b>332</b>
Source . . . . .	337
<b>51 Check if two trees are Mirror   Set 2</b>	<b>338</b>
Source . . . . .	340
<b>52 Check if two trees are mirror of each other using level order traversal</b>	<b>341</b>
Source . . . . .	345
<b>53 Check if two trees have same structure</b>	<b>346</b>
Source . . . . .	348
<b>54 Check mirror in n-ary tree</b>	<b>349</b>
Source . . . . .	351
<b>55 Check sum of Covered and Uncovered nodes of Binary Tree</b>	<b>352</b>
Source . . . . .	358
<b>56 Check whether BST contains Dead End or not</b>	<b>359</b>
Source . . . . .	362
<b>57 Check whether a binary tree is a complete tree or not   Set 2 (Recursive Solution)</b>	<b>363</b>
Source . . . . .	368
<b>58 Check whether a binary tree is a full binary tree or not</b>	<b>369</b>
Source . . . . .	374
<b>59 Check whether a binary tree is a full binary tree or not   Iterative Approach</b>	<b>375</b>
Source . . . . .	378
<b>60 Check whether a given Binary Tree is Complete or not   Set 1 (Iterative Solution)</b>	<b>379</b>
Source . . . . .	388
<b>61 Check whether a given binary tree is perfect or not</b>	<b>389</b>
Source . . . . .	392
<b>62 Clone a Binary Tree with Random Pointers</b>	<b>393</b>
Source . . . . .	401

<b>63 Closest leaf to a given node in Binary Tree</b>	<b>402</b>
Source . . . . .	409
<b>64 Combinatorics on ordered trees</b>	<b>410</b>
Source . . . . .	420
<b>65 Complexity of different operations in Binary tree, Binary Search Tree and AVL tree</b>	<b>421</b>
Source . . . . .	423
<b>66 Connect Nodes at same Level (Level Order Traversal)</b>	<b>424</b>
Source . . . . .	429
<b>67 Connect nodes at same level</b>	<b>430</b>
Source . . . . .	436
<b>68 Connect nodes at same level using constant extra space</b>	<b>437</b>
Source . . . . .	448
<b>69 Construct Ancestor Matrix from a Given Binary Tree</b>	<b>449</b>
Source . . . . .	452
<b>70 Construct BST from given preorder traversal   Set 2</b>	<b>453</b>
Source . . . . .	458
<b>71 Construct BST from its given level order traversal</b>	<b>459</b>
Source . . . . .	463
<b>72 Construct Binary Tree from String with bracket representation</b>	<b>464</b>
Source . . . . .	467
<b>73 Construct Binary Tree from given Parent Array representation</b>	<b>468</b>
Source . . . . .	475
<b>74 Construct Complete Binary Tree from its Linked List Representation</b>	<b>476</b>
Source . . . . .	485
<b>75 Construct Full Binary Tree from given preorder and postorder traversals</b>	<b>486</b>
Source . . . . .	492
<b>76 Construct Full Binary Tree using its Preorder traversal and Preorder traversal of its mirror tree</b>	<b>493</b>
Source . . . . .	497
<b>77 Construct Special Binary Tree from given Inorder traversal</b>	<b>498</b>
Source . . . . .	504
<b>78 Construct Tree from given Inorder and Preorder traversals</b>	<b>505</b>
Source . . . . .	518
<b>79 Construct a Binary Search Tree from given postorder</b>	<b>519</b>

Source . . . . .	524
<b>80 Construct a Binary Tree from Postorder and Inorder</b>	<b>525</b>
Source . . . . .	533
<b>81 Construct a complete binary tree from given array in level order fashion</b>	<b>534</b>
Source . . . . .	538
<b>82 Construct a special tree from given preorder traversal</b>	<b>539</b>
Source . . . . .	544
<b>83 Construct a tree from Inorder and Level order traversals   Set 1</b>	<b>545</b>
Source . . . . .	553
<b>84 Construct a tree from Inorder and Level order traversals   Set 2</b>	<b>554</b>
Source . . . . .	557
<b>85 Construct the full k-ary tree from its preorder traversal</b>	<b>558</b>
Source . . . . .	561
<b>86 Construct tree from ancestor matrix</b>	<b>562</b>
Source . . . . .	566
<b>87 Continuous Tree</b>	<b>567</b>
Source . . . . .	569
<b>88 Convert Ternary Expression to a Binary Tree</b>	<b>570</b>
Source . . . . .	575
<b>89 Convert a Binary Tree into Doubly Linked List in spiral fashion</b>	<b>576</b>
Source . . . . .	584
<b>90 Convert a Binary Tree into its Mirror Tree</b>	<b>585</b>
Source . . . . .	592
<b>91 Convert a Binary Tree such that every node stores the sum of all nodes in its right subtree</b>	<b>593</b>
Source . . . . .	596
<b>92 Convert a Binary Tree to Threaded binary tree   Set 1 (Using Queue)</b>	<b>597</b>
Source . . . . .	603
<b>93 Convert a Binary Tree to Threaded binary tree   Set 2 (Efficient)</b>	<b>604</b>
Source . . . . .	607
<b>94 Convert a Binary Tree to a Circular Doubly Link List</b>	<b>608</b>
Source . . . . .	615
<b>95 Convert a given Binary Tree to Doubly Linked List   Set 1</b>	<b>616</b>
Source . . . . .	624

<b>96 Convert a given Binary Tree to Doubly Linked List   Set 2</b>	<b>625</b>
Source . . . . .	633
<b>97 Convert a given Binary Tree to Doubly Linked List   Set 3</b>	<b>634</b>
Source . . . . .	639
<b>98 Convert a given Binary Tree to Doubly Linked List   Set 4</b>	<b>640</b>
Source . . . . .	645
<b>99 Convert a given Binary tree to a tree that holds Logical AND property</b>	<b>646</b>
Source . . . . .	648
<b>100 Convert a given tree to its Sum Tree</b>	<b>649</b>
Source . . . . .	653
<b>101 Convert a normal BST to Balanced BST</b>	<b>654</b>
Source . . . . .	660
<b>102 Convert a tree to forest of even nodes</b>	<b>661</b>
Source . . . . .	664
<b>103 Convert an arbitrary Binary Tree to a tree that holds Children Sum Property</b>	<b>665</b>
Source . . . . .	672
<b>104 Convert left-right representation of a binary tree to down-right</b>	<b>673</b>
Source . . . . .	676
<b>105 Count BST nodes that lie in a given range</b>	<b>677</b>
Source . . . . .	681
<b>106 Count Balanced Binary Trees of Height h</b>	<b>682</b>
Source . . . . .	685
<b>107 Count Non-Leaf nodes in a Binary Tree</b>	<b>686</b>
Source . . . . .	687
<b>108 Count elements which divide all numbers in range L-R</b>	<b>688</b>
Source . . . . .	694
<b>109 Count full nodes in a Binary tree (Iterative and Recursive)</b>	<b>695</b>
Source . . . . .	703
<b>110 Count half nodes in a Binary tree (Iterative and Recursive)</b>	<b>704</b>
Source . . . . .	711
<b>111 Count pairs in a binary tree whose sum is equal to a given value x</b>	<b>712</b>
Source . . . . .	719
<b>112 Count subtrees that sum up to a given value x</b>	<b>720</b>
Source . . . . .	726

<b>113 Count the number of nodes at given level in a tree using BFS.</b>	<b>727</b>
Source . . . . .	731
<b>114 Counting the number of words in a Trie</b>	<b>732</b>
Source . . . . .	736
<b>115 Create a Doubly Linked List from a Ternary Tree</b>	<b>737</b>
Source . . . . .	743
<b>116 Create loops of even and odd values in a binary tree</b>	<b>744</b>
Source . . . . .	747
<b>117 Creating a tree with Left-Child Right-Sibling Representation</b>	<b>748</b>
Source . . . . .	752
<b>118 Custom Tree Problem</b>	<b>753</b>
Source . . . . .	757
<b>119 DFS for a n-ary tree (acyclic graph) represented as adjacency list</b>	<b>758</b>
Source . . . . .	761
<b>120 Decision Tree</b>	<b>762</b>
Source . . . . .	764
<b>121 Deepest left leaf node in a binary tree</b>	<b>765</b>
Source . . . . .	771
<b>122 Deepest left leaf node in a binary tree   iterative approach</b>	<b>772</b>
Source . . . . .	774
<b>123 Deepest right leaf node in a binary tree   Iterative approach</b>	<b>775</b>
Source . . . . .	777
<b>124 Delete leaf nodes with value as x</b>	<b>778</b>
Source . . . . .	780
<b>125 Delete leaf nodes with value k</b>	<b>781</b>
Source . . . . .	783
<b>126 Deleting a binary tree using the delete keyword</b>	<b>784</b>
Source . . . . .	786
<b>127 Deletion in a Binary Tree</b>	<b>787</b>
Source . . . . .	791
<b>128 Density of Binary Tree in One Traversal</b>	<b>792</b>
Source . . . . .	796
<b>129 Depth of an N-Ary tree</b>	<b>797</b>
Source . . . . .	800

<b>130 Depth of the deepest odd level node in Binary Tree</b>	<b>801</b>
Source . . . . .	804
<b>131 Diagonal Sum of a Binary Tree</b>	<b>805</b>
Source . . . . .	810
<b>132 Diagonal Traversal of Binary Tree</b>	<b>811</b>
Source . . . . .	817
<b>133 Diameter of a Binary Tree</b>	<b>818</b>
Source . . . . .	827
<b>134 Diameter of a Binary Tree in O(n) [A new method]</b>	<b>828</b>
Source . . . . .	830
<b>135 Diameter of a tree using DFS</b>	<b>831</b>
Source . . . . .	836
<b>136 Diameter of an N-ary tree</b>	<b>837</b>
Source . . . . .	841
<b>137 Diameter of n-ary tree using BFS</b>	<b>842</b>
Source . . . . .	844
<b>138 Difference between sums of odd level and even level nodes of a Binary Tree</b>	<b>845</b>
Source . . . . .	851
<b>139 Disjoint Set Union on trees   Set 1</b>	<b>852</b>
Source . . . . .	855
<b>140 Disjoint Set Union on trees   Set 2</b>	<b>856</b>
Source . . . . .	860
<b>141 Double Tree</b>	<b>861</b>
Source . . . . .	866
<b>142 Dynamic Programming on Trees   Set 2</b>	<b>867</b>
Source . . . . .	874
<b>143 Dynamic Programming on Trees   Set-1</b>	<b>875</b>
Source . . . . .	879
<b>144 Enumeration of Binary Trees</b>	<b>880</b>
Source . . . . .	881
<b>145 Euler Tour of Tree</b>	<b>882</b>
Source . . . . .	886
<b>146 Euler Tour   Subtree Sum using Segment Tree</b>	<b>887</b>
Source . . . . .	894

<b>147 Euler tour of Binary Tree</b>	<b>895</b>
Source . . . . .	898
<b>148 Evaluation of Expression Tree</b>	<b>899</b>
Source . . . . .	903
<b>149 Expression Tree</b>	<b>904</b>
Source . . . . .	911
<b>150 Extract Leaves of a Binary Tree in a Doubly Linked List</b>	<b>912</b>
Source . . . . .	919
<b>151 Factor Tree of a given Number</b>	<b>920</b>
Source . . . . .	922
<b>152 Find All Duplicate Subtrees</b>	<b>923</b>
Source . . . . .	927
<b>153 Find Count of Single Valued Subtrees</b>	<b>928</b>
Source . . . . .	935
<b>154 Find Height of Binary Tree represented by Parent array</b>	<b>936</b>
Source . . . . .	941
<b>155 Find LCA in Binary Tree using RMQ</b>	<b>942</b>
Source . . . . .	954
<b>156 Find Minimum Depth of a Binary Tree</b>	<b>955</b>
Source . . . . .	963
<b>157 Find a number in minimum steps</b>	<b>964</b>
Source . . . . .	966
<b>158 Find all possible binary trees with given Inorder Traversal</b>	<b>967</b>
Source . . . . .	974
<b>159 Find depth of the deepest odd level leaf node</b>	<b>975</b>
Source . . . . .	982
<b>160 Find distance between two nodes of a Binary Tree</b>	<b>983</b>
Source . . . . .	998
<b>161 Find distance from root to given node in a binary tree</b>	<b>999</b>
Source . . . . .	1001
<b>162 Find first non matching leaves in two binary trees</b>	<b>1002</b>
Source . . . . .	1005
<b>163 Find height of a special binary tree whose leaf nodes are connected</b>	<b>1006</b>
Source . . . . .	1008

<b>164 Find if given vertical level of binary tree is sorted or not</b>	<b>1009</b>
Source . . . . .	.1013
<b>165 Find if there is a pair in root to a leaf path with sum equals to root's data</b>	<b>1014</b>
Source . . . . .	.1017
<b>166 Find largest subtree having identical left and right subtrees</b>	<b>1018</b>
Source . . . . .	.1021
<b>167 Find largest subtree sum in a tree</b>	<b>1022</b>
Source . . . . .	.1027
<b>168 Find maximum (or minimum) in Binary Tree</b>	<b>1028</b>
Source . . . . .	.1032
<b>169 Find maximum level product in Binary Tree</b>	<b>1033</b>
Source . . . . .	.1036
<b>170 Find maximum level sum in Binary Tree</b>	<b>1037</b>
Source . . . . .	.1040
<b>171 Find maximum vertical sum in binary tree</b>	<b>1041</b>
Source . . . . .	.1044
<b>172 Find mirror of a given node in Binary tree</b>	<b>1045</b>
Source . . . . .	.1050
<b>173 Find multiplication of sums of data of leaves at same levels</b>	<b>1051</b>
Source . . . . .	.1057
<b>174 Find n-th node in Postorder traversal of a Binary Tree</b>	<b>1058</b>
Source . . . . .	.1060
<b>175 Find n-th node in Preorder traversal of a Binary Tree</b>	<b>1061</b>
Source . . . . .	.1063
<b>176 Find n-th node of inorder traversal</b>	<b>1064</b>
Source . . . . .	.1066
<b>177 Find next right node of a given key</b>	<b>1067</b>
Source . . . . .	.1074
<b>178 Find next right node of a given key   Set 2</b>	<b>1075</b>
Source . . . . .	.1078
<b>179 Find right sibling of a binary tree with parent pointers</b>	<b>1079</b>
Source . . . . .	.1084
<b>180 Find root of the tree where children id sum for every node is given</b>	<b>1085</b>
Source . . . . .	.1086

<b>181 Find sum of all left leaves in a given Binary Tree</b>	<b>1087</b>
Source . . . . .	.1098
<b>182 Find sum of all nodes of the given perfect binary tree</b>	<b>1099</b>
Source . . . . .	.1106
<b>183 Find sum of all right leaves in a given Binary Tree</b>	<b>1107</b>
Source . . . . .	.1110
<b>184 Find the Deepest Node in a Binary Tree</b>	<b>1111</b>
Source . . . . .	.1115
<b>185 Find the closest element in Binary Search Tree</b>	<b>1116</b>
Source . . . . .	.1117
<b>186 Find the closest leaf in a Binary Tree</b>	<b>1118</b>
Source . . . . .	.1126
<b>187 Find the largest BST subtree in a given Binary Tree   Set 1</b>	<b>1127</b>
Source . . . . .	.1135
<b>188 Find the maximum node at a given level in a binary tree</b>	<b>1136</b>
Source . . . . .	.1140
<b>189 Find the maximum path sum between two leaves of a binary tree</b>	<b>1141</b>
Source . . . . .	.1148
<b>190 Find the maximum sum leaf to root path in a Binary Tree</b>	<b>1149</b>
Source . . . . .	.1149
<b>191 Find the node with minimum value in a Binary Search Tree</b>	<b>1150</b>
Source . . . . .	.1155
<b>192 Flatten a binary tree into linked list</b>	<b>1156</b>
Source . . . . .	.1160
<b>193 Flatten a binary tree into linked list   Set-2</b>	<b>1161</b>
Source . . . . .	.1164
<b>194 Flip Binary Tree</b>	<b>1165</b>
Source . . . . .	.1176
<b>195 Foldable Binary Trees</b>	<b>1177</b>
Source . . . . .	.1187
<b>196 General Tree (Each node can have arbitrary number of children) Level Order Traversal</b>	<b>1188</b>
Source . . . . .	.1190
<b>197 Get Level of a node in a Binary Tree</b>	<b>1191</b>
Source . . . . .	.1195

<b>198 Get level of a node in binary tree   iterative approach</b>	<b>1196</b>
Source . . . . .	.1199
<b>199 Get maximum left node in binary tree</b>	<b>1200</b>
Source . . . . .	.1202
<b>200 Given a binary tree, how do you remove all the half nodes?</b>	<b>1203</b>
Source . . . . .	.1209
<b>201 Given a binary tree, print all root-to-leaf paths</b>	<b>1210</b>
Source . . . . .	.1216
<b>202 Given a binary tree, print out all of its root-to-leaf paths one per line.</b>	<b>1217</b>
Source . . . . .	.1223
<b>203 Given a n-ary tree, count number of nodes which have more number of children than parents</b>	<b>1224</b>
Source . . . . .	.1226
<b>204 Given level order traversal of a Binary Tree, check if the Tree is a Min-Heap</b>	<b>1227</b>
Source . . . . .	.1230
<b>205 Handshaking Lemma and Interesting Tree Properties</b>	<b>1231</b>
Source . . . . .	.1233
<b>206 HashSet vs TreeSet in Java</b>	<b>1234</b>
Source . . . . .	.1236
<b>207 Height of a complete binary tree (or Heap) with N nodes</b>	<b>1237</b>
Source . . . . .	.1240
<b>208 Height of a generic tree from parent array</b>	<b>1241</b>
Source . . . . .	.1246
<b>209 Height of binary tree considering even level leaves only</b>	<b>1247</b>
Source . . . . .	.1250
<b>210 Height of n-ary tree if parent array is given</b>	<b>1251</b>
Source . . . . .	.1254
<b>211 How to determine if a binary tree is height-balanced?</b>	<b>1255</b>
Source . . . . .	.1267
<b>212 If you are given two traversal sequences, can you construct the binary tree?</b>	<b>1268</b>
Source . . . . .	.1269
<b>213 Immediate Smaller element in an N-ary Tree</b>	<b>1270</b>
Source . . . . .	.1273

<b>214 Implementation of Binary Search Tree in Javascript</b>	<b>1274</b>
Source . . . . .	.1283
<b>215 Inorder Non-threaded Binary Tree Traversal without Recursion or Stack</b>	<b>284</b>
Source . . . . .	.1291
<b>216 Inorder Successor of a node in Binary Tree</b>	<b>1292</b>
Source . . . . .	.1296
<b>217 Inorder Tree Traversal without Recursion</b>	<b>1297</b>
Source . . . . .	.1307
<b>218 Inorder Tree Traversal without recursion and without stack!</b>	<b>1308</b>
Source . . . . .	.1314
<b>219 Inorder predecessor and successor for a given key in BST</b>	<b>1315</b>
Source . . . . .	.1322
<b>220 Insertion in a Binary Tree</b>	<b>1323</b>
Source . . . . .	.1327
<b>221 Iterative Method to find Height of Binary Tree</b>	<b>1328</b>
Source . . . . .	.1334
<b>222 Iterative Postorder Traversal   Set 1 (Using Two Stacks)</b>	<b>1335</b>
Source . . . . .	.1342
<b>223 Iterative Postorder Traversal   Set 2 (Using One Stack)</b>	<b>1343</b>
Source . . . . .	.1352
<b>224 Iterative Preorder Traversal</b>	<b>1353</b>
Source . . . . .	.1358
<b>225 Iterative Preorder Traversal of an N-ary Tree</b>	<b>1359</b>
Source . . . . .	.1362
<b>226 Iterative Search for a key ‘x’ in Binary Tree</b>	<b>1363</b>
Source . . . . .	.1367
<b>227 Iterative Segment Tree (Range Maximum Query with Node Update)</b>	<b>1368</b>
Source . . . . .	.1371
<b>228 Iterative Segment Tree (Range Minimum Query)</b>	<b>1372</b>
Source . . . . .	.1375
<b>229 Iterative diagonal traversal of binary tree</b>	<b>1376</b>
Source . . . . .	.1378
<b>230 Iterative function to check if two trees are identical</b>	<b>1379</b>
Source . . . . .	.1382

<b>231 Iterative method to check if two trees are mirror of each other</b>	<b>1383</b>
Source . . . . .	.1386
<b>232 Iterative method to find ancestors of a given binary tree</b>	<b>1387</b>
Source . . . . .	.1390
<b>233 Iterative program to Calculate Size of a tree</b>	<b>1391</b>
Source . . . . .	.1393
<b>234 Iterative program to count leaf nodes in a Binary Tree</b>	<b>1394</b>
Source . . . . .	.1396
<b>235 K-th ancestor of a node in Binary Tree</b>	<b>1397</b>
Source . . . . .	.1400
<b>236 Kth ancestor of a node in binary tree   Set 2</b>	<b>1401</b>
Source . . . . .	.1403
<b>237 K'th Largest Element in BST when modification to BST is not allowed</b>	<b>1404</b>
Source . . . . .	.1410
<b>238 LCA for general or n-ary trees (Sparse Matrix DP approach &lt; O(nlogn), O(logn)&gt;)</b>	<b>1411</b>
Source . . . . .	.1417
<b>239 LCA for n-ary Tree   Constant Query O(1)</b>	<b>1418</b>
Source . . . . .	.1424
<b>240 Largest BST in a Binary Tree   Set 2</b>	<b>1425</b>
Source . . . . .	.1428
<b>241 Largest Independent Set Problem   DP-26</b>	<b>1429</b>
Source . . . . .	.1435
<b>242 Largest number in BST which is less than or equal to N</b>	<b>1436</b>
Source . . . . .	.1439
<b>243 Largest value in each level of Binary Tree</b>	<b>1440</b>
Source . . . . .	.1442
<b>244 Largest value in each level of Binary Tree   Set-2 (Iterative Approach)</b>	<b>1443</b>
Source . . . . .	.1446
<b>245 Leaf nodes from Preorder of a Binary Search Tree (Using Recursion)</b>	<b>1447</b>
Source . . . . .	.1450
<b>246 Left-Child Right-Sibling Representation of Tree</b>	<b>1451</b>
Source . . . . .	.1457
<b>247 Leftist Tree / Leftist Heap</b>	<b>1458</b>
Source . . . . .	.1469

<b>248 Level Ancestor Problem</b>	<b>1470</b>
Source . . . . .	.1477
<b>249 Level Order Tree Traversal</b>	<b>1478</b>
Source . . . . .	.1490
<b>250 Level order traversal in spiral form</b>	<b>1491</b>
Source . . . . .	.1500
<b>251 Level order traversal in spiral form   Using one stack and one queue</b>	<b>1501</b>
Source . . . . .	.1503
<b>252 Level order traversal line by line   Set 2 (Using Two Queues)</b>	<b>1504</b>
Source . . . . .	.1510
<b>253 Level order traversal line by line   Set 3 (Using One Queue)</b>	<b>1511</b>
Source . . . . .	.1515
<b>254 Level order traversal with direction change after every two levels</b>	<b>1516</b>
Source . . . . .	.1523
<b>255 Level with maximum number of nodes</b>	<b>1524</b>
Source . . . . .	.1527
<b>256 Levelwise Alternating GCD and LCM of nodes in Segment Tree</b>	<b>1528</b>
Source . . . . .	.1536
<b>257 Linked complete binary tree &amp; its creation</b>	<b>1537</b>
Source . . . . .	.1542
<b>258 Locking and Unlocking of Resources arranged in the form of n-ary Tree</b>	<b>1543</b>
Source . . . . .	.1545
<b>259 Longest Path with Same Values in a Binary Tree</b>	<b>1546</b>
Source . . . . .	.1548
<b>260 Longest consecutive sequence in Binary tree</b>	<b>1549</b>
Source . . . . .	.1551
<b>261 Longest path in an undirected tree</b>	<b>1552</b>
Source . . . . .	.1555
<b>262 Longest word in ternary search tree</b>	<b>1556</b>
Source . . . . .	.1561
<b>263 Lowest Common Ancestor in Parent Array Representation</b>	<b>1562</b>
Source . . . . .	.1564
<b>264 Lowest Common Ancestor in a Binary Search Tree.</b>	<b>1565</b>
Source . . . . .	.1571

<b>265 Lowest Common Ancestor in a Binary Tree   Set 1</b>	<b>1572</b>
Source . . . . .	.1591
<b>266 Lowest Common Ancestor in a Binary Tree   Set 2 (Using Parent Pointer)</b>	<b>1592</b>
Source . . . . .	.1593
<b>267 Lowest Common Ancestor in a Binary Tree   Set 3 (Using RMQ)</b>	<b>1594</b>
Source . . . . .	.1600
<b>268 Maximum Consecutive Increasing Path Length in Binary Tree</b>	<b>1601</b>
Source . . . . .	.1605
<b>269 Maximum Path Sum in a Binary Tree</b>	<b>1606</b>
Source . . . . .	.1612
<b>270 Maximum difference between node and its ancestor in Binary Tree</b>	<b>1613</b>
Source . . . . .	.1618
<b>271 Maximum edge removal from tree to make even forest</b>	<b>1619</b>
Source . . . . .	.1622
<b>272 Maximum parent children sum in Binary tree</b>	<b>1623</b>
Source . . . . .	.1626
<b>273 Maximum spiral sum in Binary Tree</b>	<b>1627</b>
Source . . . . .	.1631
<b>274 Maximum sum from a tree with adjacent levels not allowed</b>	<b>1632</b>
Source . . . . .	.1635
<b>275 Maximum sum of nodes in Binary tree such that no two are adjacent</b>	<b>1636</b>
Source . . . . .	.1640
<b>276 Maximum width of a binary tree</b>	<b>1641</b>
Source . . . . .	.1661
<b>277 Merge Sort Tree for Range Order Statistics</b>	<b>1662</b>
Source . . . . .	.1666
<b>278 Merge Two Binary Trees by doing Node Sum (Recursive and Iterative)</b>	<b>1667</b>
Source . . . . .	.1675
<b>279 Merge two BSTs with limited extra space</b>	<b>1676</b>
Source . . . . .	.1681
<b>280 Minimum no. of iterations to pass information to all nodes in the tree</b>	<b>1682</b>
Source . . . . .	.1689
<b>281 Minimum swap required to convert binary tree to binary search tree</b>	<b>1690</b>
Source . . . . .	.1691

<b>282 Mirror of n-ary Tree</b>	<b>1692</b>
Source . . . . .	1697
<b>283 Modify a binary tree to get preorder traversal using right pointers only</b>	<b>1698</b>
Source . . . . .	1703
<b>284 Morris traversal for Preorder</b>	<b>1704</b>
Source . . . . .	1710
<b>285 Next Larger element in n-ary tree</b>	<b>1711</b>
Source . . . . .	1713
<b>286 Node having maximum sum of immediate children and itself in n-ary tree</b>	<b>1714</b>
Source . . . . .	1717
<b>287 Non-recursive program to delete an entire binary tree</b>	<b>1718</b>
Source . . . . .	1723
<b>288 Number of Binary Trees for given Preorder Sequence length</b>	<b>1724</b>
Source . . . . .	1728
<b>289 Number of children of given node in n-ary Tree</b>	<b>1729</b>
Source . . . . .	1732
<b>290 Number of full binary trees such that each node is product of its children</b>	<b>1733</b>
Source . . . . .	1741
<b>291 Number of nodes greater than a given value in n-ary tree</b>	<b>1742</b>
Source . . . . .	1744
<b>292 Number of siblings of a given Node in n-ary Tree</b>	<b>1745</b>
Source . . . . .	1748
<b>293 Number of subtrees having odd count of even numbers</b>	<b>1749</b>
Source . . . . .	1752
<b>294 Number of turns to reach from one node to other in binary tree</b>	<b>1753</b>
Source . . . . .	1761
<b>295 Number of ways to traverse an N-ary tree</b>	<b>1762</b>
Source . . . . .	1766
<b>296 Overview of Data Structures   Set 2 (Binary Tree, BST, Heap and Hash)</b>	<b>1767</b>
Source . . . . .	1770
<b>297 Pairs involved in Balanced Parentheses</b>	<b>1771</b>
Source . . . . .	1775
<b>298 Pairwise Swap leaf nodes in a binary tree</b>	<b>1776</b>
Source . . . . .	1780

<b>299 Palindromic Tree   Introduction &amp; Implementation</b>	<b>1781</b>
Source . . . . .	.1794
<b>300 Path length having maximum number of bends</b>	<b>1795</b>
Source . . . . .	.1799
<b>301 Perfect Binary Tree Specific Level Order Traversal</b>	<b>1800</b>
Source . . . . .	.1809
<b>302 Perfect Binary Tree Specific Level Order Traversal   Set 2</b>	<b>1810</b>
Source . . . . .	.1819
<b>303 Persistent Segment Tree   Set 1 (Introduction)</b>	<b>1820</b>
Source . . . . .	.1826
<b>304 Populate Inorder Successor for all nodes</b>	<b>1827</b>
Source . . . . .	.1832
<b>305 Possible edges of a tree for given diameter, height and vertices</b>	<b>1833</b>
Source . . . . .	.1836
<b>306 Postorder predecessor of a Node in Binary Search Tree</b>	<b>1837</b>
Source . . . . .	.1840
<b>307 Postorder successor of a Node in Binary Tree</b>	<b>1841</b>
Source . . . . .	.1844
<b>308 Postorder traversal of Binary Tree without recursion and without stack</b>	<b>1845</b>
Source . . . . .	.1848
<b>309 Practice questions on Height balanced/AVL Tree</b>	<b>1849</b>
Source . . . . .	.1854
<b>310 Preorder Successor of a Node in Binary Tree</b>	<b>1855</b>
Source . . . . .	.1858
<b>311 Preorder Traversal of N-ary Tree Without Recursion</b>	<b>1859</b>
Source . . . . .	.1861
<b>312 Preorder from Inorder and Postorder traversals</b>	<b>1862</b>
Source . . . . .	.1867
<b>313 Preorder predecessor of a Node in Binary Tree</b>	<b>1868</b>
Source . . . . .	.1871
<b>314 Print Ancestors of a given node in Binary Tree</b>	<b>1872</b>
Source . . . . .	.1877
<b>315 Print BST keys in the given range</b>	<b>1878</b>
Source . . . . .	.1882

<b>316 Print Binary Tree in 2-Dimensions</b>	<b>1883</b>
Source . . . . .	.1886
<b>317 Print Binary Tree levels in sorted order</b>	<b>1887</b>
Source . . . . .	.1890
<b>318 Print Binary Tree levels in sorted order   Set 2 (Using set)</b>	<b>1891</b>
Source . . . . .	.1893
<b>319 Print Common Nodes in Two Binary Search Trees</b>	<b>1894</b>
Source . . . . .	.1898
<b>320 Print Left View of a Binary Tree</b>	<b>1899</b>
Source . . . . .	.1904
<b>321 Print Levels of all nodes in a Binary Tree</b>	<b>1905</b>
Source . . . . .	.1909
<b>322 Print Nodes in Top View of Binary Tree</b>	<b>1910</b>
Source . . . . .	.1915
<b>323 Print Postorder traversal from given Inorder and Preorder traversals</b>	<b>1916</b>
Source . . . . .	.1921
<b>324 Print Right View of a Binary Tree</b>	<b>1922</b>
Source . . . . .	.1927
<b>325 Print a Binary Tree in Vertical Order   Set 1</b>	<b>1928</b>
Source . . . . .	.1936
<b>326 Print a Binary Tree in Vertical Order   Set 2 (Map based Method)</b>	<b>1937</b>
Source . . . . .	.1944
<b>327 Print a Binary Tree in Vertical Order   Set 3 (Using Level Order Traversal)</b>	<b>1945</b>
Source . . . . .	.1952
<b>328 Print all full nodes in a Binary Tree</b>	<b>1953</b>
Source . . . . .	.1955
<b>329 Print all k-sum paths in a binary tree</b>	<b>1956</b>
Source . . . . .	.1959
<b>330 Print all leaf nodes of a Binary Tree from left to right</b>	<b>1960</b>
Source . . . . .	.1962
<b>331 Print all nodes at distance k from a given node</b>	<b>1963</b>
Source . . . . .	.1971
<b>332 Print all nodes in a binary tree having K leaves</b>	<b>1972</b>
Source . . . . .	.1974

<b>333 Print all nodes that are at distance k from a leaf node</b>	<b>1975</b>
Source . . . . .	.1979
<b>334 Print all nodes that don't have sibling</b>	<b>1980</b>
Source . . . . .	.1985
<b>335 Print all root to leaf paths with there relative positions</b>	<b>1986</b>
Source . . . . .	.1990
<b>336 Print all the paths from root, with a specified sum in Binary tree</b>	<b>1991</b>
Source . . . . .	.1994
<b>337 Print common nodes on path from root (or common ancestors)</b>	<b>1995</b>
Source . . . . .	.2000
<b>338 Print cousins of a given node in Binary Tree</b>	<b>2001</b>
Source . . . . .	.2004
<b>339 Print cousins of a given node in Binary Tree   Single Traversal</b>	<b>2005</b>
Source . . . . .	.2011
<b>340 Print extreme nodes of each level of Binary Tree in alternate order</b>	<b>2012</b>
Source . . . . .	.2015
<b>341 Print leftmost and rightmost nodes of a Binary Tree</b>	<b>2016</b>
Source . . . . .	.2022
<b>342 Print level order traversal line by line   Set 1</b>	<b>2023</b>
Source . . . . .	.2029
<b>343 Print middle level of perfect binary tree without finding height</b>	<b>2030</b>
Source . . . . .	.2034
<b>344 Print nodes at k distance from root</b>	<b>2035</b>
Source . . . . .	.2039
<b>345 Print nodes at k distance from root   Iterative</b>	<b>2040</b>
Source . . . . .	.2043
<b>346 Print nodes between two given level numbers of a binary tree</b>	<b>2044</b>
Source . . . . .	.2050
<b>347 Print nodes in top view of Binary Tree   Set 2</b>	<b>2051</b>
Source . . . . .	.2054
<b>348 Print path from root to a given node in a binary tree</b>	<b>2055</b>
Source . . . . .	.2058
<b>349 Print root to leaf paths without using recursion</b>	<b>2059</b>
Source . . . . .	.2062

<b>350 Print the longest leaf to leaf path in a Binary tree</b>	<b>2063</b>
Source . . . . .	.2068
<b>351 Print the nodes at odd levels of a tree</b>	<b>2069</b>
Source . . . . .	.2072
<b>352 Print the path common to the two paths from the root to the two given nodes</b>	<b>2073</b>
Source . . . . .	.2078
<b>353 Product of nodes at k-th level in a tree represented as string</b>	<b>2079</b>
Source . . . . .	.2085
<b>354 Program to count leaf nodes in a binary tree</b>	<b>2086</b>
Source . . . . .	.2090
<b>355 Prüfer Code to Tree Creation</b>	<b>2091</b>
Source . . . . .	.2095
<b>356 Quad Tree</b>	<b>2096</b>
Source . . . . .	.2102
<b>357 Queries for DFS of a subtree in a tree</b>	<b>2103</b>
Source . . . . .	.2107
<b>358 Queries for M-th node in the DFS of subtree</b>	<b>2108</b>
Source . . . . .	.2112
<b>359 Queries to find distance between two nodes of a Binary tree</b>	<b>2113</b>
Source . . . . .	.2115
<b>360 Queries to find distance between two nodes of a Binary tree – O(logn) method</b>	<b>2116</b>
Source . . . . .	.2124
<b>361 Query for ancestor-descendant relationship in a tree</b>	<b>2125</b>
Source . . . . .	.2127
<b>362 Range LCM Queries</b>	<b>2128</b>
Source . . . . .	.2131
<b>363 Range and Update Query for Chessboard Pieces</b>	<b>2132</b>
Source . . . . .	.2140
<b>364 Range query for Largest Sum Contiguous Subarray</b>	<b>2141</b>
Source . . . . .	.2147
<b>365 Relationship between number of nodes and height of binary tree</b>	<b>2148</b>
Source . . . . .	.2150
<b>366 Remove all nodes which don't lie in any path with sum<math>\geq</math> k</b>	<b>2151</b>

Source . . . . .	2163
<b>367 Remove nodes on root to leaf paths of length &lt; K</b>	<b>2164</b>
Source . . . . .	.2169
<b>368 Replace each node in binary tree with the sum of its inorder predecessor and successor</b>	<b>2170</b>
Source . . . . .	.2174
<b>369 Replace node with depth in a binary tree</b>	<b>2175</b>
Source . . . . .	.2177
<b>370 Reverse Level Order Traversal</b>	<b>2178</b>
Source . . . . .	.2189
<b>371 Reverse Morris traversal using Threaded Binary Tree</b>	<b>2190</b>
Source . . . . .	.2194
<b>372 Reverse alternate levels of a perfect binary tree</b>	<b>2195</b>
Source . . . . .	.2204
<b>373 Reverse tree path</b>	<b>2205</b>
Source . . . . .	.2214
<b>374 Right view of Binary Tree using Queue</b>	<b>2215</b>
Source . . . . .	.2219
<b>375 Root to leaf path sum equal to a given number</b>	<b>2220</b>
Source . . . . .	.2226
<b>376 Root to leaf path with maximum distinct nodes</b>	<b>2227</b>
Source . . . . .	.2229
<b>377 Root to leaf paths having equal lengths in a Binary Tree</b>	<b>2230</b>
Source . . . . .	.2232
<b>378 ScapeGoat Tree   Set 1 (Introduction and Insertion)</b>	<b>2233</b>
Source . . . . .	.2244
<b>379 Second Largest element in n-ary tree</b>	<b>2245</b>
Source . . . . .	.2248
<b>380 Segment Tree   Set 1 (Sum of given range)</b>	<b>2249</b>
Source . . . . .	.2258
<b>381 Segment Tree   Set 2 (Range Minimum Query)</b>	<b>2259</b>
Source . . . . .	.2266
<b>382 Segment Tree   Set 3 (XOR of given range)</b>	<b>2267</b>
Source . . . . .	.2271

<b>383 Select a Random Node from a tree with equal probability</b>	<b>2272</b>
Source . . . . .	.2273
<b>384 Serialize and Deserialize a Binary Tree</b>	<b>2276</b>
Source . . . . .	.2281
<b>385 Serialize and Deserialize an N-ary Tree</b>	<b>2282</b>
Source . . . . .	.2286
<b>386 Shortest distance between two nodes in an infinite binary tree</b>	<b>2287</b>
Source . . . . .	.2291
<b>387 Shortest path between two nodes in array like representation of binary tree</b>	<b>2292</b>
Source . . . . .	.2295
<b>388 Sink Odd nodes in Binary Tree</b>	<b>2296</b>
Source . . . . .	.2300
<b>389 Smallest Subarray with given GCD</b>	<b>2301</b>
Source . . . . .	.2307
<b>390 Smallest number in BST which is greater than or equal to N</b>	<b>2308</b>
Source . . . . .	.2311
<b>391 Smallest number in BST which is greater than or equal to N ( Iterative Approach)</b>	<b>2312</b>
Source . . . . .	.2315
<b>392 Smallest value in each level of Binary Tree</b>	<b>2316</b>
Source . . . . .	.2322
<b>393 Sorted Array to Balanced BST</b>	<b>2323</b>
Source . . . . .	.2328
<b>394 Splay Tree   Set 3 (Delete)</b>	<b>2329</b>
Source . . . . .	.2334
<b>395 Sqrt (or Square Root) Decomposition   Set 2 (LCA of Tree in O(sqrt(height)) time)</b>	<b>2335</b>
Source . . . . .	.2345
<b>396 Sub-tree with minimum color difference in a 2-coloured tree</b>	<b>2346</b>
Source . . . . .	.2349
<b>397 Subtree of all nodes in a tree using DFS</b>	<b>2350</b>
Source . . . . .	.2354
<b>398 Subtree with given sum in a Binary Tree</b>	<b>2355</b>
Source . . . . .	.2360

<b>399 Subtrees formed after bursting nodes</b>	<b>2361</b>
Source . . . . .	.2364
<b>400 Succinct Encoding of Binary Tree</b>	<b>2365</b>
Source . . . . .	.2371
<b>401 Sudo Placement[1.4]   BST Traversal</b>	<b>2372</b>
Source . . . . .	.2374
<b>402 Sudo Placement[1.4]   Jumping the Subtree</b>	<b>2375</b>
Source . . . . .	.2382
<b>403 Sum of Interval and Update with Number of Divisors</b>	<b>2383</b>
Source . . . . .	.2387
<b>404 Sum of all elements of N-ary Tree</b>	<b>2388</b>
Source . . . . .	.2391
<b>405 Sum of all leaf nodes of binary tree</b>	<b>2392</b>
Source . . . . .	.2395
<b>406 Sum of all nodes in a binary tree</b>	<b>2396</b>
Source . . . . .	.2398
<b>407 Sum of all the numbers that are formed from root to leaf paths</b>	<b>2399</b>
Source . . . . .	.2404
<b>408 Sum of all the parent nodes having child node x</b>	<b>2405</b>
Source . . . . .	.2409
<b>409 Sum of heights of all individual nodes in a binary tree</b>	<b>2410</b>
Source . . . . .	.2414
<b>410 Sum of k largest elements in BST</b>	<b>2415</b>
Source . . . . .	.2418
<b>411 Sum of leaf nodes at minimum level</b>	<b>2419</b>
Source . . . . .	.2422
<b>412 Sum of nodes at k-th level in a tree represented as string</b>	<b>2423</b>
Source . . . . .	.2425
<b>413 Sum of nodes at maximum depth of a Binary Tree</b>	<b>2426</b>
Source . . . . .	.2429
<b>414 Sum of nodes at maximum depth of a Binary Tree   Iterative Approach</b>	<b>2430</b>
Source . . . . .	.2432
<b>415 Sum of nodes on the longest path from root to leaf node</b>	<b>2433</b>
Source . . . . .	.2438

<b>416 Swap Nodes in Binary tree of every k'th level</b>	<b>2439</b>
Source . . . . .	.2444
<b>417 Symmetric Tree (Mirror Image of itself)</b>	<b>2445</b>
Source . . . . .	.2450
<b>418 Ternary Search Tree (Deletion)</b>	<b>2451</b>
Source . . . . .	.2460
<b>419 The Great Tree-List Recursion Problem.</b>	<b>2461</b>
Source . . . . .	.2461
<b>420 Threaded Binary Tree</b>	<b>2462</b>
Source . . . . .	.2465
<b>421 Threaded Binary Tree   Insertion</b>	<b>2466</b>
Source . . . . .	.2472
<b>422 Tilt of Binary Tree</b>	<b>2473</b>
Source . . . . .	.2475
<b>423 Top three elements in binary tree</b>	<b>2476</b>
Source . . . . .	.2478
<b>424 Total nodes traversed in Euler Tour Tree</b>	<b>2479</b>
Source . . . . .	.2483
<b>425 Total sum except adjacent of a given node in a Binary Tree</b>	<b>2484</b>
Source . . . . .	.2488
<b>426 Traversal of tree with k jumps allowed between nodes of same height</b>	<b>2489</b>
Source . . . . .	.2494
<b>427 Tree Isomorphism Problem</b>	<b>2495</b>
Source . . . . .	.2500
<b>428 Tree Traversals (Inorder, Preorder and Postorder)</b>	<b>2501</b>
Source . . . . .	.2512
<b>429 Two Dimensional Segment Tree   Sub-Matrix Sum</b>	<b>2513</b>
Source . . . . .	.2520
<b>430 Vertical Sum in Binary Tree   Set 2 (Space Optimized)</b>	<b>2521</b>
Source . . . . .	.2527
<b>431 Vertical Sum in a given Binary Tree   Set 1</b>	<b>2528</b>
Source . . . . .	.2533
<b>432 Vertical width of Binary tree   Set 1</b>	<b>2534</b>
Source . . . . .	.2537

<b>433 Vertical width of Binary tree   Set 2</b>	<b>2538</b>
Source . . . . .	.2541
<b>434 Ways to color a skewed tree such that parent and child have different colors</b>	<b>2542</b>
Source . . . . .	.2547
<b>435 Write Code to Determine if Two Trees are Identical</b>	<b>2548</b>
Source . . . . .	.2552
<b>436 Write a program to Delete a Tree</b>	<b>2553</b>
Source . . . . .	.2559
<b>437 Write a Program to Find the Maximum Depth or Height of a Tree</b>	<b>2560</b>
Source . . . . .	.2565
<b>438 Write a program to Calculate Size of a tree   Recursion</b>	<b>2566</b>
Source . . . . .	.2570
<b>439 XOR of numbers that appeared even number of times in given Range</b>	<b>2571</b>
Source . . . . .	.2576
<b>440 ZigZag Tree Traversal</b>	<b>2577</b>
Source . . . . .	.2583
<b>441 nth Rational number in Calkin-Wilf sequence</b>	<b>2584</b>
Source . . . . .	.2588

# Chapter 1

## A program to check if a binary tree is BST or not

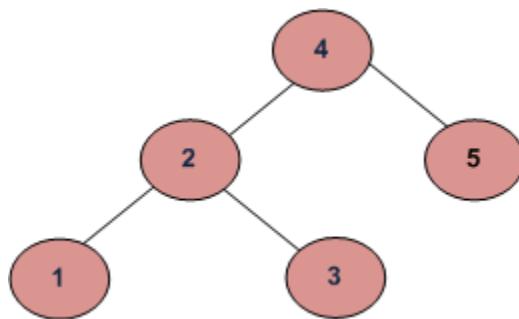
A program to check if a binary tree is BST or not - GeeksforGeeks

A binary search tree (BST) is a node based binary tree data structure which has the following properties.

- The left subtree of a node contains only nodes with keys less than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- Both the left and right subtrees must also be binary search trees.

From the above properties it naturally follows that:

- Each node (item in the tree) has a distinct key.



### METHOD 1 (Simple but Wrong)

Following is a simple program. For each node, check if left node of it is smaller than the node and right node of it is greater than the node.

```
int isBST(struct node* node)
{
    if (node == NULL)
        return 1;
```

```

/* false if left is > than node */
if (node->left != NULL && node->left->data > node->data)
    return 0;

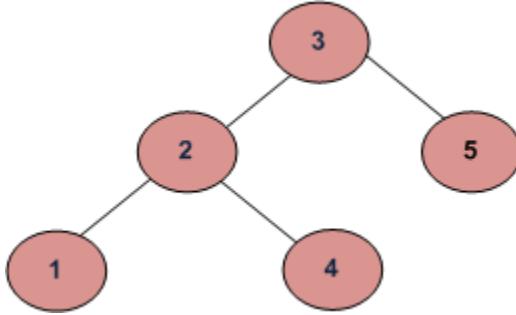
/* false if right is < than node */
if (node->right != NULL && node->right->data < node->data)
    return 0;

/* false if, recursively, the left or right is not a BST */
if (!isBST(node->left) || !isBST(node->right))
    return 0;

/* passing all that, it's a BST */
return 1;
}

```

This approach is wrong as this will return true for below binary tree (and below tree is not a BST because 4 is in left subtree of 3)



#### METHOD 2 (Correct but not efficient)

For each node, check if max value in left subtree is smaller than the node and min value in right subtree greater than the node.

```

/* Returns true if a binary tree is a binary search tree */
int isBST(struct node* node)
{
    if (node == NULL)
        return(true);

    /* false if the max of the left is > than us */
    if (node->left!=NULL && maxValue(node->left) > node->data)
        return(false);

    /* false if the min of the right is <= than us */
    if (node->right!=NULL && minValue(node->right) < node->data)
        return(false);
}

```

```
/* false if, recursively, the left or right is not a BST */
if (!isBST(node->left) || !isBST(node->right))
    return(false);

/* passing all that, it's a BST */
return(true);
}
```

It is assumed that you have helper functions minValue() and maxValue() that return the min or max int value from a non-empty tree

### METHOD 3 (Correct and Efficient)

Method 2 above runs slowly since it traverses over some parts of the tree many times. A better solution looks at each node only once. The trick is to write a utility helper function isBSTUtil(struct node\* node, int min, int max) that traverses down the tree keeping track of the narrowing min and max allowed values as it goes, looking at each node only once. The initial values for min and max should be INT\_MIN and INT\_MAX — they narrow from there.

```
/* Returns true if the given tree is a binary search tree
   (efficient version). */
int isBST(struct node* node)
{
    return(isBSTUtil(node, INT_MIN, INT_MAX));
}

/* Returns true if the given tree is a BST and its
   values are >= min and <= max. */
int isBSTUtil(struct node* node, int min, int max)
```

#### Implementation:

C

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};
```

```
int isBSTUtil(struct node* node, int min, int max);

/* Returns true if the given tree is a binary search tree
 * (efficient version). */
int isBST(struct node* node)
{
    return(isBSTUtil(node, INT_MIN, INT_MAX));
}

/* Returns true if the given tree is a BST and its
 * values are >= min and <= max. */
int isBSTUtil(struct node* node, int min, int max)
{
    /* an empty tree is BST */
    if (node==NULL)
        return 1;

    /* false if this node violates the min/max constraint */
    if (node->data < min || node->data > max)
        return 0;

    /* otherwise check the subtrees recursively,
     * tightening the min or max constraint */
    return
        isBSTUtil(node->left, min, node->data-1) && // Allow only distinct values
        isBSTUtil(node->right, node->data+1, max); // Allow only distinct values
}

/* Helper function that allocates a new node with the
 * given data and NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* node = (struct node*)
                           malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;

    return(node);
}

/* Driver program to test above functions*/
int main()
{
    struct node *root = newNode(4);
    root->left      = newNode(2);
    root->right     = newNode(5);
```

```
root->left->left = newNode(1);
root->left->right = newNode(3);

if(isBST(root))
    printf("Is BST");
else
    printf("Not a BST");

getchar();
return 0;
}
```

**Java**

```
//Java implementation to check if given Binary tree
//is a BST or not

/* Class containing left and right child of current
node and key value*/
class Node
{
    int data;
    Node left, right;

    public Node(int item)
    {
        data = item;
        left = right = null;
    }
}

public class BinaryTree
{
    //Root of the Binary Tree
    Node root;

    /* can give min and max value according to your code or
    can write a function to find min and max value of tree. */

    /* returns true if given search tree is binary
    search tree (efficient version) */
    boolean isBST()  {
        return isBSTUtil(root, Integer.MIN_VALUE,
                         Integer.MAX_VALUE);
    }

    /* Returns true if the given tree is a BST and its
    values are >= min and <= max. */
}
```

```
boolean isBSTUtil(Node node, int min, int max)
{
    /* an empty tree is BST */
    if (node == null)
        return true;

    /* false if this node violates the min/max constraints */
    if (node.data < min || node.data > max)
        return false;

    /* otherwise check the subtrees recursively
     * tightening the min/max constraints */
    // Allow only distinct values
    return (isBSTUtil(node.left, min, node.data-1) &&
            isBSTUtil(node.right, node.data+1, max));
}

/* Driver program to test above functions */
public static void main(String args[])
{
    BinaryTree tree = new BinaryTree();
    tree.root = new Node(4);
    tree.root.left = new Node(2);
    tree.root.right = new Node(5);
    tree.root.left.left = new Node(1);
    tree.root.left.right = new Node(3);

    if (tree.isBST())
        System.out.println("IS BST");
    else
        System.out.println("Not a BST");
}
}
```

### Python

```
# Python program to check if a binary tree is bst or not

INT_MAX = 4294967296
INT_MIN = -4294967296

# A binary tree node
class Node:

    # Constructor to create a new node
    def __init__(self, data):
        self.data = data
        self.left = None
```

```
self.right = None

# Returns true if the given tree is a binary search tree
# (efficient version)
def isBST(node):
    return (isBSTUtil(node, INT_MIN, INT_MAX))

# Returns true if the given tree is a BST and its values
# >= min and <= max
def isBSTUtil(node, mini, maxi):

    # An empty tree is BST
    if node is None:
        return True

    # False if this node violates min/max constraint
    if node.data < mini or node.data > maxi:
        return False

    # Otherwise check the subtrees recursively
    # tightening the min or max constraint
    return (isBSTUtil(node.left, mini, node.data -1) and
            isBSTUtil(node.right, node.data+1, maxi))

# Driver program to test above function
root = Node(4)
root.left = Node(2)
root.right = Node(5)
root.left.left = Node(1)
root.left.right = Node(3)

if (isBST(root)):
    print "Is BST"
else:
    print "Not a BST"

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

Time Complexity: O(n)

Auxiliary Space : O(1) if Function Call Stack size is not considered, otherwise O(n)

### Simplified Method 3

We can simplify method 2 using NULL pointers instead of INT\_MIN and INT\_MAX values.

```
// C++ program to check if a given tree is BST.
#include <bits/stdc++.h>
```

```
using namespace std;

/* A binary tree node has data, pointer to
   left child and a pointer to right child */
struct Node
{
    int data;
    struct Node* left, *right;
};

// Returns true if given tree is BST.
bool isBST(Node* root, Node* l=NULL, Node* r=NULL)
{
    // Base condition
    if (root == NULL)
        return true;

    // if left node exist then check it has
    // correct data or not i.e. left node's data
    // should be less than root's data
    if (l != NULL and root->data < l->data)
        return false;

    // if right node exist then check it has
    // correct data or not i.e. right node's data
    // should be greater than root's data
    if (r != NULL and root->data > r->data)
        return false;

    // check recursively for every node.
    return isBST(root->left, l, root) and
           isBST(root->right, root, r);
}

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct Node* newNode(int data)
{
    struct Node* node = new Node;
    node->data = data;
    node->left = node->right = NULL;
    return (node);
}

/* Driver program to test above functions*/
int main()
{
    struct Node *root = newNode(3);
```

```
root->left      = newNode(2);
root->right     = newNode(5);
root->left->left = newNode(1);
root->left->right = newNode(4);

if (isBST(root,NULL,NULL))
    cout << "Is BST";
else
    cout << "Not a BST";

return 0;
}
```

Output :

Not a BST

Thanks to **Abhinesh Garhwal** for suggesting above solution.

#### METHOD 4(Using In-Order Traversal)

Thanks to *LJW489* for suggesting this method.

- 1) Do In-Order Traversal of the given tree and store the result in a temp array.
- 3) Check if the temp array is sorted in ascending order, if it is, then the tree is BST.

Time Complexity: O(n)

We can avoid the use of Auxiliary Array. While doing In-Order traversal, we can keep track of previously visited node. If the value of the currently visited node is less than the previous value, then tree is not BST. Thanks to *ygos* for this space optimization.

#### C

```
bool isBST(struct node* root)
{
    static struct node *prev = NULL;

    // traverse the tree in inorder fashion and keep track of prev node
    if (root)
    {
        if (!isBST(root->left))
            return false;

        // Allows only distinct valued nodes
        if (prev != NULL && root->data <= prev->data)
            return false;

        prev = root;
    }
}
```

```
        return isBST(root->right);
    }

    return true;
}

Java

// Java implementation to check if given Binary tree
// is a BST or not

/* Class containing left and right child of current
node and key value*/
class Node
{
    int data;
    Node left, right;

    public Node(int item)
    {
        data = item;
        left = right = null;
    }
}

public class BinaryTree
{
    // Root of the Binary Tree
    Node root;

    // To keep tract of previous node in Inorder Traversal
    Node prev;

    boolean isBST()  {
        prev = null;
        return isBST(root);
    }

    /* Returns true if given search tree is binary
       search tree (efficient version) */
    boolean isBST(Node node)
    {
        // traverse the tree in inorder fashion and
        // keep a track of previous node
        if (node != null)
        {
            if (!isBST(node.left))
                return false;

            if (prev != null && prev.data > node.data)
                return false;

            prev = node;
            return isBST(node.right);
        }
        return true;
    }
}
```

```
// allows only distinct values node
if (prev != null && node.data <= prev.data )
    return false;
prev = node;
return isBST(node.right);
}
return true;
}

/* Driver program to test above functions */
public static void main(String args[])
{
    BinaryTree tree = new BinaryTree();
    tree.root = new Node(4);
    tree.root.left = new Node(2);
    tree.root.right = new Node(5);
    tree.root.left.left = new Node(1);
    tree.root.left.right = new Node(3);

    if (tree.isBST())
        System.out.println("IS BST");
    else
        System.out.println("Not a BST");
}
}
```

### Python3

```
# Python implementation to check if
# given Binary tree is a BST or not

# A binary tree node containing data
# field, left and right pointers
class Node:
    # constructor to create new node
    def __init__(self, val):
        self.data = val
        self.left = None
        self.right = None

    # global variable prev - to keep track
    # of previous node during Inorder
    # traversal
    prev = None

    # function to check if given binary
    # tree is BST
```

```
def isbst(root):

    # prev is a global variable
    global prev
    prev = None
    return isbst_rec(root)

# Helper function to test if binary
# tree is BST
# Traverse the tree in inorder fashion
# and keep track of previous node
# return true if tree is Binary
# search tree otherwise false
def isbst_rec(root):

    # prev is a global variable
    global prev

    # if tree is empty return true
    if root is None:
        return True

    if isbst_rec(root.left) is False:
        return False

    # if previous node's data is found
    # greater than the current node's
    # data return false
    if prev is not None and prev.data > root.data:
        return False

    # store the current node in prev
    prev = root
    return isbst_rec(root.right)

# driver code to test above function
root = Node(4)
root.left = Node(2)
root.right = Node(5)
root.left.left = Node(1)
root.left.right = Node(3)

if isbst(root):
    print("is BST")
else:
    print("not a BST")
```

```
# This code is contributed by
# Shweta Singh(shweta44)
```

The use of static variable can also be avoided by using reference to prev node as a parameter.

```
// C++ program to check if a given tree is BST.
#include <bits/stdc++.h>
using namespace std;

/* A binary tree node has data, pointer to
left child and a pointer to right child */
struct Node
{
    int data;
    struct Node* left, *right;

    Node(int data)
    {
        this->data = data;
        left = right = NULL;
    }
};

bool isBSTUtil(struct Node* root, Node *&prev)
{
    // traverse the tree in inorder fashion and
    // keep track of prev node
    if (root)
    {
        if (!isBSTUtil(root->left, prev))
            return false;

        // Allows only distinct valued nodes
        if (prev != NULL && root->data <= prev->data)
            return false;

        prev = root;

        return isBSTUtil(root->right, prev);
    }

    return true;
}

bool isBST(Node *root)
{
```

```
Node *prev = NULL;
return isBSTUtil(root, prev);
}

/* Driver program to test above functions*/
int main()
{
    struct Node *root = new Node(3);
    root->left      = new Node(2);
    root->right     = new Node(5);
    root->left->left = new Node(1);
    root->left->right = new Node(4);

    if (isBST(root))
        cout << "Is BST";
    else
        cout << "Not a BST";

    return 0;
}
```

**Sources:**

[http://en.wikipedia.org/wiki/Binary\\_search\\_tree](http://en.wikipedia.org/wiki/Binary_search_tree)  
<http://cslibrary.stanford.edu/110/BinaryTrees.html>

**Improved By :** [shweta44, ChandrahasAbburi](#)

**Source**

<https://www.geeksforgeeks.org/a-program-to-check-if-a-binary-tree-is-bst-or-not/>

## Chapter 2

# AA Trees | Set 1 (Introduction)

AA Trees | Set 1 (Introduction) - GeeksforGeeks

AA trees are the variation of the [red-black trees](#), a form of [binary search tree](#).

AA trees use the concept of **levels** to aid in **balancing binary trees**. The **level** of node (instead of colour) is used as balancing information. A link where child and parent's levels are same, is called a horizontal link, and is analogous to a red link in the red-black tree.

- The level of every leaf node is one.
- The level of red nodes are same as the level of their parent nodes and the links are called **horizontal links**.
- The level of black nodes are one less than the level of their parent node.

Additional storage requirement with every node is  $O(\log n)$  in red black trees instead of  $O(1)$  (only color in Red Black Trees), but AA trees simplify restructuring by removing many cases.

An AA tree follows same rule as [red-black trees](#) with the addition of single new rule that red nodes cannot be present as left child.

1. Every node can be either red (linked horizontally) or black.
2. There are no two adjacent red nodes (or horizontal links).
3. Every path from root to a NULL node has same number of black nodes (ot black links).
4. **Left link cannot NOT be red (horizontal).** (*New added rule*)

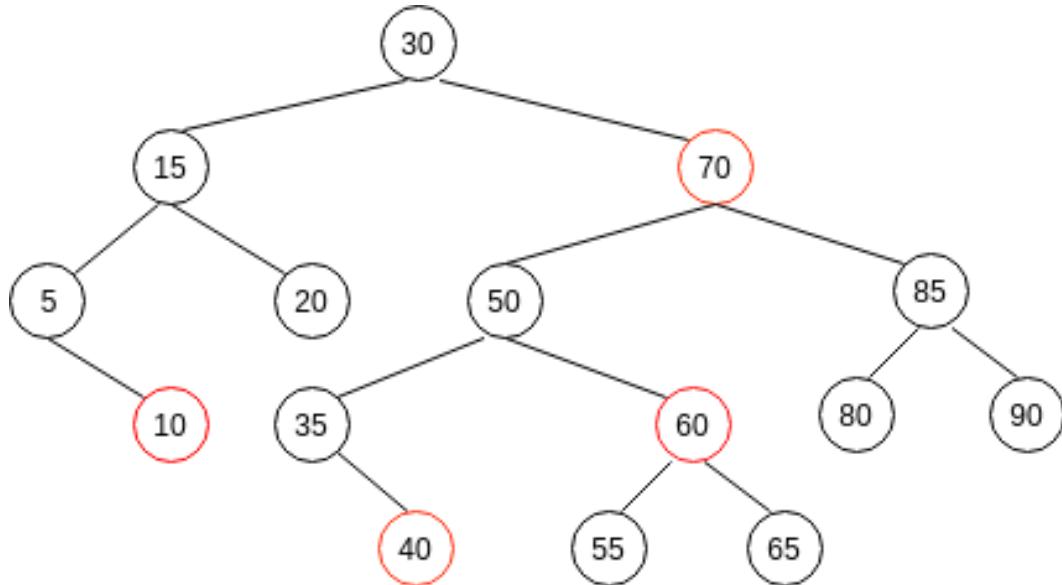
### Why AA trees :

The implementation and number of rotation cases in Red-Black Trees is complex. AA trees simplifies the algorithm.

- It eliminates half of the restructuring process by eliminating half of the rotation cases, which is easier to code.

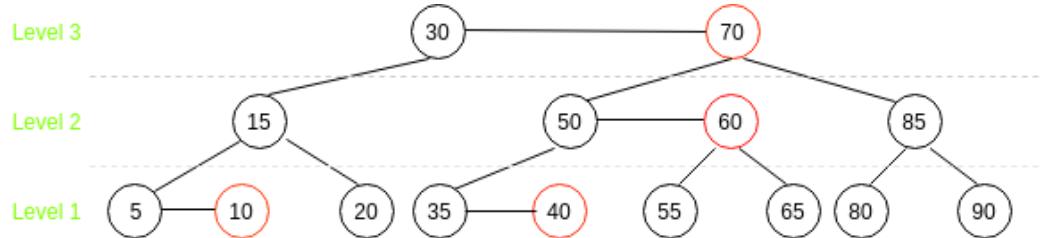
- It simplifies the deletion process by removing multiple cases.

Below tree is the example of AA tree :



Note that in the above tree there are no left red child which is the new added rule of AA Trees.

After re-drawing the above AA tree with levels and horizontal links (the red nodes are shown connected through horizontal or red links), the tree looks like:



Note that all the nodes on level 1 i.e. 5, 10, 20, 35, 40, 55, 65, 80, 90 are known as leaf nodes.

**So, in summarized way, for tree to be AA tree, it must satisfy the following five invariants:**

### Source

<https://www.geeksforgeeks.org/aa-trees-set-1-introduction/>

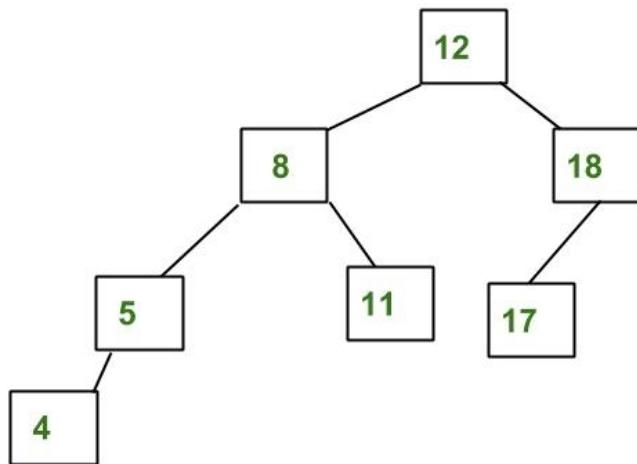
## Chapter 3

# AVL Tree | Set 1 (Insertion)

AVL Tree | Set 1 (Insertion) - GeeksforGeeks

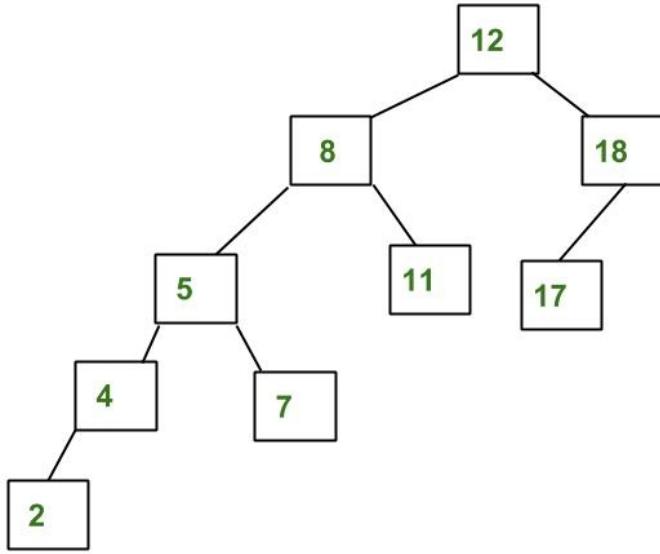
AVL tree is a self-balancing Binary Search Tree (BST) where the difference between heights of left and right subtrees cannot be more than one for all nodes.

**An Example Tree that is an AVL Tree**



The above tree is AVL because differences between heights of left and right subtrees for every node is less than or equal to 1.

**An Example Tree that is NOT an AVL Tree**



The above tree is not AVL because differences between heights of left and right subtrees for 8 and 18 is greater than 1.

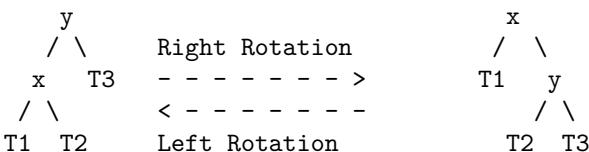
### Why AVL Trees?

Most of the BST operations (e.g., search, max, min, insert, delete.. etc) take  $O(h)$  time where  $h$  is the height of the BST. The cost of these operations may become  $O(n)$  for a skewed Binary tree. If we make sure that height of the tree remains  $O(\log n)$  after every insertion and deletion, then we can guarantee an upper bound of  $O(\log n)$  for all these operations. The height of an AVL tree is always  $O(\log n)$  where  $n$  is the number of nodes in the tree (See [this](#) video lecture for proof).

### Insertion

To make sure that the given tree remains AVL after every insertion, we must augment the standard BST insert operation to perform some re-balancing. Following are two basic operations that can be performed to re-balance a BST without violating the BST property ( $\text{keys(left)} < \text{key(root)} < \text{keys(right)}$ ). 1) Left Rotation 2) Right Rotation

T1, T2 and T3 are subtrees of the tree rooted with y (on the left side) or x (on the right side)



Keys in both of the above trees follow the following order

$\text{keys(T1)} < \text{key(x)} < \text{keys(T2)} < \text{key(y)} < \text{keys(T3)}$   
So BST property is not violated anywhere.

### Steps to follow for insertion

Let the newly inserted node be w

1) Perform standard BST insert for w.

2) Starting from w, travel up and find the first unbalanced node. Let z be the first unbalanced node, y be the child of z that comes on the path from w to z and x be the grandchild of z that comes on the path from w to z.

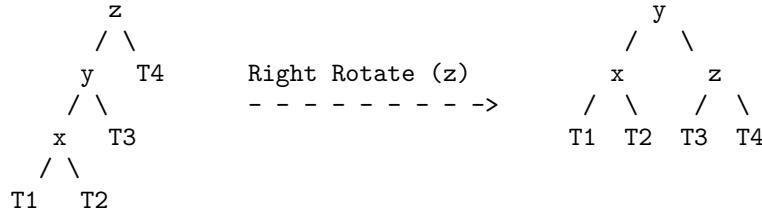
3) Re-balance the tree by performing appropriate rotations on the subtree rooted with z. There can be 4 possible cases that needs to be handled as x, y and z can be arranged in 4 ways. Following are the possible 4 arrangements:

- a) y is left child of z and x is left child of y (Left Left Case)
- b) y is left child of z and x is right child of y (Left Right Case)
- c) y is right child of z and x is right child of y (Right Right Case)
- d) y is right child of z and x is left child of y (Right Left Case)

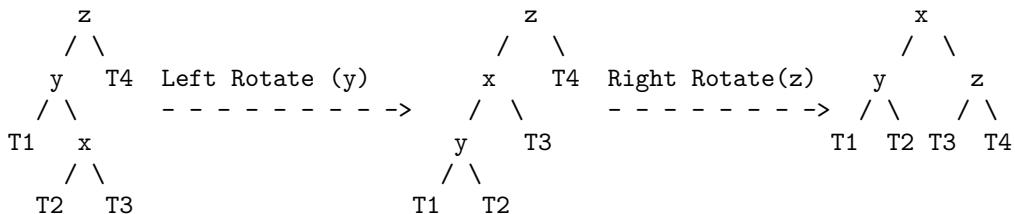
Following are the operations to be performed in above mentioned 4 cases. In all of the cases, we only need to re-balance the subtree rooted with z and the complete tree becomes balanced as the height of subtree (After appropriate rotations) rooted with z becomes same as it was before insertion. (See [this](#) video lecture for proof)

#### a) Left Left Case

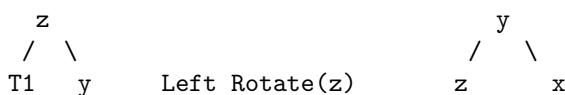
T1, T2, T3 and T4 are subtrees.

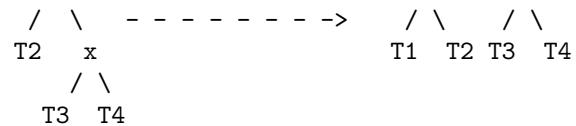


#### b) Left Right Case

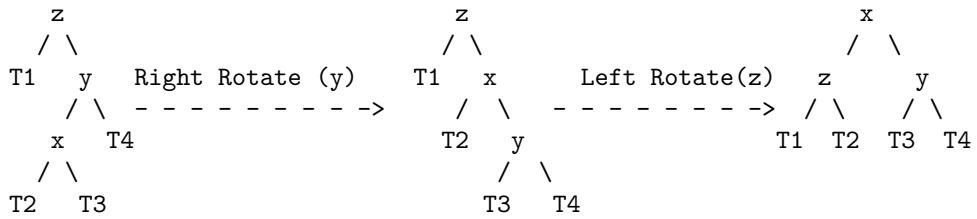


#### c) Right Right Case

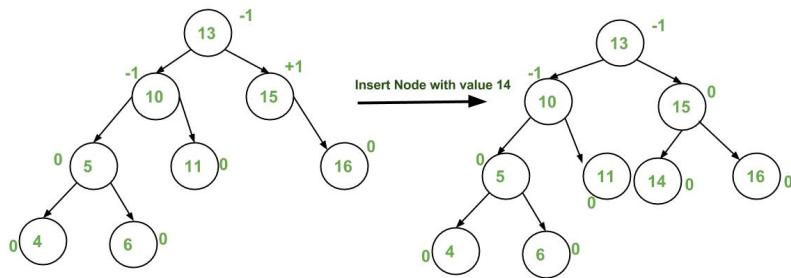


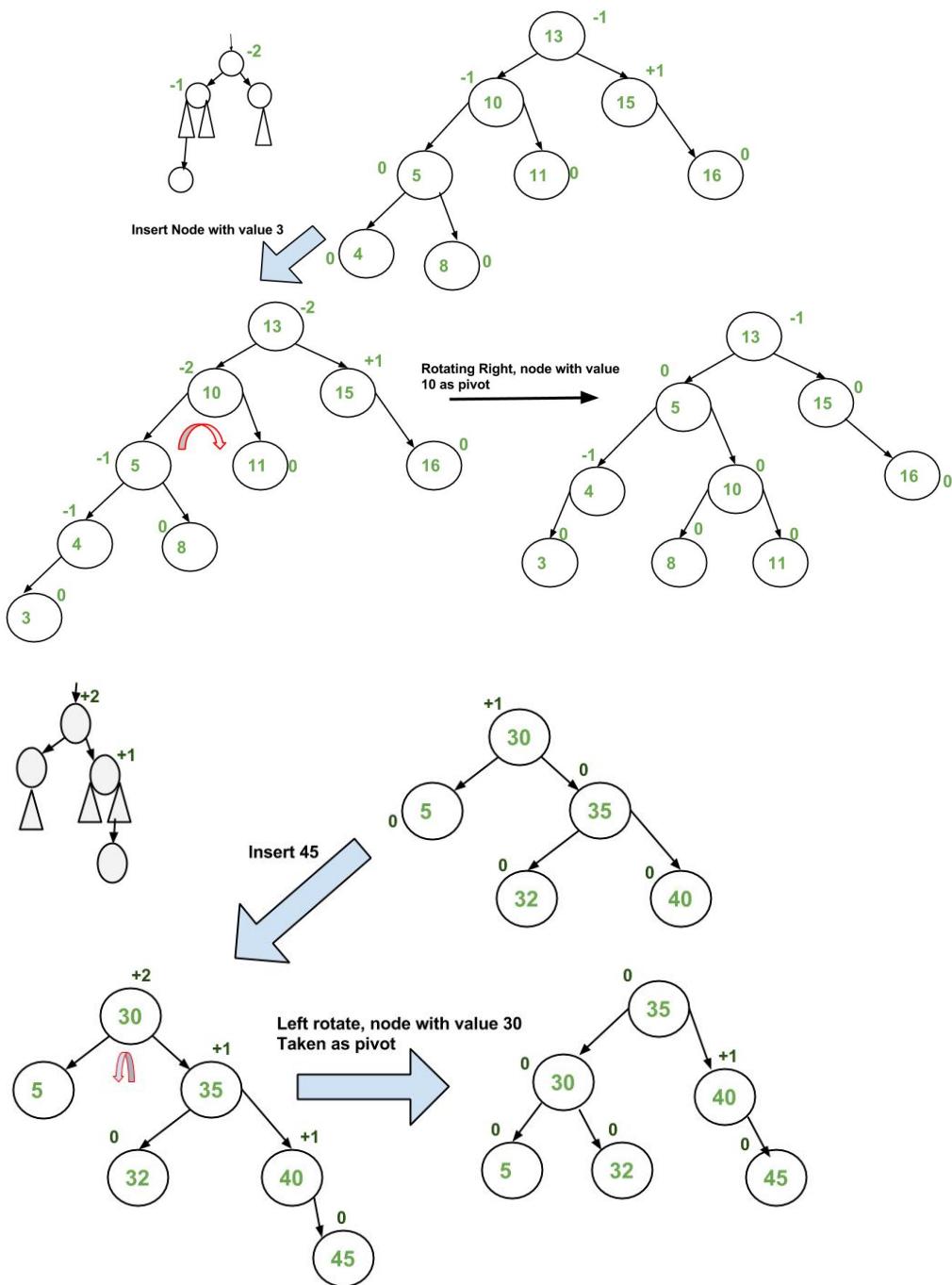


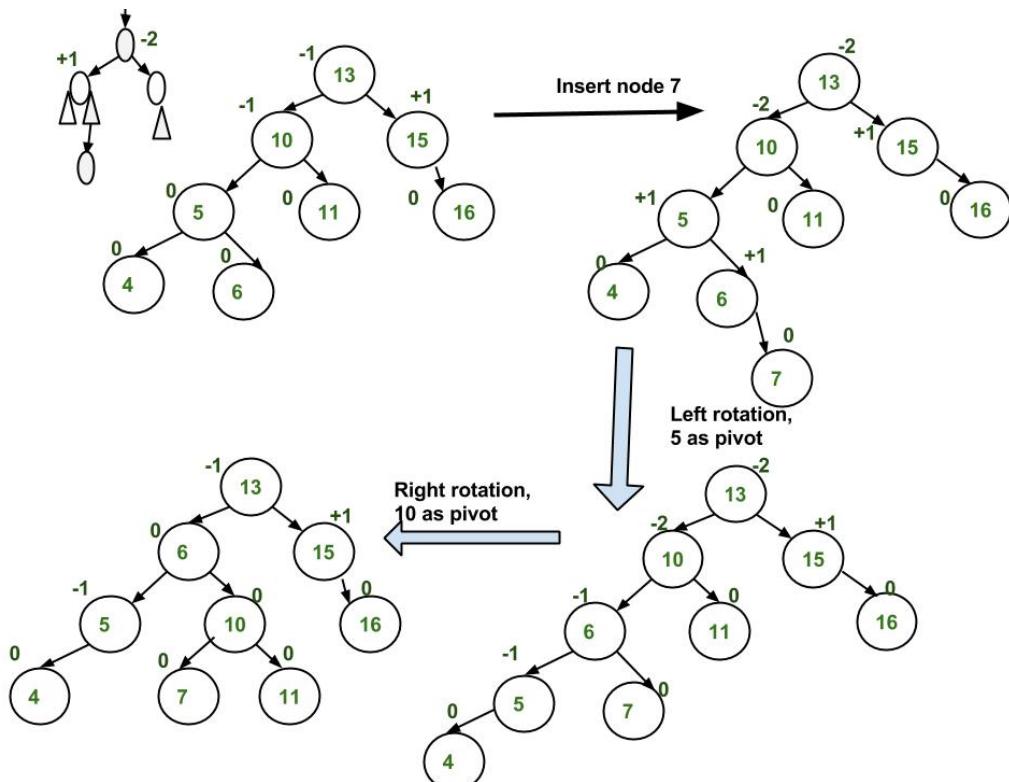
#### d) Right Left Case

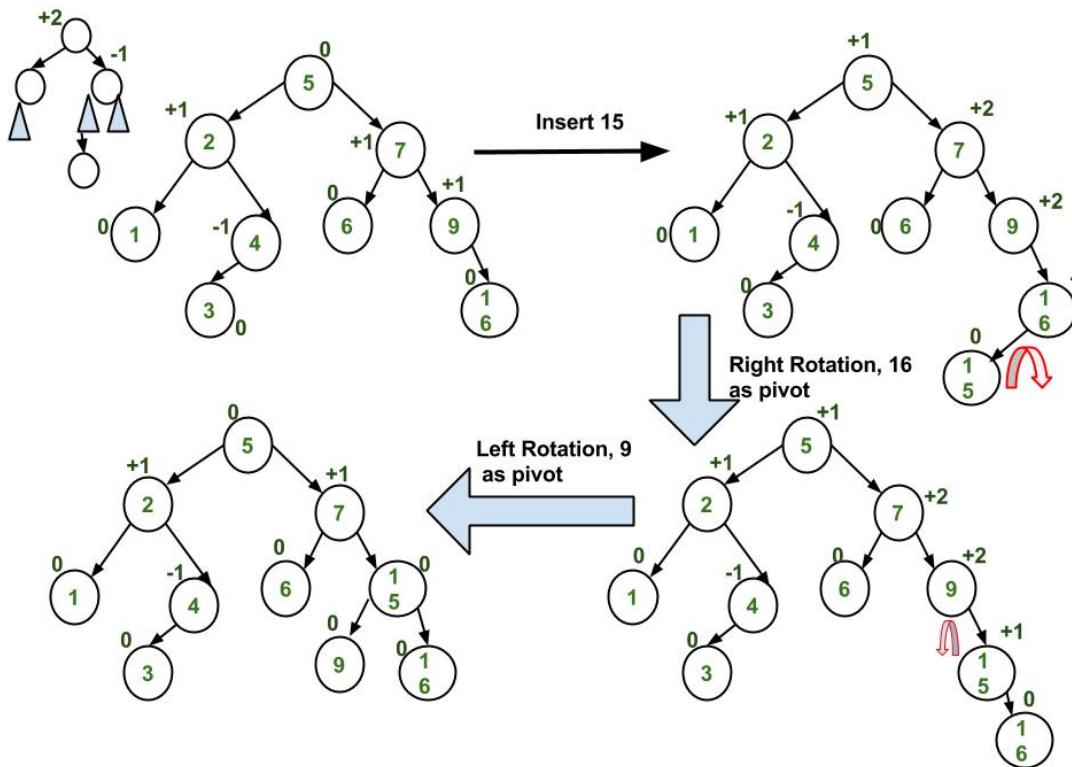


## Insertion Examples:









### implementation

Following is the implementation for AVL Tree Insertion. The following implementation uses the recursive BST insert to insert a new node. In the recursive BST insert, after insertion, we get pointers to all ancestors one by one in a bottom-up manner. So we don't need parent pointer to travel up. The recursive code itself travels up and visits all the ancestors of the newly inserted node.

- 1) Perform the normal BST insertion.
- 2) The current node must be one of the ancestors of the newly inserted node. Update the height of the current node.
- 3) Get the balance factor (left subtree height – right subtree height) of the current node.
- 4) If balance factor is greater than 1, then the current node is unbalanced and we are either in Left Left case or left Right case. To check whether it is left left case or not, compare the newly inserted key with the key in left subtree root.
- 5) If balance factor is less than -1, then the current node is unbalanced and we are either in Right Right case or Right-Left case. To check whether it is Right Right case or not, compare the newly inserted key with the key in right subtree root.

C

```

// C program to insert a node in AVL tree
#include<stdio.h>
#include<stdlib.h>
  
```

```

// An AVL tree node
struct Node
{
    int key;
    struct Node *left;
    struct Node *right;
    int height;
};

// A utility function to get maximum of two integers
int max(int a, int b);

// A utility function to get the height of the tree
int height(struct Node *N)
{
    if (N == NULL)
        return 0;
    return N->height;
}

// A utility function to get maximum of two integers
int max(int a, int b)
{
    return (a > b)? a : b;
}

/* Helper function that allocates a new node with the given key and
   NULL left and right pointers. */
struct Node* newNode(int key)
{
    struct Node* node = (struct Node*)
                           malloc(sizeof(struct Node));
    node->key    = key;
    node->left   = NULL;
    node->right  = NULL;
    node->height = 1; // new node is initially added at leaf
    return(node);
}

// A utility function to right rotate subtree rooted with y
// See the diagram given above.
struct Node *rightRotate(struct Node *y)
{
    struct Node *x = y->left;
    struct Node *T2 = x->right;

    // Perform rotation

```

```

x->right = y;
y->left = T2;

// Update heights
y->height = max(height(y->left), height(y->right))+1;
x->height = max(height(x->left), height(x->right))+1;

// Return new root
return x;
}

// A utility function to left rotate subtree rooted with x
// See the diagram given above.
struct Node *leftRotate(struct Node *x)
{
    struct Node *y = x->right;
    struct Node *T2 = y->left;

    // Perform rotation
    y->left = x;
    x->right = T2;

    // Update heights
    x->height = max(height(x->left), height(x->right))+1;
    y->height = max(height(y->left), height(y->right))+1;

    // Return new root
    return y;
}

// Get Balance factor of node N
int getBalance(struct Node *N)
{
    if (N == NULL)
        return 0;
    return height(N->left) - height(N->right);
}

// Recursive function to insert a key in the subtree rooted
// with node and returns the new root of the subtree.
struct Node* insert(struct Node* node, int key)
{
    /* 1. Perform the normal BST insertion */
    if (node == NULL)
        return(newNode(key));

    if (key < node->key)
        node->left = insert(node->left, key);
}

```

```

        else if (key > node->key)
            node->right = insert(node->right, key);
        else // Equal keys are not allowed in BST
            return node;

        /* 2. Update height of this ancestor node */
        node->height = 1 + max(height(node->left),
                               height(node->right));

        /* 3. Get the balance factor of this ancestor
           node to check whether this node became
           unbalanced */
        int balance = getBalance(node);

        // If this node becomes unbalanced, then
        // there are 4 cases

        // Left Left Case
        if (balance > 1 && key < node->left->key)
            return rightRotate(node);

        // Right Right Case
        if (balance < -1 && key > node->right->key)
            return leftRotate(node);

        // Left Right Case
        if (balance > 1 && key > node->left->key)
        {
            node->left = leftRotate(node->left);
            return rightRotate(node);
        }

        // Right Left Case
        if (balance < -1 && key < node->right->key)
        {
            node->right = rightRotate(node->right);
            return leftRotate(node);
        }

        /* return the (unchanged) node pointer */
        return node;
    }

    // A utility function to print preorder traversal
    // of the tree.
    // The function also prints height of every node
    void preOrder(struct Node *root)
    {

```

```
if(root != NULL)
{
    printf("%d ", root->key);
    preOrder(root->left);
    preOrder(root->right);
}
}

/* Drier program to test above function*/
int main()
{
    struct Node *root = NULL;

    /* Constructing tree given in the above figure */
    root = insert(root, 10);
    root = insert(root, 20);
    root = insert(root, 30);
    root = insert(root, 40);
    root = insert(root, 50);
    root = insert(root, 25);

    /* The constructed AVL Tree would be
        30
        / \
       20   40
      / \   \
     10  25   50
    */
    printf("Preorder traversal of the constructed AVL"
           " tree is \n");
    preOrder(root);

    return 0;
}
```

**Java**

```
// Java program for insertion in AVL Tree
class Node {
    int key, height;
    Node left, right;

    Node(int d) {
        key = d;
        height = 1;
    }
}
```

```

class AVLTree {

    Node root;

    // A utility function to get the height of the tree
    int height(Node N) {
        if (N == null)
            return 0;

        return N.height;
    }

    // A utility function to get maximum of two integers
    int max(int a, int b) {
        return (a > b) ? a : b;
    }

    // A utility function to right rotate subtree rooted with y
    // See the diagram given above.
    Node rightRotate(Node y) {
        Node x = y.left;
        Node T2 = x.right;

        // Perform rotation
        x.right = y;
        y.left = T2;

        // Update heights
        y.height = max(height(y.left), height(y.right)) + 1;
        x.height = max(height(x.left), height(x.right)) + 1;

        // Return new root
        return x;
    }

    // A utility function to left rotate subtree rooted with x
    // See the diagram given above.
    Node leftRotate(Node x) {
        Node y = x.right;
        Node T2 = y.left;

        // Perform rotation
        y.left = x;
        x.right = T2;

        // Update heights
        x.height = max(height(x.left), height(x.right)) + 1;
    }
}

```

```

y.height = max(height(y.left), height(y.right)) + 1;

// Return new root
return y;
}

// Get Balance factor of node N
int getBalance(Node N) {
    if (N == null)
        return 0;

    return height(N.left) - height(N.right);
}

Node insert(Node node, int key) {

    /* 1. Perform the normal BST insertion */
    if (node == null)
        return (new Node(key));

    if (key < node.key)
        node.left = insert(node.left, key);
    else if (key > node.key)
        node.right = insert(node.right, key);
    else // Duplicate keys not allowed
        return node;

    /* 2. Update height of this ancestor node */
    node.height = 1 + max(height(node.left),
                           height(node.right));

    /* 3. Get the balance factor of this ancestor
       node to check whether this node became
       unbalanced */
    int balance = getBalance(node);

    // If this node becomes unbalanced, then there
    // are 4 cases Left Left Case
    if (balance > 1 && key < node.left.key)
        return rightRotate(node);

    // Right Right Case
    if (balance < -1 && key > node.right.key)
        return leftRotate(node);

    // Left Right Case
    if (balance > 1 && key > node.left.key) {
        node.left = leftRotate(node.left);
}

```

```

        return rightRotate(node);
    }

    // Right Left Case
    if (balance < -1 && key < node.right.key) {
        node.right = rightRotate(node.right);
        return leftRotate(node);
    }

    /* return the (unchanged) node pointer */
    return node;
}

// A utility function to print preorder traversal
// of the tree.
// The function also prints height of every node
void preOrder(Node node) {
    if (node != null) {
        System.out.print(node.key + " ");
        preOrder(node.left);
        preOrder(node.right);
    }
}

public static void main(String[] args) {
    AVLTree tree = new AVLTree();

    /* Constructing tree given in the above figure */
    tree.root = tree.insert(tree.root, 10);
    tree.root = tree.insert(tree.root, 20);
    tree.root = tree.insert(tree.root, 30);
    tree.root = tree.insert(tree.root, 40);
    tree.root = tree.insert(tree.root, 50);
    tree.root = tree.insert(tree.root, 25);

    /* The constructed AVL Tree would be
       30
      / \
     20   40
    / \   \
   10  25   50
    */
    System.out.println("Preorder traversal" +
                       " of constructed tree is : ");
    tree.preOrder(tree.root);
}
}

// This code has been contributed by Mayank Jaiswal

```

**Python3**

```
# Python code to insert a node in AVL tree

# Generic tree node class
class TreeNode(object):
    def __init__(self, val):
        self.val = val
        self.left = None
        self.right = None
        self.height = 1

# AVL tree class which supports the
# Insert operation
class AVL_Tree(object):

    # Recursive function to insert key in
    # subtree rooted with node and returns
    # new root of subtree.
    def insert(self, root, key):

        # Step 1 - Perform normal BST
        if not root:
            return TreeNode(key)
        elif key < root.val:
            root.left = self.insert(root.left, key)
        else:
            root.right = self.insert(root.right, key)

        # Step 2 - Update the height of the
        # ancestor node
        root.height = 1 + max(self.getHeight(root.left),
                              self.getHeight(root.right))

        # Step 3 - Get the balance factor
        balance = self.getBalance(root)

        # Step 4 - If the node is unbalanced,
        # then try out the 4 cases
        # Case 1 - Left Left
        if balance > 1 and key < root.left.val:
            return self.rightRotate(root)

        # Case 2 - Right Right
        if balance < -1 and key > root.right.val:
            return self.leftRotate(root)

        # Case 3 - Left Right
        if balance > 1 and key > root.left.val:
            root.left = self.leftRotate(root.left)
            return self.rightRotate(root)

        # Case 4 - Right Left
        if balance < -1 and key < root.right.val:
            root.right = self.rightRotate(root.right)
            return self.leftRotate(root)

        return root
```

```

if balance > 1 and key > root.left.val:
    root.left = self.leftRotate(root.left)
    return self.rightRotate(root)

# Case 4 - Right Left
if balance < -1 and key < root.right.val:
    root.right = self.rightRotate(root.right)
    return self.leftRotate(root)

return root

def leftRotate(self, z):

    y = z.right
    T2 = y.left

    # Perform rotation
    y.left = z
    z.right = T2

    # Update heights
    z.height = 1 + max(self.getHeight(z.left),
                        self.getHeight(z.right))
    y.height = 1 + max(self.getHeight(y.left),
                        self.getHeight(y.right))

    # Return the new root
    return y

def rightRotate(self, z):

    y = z.left
    T3 = y.right

    # Perform rotation
    y.right = z
    z.left = T3

    # Update heights
    z.height = 1 + max(self.getHeight(z.left),
                        self.getHeight(z.right))
    y.height = 1 + max(self.getHeight(y.left),
                        self.getHeight(y.right))

    # Return the new root
    return y

def getHeight(self, root):

```

```
if not root:  
    return 0  
  
return root.height  
  
def getBalance(self, root):  
    if not root:  
        return 0  
  
    return self.getHeight(root.left) - self.getHeight(root.right)  
  
def preOrder(self, root):  
  
    if not root:  
        return  
  
    print("{0} ".format(root.val), end="")  
    self.preOrder(root.left)  
    self.preOrder(root.right)  
  
# Driver program to test above function  
myTree = AVL_Tree()  
root = None  
  
root = myTree.insert(root, 10)  
root = myTree.insert(root, 20)  
root = myTree.insert(root, 30)  
root = myTree.insert(root, 40)  
root = myTree.insert(root, 50)  
root = myTree.insert(root, 25)  
  
"""The constructed AVL Tree would be  
      30  
      / \br/>     20   40  
    / \   \br/>   10  25   50"""  
  
# Preorder Traversal  
print("Preorder traversal of the",  
      "constructed AVL tree is")  
myTree.preOrder(root)  
print()  
  
# This code is contributed by Ajitesh Pathak
```

Output:

Preorder traversal of the constructed AVL tree is  
30 20 10 25 40 50

**Time Complexity:** The rotation operations (left and right rotate) take constant time as only a few pointers are being changed there. Updating the height and getting the balance factor also takes constant time. So the time complexity of AVL insert remains same as BST insert which is  $O(h)$  where  $h$  is the height of the tree. Since AVL tree is balanced, the height is  $O(\log n)$ . So time complexity of AVL insert is  $O(\log n)$ .

#### **Comparison with Red Black Tree**

The AVL tree and other self-balancing search trees like Red Black are useful to get all basic operations done in  $O(\log n)$  time. The AVL trees are more balanced compared to Red-Black Trees, but they may cause more rotations during insertion and deletion. So if your application involves many frequent insertions and deletions, then Red Black trees should be preferred. And if the insertions and deletions are less frequent and search is the more frequent operation, then AVL tree should be preferred over [Red Black Tree](#).

Following is the post for delete.

[AVL Tree | Set 2 \(Deletion\)](#)

Following are some posts that have used self-balancing search trees.

[Median in a stream of integers \(running integers\)](#)

[Maximum of all subarrays of size k](#)

[Count smaller elements on right side](#)

#### **References:**

[IITD Video Lecture on AVL Tree Introduction](#)

[IITD Video Lecture on AVL Tree Insertion and Deletion](#)

#### **Source**

<https://www.geeksforgeeks.org/avl-tree-set-1-insertion/>

## Chapter 4

# AVL Tree | Set 2 (Deletion)

AVL Tree | Set 2 (Deletion) - GeeksforGeeks

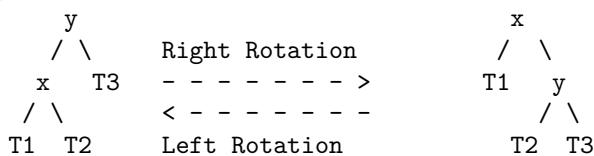
We have discussed AVL insertion in the [previous post](#). In this post, we will follow a similar approach for deletion.

### Steps to follow for deletion.

To make sure that the given tree remains AVL after every deletion, we must augment the standard BST delete operation to perform some re-balancing. Following are two basic operations that can be performed to re-balance a BST without violating the BST property ( $\text{keys(left)} < \text{key(root)} < \text{keys(right)}$ ).

- 1) Left Rotation
- 2) Right Rotation

T1, T2 and T3 are subtrees of the tree rooted with y (on left side)  
or x (on right side)



Keys in both of the above trees follow the following order

$\text{keys(T1)} < \text{key(x)} < \text{keys(T2)} < \text{key(y)} < \text{keys(T3)}$

So BST property is not violated anywhere.

Let w be the node to be deleted

- 1) Perform standard BST delete for w.
- 2) Starting from w, travel up and find the first unbalanced node. Let z be the first unbalanced node, y be the larger height child of z, and x be the larger height child of y. Note that the definitions of x and y are different from [insertion](#) here.
- 3) Re-balance the tree by performing appropriate rotations on the subtree rooted with z. There can be 4 possible cases that needs to be handled as x, y and z can be arranged in 4

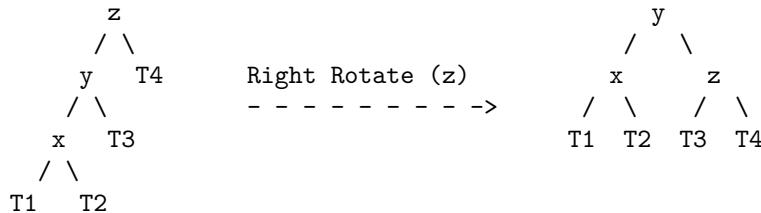
ways. Following are the possible 4 arrangements:

- a) y is left child of z and x is left child of y (Left Left Case)
- b) y is left child of z and x is right child of y (Left Right Case)
- c) y is right child of z and x is right child of y (Right Right Case)
- d) y is right child of z and x is left child of y (Right Left Case)

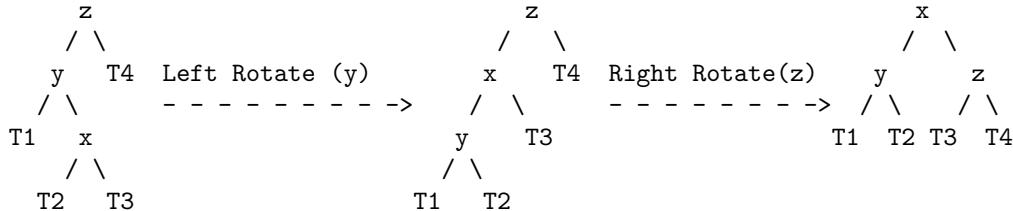
Like insertion, following are the operations to be performed in above mentioned 4 cases. Note that, unlike insertion, fixing the node z won't fix the complete AVL tree. After fixing z, we may have to fix ancestors of z as well (See [this video lecture](#) for proof)

#### a) Left Left Case

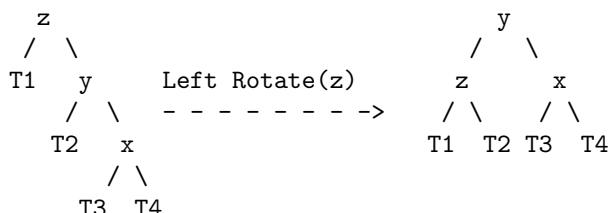
T1, T2, T3 and T4 are subtrees.



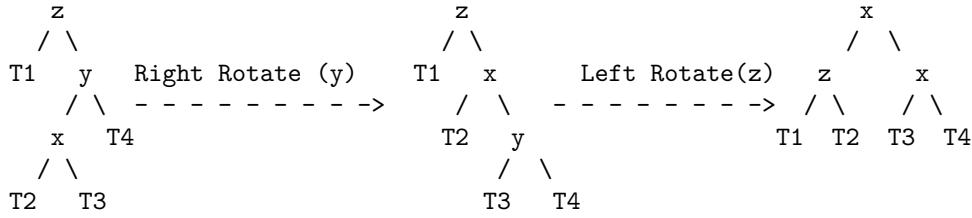
#### b) Left Right Case



#### c) Right Right Case



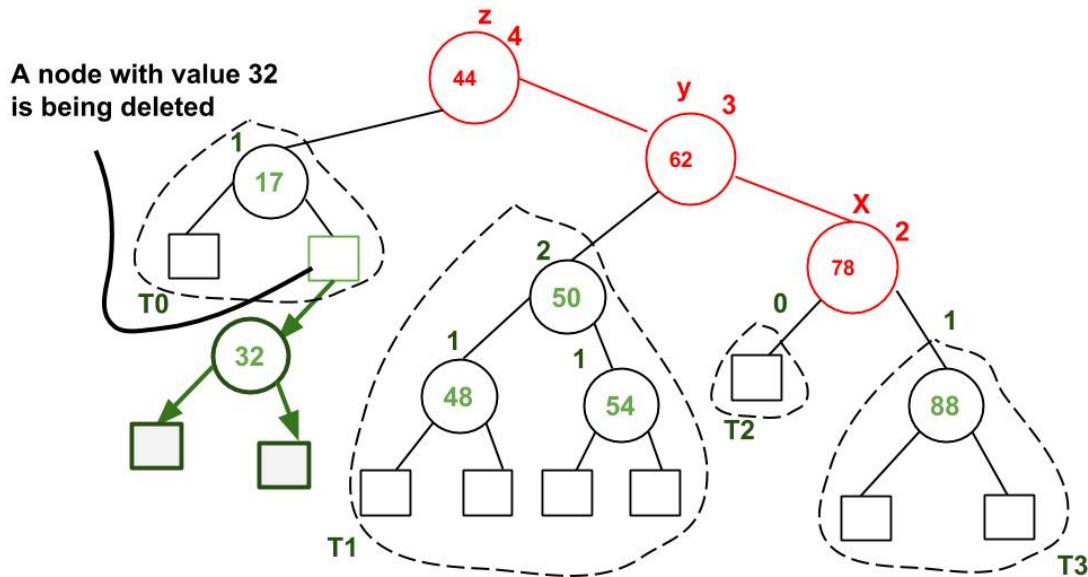
#### d) Right Left Case

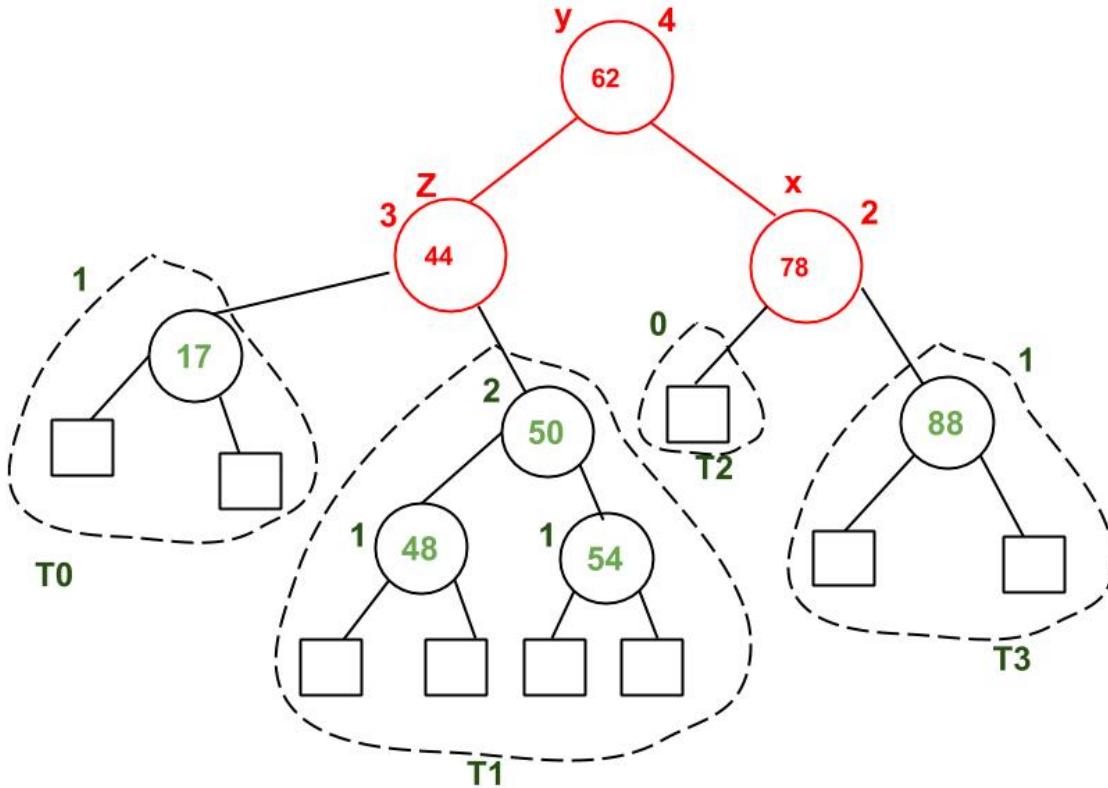


Unlike insertion, in deletion, after we perform a rotation at z, we may have to perform a rotation at ancestors of z. Thus, we must continue to trace the path until we reach the root.

**Example:**

#### Example of deletion from an AVL Tree:





A node with value 32 is being deleted. After deleting 32, we travel up and find the first unbalanced node which is 44. We mark it as z, its higher height child as y which is 52, and y's higher height child as x which could be either 78 or 50 as both are of same height. We have considered 78. Now the case is Right Right, so we perform left rotation.

### C implementation

Following is the C implementation for AVL Tree Deletion. The following C implementation uses the recursive BST delete as basis. In the recursive BST delete, after deletion, we get pointers to all ancestors one by one in bottom up manner. So we don't need parent pointer to travel up. The recursive code itself travels up and visits all the ancestors of the deleted node.

- 1) Perform the normal BST deletion.
- 2) The current node must be one of the ancestors of the deleted node. Update the height of the current node.
- 3) Get the balance factor (left subtree height – right subtree height) of the current node.
- 4) If balance factor is greater than 1, then the current node is unbalanced and we are either

in Left Left case or Left Right case. To check whether it is Left Left case or Left Right case, get the balance factor of left subtree. If balance factor of the left subtree is greater than or equal to 0, then it is Left Left case, else Left Right case.

5) If balance factor is less than -1, then the current node is unbalanced and we are either in Right Right case or Right Left case. To check whether it is Right Right case or Right Left case, get the balance factor of right subtree. If the balance factor of the right subtree is smaller than or equal to 0, then it is Right Right case, else Right Left case.

## C

```
// C program to delete a node from AVL Tree
#include<stdio.h>
#include<stdlib.h>

// An AVL tree node
struct Node
{
    int key;
    struct Node *left;
    struct Node *right;
    int height;
};

// A utility function to get maximum of two integers
int max(int a, int b);

// A utility function to get height of the tree
int height(struct Node *N)
{
    if (N == NULL)
        return 0;
    return N->height;
}

// A utility function to get maximum of two integers
int max(int a, int b)
{
    return (a > b)? a : b;
}

/* Helper function that allocates a new node with the given key and
NULL left and right pointers. */
struct Node* newNode(int key)
{
    struct Node* node = (struct Node*)
                           malloc(sizeof(struct Node));
    node->key    = key;
    node->left   = NULL;
    node->right  = NULL;
```

```

node->height = 1; // new node is initially added at leaf
return(node);
}

// A utility function to right rotate subtree rooted with y
// See the diagram given above.
struct Node *rightRotate(struct Node *y)
{
    struct Node *x = y->left;
    struct Node *T2 = x->right;

    // Perform rotation
    x->right = y;
    y->left = T2;

    // Update heights
    y->height = max(height(y->left), height(y->right))+1;
    x->height = max(height(x->left), height(x->right))+1;

    // Return new root
    return x;
}

// A utility function to left rotate subtree rooted with x
// See the diagram given above.
struct Node *leftRotate(struct Node *x)
{
    struct Node *y = x->right;
    struct Node *T2 = y->left;

    // Perform rotation
    y->left = x;
    x->right = T2;

    // Update heights
    x->height = max(height(x->left), height(x->right))+1;
    y->height = max(height(y->left), height(y->right))+1;

    // Return new root
    return y;
}

// Get Balance factor of node N
int getBalance(struct Node *N)
{
    if (N == NULL)
        return 0;
    return height(N->left) - height(N->right);
}

```

```

}

struct Node* insert(struct Node* node, int key)
{
    /* 1. Perform the normal BST rotation */
    if (node == NULL)
        return(newNode(key));

    if (key < node->key)
        node->left = insert(node->left, key);
    else if (key > node->key)
        node->right = insert(node->right, key);
    else // Equal keys not allowed
        return node;

    /* 2. Update height of this ancestor node */
    node->height = 1 + max(height(node->left),
                           height(node->right));

    /* 3. Get the balance factor of this ancestor
       node to check whether this node became
       unbalanced */
    int balance = getBalance(node);

    // If this node becomes unbalanced, then there are 4 cases

    // Left Left Case
    if (balance > 1 && key < node->left->key)
        return rightRotate(node);

    // Right Right Case
    if (balance < -1 && key > node->right->key)
        return leftRotate(node);

    // Left Right Case
    if (balance > 1 && key > node->left->key)
    {
        node->left = leftRotate(node->left);
        return rightRotate(node);
    }

    // Right Left Case
    if (balance < -1 && key < node->right->key)
    {
        node->right = rightRotate(node->right);
        return leftRotate(node);
    }
}

```

```

/* return the (unchanged) node pointer */
return node;
}

/* Given a non-empty binary search tree, return the
   node with minimum key value found in that tree.
   Note that the entire tree does not need to be
   searched. */
struct Node * minValueNode(struct Node* node)
{
    struct Node* current = node;

    /* loop down to find the leftmost leaf */
    while (current->left != NULL)
        current = current->left;

    return current;
}

// Recursive function to delete a node with given key
// from subtree with given root. It returns root of
// the modified subtree.
struct Node* deleteNode(struct Node* root, int key)
{
    // STEP 1: PERFORM STANDARD BST DELETE

    if (root == NULL)
        return root;

    // If the key to be deleted is smaller than the
    // root's key, then it lies in left subtree
    if (key < root->key)
        root->left = deleteNode(root->left, key);

    // If the key to be deleted is greater than the
    // root's key, then it lies in right subtree
    else if( key > root->key )
        root->right = deleteNode(root->right, key);

    // if key is same as root's key, then This is
    // the node to be deleted
    else
    {
        // node with only one child or no child
        if( (root->left == NULL) || (root->right == NULL) )
        {
            struct Node *temp = root->left ? root->left :
                                         root->right;

```

```

// No child case
if (temp == NULL)
{
    temp = root;
    root = NULL;
}
else // One child case
    *root = *temp; // Copy the contents of
                    // the non-empty child
    free(temp);
}
else
{
    // node with two children: Get the inorder
    // successor (smallest in the right subtree)
    struct Node* temp = minValueNode(root->right);

    // Copy the inorder successor's data to this node
    root->key = temp->key;

    // Delete the inorder successor
    root->right = deleteNode(root->right, temp->key);
}
}

// If the tree had only one node then return
if (root == NULL)
    return root;

// STEP 2: UPDATE HEIGHT OF THE CURRENT NODE
root->height = 1 + max(height(root->left),
                        height(root->right));

// STEP 3: GET THE BALANCE FACTOR OF THIS NODE (to
// check whether this node became unbalanced)
int balance = getBalance(root);

// If this node becomes unbalanced, then there are 4 cases

// Left Left Case
if (balance > 1 && getBalance(root->left) >= 0)
    return rightRotate(root);

// Left Right Case
if (balance > 1 && getBalance(root->left) < 0)
{
    root->left = leftRotate(root->left);
}

```

```

        return rightRotate(root);
    }

    // Right Right Case
    if (balance < -1 && getBalance(root->right) <= 0)
        return leftRotate(root);

    // Right Left Case
    if (balance < -1 && getBalance(root->right) > 0)
    {
        root->right = rightRotate(root->right);
        return leftRotate(root);
    }

    return root;
}

// A utility function to print preorder traversal of
// the tree.
// The function also prints height of every node
void preOrder(struct Node *root)
{
    if(root != NULL)
    {
        printf("%d ", root->key);
        preOrder(root->left);
        preOrder(root->right);
    }
}

/* Driver program to test above function*/
int main()
{
    struct Node *root = NULL;

    /* Constructing tree given in the above figure */
    root = insert(root, 9);
    root = insert(root, 5);
    root = insert(root, 10);
    root = insert(root, 0);
    root = insert(root, 6);
    root = insert(root, 11);
    root = insert(root, -1);
    root = insert(root, 1);
    root = insert(root, 2);

    /* The constructed AVL Tree would be

```

```
        /   \
      1     10
     / \   /
    0   5   11
   /   / \
  -1   2   6
 */
printf("Preorder traversal of the constructed AVL "
       "tree is \n");
preOrder(root);

root = deleteNode(root, 10);

/* The AVL Tree after deletion of 10
     1
    / \
   0   9
  /   / \
 -1   5   11
  / \
 2   6
*/
printf("\nPreorder traversal after deletion of 10 \n");
preOrder(root);

return 0;
}
```

**Java**

```
// Java program for deletion in AVL Tree

class Node
{
    int key, height;
    Node left, right;

    Node(int d)
    {
        key = d;
        height = 1;
    }
}

class AVLTree
{
```

```

Node root;

// A utility function to get height of the tree
int height(Node N)
{
    if (N == null)
        return 0;
    return N.height;
}

// A utility function to get maximum of two integers
int max(int a, int b)
{
    return (a > b) ? a : b;
}

// A utility function to right rotate subtree rooted with y
// See the diagram given above.
Node rightRotate(Node y)
{
    Node x = y.left;
    Node T2 = x.right;

    // Perform rotation
    x.right = y;
    y.left = T2;

    // Update heights
    y.height = max(height(y.left), height(y.right)) + 1;
    x.height = max(height(x.left), height(x.right)) + 1;

    // Return new root
    return x;
}

// A utility function to left rotate subtree rooted with x
// See the diagram given above.
Node leftRotate(Node x)
{
    Node y = x.right;
    Node T2 = y.left;

    // Perform rotation
    y.left = x;
    x.right = T2;

    // Update heights
    x.height = max(height(x.left), height(x.right)) + 1;
}

```

```

y.height = max(height(y.left), height(y.right)) + 1;

// Return new root
return y;
}

// Get Balance factor of node N
int getBalance(Node N)
{
    if (N == null)
        return 0;
    return height(N.left) - height(N.right);
}

Node insert(Node node, int key)
{
    /* 1. Perform the normal BST rotation */
    if (node == null)
        return (new Node(key));

    if (key < node.key)
        node.left = insert(node.left, key);
    else if (key > node.key)
        node.right = insert(node.right, key);
    else // Equal keys not allowed
        return node;

    /* 2. Update height of this ancestor node */
    node.height = 1 + max(height(node.left),
                           height(node.right));

    /* 3. Get the balance factor of this ancestor
       node to check whether this node became
       Unbalanced */
    int balance = getBalance(node);

    // If this node becomes unbalanced, then
    // there are 4 cases Left Left Case
    if (balance > 1 && key < node.left.key)
        return rightRotate(node);

    // Right Right Case
    if (balance < -1 && key > node.right.key)
        return leftRotate(node);

    // Left Right Case
    if (balance > 1 && key > node.left.key)
    {

```

```

        node.left = leftRotate(node.left);
        return rightRotate(node);
    }

    // Right Left Case
    if (balance < -1 && key < node.right.key)
    {
        node.right = rightRotate(node.right);
        return leftRotate(node);
    }

    /* return the (unchanged) node pointer */
    return node;
}

/* Given a non-empty binary search tree, return the
node with minimum key value found in that tree.
Note that the entire tree does not need to be
searched. */
Node minValueNode(Node node)
{
    Node current = node;

    /* loop down to find the leftmost leaf */
    while (current.left != null)
        current = current.left;

    return current;
}

Node deleteNode(Node root, int key)
{
    // STEP 1: PERFORM STANDARD BST DELETE
    if (root == null)
        return root;

    // If the key to be deleted is smaller than
    // the root's key, then it lies in left subtree
    if (key < root.key)
        root.left = deleteNode(root.left, key);

    // If the key to be deleted is greater than the
    // root's key, then it lies in right subtree
    else if (key > root.key)
        root.right = deleteNode(root.right, key);

    // if key is same as root's key, then this is the node
    // to be deleted
}

```

```

else
{

    // node with only one child or no child
    if ((root.left == null) || (root.right == null))
    {
        Node temp = null;
        if (temp == root.left)
            temp = root.right;
        else
            temp = root.left;

        // No child case
        if (temp == null)
        {
            temp = root;
            root = null;
        }
        else // One child case
            root = temp; // Copy the contents of
                           // the non-empty child
    }
    else
    {

        // node with two children: Get the inorder
        // successor (smallest in the right subtree)
        Node temp = minValueNode(root.right);

        // Copy the inorder successor's data to this node
        root.key = temp.key;

        // Delete the inorder successor
        root.right = deleteNode(root.right, temp.key);
    }
}

// If the tree had only one node then return
if (root == null)
    return root;

// STEP 2: UPDATE HEIGHT OF THE CURRENT NODE
root.height = max(height(root.left), height(root.right)) + 1;

// STEP 3: GET THE BALANCE FACTOR OF THIS NODE (to check whether
// this node became unbalanced)
int balance = getBalance(root);

```

```

// If this node becomes unbalanced, then there are 4 cases
// Left Left Case
if (balance > 1 && getBalance(root.left) >= 0)
    return rightRotate(root);

// Left Right Case
if (balance > 1 && getBalance(root.left) < 0)
{
    root.left = leftRotate(root.left);
    return rightRotate(root);
}

// Right Right Case
if (balance < -1 && getBalance(root.right) <= 0)
    return leftRotate(root);

// Right Left Case
if (balance < -1 && getBalance(root.right) > 0)
{
    root.right = rightRotate(root.right);
    return leftRotate(root);
}

return root;
}

// A utility function to print preorder traversal of
// the tree. The function also prints height of every
// node
void preOrder(Node node)
{
    if (node != null)
    {
        System.out.print(node.key + " ");
        preOrder(node.left);
        preOrder(node.right);
    }
}

public static void main(String[] args)
{
    AVLTree tree = new AVLTree();

    /* Constructing tree given in the above figure */
    tree.root = tree.insert(tree.root, 9);
    tree.root = tree.insert(tree.root, 5);
    tree.root = tree.insert(tree.root, 10);
    tree.root = tree.insert(tree.root, 0);
}

```

```
tree.root = tree.insert(tree.root, 6);
tree.root = tree.insert(tree.root, 11);
tree.root = tree.insert(tree.root, -1);
tree.root = tree.insert(tree.root, 1);
tree.root = tree.insert(tree.root, 2);

/* The constructed AVL Tree would be
9
/
1 10
/ \ \
0 5 11
/ / \
-1 2 6
*/
System.out.println("Preorder traversal of "+
                     "constructed tree is : ");
tree.preOrder(tree.root);

tree.root = tree.deleteNode(tree.root, 10);

/* The AVL Tree after deletion of 10
1
/
0 9
/   / \
-1 5 11
/ \
2 6
*/
System.out.println("");
System.out.println("Preorder traversal after "+
                     "deletion of 10 :");
tree.preOrder(tree.root);
}

}

// This code has been contributed by Mayank Jaiswal
```

### Python3

```
# Python code to delete a node in AVL tree
# Generic tree node class
class TreeNode(object):
    def __init__(self, val):
        self.val = val
        self.left = None
        self.right = None
```

```

        self.height = 1

# AVL tree class which supports insertion,
# deletion operations
class AVL_Tree(object):

    def insert(self, root, key):

        # Step 1 - Perform normal BST
        if not root:
            return TreeNode(key)
        elif key < root.val:
            root.left = self.insert(root.left, key)
        else:
            root.right = self.insert(root.right, key)

        # Step 2 - Update the height of the
        # ancestor node
        root.height = 1 + max(self.getHeight(root.left),
                              self.getHeight(root.right))

        # Step 3 - Get the balance factor
        balance = self.getBalance(root)

        # Step 4 - If the node is unbalanced,
        # then try out the 4 cases
        # Case 1 - Left Left
        if balance > 1 and key < root.left.val:
            return self.rightRotate(root)

        # Case 2 - Right Right
        if balance < -1 and key > root.right.val:
            return self.leftRotate(root)

        # Case 3 - Left Right
        if balance > 1 and key > root.left.val:
            root.left = self.leftRotate(root.left)
            return self.rightRotate(root)

        # Case 4 - Right Left
        if balance < -1 and key < root.right.val:
            root.right = self.rightRotate(root.right)
            return self.leftRotate(root)

    return root

# Recursive function to delete a node with
# given key from subtree with given root.

```

```

# It returns root of the modified subtree.
def delete(self, root, key):

    # Step 1 - Perform standard BST delete
    if not root:
        return root

    elif key < root.val:
        root.left = self.delete(root.left, key)

    elif key > root.val:
        root.right = self.delete(root.right, key)

    else:
        if root.left is None:
            temp = root.right
            root = None
            return temp

        elif root.right is None:
            temp = root.left
            root = None
            return temp

        temp = self.getMinValueNode(root.right)
        root.val = temp.val
        root.right = self.delete(root.right,
                               temp.val)

    # If the tree has only one node,
    # simply return it
    if root is None:
        return root

    # Step 2 - Update the height of the
    # ancestor node
    root.height = 1 + max(self.getHeight(root.left),
                          self.getHeight(root.right))

    # Step 3 - Get the balance factor
    balance = self.getBalance(root)

    # Step 4 - If the node is unbalanced,
    # then try out the 4 cases
    # Case 1 - Left Left
    if balance > 1 and self.getBalance(root.left) >= 0:
        return self.rightRotate(root)

```

```

# Case 2 - Right Right
if balance < -1 and self.getBalance(root.right) <= 0:
    return self.leftRotate(root)

# Case 3 - Left Right
if balance > 1 and self.getBalance(root.left) < 0:
    root.left = self.leftRotate(root.left)
    return self.rightRotate(root)

# Case 4 - Right Left
if balance < -1 and self.getBalance(root.right) > 0:
    root.right = self.rightRotate(root.right)
    return self.leftRotate(root)

return root

def leftRotate(self, z):

    y = z.right
    T2 = y.left

    # Perform rotation
    y.left = z
    z.right = T2

    # Update heights
    z.height = 1 + max(self.getHeight(z.left),
                        self.getHeight(z.right))
    y.height = 1 + max(self.getHeight(y.left),
                        self.getHeight(y.right))

    # Return the new root
    return y

def rightRotate(self, z):

    y = z.left
    T3 = y.right

    # Perform rotation
    y.right = z
    z.left = T3

    # Update heights
    z.height = 1 + max(self.getHeight(z.left),
                        self.getHeight(z.right))
    y.height = 1 + max(self.getHeight(y.left),
                        self.getHeight(y.right))

```

```
# Return the new root
return y

def getHeight(self, root):
    if not root:
        return 0

    return root.height

def getBalance(self, root):
    if not root:
        return 0

    return self.getHeight(root.left) - self.getHeight(root.right)

def getMinValueNode(self, root):
    if root is None or root.left is None:
        return root

    return self.getMinValueNode(root.left)

def preOrder(self, root):

    if not root:
        return

    print("{0} ".format(root.val), end="")
    self.preOrder(root.left)
    self.preOrder(root.right)

myTree = AVL_Tree()
root = None
nums = [9, 5, 10, 0, 6, 11, -1, 1, 2]

for num in nums:
    root = myTree.insert(root, num)

# Preorder Traversal
print("Preorder Traversal after insertion -")
myTree.preOrder(root)
print()

# Delete
key = 10
root = myTree.delete(root, key)
```

```
# Preorder Traversal
print("Preorder Traversal after deletion -")
myTree.preOrder(root)
print()

# This code is contributed by Ajitesh Pathak
```

Output:

```
Preorder traversal of the constructed AVL tree is
9 1 0 -1 5 2 6 10 11
Preorder traversal after deletion of 10
1 0 -1 9 5 2 6 11
```

**Time Complexity:** The rotation operations (left and right rotate) take constant time as only few pointers are being changed there. Updating the height and getting the balance factor also take constant time. So the time complexity of AVL delete remains same as BST delete which is  $O(h)$  where  $h$  is height of the tree. Since AVL tree is balanced, the height is  $O(\log n)$ . So time complexity of AVL delete is  $O(\log n)$ .

**References:**

<https://www.cs.purdue.edu/homes/ayg/CS251/slides/chap7b.pdf>  
[IITD Video Lecture on AVL Tree Insertion and Deletion](#)

**Improved By :** [AnkushRodewad, KP1975](#)

## Source

<https://www.geeksforgeeks.org/avl-tree-set-2-deletion/>

## Chapter 5

# AVL with duplicate keys

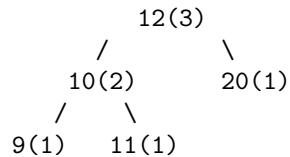
AVL with duplicate keys - GeeksforGeeks

Please refer below post before reading about AVL tree handling of duplicates.

[How to handle duplicates in Binary Search Tree?](#)

The is to augment [AVL tree](#) node to store count together with regular fields like key, left and right pointers.

Insertion of keys 12, 10, 20, 9, 11, 10, 12, 12 in an empty Binary Search Tree would create following.



Count of a key is shown in bracket

Below is C implementation of normal AVL Tree with count with every key. This code basically is taken from [code for insert and delete in AVL tree](#). The changes made for handling duplicates are highlighted, rest of the code is same.

The important thing to note is changes are very similar to simple Binary Search Tree changes.

```
// AVL tree that handles duplicates
#include<stdio.h>
#include<stdlib.h>

// An AVL tree node
struct node
{
    int key,
```

```

    struct node *left;
    struct node *right;
    int height;
    int count;
};

// A utility function to get maximum of two integers
int max(int a, int b);

// A utility function to get height of the tree
int height(struct node *N)
{
    if (N == NULL)
        return 0;
    return N->height;
}

// A utility function to get maximum of two integers
int max(int a, int b)
{
    return (a > b)? a : b;
}

/* Helper function that allocates a new node with the given key and
   NULL left and right pointers. */
struct node* newNode(int key)
{
    struct node* node = (struct node*)
                           malloc(sizeof(struct node));
    node->key    = key;
    node->left   = NULL;
    node->right  = NULL;
    node->height = 1; // new node is initially added at leaf
    node->count  = 1;
    return(node);
}

// A utility function to right rotate subtree rooted with y
// See the diagram given above.
struct node *rightRotate(struct node *y)
{
    struct node *x = y->left;
    struct node *T2 = x->right;

    // Perform rotation
    x->right = y;
    y->left = T2;
}

```

```

// Update heights
y->height = max(height(y->left), height(y->right))+1;
x->height = max(height(x->left), height(x->right))+1;

// Return new root
return x;
}

// A utility function to left rotate subtree rooted with x
// See the diagram given above.
struct node *leftRotate(struct node *x)
{
    struct node *y = x->right;
    struct node *T2 = y->left;

    // Perform rotation
    y->left = x;
    x->right = T2;

    // Update heights
    x->height = max(height(x->left), height(x->right))+1;
    y->height = max(height(y->left), height(y->right))+1;

    // Return new root
    return y;
}

// Get Balance factor of node N
int getBalance(struct node *N)
{
    if (N == NULL)
        return 0;
    return height(N->left) - height(N->right);
}

struct node* insert(struct node* node, int key)
{
    /* 1. Perform the normal BST rotation */
    if (node == NULL)
        return (newNode(key));

    // If key already exists in BST, increment count and return
    if (key == node->key)
    {
        (node->count)++;
        return node;
    }
}

```

```

/* Otherwise, recur down the tree */
if (key < node->key)
    node->left = insert(node->left, key);
else
    node->right = insert(node->right, key);

/* 2. Update height of this ancestor node */
node->height = max(height(node->left), height(node->right)) + 1;

/* 3. Get the balance factor of this ancestor node to check whether
   this node became unbalanced */
int balance = getBalance(node);

// If this node becomes unbalanced, then there are 4 cases

// Left Left Case
if (balance > 1 && key < node->left->key)
    return rightRotate(node);

// Right Right Case
if (balance < -1 && key > node->right->key)
    return leftRotate(node);

// Left Right Case
if (balance > 1 && key > node->left->key)
{
    node->left = leftRotate(node->left);
    return rightRotate(node);
}

// Right Left Case
if (balance < -1 && key < node->right->key)
{
    node->right = rightRotate(node->right);
    return leftRotate(node);
}

/* return the (unchanged) node pointer */
return node;
}

/* Given a non-empty binary search tree, return the node with minimum
   key value found in that tree. Note that the entire tree does not
   need to be searched. */
struct node * minValueNode(struct node* node)
{
    struct node* current = node;

```

```
/* loop down to find the leftmost leaf */
while (current->left != NULL)
    current = current->left;

return current;
}

struct node* deleteNode(struct node* root, int key)
{
    // STEP 1: PERFORM STANDARD BST DELETE

    if (root == NULL)
        return root;

    // If the key to be deleted is smaller than the root's key,
    // then it lies in left subtree
    if (key < root->key)
        root->left = deleteNode(root->left, key);

    // If the key to be deleted is greater than the root's key,
    // then it lies in right subtree
    else if (key > root->key)
        root->right = deleteNode(root->right, key);

    // if key is same as root's key, then This is the node
    // to be deleted
    else
    {
        // If key is present more than once, simply decrement
        // count and return
        if (root->count > 1)
        {
            (root->count)--;
            return;
        }
        // ELSE, delete the node

        // node with only one child or no child
        if( (root->left == NULL) || (root->right == NULL) )
        {
            struct node *temp = root->left ? root->left : root->right;

            // No child case
            if(temp == NULL)
            {
                temp = root;
                root = NULL;
            }
        }
    }
}
```

```

        else // One child case
            *root = *temp; // Copy the contents of the non-empty child

            free(temp);
        }
        else
        {
            // node with two children: Get the inorder successor (smallest
            // in the right subtree)
            struct node* temp = minValueNode(root->right);

            // Copy the inorder successor's data to this node
            root->key = temp->key;

            // Delete the inorder successor
            root->right = deleteNode(root->right, temp->key);
        }
    }

    // If the tree had only one node then return
    if (root == NULL)
        return root;

    // STEP 2: UPDATE HEIGHT OF THE CURRENT NODE
    root->height = max(height(root->left), height(root->right)) + 1;

    // STEP 3: GET THE BALANCE FACTOR OF THIS NODE (to check whether
    // this node became unbalanced)
    int balance = getBalance(root);

    // If this node becomes unbalanced, then there are 4 cases

    // Left Left Case
    if (balance > 1 && getBalance(root->left) >= 0)
        return rightRotate(root);

    // Left Right Case
    if (balance > 1 && getBalance(root->left) < 0)
    {
        root->left = leftRotate(root->left);
        return rightRotate(root);
    }

    // Right Right Case
    if (balance < -1 && getBalance(root->right) <= 0)
        return leftRotate(root);

    // Right Left Case

```

```

if (balance < -1 && getBalance(root->right) > 0)
{
    root->right = rightRotate(root->right);
    return leftRotate(root);
}

return root;
}

// A utility function to print preorder traversal of the tree.
// The function also prints height of every node
void preOrder(struct node *root)
{
    if(root != NULL)
    {
        printf("%d(%d) ", root->key, root->count);
        preOrder(root->left);
        preOrder(root->right);
    }
}

/* Drier program to test above function*/
int main()
{
    struct node *root = NULL;

    /* Constructing tree given in the above figure */
    root = insert(root, 9);
    root = insert(root, 5);
    root = insert(root, 10);
    root = insert(root, 5);
    root = insert(root, 9);
    root = insert(root, 7);
    root = insert(root, 17);

    printf("Pre order traversal of the constructed AVL tree is \n");
    preOrder(root);

    root = deleteNode(root, 9);

    printf("\nPre order traversal after deletion of 10 \n");
    preOrder(root);

    return 0;
}

```

Output:

```
Pre order traversal of the constructed AVL tree is  
9(2) 5(2) 7(1) 10(1) 17(1)  
Pre order traversal after deletion of 10  
9(1) 5(2) 7(1) 10(1) 17(1)
```

Thanks to **Rounaq Jhunjhunu Wala** for sharing initial code. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## Source

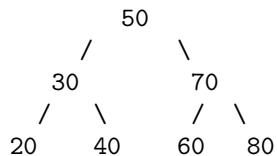
<https://www.geeksforgeeks.org/avl-with-duplicate-keys/>

# Chapter 6

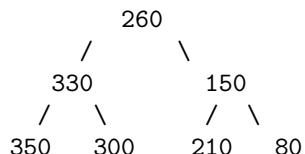
## Add all greater values to every node in a given BST

Add all greater values to every node in a given BST - GeeksforGeeks

Given a **Binary Search Tree (BST)**, modify it so that all greater values in the given BST are added to every node. For example, consider the following BST.



The above tree should be modified to following



A **simple method** for solving this is to find sum of all greater values for every node. This method would take  $O(n^2)$  time.

We can do it **using a single traversal**. The idea is to use following BST property. If we do reverse Inorder traversal of BST, we get all nodes in decreasing order. We do reverse Inorder traversal and keep track of the sum of all nodes visited so far, we add this sum to every node.

C

```
// C program to add all greater values in every node of BST
```

```
#include<stdio.h>
#include<stdlib.h>

struct Node
{
    int data;
    struct Node *left, *right;
};

// A utility function to create a new BST node
struct Node *newNode(int item)
{
    struct Node *temp = (struct Node *)malloc(sizeof(struct Node));
    temp->data = item;
    temp->left = temp->right = NULL;
    return temp;
}

// Recursive function to add all greater values in every node
void modifyBSTUtil(struct Node *root, int *sum)
{
    // Base Case
    if (root == NULL) return;

    // Recur for right subtree
    modifyBSTUtil(root->right, sum);

    // Now *sum has sum of nodes in right subtree, add
    // root->data to sum and update root->data
    *sum = *sum + root->data;
    root->data = *sum;

    // Recur for left subtree
    modifyBSTUtil(root->left, sum);
}

// A wrapper over modifyBSTUtil()
void modifyBST(struct Node *root)
{
    int sum = 0;
    modifyBSTUtil(root, &sum);
}

// A utility function to do inorder traversal of BST
void inorder(struct Node *root)
{
    if (root != NULL)
    {
```

```
    inorder(root->left);
    printf("%d ", root->data);
    inorder(root->right);
}
}

/* A utility function to insert a new node with given data in BST */
struct Node* insert(struct Node* node, int data)
{
    /* If the tree is empty, return a new node */
    if (node == NULL) return newNode(data);

    /* Otherwise, recur down the tree */
    if (data <= node->data)
        node->left = insert(node->left, data);
    else
        node->right = insert(node->right, data);

    /* return the (unchanged) node pointer */
    return node;
}

// Driver Program to test above functions
int main()
{
    /* Let us create following BST
       50
      /   \
     30   70
    / \   / \
   20 40 60 80 */
    struct Node *root = NULL;
    root = insert(root, 50);
    insert(root, 30);
    insert(root, 20);
    insert(root, 40);
    insert(root, 70);
    insert(root, 60);
    insert(root, 80);

    modifyBST(root);

    // print inoder tarversal of the modified BST
    inorder(root);

    return 0;
}
```

**Java**

```
// Java code to add all greater values to
// every node in a given BST

// A binary tree node
class Node {

    int data;
    Node left, right;

    Node(int d)
    {
        data = d;
        left = right = null;
    }
}

class BinarySearchTree {

    // Root of BST
    Node root;

    // Constructor
    BinarySearchTree()
    {
        root = null;
    }

    // Inorder traversal of the tree
    void inorder()
    {
        inorderUtil(this.root);
    }

    // Utility function for inorder traversal of
    // the tree
    void inorderUtil(Node node)
    {
        if (node == null)
            return;

        inorderUtil(node.left);
        System.out.print(node.data + " ");
        inorderUtil(node.right);
    }

    // adding new node
```

```
public void insert(int data)
{
    this.root = this.insertRec(this.root, data);
}

/* A utility function to insert a new node with
given data in BST */
Node insertRec(Node node, int data)
{
    /* If the tree is empty, return a new node */
    if (node == null) {
        this.root = new Node(data);
        return this.root;
    }

    /* Otherwise, recur down the tree */
    if (data <= node.data) {
        node.left = this.insertRec(node.left, data);
    } else {
        node.right = this.insertRec(node.right, data);
    }
    return node;
}

// This class initialises the value of sum to 0
public class Sum {
    int sum = 0;
}

// Recursive function to add all greater values in
// every node
void modifyBSTUtil(Node node, Sum S)
{
    // Base Case
    if (node == null)
        return;

    // Recur for right subtree
    this.modifyBSTUtil(node.right, S);

    // Now *sum has sum of nodes in right subtree, add
    // root->data to sum and update root->data
    S.sum = S.sum + node.data;
    node.data = S.sum;

    // Recur for left subtree
    this.modifyBSTUtil(node.left, S);
}
```

```
// A wrapper over modifyBSTUtil()
void modifyBST(Node node)
{
    Sum S = new Sum();
    this.modifyBSTUtil(node, S);
}

// Driver Function
public static void main(String[] args)
{
    BinarySearchTree tree = new BinarySearchTree();

    /* Let us create following BST
       50
      /   \
     30   70
    / \ / \
   20 40 60 80 */

    tree.insert(50);
    tree.insert(30);
    tree.insert(20);
    tree.insert(40);
    tree.insert(70);
    tree.insert(60);
    tree.insert(80);

    tree.modifyBST(tree.root);

    // print inorder traversal of the modified BST
    tree.inorder();
}
}

// This code is contributed by Kamal Rawal
```

Output

```
350 330 300 260 210 150 80
```

Time Complexity: O(n) where n is number of nodes in the given BST.

As a side note, we can also use reverse Inorder traversal to find kth largest element in a BST.

This article is contributed by [Chandra Prakash](#). Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

**Source**

<https://www.geeksforgeeks.org/add-greater-values-every-node-given-bst/>

## Chapter 7

# Advantages of Trie Data Structure

Advantages of Trie Data Structure - GeeksforGeeks

[Tries](#) is a tree that stores strings. Maximum number of children of a node is equal to size of alphabet. Trie supports search, insert and delete operations in  $O(L)$  time where  $L$  is length of key.

[Hashing](#):- In hashing, we convert key to a small value and the value is used to index data. Hashing supports search, insert and delete operations in  $O(L)$  time on average.

[Self Balancing BST](#) : The time complexity of search, insert and delete operations in a self-balancing [Binary Search Tree \(BST\)](#) (like [Red-Black Tree](#), [AVL Tree](#), [Splay Tree](#), etc) is  $O(L \log n)$  where  $n$  is total number words and  $L$  is length of word. The advantage of Self balancing BSTs is that they maintain order which makes operations like minimum, maximum, closest (floor or ceiling) and k-th largest faster. Please refer [Advantages of BST over Hash Table](#) for details.

**Why Trie? :-**

1. With Trie, we can insert and find strings in  $O(L)$  time where  $L$  represent the length of a single word. This is obviously faster than BST. This is also faster than Hashing because of the ways it is implemented. We do not need to compute any hash function. No collision handling is required (like we do in [open addressing](#) and [separate chaining](#))
2. Another advantage of Trie is, we can [easily print all words in alphabetical order](#) which is not easily possible with hashing.
3. We can efficiently do [prefix search \(or auto-complete\)](#) with Trie.

**Issues with Trie :-**

The main disadvantage of tries is that they need lot of memory for storing the strings. For each node we have too many node pointers(equal to number of characters of the alphabet), If space is concern, then [Ternary Search Tree](#) can be preferred for dictionary implementations. In Ternary Search Tree, time complexity of search operation is  $O(h)$  where  $h$  is

height of the tree. Ternary Search Trees also supports other operations supported by Trie like prefix search, alphabetical order printing and nearest neighbor search.

The final conclusion is regarding *tries data structure* is that they are faster but require *huge memory* for storing the strings.

**Improved By :** [Aashutosh Rathi](#)

## Source

<https://www.geeksforgeeks.org/advantages-trie-data-structure/>

## Chapter 8

# Applications of Minimum Spanning Tree Problem

Applications of Minimum Spanning Tree Problem - GeeksforGeeks

Minimum Spanning Tree (MST) problem: Given connected graph G with positive edge weights, find a min weight set of edges that connects all of the vertices.

MST is fundamental problem with diverse applications.

### Network design.

- *telephone, electrical, hydraulic, TV cable, computer, road*

The standard application is to a problem like phone network design. You have a business with several offices; you want to lease phone lines to connect them up with each other; and the phone company charges different amounts of money to connect different pairs of cities. You want a set of lines that connects all your offices with a minimum total cost. It should be a spanning tree, since if a network isn't a tree you can always remove some edges and save money.

### Approximation algorithms for NP-hard problems.

- *traveling salesperson problem, Steiner tree*

A less obvious application is that the minimum spanning tree can be used to approximately solve the traveling salesman problem. A convenient formal way of defining this problem is to find the shortest path that visits each point at least once.

Note that if you have a path visiting all points exactly once, it's a special kind of tree. For instance in the example above, twelve of sixteen spanning trees are actually paths. If you have a path visiting some vertices more than once, you can always drop some edges to get a tree. So in general the MST weight is less than the TSP weight, because it's a minimization over a strictly larger set.

On the other hand, if you draw a path tracing around the minimum spanning tree, you trace each edge twice and visit all points, so the TSP weight is less than twice the MST weight. Therefore this tour is within a factor of two of optimal.

**Indirect applications.**

- max bottleneck paths
- LDPC codes for error correction
- image registration with Renyi entropy
- learning salient features for real-time face verification
- reducing data storage in sequencing amino acids in a protein
- model locality of particle interactions in turbulent fluid flows
- autoconfig protocol for Ethernet bridging to avoid cycles in a network

**Cluster analysis**

k clustering problem can be viewed as finding an MST and deleting the k-1 most expensive edges.

**Sources:**

- <http://www.cs.princeton.edu/courses/archive/spr07/cos226/lectures/mst.pdf>
- <http://www.ics.uci.edu/~eppstein/161/960206.html>

**Source**

<https://www.geeksforgeeks.org/applications-of-minimum-spanning-tree/>

# Chapter 9

## Applications of tree data structure

Applications of tree data structure - GeeksforGeeks

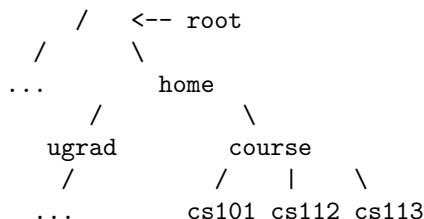
**Difficulty Level:** Rookie

### Why Tree?

Unlike Array and Linked List, which are linear data structures, tree is hierarchical (or non-linear) data structure.

1. One reason to use trees might be because you want to store information that naturally forms a hierarchy. For example, the file system on a computer:

file system



2. If we organize keys in form of a tree (with some ordering e.g., BST), we can search for a given key in moderate time (quicker than Linked List and slower than arrays). **Self-balancing search trees** like [AVL](#) and [Red-Black trees](#) guarantee an upper bound of  $O(\log n)$  for search.
3. We can insert/delete keys in moderate time (quicker than Arrays and slower than Unordered Linked Lists). **Self-balancing search trees** like [AVL](#) and [Red-Black trees](#) guarantee an upper bound of  $O(\log n)$  for insertion/deletion.

4. Like Linked Lists and unlike Arrays, Pointer implementation of trees don't have an upper limit on number of nodes as nodes are linked using pointers.

### **Other Applications :**

1. [Heap](#) is a tree data structure which is implemented using arrays and used to implement priority queues.
2. [B-Tree](#) and [B+ Tree](#) : They are used to implement indexing in databases.
3. [Syntax Tree](#): Used in Compilers.
4. [K-D Tree](#): A space partitioning tree used to organize points in K dimensional space.
5. [Trie](#) : Used to implement dictionaries with prefix lookup.
6. [Suffix Tree](#) : For quick pattern searching in a fixed text.

As per [Wikipedia](#), following are the common uses of tree.

1. Manipulate hierarchical data.
2. Make information easy to search (see tree traversal).
3. Manipulate sorted lists of data.
4. As a workflow for compositing digital images for visual effects.
5. Router algorithms

References:

<http://www.cs.bu.edu/teaching/c/tree/binary/>

[http://en.wikipedia.org/wiki/Tree\\_%28data\\_structure%29#Common\\_uses](http://en.wikipedia.org/wiki/Tree_%28data_structure%29#Common_uses)

### **Source**

<https://www.geeksforgeeks.org/applications-of-tree-data-structure/>

# Chapter 10

## Averages of Levels in Binary Tree

Averages of Levels in Binary Tree - GeeksforGeeks

Given a non-empty binary tree, print the average value of the nodes on each level.

Examples:

Input :

```
    4
   / \
  2   9
 / \   \
3   5   7
```

Output : [4 5.5 5]

The average value of nodes on level 0 is 4,  
on level 1 is 5.5, and on level 2 is 5.  
Hence, print [4 5.5 5].

The idea is based on [Level order traversal line by line | Set 2 \(Using Two Queues\)](#)

1. Start by pushing the root node into the queue. Then, remove a node from the front of the queue.
2. For every node removed from the queue, push all its children into a new temporary queue.
3. Keep on popping nodes from the queue and adding these node's children to the temporary queue till queue becomes empty.
4. Every time queue becomes empty, it indicates that one level of the tree has been considered.

5. While pushing the nodes into temporary queue, keep a track of the sum of the nodes along with the number of nodes pushed and find out the average of the nodes on each level by making use of these sum and count values.
6. After each level has been considered, again initialize the queue with temporary queue and continue the process till both queues become empty.

```

// C++ program to find averages of all levels
// in a binary tree.
#include <bits/stdc++.h>
using namespace std;

/* A binary tree node has data, pointer to
   left child and a pointer to right child */
struct Node {
    int val;
    struct Node* left, *right;
};

/* Function to print the average value of the
   nodes on each level */
void averageOfLevels(Node* root)
{
    vector<float> res;

    // Traversing level by level
    queue<Node*> q;
    q.push(root);

    while (!q.empty()) {

        // Compute sum of nodes and
        // count of nodes in current
        // level.
        int sum = 0, count = 0;
        queue<Node*> temp;
        while (!q.empty()) {
            Node* n = q.front();
            q.pop();
            sum += n->val;
            count++;
            if (n->left != NULL)
                temp.push(n->left);
            if (n->right != NULL)
                temp.push(n->right);
        }
        q = temp;
        cout << (sum * 1.0 / count) << " ";
    }
}

```

```

}

/* Helper function that allocates a
   new node with the given data and
   NULL left and right pointers. */
Node* newNode(int data)
{
    Node* temp = new Node;
    temp->val = data;
    temp->left = temp->right = NULL;
    return temp;
}

// Driver code
int main()
{
    /* Let us construct a Binary Tree
       4
      / \
     2   9
    / \   \
   3   5   7 */

    Node* root = NULL;
    root = newNode(4);
    root->left = newNode(2);
    root->right = newNode(9);
    root->left->left = newNode(3);
    root->left->right = newNode(8);
    root->right->right = newNode(7);
    averageOfLevels(root);
    return 0;
}

```

Output:

```

Average of levels:
[4 5.5 5]

```

#### Complexity Analysis:

- Time complexity :  $O(n)$ .  
The whole tree is traversed atmost once. Here, n refers to the number of nodes in the given binary tree.
- Auxiliary Space :  $O(n)$ .  
The size of queues can grow upto atmost the maximum number of nodes at any level in the given binary tree. Here, n refers to the maximum number of nodes at any level in the input tree.

**Source**

<https://www.geeksforgeeks.org/averages-levels-binary-tree/>

# Chapter 11

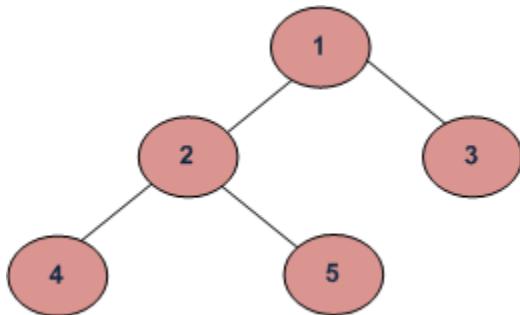
## BFS vs DFS for Binary Tree

BFS vs DFS for Binary Tree - GeeksforGeeks

**What are BFS and DFS for Binary Tree?**

A Tree is typically traversed in two ways:

- Breadth First Traversal (Or Level Order Traversal)
- Depth First Traversals
  - Inorder Traversal (Left-Root-Right)
  - Preorder Traversal (Root-Left-Right)
  - Postorder Traversal (Left-Right-Root)



BFS and DFSs of above Tree

Breadth First Traversal : 1 2 3 4 5

Depth First Traversals:

Preorder Traversal : 1 2 4 5 3

Inorder Traversal : 4 2 5 1 3

Postorder Traversal : 4 5 2 3 1

### Why do we care?

There are many tree questions that can be solved using any of the above four traversals. Examples of such questions are [size](#), [maximum](#), [minimum](#), [print left view](#), etc.

### Is there any difference in terms of Time Complexity?

All four traversals require  $O(n)$  time as they visit every node exactly once.

### Is there any difference in terms of Extra Space?

There is difference in terms of extra space required.

1. Extra Space required for Level Order Traversal is  $O(w)$  where  $w$  is maximum width of Binary Tree. In level order traversal, queue one by one stores nodes of different level.
2. Extra Space required for Depth First Traversals is  $O(h)$  where  $h$  is maximum height of Binary Tree. In Depth First Traversals, stack (or function call stack) stores all ancestors of a node.

Maximum Width of a Binary Tree at depth (or height)  $h$  can be  $2^h$  where  $h$  starts from 0. So the maximum number of nodes can be at the last level. And worst case occurs when Binary Tree is a perfect Binary Tree with numbers of nodes like 1, 3, 7, 15, ...etc. In worst case, value of  $2^h$  is **Ceil( $n/2$ )**.

Height for a Balanced Binary Tree is  $O(\log n)$ . Worst case occurs for skewed tree and worst case height becomes  $O(n)$ .

So in worst case extra space required is  $O(n)$  for both. But worst cases occur for different types of trees.

*It is evident from above points that extra space required for Level order traversal is likely to be more when tree is more balanced and extra space for Depth First Traversal is likely to be more when tree is less balanced.*

### How to Pick One?

1. Extra Space can be one factor (Explained above)
2. Depth First Traversals are typically recursive and recursive code requires function call overheads.
3. The most important points is, BFS starts visiting nodes from root while DFS starts visiting nodes from leaves. So if our problem is to search something that is more likely to closer to root, we would prefer BFS. And if the target node is close to a leaf, we would prefer DFS.

### Exercise:

Which traversal should be used to print leaves of Binary Tree and why?

Which traversal should be used to print nodes at  $k$ 'th level where  $k$  is much less than total number of levels?

This article is contributed by **Dheeraj Gupta**. This Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

### Source

<https://www.geeksforgeeks.org/bfs-vs-dfs-binary-tree/>

## Chapter 12

# BK-Tree | Introduction & Implementation

BK-Tree | Introduction & Implementation - GeeksforGeeks

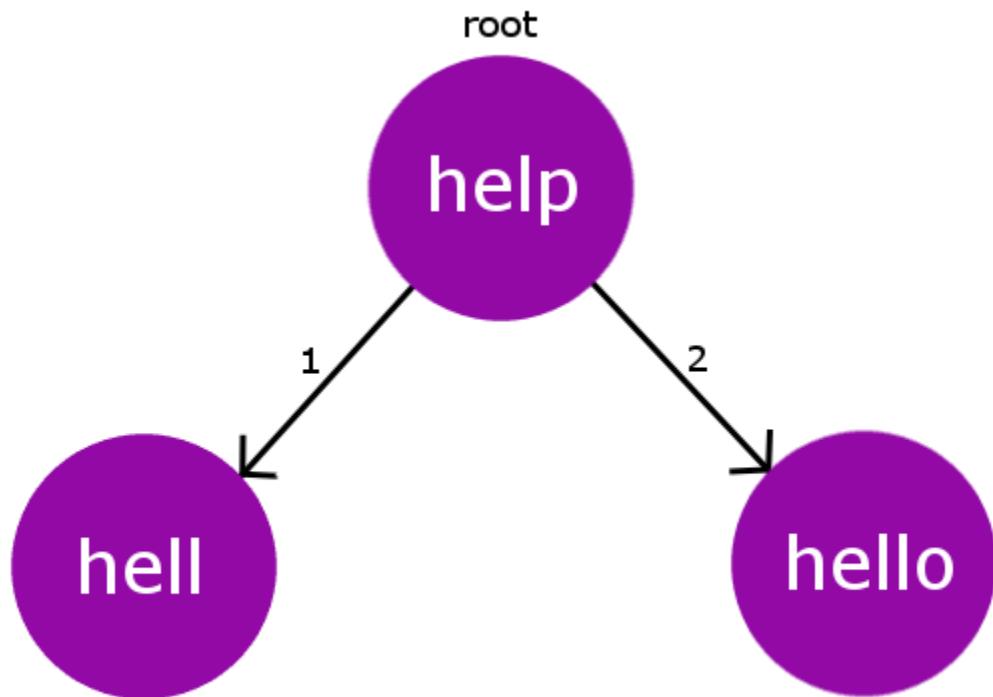
BK Tree or Burkhard Keller Tree is a data structure that is used to perform spell check based on Edit Distance (Levenshtein distance) concept. BK trees are also used for approximate string matching. Various auto correct feature in many softwares can be implemented based on this data structure.

Pre-requisites : Edit distance Problem  
Metric tree

Let's say we have a dictionary of words and then we have some other words which are to be checked in the dictionary for spelling errors. We need to have collection of all words in the dictionary which are very close to the given word. For instance if we are checking a word “ruk” we will have {“truck”, “buck”, “duck”, .....}. Therefore, spelling mistake can be corrected by deleting a character from the word or adding a new character in the word or by replacing the character in the word by some appropriate one. Therefore, we will be using the edit distance as a measure for correctness and matching of the misspelled word from the words in our dictionary.

Now, let's see the structure of our BK Tree. Like all other trees, BK Tree consists of nodes and edges. The nodes in the BK Tree will represent the individual words in our dictionary and there will be exactly the same number of nodes as the number of words in our dictionary. The edge will contain some integer weight that will tell us about the edit-distance from one node to another. Lets say we have an edge from node **u** to node **v** having some edge-weight **w**, then **w** is the edit-distance required to turn the string **u** to **v**.

Consider our dictionary with words : { “help” , “hell” , “hello”}. Therefore, for this dictionary our BK Tree will look like the below one.

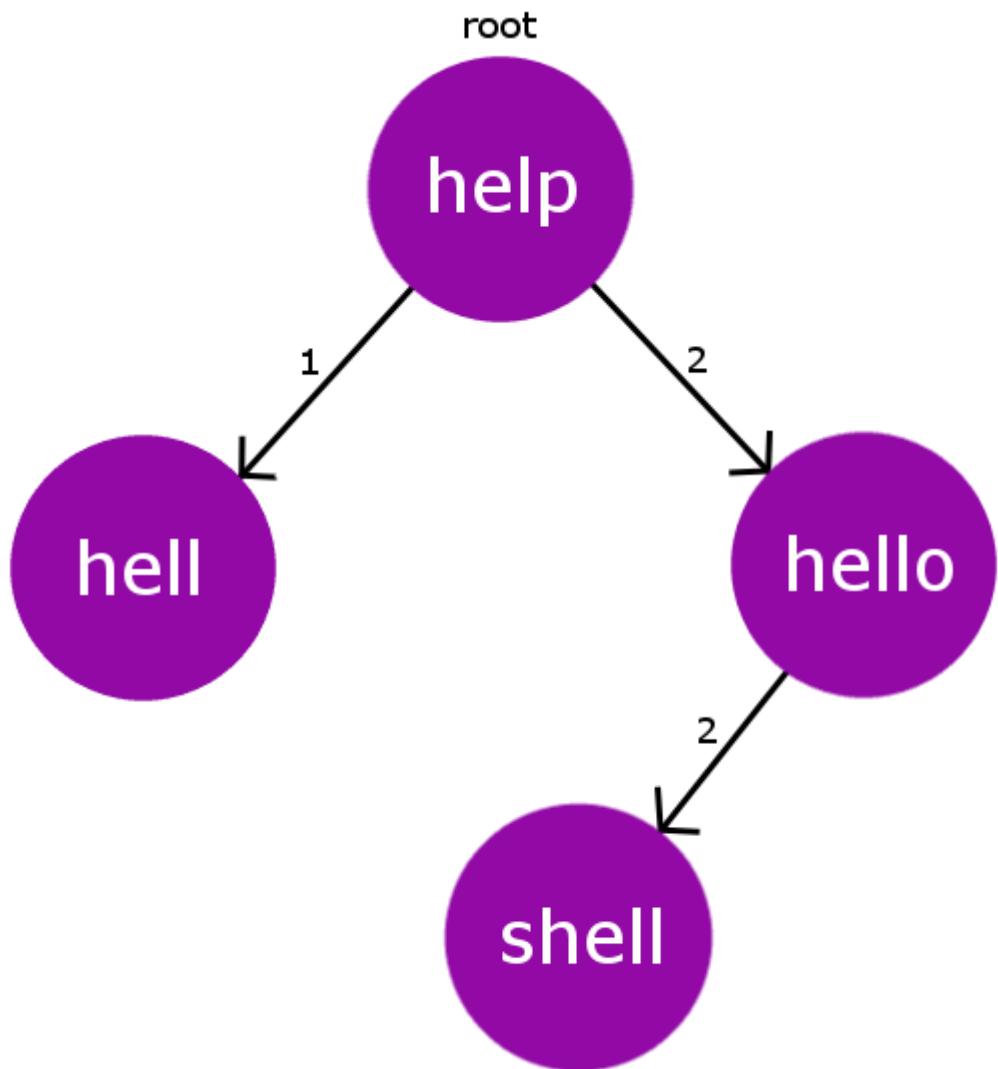


Every node in the BK Tree will have exactly one child with same edit-distance. In case, if we encounter some collision for edit-distance while inserting, we will then propagate the insertion process down the children until we find an appropriate parent for the string node.

Every insertion in the BK Tree will start from our root node. Root node can be any word from our dictionary.

For example, let's add another word "shell" to the above dictionary. Now our **Dict[] = {"help" , "hell" , "hello" , "shell"}.** It is now evident that "shell" has same edit-distance as "hello" has from the root node "help" i.e 2. Hence, we encounter a collision. Therefore, we deal this collision by recursively doing this insertion process on the pre-existing colliding node.

So, now instead of inserting "shell" at the root node "help", we will now insert it to the colliding node "hello". Therefore, now the new node "shell" is added to the tree and it has node "hello" as its parent with the edge-weight of 2(edit-distance). Below pictorial representation describes the BK Tree after this insertion.

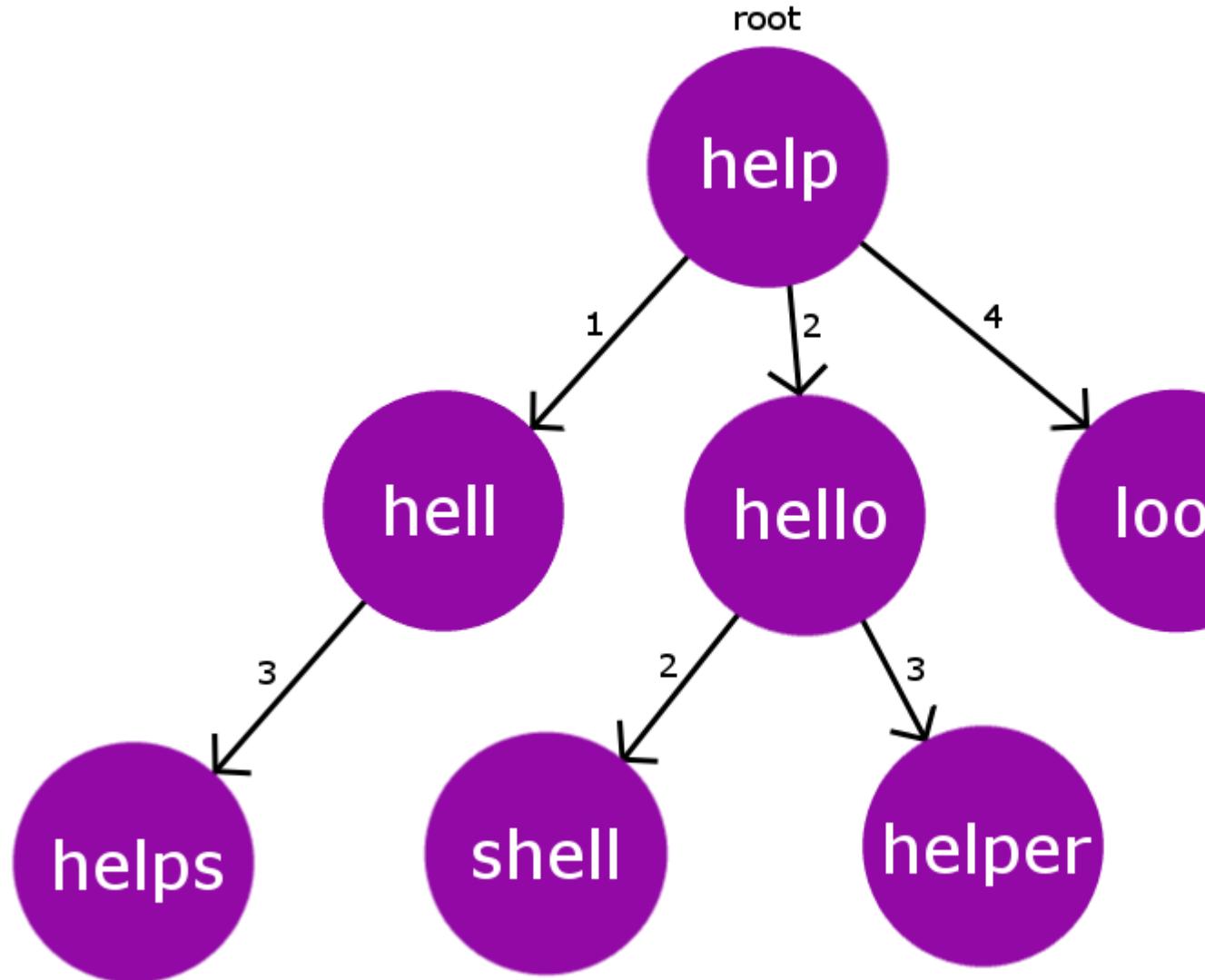


So, till now we have understood how we will build our BK Tree. Now, the question arises that how to find the closest correct word for our misspelled word? First of all, we need to set a tolerance value. This **tolerance value** is simply the maximum edit distance from our misspelled word to the correct words in our dictionary. So, to find the eligible correct words within the tolerance limit, Naive approach will be to iterate over all the words in the dictionary and collect the words which are within the tolerance limit. But this approach has  $O(n*m*n)$  time complexity( $n$  is the number of words in  $\text{dict}[]$ ,  $m$  is average size of correct word and  $n$  is length of misspelled word) which times out for larger size of dictionary.

Therefore, now the BK Tree comes into action. As we know that each node in BK Tree is constructed on basis of edit-distance measure from its parent. Therefore, we will directly

be going from root node to specific nodes that lie within the tolerance limit. Lets, say our tolerance limit is **TOL** and the edit-distance of the current node from the misspelled word is **dist**. Therefore, now instead of iterating over all its children we will only iterate over its children that have edit distance in range  $[dist-TOL, dist+TOL]$ . This will reduce our complexity by a large extent. We will discuss this in our time complexity analysis.

Consider the below constructed BK Tree.



Let's say we have a misspelled word "**oop**" and the tolerance limit is 2. Now, we will see how we will collect the expected correct for the given misspelled word.

Iteration 1: We will start checking the edit distance from the root node.  $D("oop" \rightarrow "help")$

= 3. Now we will iterate over its children having edit distance in range [ D-TOL , D+TOL ] i.e [1,5]

Iteration 2: Let's start iterating from the highest possible edit distance child i.e node "loop" with edit distance 4. Now once again we will find its edit distance from our misspelled word.  $D("oop", "loop") = 1$ .

here  $D = 1$  i.e  $D \leq TOL$ , so we will add "loop" to the expected correct word list and process its child nodes having edit distance in range [D-TOL,D+TOL] i.e [1,3]

Iteration 3: Now, we are at node "troop". Once again we will check its edit distance from misspelled word .  $D("oop", "troop")=2$ . Here again  $D \leq TOL$ , hence again we will add "troop" to the expected correct word list.

We will proceed the same for all the words in the range [D-TOL,D+TOL] starting from the root node till the bottom most leaf node. This, is similar to a DFS traversal on a tree, with selectively visiting the child nodes whose edge weight lie in some given range.

Therefore, at the end we will be left with only 2 expected words for the misspelled word "oop" i.e {"loop", "troop"}

```
// C++ program to demonstrate working of BK-Tree
#include "bits/stdc++.h"
using namespace std;

// maximum number of words in dict[]
#define MAXN 100

// defines the tolerance value
#define TOL 2

// defines maximum length of a word
#define LEN 10

struct Node
{
    // stores the word of the current Node
    string word;

    // links to other Node in the tree
    int next[2*LEN];

    // constructors
    Node(string x):word(x)
    {
        // initializing next[i] = 0
        for(int i=0; i<2*LEN; i++)
            next[i] = 0;
    }
    Node() {}
}
```

```

};

// stores the root Node
Node RT;

// stores every Node of the tree
Node tree[MAXN];

// index for current Node of tree
int ptr;

int min(int a, int b, int c)
{
    return min(a, min(b, c));
}

// Edit Distance
// Dynamic-Approach O(m*n)
int editDistance(string& a, string& b)
{
    int m = a.length(), n = b.length();
    int dp[m+1][n+1];

    // filling base cases
    for (int i=0; i<=m; i++)
        dp[i][0] = i;
    for (int j=0; j<=n; j++)
        dp[0][j] = j;

    // populating matrix using dp-approach
    for (int i=1; i<=m; i++)
    {
        for (int j=1; j<=n; j++)
        {
            if (a[i-1] != b[j-1])
            {
                dp[i][j] = min( 1 + dp[i-1][j],    // deletion
                                1 + dp[i][j-1],   // insertion
                                1 + dp[i-1][j-1] // replacement
                            );
            }
            else
                dp[i][j] = dp[i-1][j-1];
        }
    }
    return dp[m][n];
}

```

```

// adds curr Node to the tree
void add(Node& root,Node& curr)
{
    if (root.word == "") 
    {
        // if it is the first Node
        // then make it the root Node
        root = curr;
        return;
    }

    // get its editDist from the Root Node
    int dist = editDistance(curr.word,root.word);

    if (tree[root.next[dist]].word == "") 
    {
        /* if no Node exists at this dist from root
         * make it child of root Node*/
        
        // incrementing the pointer for curr Node
        ptr++;

        // adding curr Node to the tree
        tree[ptr] = curr;

        // curr as child of root Node
        root.next[dist] = ptr;
    }
    else
    {
        // recursively find the parent for curr Node
        add(tree[root.next[dist]],curr);
    }
}

vector <string> getSimilarWords(Node& root,string& s)
{
    vector < string > ret;
    if (root.word == "") 
        return ret;

    // calculating editdistance of s from root
    int dist = editDistance(root.word,s);

    // if dist is less than tolerance value
    // add it to similar words
    if (dist <= TOL) ret.push_back(root.word);
}

```

```

// iterate over the string havinng tolerane
// in range (dist-TOL , dist+TOL)
int start = dist - TOL;
if (start < 0)
    start = 1;

while (start < dist + TOL)
{
    vector <string> tmp =
        getSimilarWords(tree[root.next[start]],s);
    for (auto i : tmp)
        ret.push_back(i);
    start++;
}
return ret;
}

// driver program to run above functions
int main(int argc, char const *argv[])
{
    // dictionary words
    string dictionary[] = {"hell","help","shel","smell",
                           "fell","felt","oops","pop","oouch","halt"
                           };
    ptr = 0;
    int sz = sizeof(dictionary)/sizeof(string);

    // adding dict[] words on to tree
    for(int i=0; i<sz; i++)
    {
        Node tmp = Node(dictionary[i]);
        add(RT,tmp);
    }

    string w1 = "ops";
    string w2 = "helpt";
    vector < string > match = getSimilarWords(RT,w1);
    cout << "similar words in dictionary for : " << w1 << ":\n";
    for (auto x : match)
        cout << x << endl;

    match = getSimilarWords(RT,w2);
    cout << "Correct words in dictionary for " << w2 << ":\n";
    for (auto x : match)
        cout << x << endl;

    return 0;
}

```

Output:

```
Correct words in dictionary for ops:  
oops  
pop  
Correct words in dictionary for helt:  
hell  
help  
fell  
shel  
felt  
halt
```

**Time Complexity :** It is quite evident that the time complexity majorly depends on the tolerance limit. We will be considering **tolerance limit** to be **2**. Now, roughly estimating, the depth of BK Tree will be  $\log n$ , where  $n$  is the size of dictionary. At every level we are visiting 2 nodes in the tree and performing edit distance calculation. Therefore, our Time Complexity will be  **$O(L1*L2*\log n)$** , here **L1** is the average length of word in our dictionary and **L2** is the length of misspelled. Generally L1 and L2 will be small.

### References

- <https://en.wikipedia.org/wiki/BK-tree>
- <https://issues.apache.org/jira/browse/LUCENE-2230>

### Source

<https://www.geeksforgeeks.org/bk-tree-introduction-implementation/>

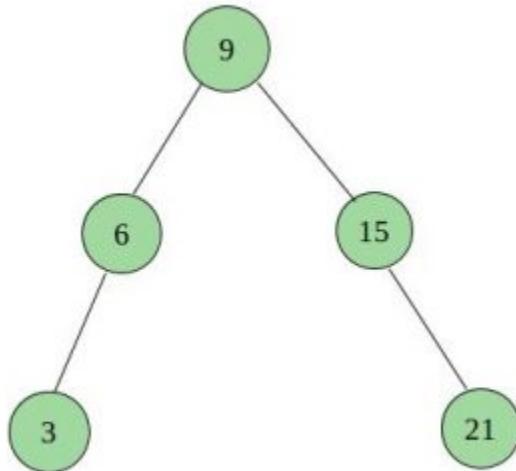
## Chapter 13

### BST to a Tree with sum of all smaller keys

BST to a Tree with sum of all smaller keys - GeeksforGeeks

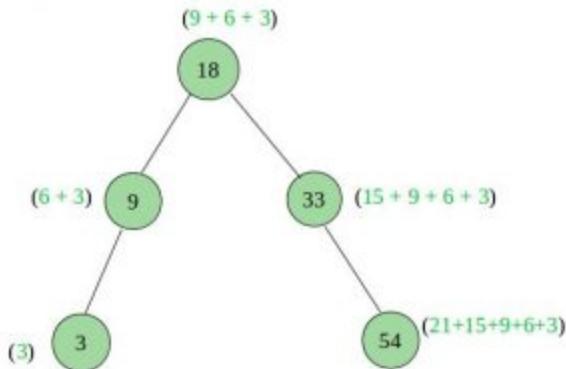
Given a Binary Search Tree(BST), convert it to a Binary Tree such that every key of the original BST is changed to key plus sum of all smaller keys in BST.

Given a BST with N Nodes we have to convert into Binary Tree



Given above BST with **N=5** Nodes. The values at Node being **9, 6, 15, 3, 21**

Binary Tree after conversion



Binary Tree after conversion, the values at Node being **18, 9, 33, 3, 54**

**Solution:** We will perform a regular Inorder traversal in which we keep track of sum of Nodes visited. Let this sum be  $sum$ . The Node which is being visited, add that key of Node to  $sum$  i.e.  $sum = sum + Node->key$ . Change the key of current Node to  $sum$  i.e.  $Node->key = sum$ .

When a BST is being traversed in inorder, for every key currently being visited, all keys that are already visited are all smaller keys.

C++

```

// Program to change a BST to Binary Tree such
// that key of a Node becomes original key plus
// sum of all smaller keys in BST
#include <stdio.h>
#include <stdlib.h>

/* A BST Node has key, left child and
   right child */
struct Node {
    int key;
    struct Node* left;
    struct Node* right;
};

/* Helper function that allocates a new
   node with the given key and NULL left
   and right pointers.*/
struct Node* newNode(int key)
{
    struct Node* node = new Node;
    node->key = key;
    node->left = NULL;
    node->right = NULL;
    return (node);
}
    
```

```
}

// A recursive function that traverses the
// given BST in inorder and for every key,
// adds all smaller keys to it
void addSmallerUtil(struct Node* root, int* sum)
{
    // Base Case
    if (root == NULL)
        return;

    // Recur for left subtree first so that
    // sum of all smaller Nodes is stored
    addSmallerUtil(root->left, sum);

    // Update the value at sum
    *sum = *sum + root->key;

    // Update key of this Node
    root->key = *sum;

    // Recur for right subtree so that
    // the updated sum is added
    // to greater Nodes
    addSmallerUtil(root->right, sum);
}

// A wrapper over addSmallerUtil(). It
// initializes sum and calls addSmallerUtil()
// to recursively update and use value of
void addSmaller(struct Node* root)
{
    int sum = 0;
    addSmallerUtil(root, &sum);
}

// A utility function to print inorder
// traversal of Binary Tree
void printInorder(struct Node* node)
{
    if (node == NULL)
        return;
    printInorder(node->left);
    printf("%d ", node->key);
    printInorder(node->right);
}

// Driver program to test above function
```

```
int main()
{
    /* Create following BST
       9
      / \
     6   15 */
    Node* root = newNode(9);
    root->left = newNode(6);
    root->right = newNode(15);

    printf(" Original BST\n");
    printInorder(root);

    addSmaller(root);

    printf("\n BST To Binary Tree\n");
    printInorder(root);

    return 0;
}
```

**Java**

```
// Java program to convert BST to binary tree
// such that sum of all smaller keys is added
// to every key

class Node {

    int data;
    Node left, right;

    Node(int d)
    {
        data = d;
        left = right = null;
    }
}

class Sum {

    int addvalue = 0;
}

class BSTtoBinaryTree {

    static Node root;
    Sum add = new Sum();
```

```
// A recursive function that traverses
// the given BST in inorder and for every
// key, adds all smaller keys to it
void addSmallerUtil(Node node, Sum sum)
{
    // Base Case
    if (node == null) {
        return;
    }

    // Recur for left subtree first so that
    // sum of all smaller Nodes is stored at sum
    addSmallerUtil(node.left, sum);

    // Update the value at sum
    sum.addValue = sum.addValue + node.data;

    // Update key of this Node
    node.data = sum.addValue;

    // Recur for right subtree so that the
    // updated sum is added to greater Nodes
    addSmallerUtil(node.right, sum);
}

// A wrapper over addSmallerUtil(). It
// initializes addvalue and calls
// addSmallerUtil() to recursively update
// and use value of addvalue
Node addSmaller(Node node)
{
    addSmallerUtil(node, add);
    return node;
}

// A utility function to print inorder
// traversal of Binary Tree
void printInorder(Node node)
{
    if (node == null) {
        return;
    }
    printInorder(node.left);
    System.out.print(node.data + " ");
    printInorder(node.right);
}
```

```
// Driver program to test the above functions
public static void main(String[] args)
{
    BSTtoBinaryTree tree = new BSTtoBinaryTree();
    tree.root = new Node(9);
    tree.root.left = new Node(6);
    tree.root.right = new Node(15);

    System.out.println("Original BST");
    tree.printInorder(root);
    Node Node = tree.addSmaller(root);
    System.out.println("");
    System.out.println("BST To Binary Tree");
    tree.printInorder(Node);
}
}
```

## Source

<https://www.geeksforgeeks.org/bst-tree-sum-smaller-keys/>

## Chapter 14

# Binary Indexed Tree : Range Updates and Point Queries

Binary Indexed Tree : Range Updates and Point Queries - GeeksforGeeks

Given an array arr[0..n-1]. The following operations need to be performed.

**update(l, r, val)** : Add ‘val’ to all the elements in the array from [l, r].

**getElement(i)** : Find element in the array indexed at ‘i’.

Initially all the elements in the array are 0. Queries can be in any order, i.e., there can be many updates before point query.

**Example:**

```
Input : arr = {0, 0, 0, 0, 0}
Queries: update : l = 0, r = 4, val = 2
         getElement : i = 3
         update : l = 3, r = 4, val = 3
         getElement : i = 3
```

```
Output: Element at 3 is 2
        Element at 3 is 5
```

```
Explanation : Array after first update becomes
              {2, 2, 2, 2, 2}
              Array after second update becomes
              {2, 2, 2, 5, 5}
```

**Method 1** [update : O(n), getElement() : O(1)]

1. **update(l, r, val)** : Iterate over the subarray from l to r and increase all the elements by val.
2. **getElement(i)** : To get the element at i'th index, simply return arr[i].

The time complexity in worst case is  $O(q*n)$  where q is number of queries and n is number of elements.

### Method 2 [update : O(1), getElement() : O(n)]

We can avoid updating all elements and can update only 2 indexes of the array!

1. **update(l, r, val)** : Add 'val' to the l<sup>th</sup> element and subtract 'val' from the (r+1)<sup>th</sup> element, do this for all the update queries.

```
arr[l] = arr[l] + val
arr[r+1] = arr[r+1] - val
```

2. **getElement(i)** : To get i<sup>th</sup> element in the array find the sum of all integers in the array from 0 to i.(Prefix Sum).

Let's analyze the update query. **Why to add val to l<sup>th</sup> index?** Adding val to l<sup>th</sup> index means that all the elements after l are increased by val, since we will be computing the prefix sum for every element. **Why to subtract val from (r+1)<sup>th</sup> index?** A range update was required from [l,r] but what we have updated is [l, n-1] so we need to remove val from all the elements after r i.e., subtract val from (r+1)<sup>th</sup> index. Thus the val is added to range [l,r]. Below is implementation of above approach.

C++

```
// C++ program to demonstrate Range Update // and Point Queries Without using BIT #include <iost
```

**Output:**

```
Element at index 4 is 2
Element at index 3 is 6
```

**Time complexity :**  $O(q*n)$  where q is number of queries.

### Method 3 (Using Binary Indexed Tree)

In method 2, we have seen that the problem can reduced to update and prefix sum queries. We have seen that [BIT can be used to do update and prefix sum queries in  \$O\(\log n\)\$  time.](#)

Below is C++ implementation.

C++

```

// C++ code to demonstrate Range Update and
// Point Queries on a Binary Index Tree
#include <iostream>
using namespace std;

// Updates a node in Binary Index Tree (BITree) at given index
// in BITree. The given value 'val' is added to BITree[i] and
// all of its ancestors in tree.
void updateBIT(int BITree[], int n, int index, int val)
{
    // index in BITree[] is 1 more than the index in arr[]
    index = index + 1;

    // Traverse all ancestors and add 'val'
    while (index <= n)
    {
        // Add 'val' to current node of BI Tree
        BITree[index] += val;

        // Update index to that of parent in update View
        index += index & (-index);
    }
}

// Constructs and returns a Binary Indexed Tree for given
// array of size n.
int *constructBITree(int arr[], int n)
{
    // Create and initialize BITree[] as 0
    int *BITree = new int[n+1];
    for (int i=1; i<=n; i++)
        BITree[i] = 0;

    // Store the actual values in BITree[] using update()
    for (int i=0; i<n; i++)
        updateBIT(BITree, n, i, arr[i]);

    // Uncomment below lines to see contents of BITree[]
    //for (int i=1; i<=n; i++)
    //    cout << BITree[i] << " ";
}

return BITree;
}

// SERVES THE PURPOSE OF getElement()
// Returns sum of arr[0..index]. This function assumes
// that the array is preprocessed and partial sums of
// array elements are stored in BITree[]

```

```

int getSum(int BITree[], int index)
{
    int sum = 0; // Initialize result

    // index in BITree[] is 1 more than the index in arr[]
    index = index + 1;

    // Traverse ancestors of BITree[index]
    while (index>0)
    {
        // Add current element of BITree to sum
        sum += BITree[index];

        // Move index to parent node in getSum View
        index -= index & (-index);
    }
    return sum;
}

// Updates such that getElement() gets an increased
// value when queried from l to r.
void update(int BITree[], int l, int r, int n, int val)
{
    // Increase value at 'l' by 'val'
    updateBIT(BITree, n, l, val);

    // Decrease value at 'r+1' by 'val'
    updateBIT(BITree, n, r+1, -val);
}

// Driver program to test above function
int main()
{
    int arr[] = {0, 0, 0, 0, 0};
    int n = sizeof(arr)/sizeof(arr[0]);
    int *BITree = constructBITree(arr, n);

    // Add 2 to all the element from [2,4]
    int l = 2, r = 4, val = 2;
    update(BITree, l, r, n, val);

    // Find the element at Index 4
    int index = 4;
    cout << "Element at index " << index << " is " <<
        getSum(BITree, index) << "\n";

    // Add 2 to all the element from [0,3]
    l = 0, r = 3, val = 4;
}

```

```
update(BITree, l, r, n, val);

// Find the element at Index 3
index = 3;
cout << "Element at index " << index << " is " <<
    getSum(BITree, index) << "\n" ;

return 0;
}
```

**Java**

```
/* Java code to demonstrate Range Update and
* Point Queries on a Binary Index Tree.
* This method only works when all array
* values are initially 0.*/
class GFG
{

    // Max tree size
    final static int MAX = 1000;

    static int BITree[] = new int[MAX];

    // Updates a node in Binary Index
    // Tree (BITree) at given index
    // in BITree. The given value 'val'
    // is added to BITree[i] and
    // all of its ancestors in tree.
    public static void updateBIT(int n,
                                int index,
                                int val)
    {
        // index in BITree[] is 1
        // more than the index in arr[]
        index = index + 1;

        // Traverse all ancestors
        // and add 'val'
        while (index <= n)
        {
            // Add 'val' to current
            // node of BITree
            BITree[index] += val;

            // Update index to that
            // of parent in update View
            index += index & (-index);
        }
    }
}
```

```
        }
    }

// Constructs Binary Indexed Tree
// for given array of size n.

public static void constructBITree(int arr[],
                                   int n)
{
    // Initialize BITree[] as 0
    for(int i = 1; i <= n; i++)
        BITree[i] = 0;

    // Store the actual values
    // in BITree[] using update()
    for(int i = 0; i < n; i++)
        updateBIT(n, i, arr[i]);

    // Uncomment below lines to
    // see contents of BITree[]
    // for (int i=1; i<=n; i++)
    //     cout << BITree[i] << " ";
}

// SERVES THE PURPOSE OF getElement()
// Returns sum of arr[0..index]. This
// function assumes that the array is
// preprocessed and partial sums of
// array elements are stored in BITree[]
public static int getSum(int index)
{
    int sum = 0; //Initialize result

    // index in BITree[] is 1 more
    // than the index in arr[]
    index = index + 1;

    // Traverse ancestors
    // of BITree[index]
    while (index > 0)
    {

        // Add current element
        // of BITree to sum
        sum += BITree[index];

        // Move index to parent
        // node in getSum View
```

```

        index -= index & (-index);
    }

    // Return the sum
    return sum;
}

// Updates such that getElement()
// gets an increased value when
// queried from l to r.
public static void update(int l, int r,
                           int n, int val)
{
    // Increase value at
    // 'l' by 'val'
    updateBIT(n, l, val);

    // Decrease value at
    // 'r+1' by 'val'
    updateBIT(n, r + 1, -val);
}

// Driver Code
public static void main(String args[])
{
    int arr[] = {0, 0, 0, 0, 0};
    int n = arr.length;

    constructBITree(arr,n);

    // Add 2 to all the
    // element from [2,4]
    int l = 2, r = 4, val = 2;
    update(l, r, n, val);

    int index = 4;

    System.out.println("Element at index "+
                       index + " is "+
                       getSum(index));

    // Add 2 to all the
    // element from [0,3]
    l = 0; r = 3; val = 4;
    update(l, r, n, val);

    // Find the element
}

```

```
// at Index 3
index = 3;
System.out.println("Element at index "+
                     index + " is "+
                     getSum(index));
}
}
// This code is contributed
// by Puneet Kumar.
```

**Output:**

```
Element at index 4 is 2
Element at index 3 is 6
```

**Time Complexity :**  $O(q * \log n) + O(n * \log n)$  where q is number of queries.

Method 1 is efficient when most of the queries are getElement(), method 2 is efficient when most of the queries are updates() and method 3 is preferred when there is mix of both queries.

**Improved By :** [p\\_unit](#)

**Source**

<https://www.geeksforgeeks.org/binary-indexed-tree-range-updates-point-queries/>

# Chapter 15

## Binary Tree (Array implementation)

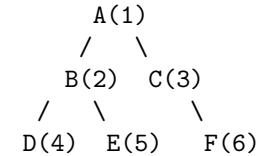
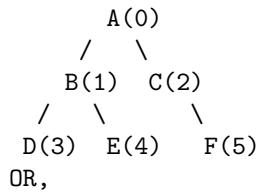
Binary Tree (Array implementation) - GeeksforGeeks

Talking about representation, trees can be represented in two way:

- 1) Dynamic Node Representation ([Linked Representation](#)).
- 2) Array Representation (Sequential Representation).

We are going to talk about sequential representation of the trees.

To represent tree using array, numbering of nodes can start either from 0—(n-1) or 1—n .



For first case(0—n-1),

```
if (say)father=p;  
then left_son=(2*p)+1;  
and right_son=(2*p)+2;
```

For second case(1—n),

```
if (say)father=p;  
then left_son=(2*p);
```

and right\_son=(2\*p)+1;  
where father, left\_son and right\_son are the values of indices of the array.

**Code –**

```
// JAVA implementation of tree using array
// numbering starting from 0 to n-1.
import java.util.*;
import java.lang.*;
import java.io.*;

class Tree {
    public static void main(String[] args)
    {
        Array_imp obj = new Array_imp();
        obj.Root("A");
        //    obj.set_Left("B", 0);
        obj.set_Right("C", 0);
        obj.set_Left("D", 1);
        obj.set_Right("E", 1);
        obj.set_Left("F", 2);
        obj.print_Tree();
    }
}

class Array_imp {
    static int root = 0;
    static String[] str = new String[10];

    /*create root*/
    public void Root(String key)
    {
        str[0] = key;
    }

    /*create left son of root*/
    public void set_Left(String key, int root)
    {
        int t = (root * 2) + 1;

        if(str[root] == null){
            System.out.printf("Can't set child at %d, no parent found\n",t);
        }else{
            str[t] = key;
        }
    }

    /*create right son of root*/
    public void set_Right(String key, int root)
```

```
{  
    int t = (root * 2) + 2;  
  
    if(str[root] == null){  
        System.out.printf("Can't set child at %d, no parent found\n",t);  
    }else{  
        str[t] = key;  
    }  
}  
  
public void print_Tree()  
{  
    for (int i = 0; i < 10; i++) {  
        if (str[i] != null)  
            System.out.print(str[i]);  
        else  
            System.out.print("-");  
  
    }  
}
```

Output:

```
Can't set child at 3, no parent found  
Can't set child at 4, no parent found  
A-C--F----
```

**Note** – Please refer [this](#) if you want to construct tree from the given parent array.

**Improved By :** [Sachin Jain 1](#)

## Source

<https://www.geeksforgeeks.org/binary-tree-array-implementation/>

## Chapter 16

# Binary Tree to Binary Search Tree Conversion

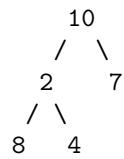
Binary Tree to Binary Search Tree Conversion - GeeksforGeeks

Given a Binary Tree, convert it to a Binary Search Tree. The conversion must be done in such a way that keeps the original structure of Binary Tree.

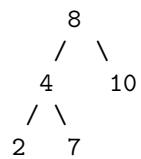
Examples.

Example 1

Input:

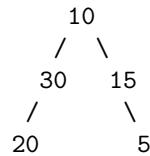


Output:

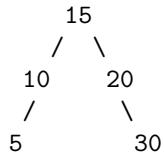


Example 2

Input:



Output:



### Solution

Following is a 3 step solution for converting Binary tree to Binary Search Tree.

- 1) Create a temp array arr[] that stores inorder traversal of the tree. This step takes O(n) time.
- 2) Sort the temp array arr[]. Time complexity of this step depends upon the sorting algorithm. In the following implementation, Quick Sort is used which takes  $(n^2)$  time. This can be done in  $O(n \log n)$  time using Heap Sort or Merge Sort.
- 3) Again do inorder traversal of tree and copy array elements to tree nodes one by one. This step takes O(n) time.

Following is C implementation of the above approach. The main function to convert is highlighted in the following code.

C

```

/* A program to convert Binary Tree to Binary Search Tree */
#include<stdio.h>
#include<stdlib.h>

/* A binary tree node structure */
struct node
{
    int data;
    struct node *left;
    struct node *right;
};

/* A helper function that stores inorder traversal of a tree rooted
   with node */
void storeInorder (struct node* node, int inorder[], int *index_ptr)
{
    // Base Case
    if (node == NULL)
        return;

    /* first store the left subtree */
    storeInorder (node->left, inorder, index_ptr);

    /* Copy the root's data */
    inorder[*index_ptr] = node->data;
    (*index_ptr)++; // increase index for next entry
}
  
```

```

/* finally store the right subtree */
storeInorder (node->right, inorder, index_ptr);
}

/* A helper function to count nodes in a Binary Tree */
int countNodes (struct node* root)
{
    if (root == NULL)
        return 0;
    return countNodes (root->left) +
           countNodes (root->right) + 1;
}

// Following function is needed for library function qsort()
int compare (const void * a, const void * b)
{
    return ( *(int*)a - *(int*)b );
}

/* A helper function that copies contents of arr[] to Binary Tree.
   This function basically does Inorder traversal of Binary Tree and
   one by one copy arr[] elements to Binary Tree nodes */
void arrayToBST (int *arr, struct node* root, int *index_ptr)
{
    // Base Case
    if (root == NULL)
        return;

    /* first update the left subtree */
    arrayToBST (arr, root->left, index_ptr);

    /* Now update root's data and increment index */
    root->data = arr[*index_ptr];
    (*index_ptr)++;

    /* finally update the right subtree */
    arrayToBST (arr, root->right, index_ptr);
}

// This function converts a given Binary Tree to BST
void binaryTreeToBST (struct node *root)
{
    // base case: tree is empty
    if (root == NULL)
        return;

    /* Count the number of nodes in Binary Tree so that
       we know the size of temporary array to be created */

```

```
int n = countNodes (root);

// Create a temp array arr[] and store inorder traversal of tree in arr[]
int *arr = new int[n];
int i = 0;
storeInorder (root, arr, &i);

// Sort the array using library function for quick sort
qsort (arr, n, sizeof(arr[0]), compare);

// Copy array elements back to Binary Tree
i = 0;
arrayToBST (arr, root, &i);

// delete dynamically allocated memory to avoid meory leak
delete [] arr;
}

/* Utility function to create a new Binary Tree node */
struct node* newNode (int data)
{
    struct node *temp = new struct node;
    temp->data = data;
    temp->left = NULL;
    temp->right = NULL;
    return temp;
}

/* Utility function to print inorder traversal of Binary Tree */
void printInorder (struct node* node)
{
    if (node == NULL)
        return;

    /* first recur on left child */
    printInorder (node->left);

    /* then print the data of node */
    printf("%d ", node->data);

    /* now recur on right child */
    printInorder (node->right);
}

/* Driver function to test above functions */
int main()
{
    struct node *root = NULL;
```

```
/* Constructing tree given in the above figure
      10
     /   \
    30   15
   /     \
  20     5 */
root = newNode(10);
root->left = newNode(30);
root->right = newNode(15);
root->left->left = newNode(20);
root->right->right = newNode(5);

// convert Binary Tree to BST
binaryTreeToBST (root);

printf("Following is Inorder Traversal of the converted BST: \n");
printInorder (root);

return 0;
}
```

### Python

```
# Program to convert binary tree to BST

# A binary tree node
class Node:

    # Constructor to create a new node
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

    # Helper function to store the inroder traversal of a tree
    def storeInorder(root, inorder):

        # Base Case
        if root is None:
            return

        # First store the left subtree
        storeInorder(root.left, inorder)

        # Copy the root's data
        inorder.append(root.data)
```

```
# Finally store the right subtree
storeInorder(root.right, inorder)

# A helper function to count nodes in a binary tree
def countNodes(root):
    if root is None:
        return 0

    return countNodes(root.left) + countNodes(root.right) + 1

# Helper function that copies contents of sorted array
# to Binary tree
def arrayToBST(arr, root):

    # Base Case
    if root is None:
        return

    # First update the left subtree
    arrayToBST(arr, root.left)

    # now update root's data delete the value from array
    root.data = arr[0]
    arr.pop(0)

    # Finally update the right subtree
    arrayToBST(arr, root.right)

# This function converts a given binary tree to BST
def binaryTreeToBST(root):

    # Base Case: Tree is empty
    if root is None:
        return

    # Count the number of nodes in Binary Tree so that
    # we know the size of temporary array to be created
    n = countNodes(root)

    # Create the temp array and store the inorder traversal
    # of tree
    arr = []
    storeInorder(root, arr)

    # Sort the array
    arr.sort()

    # copy array elements back to binary tree
```

```
arrayToBST(arr, root)

# Print the inorder traversal of the tree
def printInorder(root):
    if root is None:
        return
    printInorder(root.left)
    print root.data,
    printInorder(root.right)

# Driver program to test above function
root = Node(10)
root.left = Node(30)
root.right = Node(15)
root.left.left = Node(20)
root.right.right= Node(5)

# Convert binary tree to BST
binaryTreeToBST(root)

print "Following is the inorder traversal of the converted BST"
printInorder(root)

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

Output:

```
Following is Inorder Traversal of the converted BST:
5 10 15 20 30
```

We will be covering another method for this problem which converts the tree using  $O(\text{height of tree})$  extra space.

## Source

<https://www.geeksforgeeks.org/binary-tree-to-binary-search-tree-conversion/>

## Chapter 17

# Binary Tree to Binary Search Tree Conversion using STL set

Binary Tree to Binary Search Tree Conversion using STL set - GeeksforGeeks

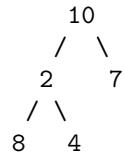
Given a Binary Tree, convert it to a [Binary Search Tree](#). The conversion must be done in such a way that keeps the original structure of Binary Tree.

This solution will use [Sets of C++ STL](#) instead of array based solution.

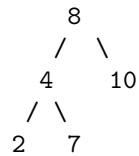
Examples:

Example 1

Input:

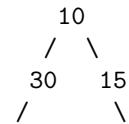


Output:



Example 2

Input:



20            5  
Output:  
               15  
             / \     
         10    20  
         /    \     
       5      30

### Solution

1. Copy the items of binary tree in a **set** while doing inorder traversal. This takes  $O(n \log n)$  time. Note that set in C++ STL is implemented using a Self Balancing Binary Search Tree like [Red Black Tree](#), [AVL Tree](#), etc
2. There is no need to sort the set as **sets** in C++ are implemented using Self-balancing binary search trees due to which each operation such as insertion, searching, deletion etc takes  $O(\log n)$  time.
3. Now simply copy the items of **set** one by one from beginning to the tree while doing inorder traversal of tree. Care should be taken as when copying each item of **set** from its beginning, we first copy it to the tree while doing inorder traversal, then remove it from the set as well.

Now the above solution is simpler and easier to implement than the array based conversion of Binary tree to Binary search tree explained here- [Conversion of Binary Tree to Binary Search tree \(Set-1\)](#), where we had to separately make a function to sort the items of the array after copying the items from tree to it.

C++ program to convert a binary tree to binary search tree using set.

```
/* CPP program to convert a Binary tree to BST
   using sets as containers. */
#include <bits/stdc++.h>
using namespace std;

struct Node {
    int data;
    struct Node *left, *right;
};

// function to store the nodes in set while
// doing inorder traversal.
void storeinorderInSet(Node* root, set<int>& s)
{
    if (!root)
        return;

    // visit the left subtree first
    storeinorderInSet(root->left, s);

    // visit the right subtree
    storeinorderInSet(root->right, s);
}
```

```
// insertion takes order of O(logn) for sets
s.insert(root->data);

// visit the right subtree
storeinorderInSet(root->right, s);

} // Time complexity = O(nlogn)

// function to copy items of set one by one
// to the tree while doing inorder traversal
void setToBST(set<int>& s, Node* root)
{
    // base condition
    if (!root)
        return;

    // first move to the left subtree and
    // update items
    setToBST(s, root->left);

    // iterator initially pointing to the
    // beginning of set
    auto it = s.begin();

    // copying the item at beginning of
    // set(sorted) to the tree.
    root->data = *it;

    // now erasing the beginning item from set.
    s.erase(it);

    // now move to right subtree and update items
    setToBST(s, root->right);

} // T(n) = O(nlogn) time

// Converts Binary tree to BST.
void binaryTreeToBST(Node* root)
{
    set<int> s;

    // populating the set with the tree's
    // inorder traversal data
    storeinorderInSet(root, s);

    // now sets are by default sorted as
    // they are implemented using self-
```

```

// balancing BST

// copying items from set to the tree
// while inorder traversal which makes a BST
setToBST(s, root);

} // Time complexity = O(nlogn),
// Auxiliary Space = O(n) for set.

// helper function to create a node
Node* newNode(int data)
{
    // dynamically allocating memory
    Node* temp = new Node();
    temp->data = data;
    temp->left = temp->right = NULL;
    return temp;
}

// function to do inorder traversal
void inorder(Node* root)
{
    if (!root)
        return;
    inorder(root->left);
    cout << root->data << " ";
    inorder(root->right);
}

int main()
{
    Node* root = newNode(5);
    root->left = newNode(7);
    root->right = newNode(9);
    root->right->left = newNode(10);
    root->left->left = newNode(1);
    root->left->right = newNode(6);
    root->right->right = newNode(11);

    /* Constructing tree given in the above figure
      5
      /   \
     7   9
     /\   / \
    1 6   10 11 */
}

// converting the above Binary tree to BST
binaryTreeToBST(root);

```

```
cout << "Inorder traversal of BST is: " << endl;
inorder(root);
return 0;
}
```

**Output:**

```
Inorder traversal of BST is:
1 5 6 7 9 10 11
```

Time Complexity :  $O(n \log n)$   
Auxiliary Space :  $(n)$

**Source**

<https://www.geeksforgeeks.org/binary-tree-binary-search-tree-conversion-using-stl-set/>

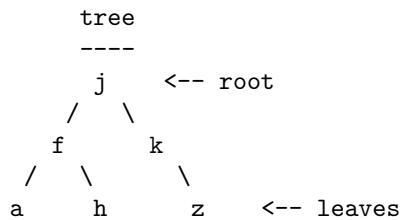
# Chapter 18

## Binary Tree | Set 1 (Introduction)

Binary Tree | Set 1 (Introduction) - GeeksforGeeks

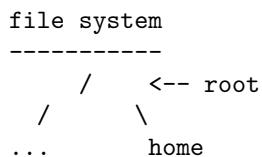
**Trees:** Unlike Arrays, Linked Lists, Stack and queues, which are linear data structures, trees are hierarchical data structures.

**Tree Vocabulary:** The topmost node is called root of the tree. The elements that are directly under an element are called its children. The element directly above something is called its parent. For example, ‘a’ is a child of ‘f’, and ‘f’ is the parent of ‘a’. Finally, elements with no children are called leaves.



### Why Trees?

1. One reason to use trees might be because you want to store information that naturally forms a hierarchy. For example, the file system on a computer:



```
      /
ugrad      \
      /   |
      cs101  cs112  cs113
```

2. Trees (with some ordering e.g., BST) provide moderate access/search (quicker than Linked List and slower than arrays).
3. Trees provide moderate insertion/deletion (quicker than Arrays and slower than Unordered Linked Lists).
4. Like Linked Lists and unlike Arrays, Trees don't have an upper limit on number of nodes as nodes are linked using pointers.

**Main applications of trees include:**

1. Manipulate hierarchical data.
2. Make information easy to search (see tree traversal).
3. Manipulate sorted lists of data.
4. As a workflow for compositing digital images for visual effects.
5. Router algorithms
6. Form of a multi-stage decision-making (see business chess).

**Binary Tree:** A tree whose elements have at most 2 children is called a binary tree. Since each element in a binary tree can have only 2 children, we typically name them the left and right child.

**Binary Tree Representation in C:** A tree is represented by a pointer to the topmost node in tree. If the tree is empty, then value of root is NULL.

A Tree node contains following parts.

1. Data
2. Pointer to left child
3. Pointer to right child

In C, we can represent a tree node using structures. Below is an example of a tree node with an integer data.

**C**

```
struct node
{
    int data;
    struct node *left;
    struct node *right;
};
```

**Python**

```
# A Python class that represents an individual node
# in a Binary Tree
class Node:
    def __init__(self, key):
        self.left = None
```

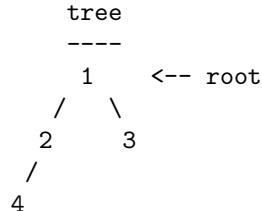
```
self.right = None  
self.val = key
```

**Java**

```
/* Class containing left and right child of current  
node and key value*/  
class Node  
{  
    int key;  
    Node left, right;  
  
    public Node(int item)  
    {  
        key = item;  
        left = right = null;  
    }  
}
```

**First Simple Tree in C**

Let us create a simple tree with 4 nodes in C. The created tree would be as following.

**C**

```
struct node  
{  
    int data;  
    struct node *left;  
    struct node *right;  
};  
  
/* newNode() allocates a new node with the given data and NULL left and  
right pointers. */  
struct node* newNode(int data)  
{  
    // Allocate memory for new node  
    struct node* node = (struct node*)malloc(sizeof(struct node));
```

```
// Assign data to this node
node->data = data;

// Initialize left and right children as NULL
node->left = NULL;
node->right = NULL;
return(node);
}

int main()
{
    /*create root*/
    struct node *root = newNode(1);
    /* following is the tree after above statement

        1
       / \
      NULL NULL
    */

    root->left      = newNode(2);
    root->right     = newNode(3);
    /* 2 and 3 become left and right children of 1
        1
       / \
      2     3
     / \   / \
    NULL NULL NULL NULL
    */

    root->left->left  = newNode(4);
    /* 4 becomes left child of 2
        1
       / \
      2     3
     / \   / \
    4   NULL NULL NULL
    */

    getchar();
    return 0;
}
```

**Python**

```
# Python program to introduce Binary Tree

# A class that represents an individual node in a
# Binary Tree
class Node:
    def __init__(self, key):
        self.left = None
        self.right = None
        self.val = key

# create root
root = Node(1)
''' following is the tree after above statement
      1
     / \
None  None'''

root.left      = Node(2);
root.right     = Node(3);

''' 2 and 3 become left and right children of 1
      1
     / \
      2      3
     / \   / \
None None None None'''

root.left.left  = Node(4);
'''4 becomes left child of 2
      1
     / \
      2      3
     / \   / \
      4    None None None
     / \
None None'''
```

**Java**

```
/* Class containing left and right child of current
   node and key value*/
class Node
{
```

```
int key;
Node left, right;

public Node(int item)
{
    key = item;
    left = right = null;
}

// A Java program to introduce Binary Tree
class BinaryTree
{
    // Root of Binary Tree
    Node root;

    // Constructors
    BinaryTree(int key)
    {
        root = new Node(key);
    }

    BinaryTree()
    {
        root = null;
    }

    public static void main(String[] args)
    {
        BinaryTree tree = new BinaryTree();

        /*create root*/
        tree.root = new Node(1);

        /* following is the tree after above statement

            1
            /
            \
           null  null      */

        tree.root.left = new Node(2);
        tree.root.right = new Node(3);

        /* 2 and 3 become left and right children of 1
            1
            /
            \
           2      3
           /      \
          /      \
```

```
    null null null null */  
  
    tree.root.left.left = new Node(4);  
    /* 4 becomes left child of 2  
        1  
       /   \  
      2     3  
     / \   / \  
    4  null  null  null  
   / \  
null  null  
*/  
}  
}  
}
```

**Summary:** Tree is a hierarchical data structure. Main uses of trees include maintaining hierarchical data, providing moderate access and insert/delete operations. Binary trees are special cases of tree where every node has at most two children.

Below are set 2 and set 3 of this post.

[Properties of Binary Tree](#)

[Types of Binary Tree](#)

**Improved By :** [nsp92](#)

## Source

<https://www.geeksforgeeks.org/binary-tree-set-1-introduction/>

## Chapter 19

# Binary Tree | Set 2 (Properties)

Binary Tree | Set 2 (Properties) - GeeksforGeeks

We have discussed [Introduction to Binary Tree in set 1](#). In this post, properties of binary are discussed.

**1) The maximum number of nodes at level 'l' of a binary tree is  $2^{l-1}$ .**

Here level is number of nodes on path from root to the node (including root and node). Level of root is 1.

This can be proved by induction.

For root, l = 1, number of nodes =  $2^{1-1} = 1$

Assume that maximum number of nodes on level l is  $2^{l-1}$

Since in Binary tree every node has at most 2 children, next level would have twice nodes, i.e.  $2 * 2^{l-1}$

**2) Maximum number of nodes in a binary tree of height 'h' is  $2^h - 1$ .**

Here height of a tree is maximum number of nodes on root to leaf path. Height of a tree with single node is considered as 1.

This result can be derived from point 2 above. A tree has maximum nodes if all levels have maximum nodes. So maximum number of nodes in a binary tree of height h is  $1 + 2 + 4 + \dots + 2^{h-1}$ . This is a simple geometric series with h terms and sum of this series is  $2^h - 1$ .

In some books, height of a leaf is considered as 0. In this convention, the above formula becomes  $2^{h+1} - 1$

**3) In a Binary Tree with N nodes, minimum possible height or minimum number of levels is  $\log_2(N+1)$**

This can be directly derived from point 2 above. If we consider the convention where height of a leaf node is considered as 0, then above formula for minimum possible height becomes  $\log_2(N+1) - 1$

**4) A Binary Tree with L leaves has at least  $\log_2 L + 1$  levels**

A Binary tree has maximum number of leaves (and minimum number of levels) when all levels are fully filled. Let all leaves be at level l, then below is true for number of leaves L.

```
L    <=  2l-1 [From Point 1]  
l = Log2L + 1  
where l is the minimum number of levels.
```

5) In Binary tree where every node has 0 or 2 children, number of leaf nodes is always one more than nodes with two children.

```
L = T + 1  
Where L = Number of leaf nodes  
T = Number of internal nodes with two children
```

See [Handshaking Lemma and Tree](#) for proof.

In the next article on tree series, we will be discussing [different types of Binary Trees and their properties](#).

## Source

<https://www.geeksforgeeks.org/binary-tree-set-2-properties/>

## Chapter 20

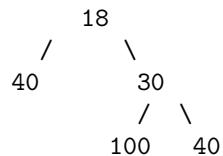
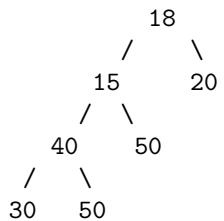
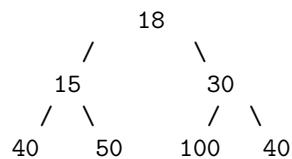
# Binary Tree | Set 3 (Types of Binary Tree)

Binary Tree | Set 3 (Types of Binary Tree) - GeeksforGeeks

We have discussed [Introduction to Binary Tree in set 1](#) and [Properties of Binary Tree in Set 2](#). In this post, common types of binary is discussed.

Following are common types of Binary Trees.

**Full Binary Tree** A Binary Tree is full if every node has 0 or 2 children. Following are examples of full binary tree. We can also say a full binary tree is a binary tree in which all nodes except leaves have two children.



**In a Full Binary, number of leaf nodes is number of internal nodes plus 1**

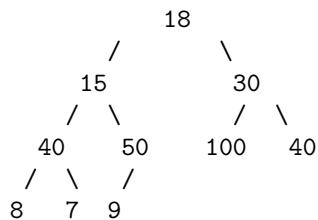
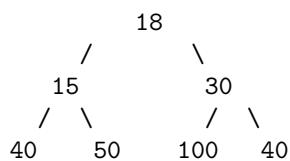
$$L = I + 1$$

Where L = Number of leaf nodes, I = Number of internal nodes

See [Handshaking Lemma and Tree](#) for proof.

**Complete Binary Tree:** A Binary Tree is complete Binary Tree if all levels are completely filled except possibly the last level and the last level has all keys as left as possible

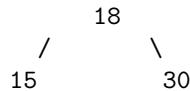
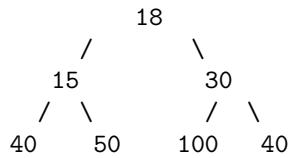
Following are examples of Complete Binary Trees



Practical example of Complete Binary Tree is [Binary Heap](#).

**Perfect Binary Tree** A Binary tree is Perfect Binary Tree in which all internal nodes have two children and all leaves are at same level.

Following are examples of Perfect Binary Trees.



A Perfect Binary Tree of height h (where height is number of nodes on path from root to leaf) has  $2^h - 1$  node.

Example of Perfect binary tree is ancestors in family. Keep a person at root, parents as children, parents of parents as their children.

### Balanced Binary Tree

A binary tree is balanced if height of the tree is  $O(\log n)$  where  $n$  is number of nodes. For Example, AVL tree maintain  $O(\log n)$  height by making sure that the difference between heights of left and right subtrees is 1. Red-Black trees maintain  $O(\log n)$  height by making sure that the number of Black nodes on every root to leaf paths are same and there are no adjacent red nodes. Balanced Binary Search trees are performance wise good as they provide  $O(\log n)$  time for search, insert and delete.

**A degenerate (or pathological) tree** A Tree where every internal node has one child. Such trees are performance-wise same as linked list.

```
    10
     /
    20
   \ 
  30
  \ 
 40
```

### Source:

[https://en.wikipedia.org/wiki/Binary\\_tree#Types\\_of\\_binary\\_trees](https://en.wikipedia.org/wiki/Binary_tree#Types_of_binary_trees)

### Source

<https://www.geeksforgeeks.org/binary-tree-set-3-types-of-binary-tree/>

## Chapter 21

# Binary tree to string with brackets

Binary tree to string with brackets - GeeksforGeeks

Construct a string consists of parenthesis and integers from a binary tree with the preorder traversing way.

The null node needs to be represented by empty parenthesis pair “()”. Omit all the empty parenthesis pairs that don't affect the one-to-one mapping relationship between the string and the original binary tree.

Examples:

```
Input : Preorder: [1, 2, 3, 4]
        1
        /   \
       2     3
      /
     4
Output: "1(2(4))(3)"
Explanation: Originally it needs to be "1(2(4)
()()(())()", but we need to omit all the
unnecessary empty parenthesis pairs.
And it will be "1(2(4))(3)".
```

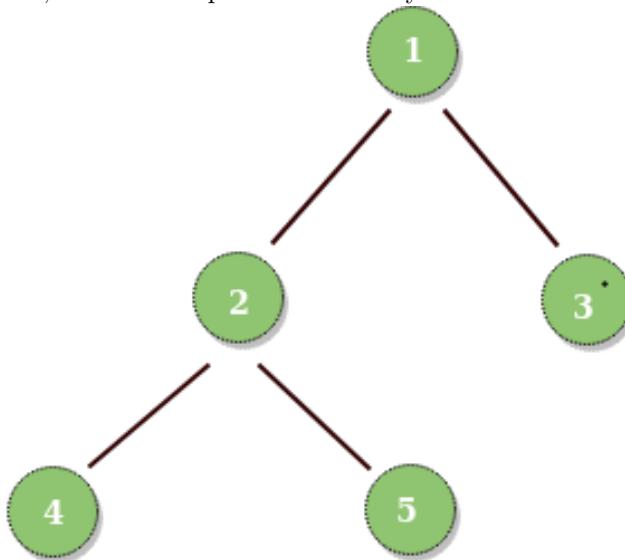
```
Input : Preorder: [1, 2, 3, null, 4]
        1
        /   \
       2     3
      \
     4
Output: "1(2()(4))(3)"
```

This is opposite of [Construct Binary Tree from String with bracket representation](#)

The idea is to do the preorder traversal of the given Binary Tree along with this, we need to make use of braces at appropriate positions. But, we also need to make sure that we omit the unnecessary braces. We print the current node and call the same given function for the left and the right children of the node in that order(if they exist). For every node encountered, the following cases are possible.

**Case 1:** Both the left child and the right child exist for the current node. In this case, we need to put the braces () around both the left child's preorder traversal output and the right child's preorder traversal output.

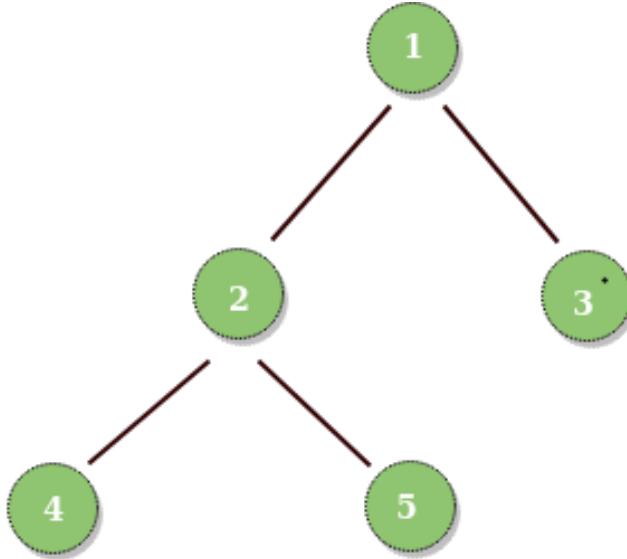
**Case 2:** None of the left or the right child exist for the current node. In this case, as shown in the figure below, considering empty braces for the null left and right children is redundant. Hence, we need not put braces for any of them.



Output :

$\mathbf{1(2(4)(5))(3()())}$   
 $\downarrow$   
 $\mathbf{1(2(4)(5))(3)}$

**Case 3:** Only the left child exists for the current node. As the figure below shows, putting empty braces for the right child in this case is unnecessary while considering the preorder traversal. This is because the right child will always come after the left child in the preorder traversal. Thus, omitting the empty braces for the right child also leads to same mapping between the string and the binary tree.



Output :

**1(2(4)(5))(3()())**  
 ↓  
**1(2(4)(5))(3)**

**Case 4:** Only the right child exists for the current node. In this case, we need to consider the empty braces for the left child. This is because, during the preorder traversal, the left child needs to be considered first. Thus, to indicate that the child following the current node is a right child we need to put a pair of empty braces for the left child.

```

/* C++ program to construct string from binary tree*/
#include <bits/stdc++.h>
using namespace std;

/* A binary tree node has data, pointer to left
   child and a pointer to right child */
struct Node {
    int data;
    Node *left, *right;
};

/* Helper function that allocates a new node */
Node* newNode(int data)
{

```

```
Node* newNode(int data)
{
    Node* node = (Node*)malloc(sizeof(Node));
    node->data = data;
    node->left = node->right = NULL;
    return (node);
}

// Function to construct string from binary tree
void treeToString(Node* root, string& str)
{
    // bases case
    if (root == NULL)
        return;

    // push the root data as character
    str.push_back(root->data + '0');

    // if leaf node, then return
    if (!root->left && !root->right)
        return;

    // for left subtree
    str.push_back('(');
    treeToString(root->left, str);
    str.push_back(')');

    // only if right child is present to
    // avoid extra parenthesis
    if (root->right) {
        str.push_back('(');
        treeToString(root->right, str);
        str.push_back(')');
    }
}

// Driver Code
int main()
{
    /* Let us construct below tree
       1
      / \
     2   3
    / \   \
   4   5   6   */
    struct Node* root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
```

```
root->right->right = newNode(6);
string str = "";
treeToString(root, str);
cout << str;
}
```

Output:

1(2(4)(5))(3()(6))

**Time complexity :**  $O(n)$  The preorder traversal is done over the n nodes.

**Space complexity :**  $O(n)$ . The depth of the recursion tree can go upto n in case of a skewed tree.

## Source

<https://www.geeksforgeeks.org/binary-tree-string-brackets/>

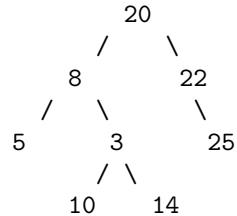
## Chapter 22

# Bottom View of a Binary Tree

Bottom View of a Binary Tree - GeeksforGeeks

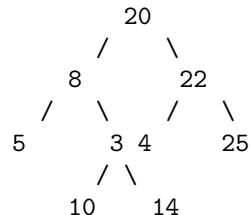
Given a Binary Tree, we need to print the bottom view from left to right. A node x is there in output if x is the bottommost node at its horizontal distance. Horizontal distance of left child of a node x is equal to horizontal distance of x minus 1, and that of right child is horizontal distance of x plus 1.

Examples:



For the above tree the output should be 5, 10, 3, 14, 25.

If there are multiple bottom-most nodes for a horizontal distance from root, then print the later one in level traversal. For example, in the below diagram, 3 and 4 are both the bottom-most nodes at horizontal distance 0, we need to print 4.



For the above tree the output should be 5, 10, 4, 14, 25.

### **Method 1 – Using Queue**

The following are steps to print Bottom View of Binary Tree.

1. We put tree nodes in a queue for the level order traversal.
2. Start with the horizontal distance(hd) 0 of the root node, keep on adding left child to queue along with the horizontal distance as hd-1 and right child as hd+1.
3. Also, use a TreeMap which stores key value pair sorted on key.
4. Every time, we encounter a new horizontal distance or an existing horizontal distance put the node data for the horizontal distance as key. For the first time it will add to the map, next time it will replace the value. This will make sure that the bottom most element for that horizontal distance is present in the map and if you see the tree from beneath that you will see that element.

A Java based implementation is below :

### **Source**

<https://www.geeksforgeeks.org/bottom-view-binary-tree/>

C++

```
// C++ Program to print Bottom View of Binary Tree
#include<bits/stdc++.h>
using namespace std;

// Tree node class
struct Node
{
    int data; //data of the node
    int hd; //horizontal distance of the node
    Node *left, *right; //left and right references

    // Constructor of tree node
    Node(int key)
    {
        data = key;
        hd = INT_MAX;
        left = right = NULL;
    }
};

// Method that prints the bottom view.
void bottomView(Node *root)
{
    if (root == NULL)
        return;
```

```
// Initialize a variable 'hd' with 0
// for the root element.
int hd = 0;

// TreeMap which stores key value pair
// sorted on key value
map<int, int> m;

// Queue to store tree nodes in level
// order traversal
queue<Node *> q;

// Assign initialized horizontal distance
// value to root node and add it to the queue.
root->hd = hd;
q.push(root); // In STL, push() is used enqueue an item

// Loop until the queue is empty (standard
// level order loop)
while (!q.empty())
{
    Node *temp = q.front();
    q.pop(); // In STL, pop() is used dequeue an item

    // Extract the horizontal distance value
    // from the dequeued tree node.
    hd = temp->hd;

    // Put the dequeued tree node to TreeMap
    // having key as horizontal distance. Every
    // time we find a node having same horizontal
    // distance we need to replace the data in
    // the map.
    m[hd] = temp->data;

    // If the dequeued node has a left child, add
    // it to the queue with a horizontal distance hd-1.
    if (temp->left != NULL)
    {
        temp->left->hd = hd-1;
        q.push(temp->left);
    }

    // If the dequeued node has a right child, add
    // it to the queue with a horizontal distance
    // hd+1.
    if (temp->right != NULL)
    {
```

```
    temp->right->hd = hd+1;
    q.push(temp->right);
}
}

// Traverse the map elements using the iterator.
for (auto i = m.begin(); i != m.end(); ++i)
    cout << i->second << " ";
}

// Driver Code
int main()
{
    Node *root = new Node(20);
    root->left = new Node(8);
    root->right = new Node(22);
    root->left->left = new Node(5);
    root->left->right = new Node(3);
    root->right->left = new Node(4);
    root->right->right = new Node(25);
    root->left->right->left = new Node(10);
    root->left->right->right = new Node(14);
    cout << "Bottom view of the given binary tree :\n";
    bottomView(root);
    return 0;
}
```

### Java

```
// Java Program to print Bottom View of Binary Tree
import java.util.*;
import java.util.Map.Entry;

// Tree node class
class Node
{
    int data; //data of the node
    int hd; //horizontal distance of the node
    Node left, right; //left and right references

    // Constructor of tree node
    public Node(int key)
    {
        data = key;
        hd = Integer.MAX_VALUE;
        left = right = null;
    }
}
```

```
//Tree class
class Tree
{
    Node root; //root node of tree

    // Default constructor
    public Tree() {}

    // Parameterized tree constructor
    public Tree(Node node)
    {
        root = node;
    }

    // Method that prints the bottom view.
    public void bottomView()
    {
        if (root == null)
            return;

        // Initialize a variable 'hd' with 0 for the root element.
        int hd = 0;

        // TreeMap which stores key value pair sorted on key value
        Map<Integer, Integer> map = new TreeMap<>();

        // Queue to store tree nodes in level order traversal
        Queue<Node> queue = new LinkedList<Node>();

        // Assign initialized horizontal distance value to root
        // node and add it to the queue.
        root.hd = hd;
        queue.add(root);

        // Loop until the queue is empty (standard level order loop)
        while (!queue.isEmpty())
        {
            Node temp = queue.remove();

            // Extract the horizontal distance value from the
            // dequeued tree node.
            hd = temp.hd;

            // Put the dequeued tree node to TreeMap having key
            // as horizontal distance. Every time we find a node
            // having same horizontal distance we need to replace
            // the data in the map.
        }
    }
}
```

```
map.put(hd, temp.data);

// If the dequeued node has a left child add it to the
// queue with a horizontal distance hd-1.
if (temp.left != null)
{
    temp.left.hd = hd-1;
    queue.add(temp.left);
}
// If the dequeued node has a left child add it to the
// queue with a horizontal distance hd+1.
if (temp.right != null)
{
    temp.right.hd = hd+1;
    queue.add(temp.right);
}

// Extract the entries of map into a set to traverse
// an iterator over that.
Set<Entry<Integer, Integer>> set = map.entrySet();

// Make an iterator
Iterator<Entry<Integer, Integer>> iterator = set.iterator();

// Traverse the map elements using the iterator.
while (iterator.hasNext())
{
    Map.Entry<Integer, Integer> me = iterator.next();
    System.out.print(me.getValue()+" ");
}
}

// Main driver class
public class BottomView
{
    public static void main(String[] args)
    {
        Node root = new Node(20);
        root.left = new Node(8);
        root.right = new Node(22);
        root.left.left = new Node(5);
        root.left.right = new Node(3);
        root.right.left = new Node(4);
        root.right.right = new Node(25);
        root.left.right.left = new Node(10);
        root.left.right.right = new Node(14);
```

```
Tree tree = new Tree(root);
System.out.println("Bottom view of the given binary tree:");
tree.bottomView();
}
}
```

Output:

```
Bottom view of the given binary tree:
5 10 4 14 25
```

### Method 2- Using HashMap()

This method is contributed by [Ekta Goel](#).

#### Approach:

Create a map like, map where key is the horizontal distance and value is a pair(a, b) where a is the value of the node and b is the height of the node. Perform a pre-order traversal of the tree. If the current node at a horizontal distance of h is the first we've seen, insert it in the map. Otherwise, compare the node with the existing one in map and if the height of the new node is greater, update in the Map.

Below is the implementation of the above:

#### C++

```
// C++ Program to print Bottom View of Binary Tree
#include < bits / stdc++.h >
#include < map >
using namespace std;

// Tree node class
struct Node
{
    // data of the node
    int data;

    // horizontal distance of the node
    int hd;

    //left and right references
    Node * left, * right;

    // Constructor of tree node
    Node(int key)
    {
        data = key;
        hd = INT_MAX;
        left = right = NULL;
    }
}
```

```

        }

};

void printBottomViewUtil(Node * root, int curr, int hd, map <int, pair <int, int>> & m)
{
    // Base case
    if (root == NULL)
        return;

    // If node for a particular
    // horizontal distance is not
    // present, add to the map.
    if (m.find(hd) == m.end())
    {
        m[hd] = make_pair(root -> data, curr);
    }
    // Compare height for already
    // present node at similar horizontal
    // distance
    else
    {
        pair < int, int > p = m[hd];
        if (p.second <= curr)
        {
            m[hd].second = curr;
            m[hd].first = root -> data;
        }
    }

    // Recur for left subtree
    printBottomViewUtil(root -> left, curr + 1, hd - 1, m);

    // Recur for right subtree
    printBottomViewUtil(root -> right, curr + 1, hd + 1, m);
}

void printBottomView(Node * root)
{
    // Map to store Horizontal Distance,
    // Height and Data.
    map < int, pair < int, int > > m;

    printBottomViewUtil(root, 0, 0, m);

    // Prints the values stored by printBottomViewUtil()
    map < int, pair < int, int > > ::iterator it;
    for (it = m.begin(); it != m.end(); ++it)
}

```

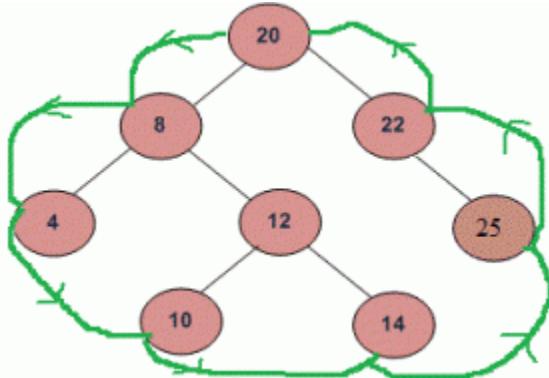
```
{  
    pair < int, int > p = it -> second;  
    cout << p.first << " ";  
}  
}  
  
int main()  
{  
    Node * root = new Node(20);  
    root -> left = new Node(8);  
    root -> right = new Node(22);  
    root -> left -> left = new Node(5);  
    root -> left -> right = new Node(3);  
    root -> right -> left = new Node(4);  
    root -> right -> right = new Node(25);  
    root -> left -> right -> left = new Node(10);  
    root -> left -> right -> right = new Node(14);  
    cout << "Bottom view of the given binary tree :\n";  
    printBottomView(root);  
    return 0;  
}
```

## Chapter 23

# Boundary Traversal of binary tree

Boundary Traversal of binary tree - GeeksforGeeks

Given a binary tree, print boundary nodes of the binary tree Anti-Clockwise starting from the root. For example, boundary traversal of the following tree is “20 8 4 10 14 25 22”



We break the problem in 3 parts:

1. Print the left boundary in top-down manner.
2. Print all leaf nodes from left to right, which can again be sub-divided into two sub-parts:
  - .....2.1 Print all leaf nodes of left sub-tree from left to right.
  - .....2.2 Print all leaf nodes of right subtree from left to right.
3. Print the right boundary in bottom-up manner.

We need to take care of one thing that nodes are not printed again. e.g. The left most node is also the leaf node of the tree.

Based on the above cases, below is the implementation:

C++

```
/* program for boundary traversal of a binary tree */
```

```
#include <stdio.h>
#include <stdlib.h>

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
    int data;
    struct node *left, *right;
};

// A simple function to print leaf nodes of a binary tree
void printLeaves(struct node* root)
{
    if ( root )
    {
        printLeaves(root->left);

        // Print it if it is a leaf node
        if ( !(root->left)  && !(root->right) )
            printf("%d ", root->data);

        printLeaves(root->right);
    }
}

// A function to print all left boundary nodes, except a leaf node.
// Print the nodes in TOP DOWN manner
void printBoundaryLeft(struct node* root)
{
    if (root)
    {
        if (root->left)
        {
            // to ensure top down order, print the node
            // before calling itself for left subtree
            printf("%d ", root->data);
            printBoundaryLeft(root->left);
        }
        else if( root->right )
        {
            printf("%d ", root->data);
            printBoundaryLeft(root->right);
        }
        // do nothing if it is a leaf node, this way we avoid
        // duplicates in output
    }
}
```

```
// A function to print all right boundary nodes, except a leaf node
// Print the nodes in BOTTOM UP manner
void printBoundaryRight(struct node* root)
{
    if (root)
    {
        if ( root->right )
        {
            // to ensure bottom up order, first call for right
            // subtree, then print this node
            printBoundaryRight(root->right);
            printf("%d ", root->data);
        }
        else if ( root->left )
        {
            printBoundaryRight(root->left);
            printf("%d ", root->data);
        }
        // do nothing if it is a leaf node, this way we avoid
        // duplicates in output
    }
}

// A function to do boundary traversal of a given binary tree
void printBoundary (struct node* root)
{
    if (root)
    {
        printf("%d ",root->data);

        // Print the left boundary in top-down manner.
        printBoundaryLeft(root->left);

        // Print all leaf nodes
        printLeaves(root->left);
        printLeaves(root->right);

        // Print the right boundary in bottom-up manner
        printBoundaryRight(root->right);
    }
}

// A utility function to create a node
struct node* newNode( int data )
{
    struct node* temp = (struct node *) malloc( sizeof(struct node) );

```

```
temp->data = data;
temp->left = temp->right = NULL;

return temp;
}

// Driver program to test above functions
int main()
{
    // Let us construct the tree given in the above diagram
    struct node *root      = newNode(20);
    root->left           = newNode(8);
    root->left->left     = newNode(4);
    root->left->right    = newNode(12);
    root->left->right->left = newNode(10);
    root->left->right->right = newNode(14);
    root->right          = newNode(22);
    root->right->right   = newNode(25);

    printBoundary( root );

    return 0;
}
```

### Java

```
//Java program to print boundary traversal of binary tree

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
class Node
{
    int data;
    Node left, right;

    Node(int item)
    {
        data = item;
        left = right = null;
    }
}

class BinaryTree
{
    Node root;

    // A simple function to print leaf nodes of a binary tree
```

```
void printLeaves(Node node)
{
    if (node != null)
    {
        printLeaves(node.left);

        // Print it if it is a leaf node
        if (node.left == null && node.right == null)
            System.out.print(node.data + " ");
        printLeaves(node.right);
    }
}

// A function to print all left boundary nodes, except a leaf node.
// Print the nodes in TOP DOWN manner
void printBoundaryLeft(Node node)
{
    if (node != null)
    {
        if (node.left != null)
        {

            // to ensure top down order, print the node
            // before calling itself for left subtree
            System.out.print(node.data + " ");
            printBoundaryLeft(node.left);
        }
        else if (node.right != null)
        {
            System.out.print(node.data + " ");
            printBoundaryLeft(node.right);
        }
    }

    // do nothing if it is a leaf node, this way we avoid
    // duplicates in output
}
}

// A function to print all right boundary nodes, except a leaf node
// Print the nodes in BOTTOM UP manner
void printBoundaryRight(Node node)
{
    if (node != null)
    {
        if (node.right != null)
        {
            // to ensure bottom up order, first call for right
            // subtree, then print this node
        }
    }
}
```

```
        printBoundaryRight(node.right);
        System.out.print(node.data + " ");
    }
    else if (node.left != null)
    {
        printBoundaryRight(node.left);
        System.out.print(node.data + " ");
    }
    // do nothing if it is a leaf node, this way we avoid
    // duplicates in output
}
}

// A function to do boundary traversal of a given binary tree
void printBoundary(Node node)
{
    if (node != null)
    {
        System.out.print(node.data + " ");

        // Print the left boundary in top-down manner.
        printBoundaryLeft(node.left);

        // Print all leaf nodes
        printLeaves(node.left);
        printLeaves(node.right);

        // Print the right boundary in bottom-up manner
        printBoundaryRight(node.right);
    }
}

// Driver program to test above functions
public static void main(String args[])
{
    BinaryTree tree = new BinaryTree();
    tree.root = new Node(20);
    tree.root.left = new Node(8);
    tree.root.left.left = new Node(4);
    tree.root.left.right = new Node(12);
    tree.root.left.right.left = new Node(10);
    tree.root.left.right.right = new Node(14);
    tree.root.right = new Node(22);
    tree.root.right.right = new Node(25);
    tree.printBoundary(tree.root);

}
```

```
// This code has been contributed by Mayank Jaiswal(mayank_24)
```

### Python

```
# Python program for binary traversal of binary tree

# A binary tree node
class Node:

    # Constructor to create a new node
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

# A simple function to print leaf nodes of a Binary Tree
def printLeaves(root):
    if(root):
        printLeaves(root.left)

        # Print it if it is a leaf node
        if root.left is None and root.right is None:
            print root.data,

        printLeaves(root.right)

# A function to print all left boundary nodes, except a
# leaf node. Print the nodes in TOP DOWN manner
def printBoundaryLeft(root):

    if(root):
        if (root.left):
            # to ensure top down order, print the node
            # before calling itself for left subtree
            print root.data,
            printBoundaryLeft(root.left)

        elif(root.right):
            print root.data,
            printBoundaryRight(root.right)

        # do nothing if it is a leaf node, this way we
        # avoid duplicates in output

# A function to print all right boundary nodes, except
```

```
# a leaf node. Print the nodes in BOTTOM UP manner
def printBoundaryRight(root):

    if(root):
        if (root.right):
            # to ensure bottom up order, first call for
            # right subtree, then print this node
            printBoundaryRight(root.right)
            print root.data,

    elif(root.left):
        printBoundaryRight(root.left)
        print root.data,

    # do nothing if it is a leaf node, this way we
    # avoid duplicates in output

# A function to do boundary traversal of a given binary tree
def printBoundary(root):
    if (root):
        print root.data,

        # Print the left boundary in top-down manner
        printBoundaryLeft(root.left)

        # Print all leaf nodes
        printLeaves(root.left)
        printLeaves(root.right)

        # Print the right boundary in bottom-up manner
        printBoundaryRight(root.right)

# Driver program to test above function
root = Node(20)
root.left = Node(8)
root.left.left = Node(4)
root.left.right = Node(12)
root.left.right.left = Node(10)
root.left.right.right = Node(14)
root.right = Node(22)
root.right.right = Node(25)
printBoundary(root)

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

Output:

20 8 4 10 14 25 22

Time Complexity: O(n) where n is the number of nodes in binary tree.

### Source

<https://www.geeksforgeeks.org/boundary-traversal-of-binary-tree/>

## Chapter 24

# Calculate depth of a full Binary tree from Preorder

Calculate depth of a full Binary tree from Preorder - GeeksforGeeks

Given preorder of a binary tree, calculate its [depth\(or height\)](#) [starting from depth 0]. The preorder is given as a string with two possible characters.

1. 'l' denotes the leaf
2. 'n' denotes internal node

The given tree can be seen as a full binary tree where every node has 0 or two children. The two children of a node can 'n' or 'l' or mix of both.

**Examples :**

Input : nlndl

Output : 2

Explanation :

Input : nlnnlll

Output : 3

Preorder of the binary tree is given so traverse

Also, we would be given a string of char (formed of 'n' and 'l'), so there is no need to implement tree also.

The recursion function would be:

- 1) Base Case: return 0; when tree[i] = 'l' or i >= strlen(tree)

2) find\_depth( tree[i++] ) //left subtree  
3) find\_depth( tree[i++] ) //right subtree

Where i is the index of the string tree.

C++

```
// C++ program to find height of full binary tree
// using preorder
#include <bits/stdc++.h>
using namespace std;

// function to return max of left subtree height
// or right subtree height
int findDepthRec(char tree[], int n, int& index)
{
    if (index >= n || tree[index] == 'l')
        return 0;

    // calc height of left subtree (In preorder
    // left subtree is processed before right)
    index++;
    int left = findDepthRec(tree, n, index);

    // calc height of right subtree
    index++;
    int right = findDepthRec(tree, n, index);

    return max(left, right) + 1;
}

// Wrapper over findDepthRec()
int findDepth(char tree[], int n)
{
    int index = 0;
    findDepthRec(tree, n, index);
}

// Driver program
int main()
{
    // Your C++ Code
    char tree[] = "nlnnlll";
    int n = strlen(tree);

    cout << findDepth(tree, n) << endl;

    return 0;
}
```

}

**Java**

```
// Java program to find height
// of full binary tree using
// preorder
import java .io.*;

class GFG
{
    // function to return max
    // of left subtree height
    // or right subtree height
    static int findDepthRec(String tree,
                           int n, int index)
    {
        if (index >= n ||
            tree.charAt(index) == 'l')
            return 0;

        // calc height of left subtree
        // (In preorder left subtree
        // is processed before right)
        index++;
        int left = findDepthRec(tree,
                               n, index);

        // calc height of
        // right subtree
        index++;
        int right = findDepthRec(tree, n, index);

        return Math.max(left, right) + 1;
    }

    // Wrapper over findDepthRec()
    static int findDepth(String tree,
                         int n)
    {
        int index = 0;
        return (findDepthRec(tree,
                           n, index));
    }

    // Driver Code
    static public void main(String[] args)
    {
```

```
String tree = "nlnnlll";
int n = tree.length();
System.out.println(findDepth(tree, n));
}
}

// This code is contributed
// by anuj_67.
```

C#

```
// C# program to find height of
// full binary tree using preorder
using System;

class GFG {

    // function to return max of left subtree
    // height or right subtree height
    static int findDepthRec(char[] tree, int n, int index)
    {
        if (index >= n || tree[index] == 'l')
            return 0;

        // calc height of left subtree (In preorder
        // left subtree is processed before right)
        index++;
        int left = findDepthRec(tree, n, index);

        // calc height of right subtree
        index++;
        int right = findDepthRec(tree, n, index);

        return Math.Max(left, right) + 1;
    }

    // Wrapper over findDepthRec()
    static int findDepth(char[] tree, int n)
    {
        int index = 0;
        return (findDepthRec(tree, n, index));
    }

    // Driver program
    static public void Main()
    {
        char[] tree = "nlnnlll".ToCharArray();
        int n = tree.Length;
```

```
        Console.WriteLine(findDepth(tree, n));
    }
}

// This code is contributed by vt_m.
```

Output:

3

Improved By : [vt\\_m](#)

## Source

<https://www.geeksforgeeks.org/calculate-depth-full-binary-tree-preorder/>

## Chapter 25

# Calculate number of nodes in all subtrees | Using DFS

Calculate number of nodes in all subtrees | Using DFS - GeeksforGeeks

Given a tree in form of adjacency list we have to calculate the number of nodes in the subtree of each node, while calculating the number of nodes in the subtree of a particular node that node will also be added as a node in subtree hence number of nodes in subtree of leaves is 1.

Examples:

Input : Consider the Graph mentioned below:

```
Output : Nodes in subtree of 1 : 5
          Nodes in subtree of 2 : 1
          Nodes in subtree of 3 : 1
          Nodes in subtree of 4 : 3
          Nodes in subtree of 5 : 1
```

Input : Consider the Graph mentioned below:

```
Output : Nodes in subtree of 1 : 7
          Nodes in subtree of 2 : 2
          Nodes in subtree of 3 : 1
          Nodes in subtree of 4 : 3
          Nodes in subtree of 5 : 1
          Nodes in subtree of 6 : 1
          Nodes in subtree of 7 : 1
```

**Explanation:** First we should calculate value count[s] : the number of nodes in subtree of node s. Where subtree contains the node itself and all the nodes in the subtree of its

children. Thus, we can calculate the number of nodes recursively using concept of DFS and DP, where we should process each edge only once and count[] value of a children used in calculating count[] of its parent expressing the concept of DP(Dynamic programming).

**Time Complexity :**  $O(n)$  [in processing of all  $(n-1)$  edges].

**Algorithm :**

```

void numberOfNodes(int s, int e)
{
    vector::iterator u;
    count1[s] = 1;
    for (u = adj[s].begin(); u != adj[s].end(); u++)
    {
        // condition to omit reverse path
        // path from children to parent
        if (*u == e)
            continue;

        // recursive call for DFS
        numberOfNodes(*u, s);

        // update count[] value of parent using
        // its children
        count1[s] += count1[*u];
    }
}
    
```

C++

```

// CPP code to find number of nodes
// in subtree of each node
#include <bits/stdc++.h>
using namespace std;

const int N = 8;

// variables used to store data globally
int count1[N];

// adjacency list representation of tree
vector<int> adj[N];

// function to calculate no. of nodes in subtree
void numberOfNodes(int s, int e)
{
    vector<int>::iterator u;
    count1[s] = 1;
    
```

```
for (u = adj[s].begin(); u != adj[s].end(); u++) {  
  
    // condition to omit reverse path  
    // path from children to parent  
    if (*u == e)  
        continue;  
  
    // recursive call for DFS  
    numberOfNodes(*u, s);  
  
    // update count[] value of parent using  
    // its children  
    count1[s] += count1[*u];  
}  
}  
  
// function to add edges in graph  
void addEdge(int a, int b)  
{  
    adj[a].push_back(b);  
    adj[b].push_back(a);  
}  
  
// function to print result  
void printNumberOfNodes()  
{  
    for (int i = 1; i < N; i++) {  
        cout << "\nNodes in subtree of " << i;  
        cout << ": " << count1[i];  
    }  
}  
  
// driver function  
int main()  
{  
    // insertion of nodes in graph  
    addEdge(1, 2);  
    addEdge(1, 4);  
    addEdge(1, 5);  
    addEdge(2, 6);  
    addEdge(4, 3);  
    addEdge(4, 7);  
  
    // call to perform dfs calculation  
    // making 1 as root of tree  
    numberOfNodes(1, 0);  
  
    // print result
```

```
    printNumberOfNodes();
    return 0;
}
```

**Java**

```
// A Java code to find number of nodes
// in subtree of each node
import java.util.ArrayList;

public class NodesInSubtree
{
    // variables used to store data globally
    static final int N = 8;
    static int count1[] = new int[N];

    // adjacency list representation of tree
    static ArrayList<Integer> adj[] = new ArrayList[N];

    // function to calculate no. of nodes in subtree
    static void numberOfWork(int s, int e)
    {
        count1[s] = 1;
        for(Integer u: adj[s])
        {
            // condition to omit reverse path
            // path from children to parent
            if(u == e)
                continue;

            // recursive call for DFS
            numberOfWork(u ,s);

            // update count[] value of parent using
            // its children
            count1[s] += count1[u];
        }
    }

    // function to add edges in graph
    static void addEdge(int a, int b)
    {
        adj[a].add(b);
        adj[b].add(a);
    }

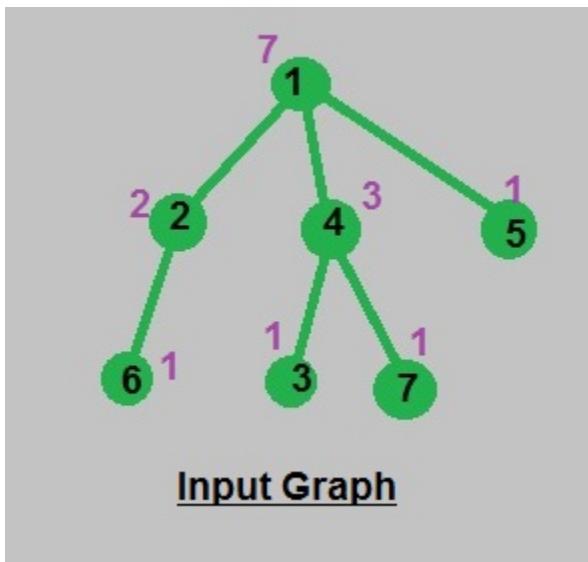
    // function to print result
    static void printNumberOfNodes()
```

```
{  
    for (int i = 1; i < N; i++)  
        System.out.println("Node of a subtree of "+ i+  
                           " : "+ count1[i]);  
}  
  
// Driver function  
public static void main(String[] args)  
{  
    // Creating list for all nodes  
    for(int i = 0; i < N; i++)  
        adj[i] = new ArrayList<>();  
  
    // insertion of nodes in graph  
    addEdge(1, 2);  
    addEdge(1, 4);  
    addEdge(1, 5);  
    addEdge(2, 6);  
    addEdge(4, 3);  
    addEdge(4, 7);  
  
    // call to perform dfs calculation  
    // making 1 as root of tree  
    numberOfNodes(1, 0);  
  
    // print result  
    printNumberOfNodes();  
}  
  
}  
// This code is contributed by Sumit Ghosh
```

Output:

```
Nodes in subtree of 1: 7  
Nodes in subtree of 2: 2  
Nodes in subtree of 3: 1  
Nodes in subtree of 4: 3  
Nodes in subtree of 5: 1  
Nodes in subtree of 6: 1  
Nodes in subtree of 7: 1
```

**Input and Output illustration:**



## Source

<https://www.geeksforgeeks.org/calculate-number-nodes-subtrees-using-dfs/>

## Chapter 26

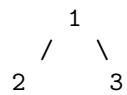
# Change a Binary Tree so that every node stores sum of all nodes in left subtree

Change a Binary Tree so that every node stores sum of all nodes in left subtree - Geeks-forGeeks

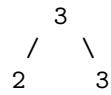
Given a Binary Tree, change the value in each node to sum of all the values in the nodes in the left subtree including its own.

Example

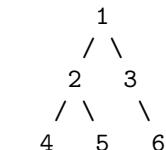
Input :



Output :

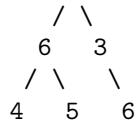


Input



Output:

12



We strongly recommend you to minimize your browser and try this yourself first.

The idea is to traverse the given tree in bottom up manner. For every node, recursively compute sum of nodes in left and right subtrees. Add sum of nodes in left subtree to current node and return sum of nodes under current subtree.

Below is C++ implementation of above idea.

```
// C++ program to store sum of nodes in left subtree in every
// node
#include<bits/stdc++.h>
using namespace std;

// A tree node
struct node
{
    int data;
    struct node* left, *right;
};

// Function to modify a Binary Tree so that every node
// stores sum of values in its left child including its
// own value
int updatetree(node *root)
{
    // Base cases
    if (!root)
        return 0;
    if (root->left == NULL && root->right == NULL)
        return root->data;

    // Update left and right subtrees
    int leftsum = updatetree(root->left);
    int rightsum = updatetree(root->right);

    // Add leftsum to current node
    root->data += leftsum;

    // Return sum of values under root
    return root->data + rightsum;
}

// Utility function to do inorder traversal
```

```
void inorder(struct node* node)
{
    if (node == NULL)
        return;
    inorder(node->left);
    printf("%d ", node->data);
    inorder(node->right);
}

// Utility function to create a new node
struct node* newNode(int data)
{
    struct node* node =
        (struct node*)malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;
    return(node);
}

// Driver program
int main()
{
    /* Let us construct below tree
           1
         /   \
        2     3
       / \   \
      4   5   6   */
    struct node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->right = newNode(6);

    updatetree(root);

    cout << "Inorder traversal of the modified tree is \n";
    inorder(root);
    return 0;
}
```

Output:

```
Inorder traversal of the modified tree is
4 6 5 12 3 6
```

Time Complexity:  $O(n)$

Thanks to Gaurav Ahirwar for suggesting this solution.

## Source

<https://www.geeksforgeeks.org/change-a-binary-tree-so-that-every-node-stores-sum-of-all-nodes-in-left-subtree/>

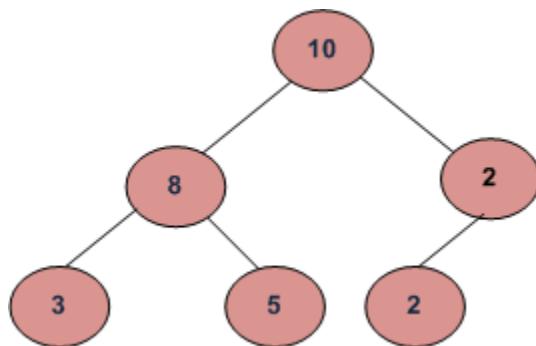
## Chapter 27

# Check for Children Sum Property in a Binary Tree

Check for Children Sum Property in a Binary Tree - GeeksforGeeks

Given a binary tree, write a function that returns true if the tree satisfies below property.

For every node, data value must be equal to sum of data values in left and right children.  
Consider data value as 0 for NULL children. Below tree is an example



### Algorithm:

Traverse the given binary tree. For each node check (recursively) if the node and both its children satisfy the Children Sum Property, if so then return true else return false.

### Implementation:

C

```
/* Program to check children sum property */
#include <stdio.h>
#include <stdlib.h>
```

```
/* A binary tree node has data, left child and right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

/* returns 1 if children sum property holds for the given
   node and both of its children*/
int isSumProperty(struct node* node)
{
    /* left_data is left child data and right_data is for right
       child data*/
    int left_data = 0, right_data = 0;

    /* If node is NULL or it's a leaf node then
       return true */
    if(node == NULL ||
       (node->left == NULL && node->right == NULL))
        return 1;
    else
    {
        /* If left child is not present then 0 is used
           as data of left child */
        if(node->left != NULL)
            left_data = node->left->data;

        /* If right child is not present then 0 is used
           as data of right child */
        if(node->right != NULL)
            right_data = node->right->data;

        /* if the node and both of its children satisfy the
           property return 1 else 0*/
        if((node->data == left_data + right_data)&&
           isSumProperty(node->left) &&
           isSumProperty(node->right))
            return 1;
        else
            return 0;
    }
}

/*
Helper function that allocates a new node
with the given data and NULL left and right
pointers.
```

```
/*
struct node* newNode(int data)
{
    struct node* node =
        (struct node*)malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;
    return(node);
}

/* Driver program to test above function */
int main()
{
    struct node *root = newNode(10);
    root->left = newNode(8);
    root->right = newNode(2);
    root->left->left = newNode(3);
    root->left->right = newNode(5);
    root->right->right = newNode(2);
    if(isSumProperty(root))
        printf("The given tree satisfies the children sum property ");
    else
        printf("The given tree does not satisfy the children sum property ");

    getchar();
    return 0;
}
```

### Java

```
// Java program to check children sum property

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
class Node
{
    int data;
    Node left, right;

    public Node(int d)
    {
        data = d;
        left = right = null;
    }
}

class BinaryTree
```

```
{  
    Node root;  
  
    /* returns 1 if children sum property holds for the given  
    node and both of its children*/  
    int isSumProperty(Node node)  
{  
  
        /* left_data is left child data and right_data is for right  
           child data*/  
        int left_data = 0, right_data = 0;  
  
        /* If node is NULL or it's a leaf node then  
        return true */  
        if (node == null  
            || (node.left == null && node.right == null))  
            return 1;  
        else  
        {  
  
            /* If left child is not present then 0 is used  
               as data of left child */  
            if (node.left != null)  
                left_data = node.left.data;  
  
            /* If right child is not present then 0 is used  
               as data of right child */  
            if (node.right != null)  
                right_data = node.right.data;  
  
            /* if the node and both of its children satisfy the  
               property return 1 else 0*/  
            if ((node.data == left_data + right_data)  
                && (isSumProperty(node.left)!=0)  
                && isSumProperty(node.right)!=0)  
                return 1;  
            else  
                return 0;  
        }  
    }  
  
    /* driver program to test the above functions */  
    public static void main(String[] args)  
    {  
        BinaryTree tree = new BinaryTree();  
        tree.root = new Node(10);  
        tree.root.left = new Node(8);  
        tree.root.right = new Node(2);  
    }
```

```
tree.root.left.left = new Node(3);
tree.root.left.right = new Node(5);
tree.root.right.right = new Node(2);
if (tree.isSumProperty(tree.root) != 0)
    System.out.println("The given tree satisfies children"
                       + " sum property");
else
    System.out.println("The given tree does not satisfy children"
                       + " sum property");
}
}
```

Output:

```
The given tree satisfies the children sum property
```

**Time Complexity:** O(n), we are doing a complete traversal of the tree.

As an exercise, extend the above question for an n-ary tree.

This question was asked by Shekhar.

## Source

<https://www.geeksforgeeks.org/check-for-children-sum-property-in-a-binary-tree/>

## Chapter 28

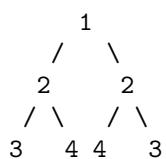
# Check for Symmetric Binary Tree (Iterative Approach)

Check for Symmetric Binary Tree (Iterative Approach) - GeeksforGeeks

Given a binary tree, check whether it is a mirror of itself without recursion.

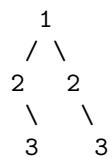
Examples:

Input :



Output : Symmetric

Input :



Output : Not Symmetric

We have discussed recursive approach to solve this problem in below post :

[Symmetric Tree \(Mirror Image of itself\)](#)

In this post, iterative approach is discussed. We use Queue here. Note that for a symmetric that elements at every level are palindromic. In example 2, at the leaf level- the elements are which is not palindromic.

In other words,

1. The left child of left subtree = right child of right subtree.
2. The right child of left subtree = left child of right subtree.

If we insert the left child of left subtree first followed by right child of the right subtree in the queue, we only need to ensure that these are equal.

Similarly, If we insert the right child of left subtree followed by left child of the right subtree in the queue, we again need to ensure that these are equal.

Below is the implementation based on above idea.

C++

```
// C++ program to check if a given Binary
// Tree is symmetric or not
#include<bits/stdc++.h>
using namespace std;

// A Binary Tree Node
struct Node
{
    int key;
    struct Node* left, *right;
};

// Utility function to create new Node
Node *newNode(int key)
{
    Node *temp = new Node;
    temp->key = key;
    temp->left = temp->right = NULL;
    return (temp);
}

// Returns true if a tree is symmetric
// i.e. mirror image of itself
bool isSymmetric(struct Node* root)
{
    if(root == NULL)
        return true;

    // If it is a single tree node, then
    // it is a symmetric tree.
    if(!root->left && !root->right)
        return true;

    queue <Node*> q;
```

```
// Add root to queue two times so that
// it can be checked if either one child
// alone is NULL or not.
q.push(root);
q.push(root);

// To store two nodes for checking their
// symmetry.
Node* leftNode, *rightNode;

while(!q.empty()){

    // Remove first two nodes to check
    // their symmetry.
    leftNode = q.front();
    q.pop();

    rightNode = q.front();
    q.pop();

    // if both left and right nodes
    // exist, but have different
    // values--> inequality, return false
    if(leftNode->key != rightNode->key){
        return false;
    }

    // Push left child of left subtree node
    // and right child of right subtree
    // node in queue.
    if(leftNode->left && rightNode->right){
        q.push(leftNode->left);
        q.push(rightNode->right);
    }

    // If only one child is present alone
    // and other is NULL, then tree
    // is not symmetric.
    else if ((leftNode->left || rightNode->right)
        return false;

    // Push right child of left subtree node
    // and left child of right subtree node
    // in queue.
    if(leftNode->right && rightNode->left){
        q.push(leftNode->right);
        q.push(rightNode->left);
    }
}
```

```
}

// If only one child is present alone
// and other is NULL, then tree
// is not symmetric.
else if(leftNode->right || rightNode->left)
    return false;
}

return true;
}

// Driver program
int main()
{
    // Let us construct the Tree shown in
    // the above figure
    Node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(2);
    root->left->left = newNode(3);
    root->left->right = newNode(4);
    root->right->left = newNode(4);
    root->right->right = newNode(3);

    if(isSymmetric(root))
        cout << "The given tree is Symmetric";
    else
        cout << "The given tree is not Symmetric";
    return 0;
}

// This code is contributed by Nikhil jindal.
```

**Java**

```
// Iterative Java program to check if
// given binary tree symmetric
import java.util.* ;

public class BinaryTree
{
    Node root;
    static class Node
    {
        int val;
        Node left, right;
        Node(int v)
```

```
{  
    val = v;  
    left = null;  
    right = null;  
}  
}  
  
/* constructor to initialise the root */  
BinaryTree(Node r) { root = r; }  
  
/* empty constructor */  
BinaryTree() { }  
  
/* function to check if the tree is Symmetric */  
public boolean isSymmetric(Node root)  
{  
    /* This allows adding null elements to the queue */  
    Queue<Node> q = new LinkedList<Node>();  
  
    /* Initially, add left and right nodes of root */  
    q.add(root.left);  
    q.add(root.right);  
  
    while (!q.isEmpty())  
    {  
        /* remove the front 2 nodes to  
           check for equality */  
        Node tempLeft = q.remove();  
        Node tempRight = q.remove();  
  
        /* if both are null, continue and check  
           for further elements */  
        if (tempLeft==null && tempRight==null)  
            continue;  
  
        /* if only one is null---inequality, return false */  
        if ((tempLeft==null && tempRight!=null) ||  
            (tempLeft!=null && tempRight==null))  
            return false;  
  
        /* if both left and right nodes exist, but  
           have different values-- inequality,  
           return false*/  
        if (tempLeft.val != tempRight.val)  
            return 0;  
  
        /* Note the order of insertion of elements
```

```
        to the queue :  
        1) left child of left subtree  
        2) right child of right subtree  
        3) right child of left subtree  
        4) left child of right subtree */  
        q.add(tempLeft.left);  
        q.add(tempRight.right);  
        q.add(tempLeft.right);  
        q.add(tempRight.left);  
    }  
  
    /* if the flow reaches here, return true*/  
    return true;  
}  
  
/* driver function to test other functions */  
public static void main(String[] args)  
{  
    Node n = new Node(1);  
    BinaryTree bt = new BinaryTree(n);  
    bt.root.left = new Node(2);  
    bt.root.right = new Node(2);  
    bt.root.left.left = new Node(3);  
    bt.root.left.right = new Node(4);  
    bt.root.right.left = new Node(4);  
    bt.root.right.right = new Node(3);  
  
    if (bt.isSymmetric(bt.root))  
        System.out.println("The given tree is Symmetric");  
    else  
        System.out.println("The given tree is not Symmetric");  
}  
}
```

Output: The given tree is Symmetric

Improved By : [nik1996](#)

## Source

<https://www.geeksforgeeks.org/check-symmetric-binary-tree-iterative-approach/>

## Chapter 29

# Check given array of size n can represent BST of n levels or not

Check given array of size n can represent BST of n levels or not - GeeksforGeeks

Given an array of size n, the task is to find whether array can represent a BST with n levels.

Since levels are n, we construct a tree in the following manner.

Assuming a number X,

- Number higher than X is on the right side
- Number lower than X is on the left side.

*Note: during the insertion, we never go beyond a number already visited.*

Examples:

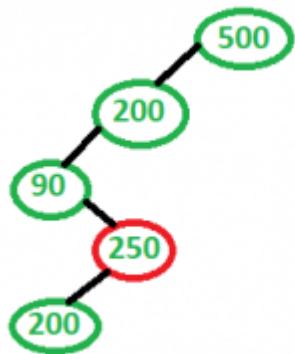
Input : 500, 200, 90, 250, 100

Output : No

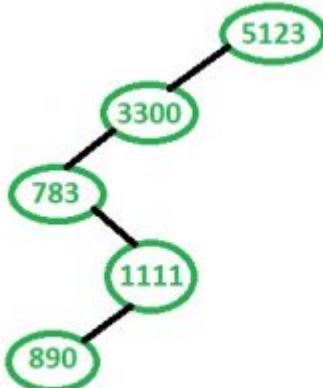
Input : 5123, 3300, 783, 1111, 890

Output : Yes

Explanation :



For the sequence **500, 200, 90, 250, 100** formed tree(in above image) can't represent BST.



The sequence **5123, 3300, 783, 1111, 890** forms a binary search tree hence its a correct sequence.

#### Method 1 : By constructing BST

We first insert all array values level by level in a Tree. To insert, we check if current value is less than previous value or greater. After constructing the tree, we check if the constructed tree is [Binary Search Tree or not](#).

```
// C++ program to Check given array
// can represent BST or not
#include <bits/stdc++.h>
using namespace std;

// structure for Binary Node
struct Node {
    int key;
    struct Node *right, *left;
};

// Function to check if a binary tree is BST or not
bool isBST(struct Node* node, int min, int max) {
    if (node == NULL)
        return true;

    if (node->key < min || node->key > max)
        return false;

    return isBST(node->left, min, node->key) && isBST(node->right, node->key, max);
}
```

```
Node* newNode(int num)
{
    Node* temp = new Node;
    temp->key = num;
    temp->left = NULL;
    temp->right = NULL;
    return temp;
}

// To create a Tree with n levels. We always
// insert new node to left if it is less than
// previous value.
Node* createNLevelTree(int arr[], int n)
{
    Node* root = newNode(arr[0]);
    Node* temp = root;
    for (int i = 1; i < n; i++) {
        if (temp->key > arr[i]) {
            temp->left = newNode(arr[i]);
            temp = temp->left;
        }
        else {
            temp->right = newNode(arr[i]);
            temp = temp->right;
        }
    }
    return root;
}

// Please refer below post for details of this
// function.
// https://www.geeksforgeeks.org/a-program-to-check-if-a-binary-tree-is-bst-or-not/
bool isBST(Node* root, int min, int max)
{
    if (root == NULL)
        return true;

    if (root->key < min || root->key > max)
        return false;

    // Allow only distinct values
    return (isBST(root->left, min,
                  (root->key) - 1)
            && isBST(root->right,
                      (root->key) + 1, max));
}
```

```
// Returns tree if given array of size n can
// represent a BST of n levels.
bool canRepresentNLevelBST(int arr[], int n)
{
    Node* root = createNLevelTree(arr, n);
    return isBST(root, INT_MIN, INT_MAX);
}

// Driver code
int main()
{
    int arr[] = { 512, 330, 78, 11, 8 };
    int n = sizeof(arr) / sizeof(arr[0]);

    if (canRepresentNLevelBST(arr, n))
        cout << "Yes";
    else
        cout << "No";

    return 0;
}
```

**Output:**

Yes

**Method 2 (Array Based)**

1. Take two variables max = INT\_MAX to mark the maximum limit for left subtree and min = INT\_MIN to mark the minimum limit for right subtree.
2. Loop from arr[1] to arr[n-1]
3. for each element check
  - a. If ( arr[i] > arr[i-1] && arr[i] > min && arr[i] < max ), update min = arr[i-1]
  - b. Else if ( arr[i] < min && arr[i] < max ), update max = arr[i]
  - c. If none of the above two conditions hold, then element will not be inserted in a new level, so break.

```
// C++ program to Check given array
// can represent BST or not
#include <bits/stdc++.h>
using namespace std;

// Driver code
int main()
{
    int arr[] = { 5123, 3300, 783, 1111, 890 };
    int n = sizeof(arr) / sizeof(arr[0]);
```

```
int max = INT_MAX;
int min = INT_MIN;
bool flag = true;

for (int i = 1; i < n; i++) {

    // This element can be inserted to the right
    // of the previous element, only if it is greater
    // than the previous element and in the range.
    if (arr[i] > arr[i - 1] && arr[i] > min && arr[i] < max) {
        // max remains same, update min
        min = arr[i - 1];
    }
    // This element can be inserted to the left
    // of the previous element, only if it is lesser
    // than the previous element and in the range.
    else if (arr[i] < arr[i - 1] && arr[i] > min && arr[i] < max) {
        // min remains same, update max
        max = arr[i - 1];
    }
    else {
        flag = false;
        break;
    }
}

if (flag) {
    cout << "Yes";
}
else {
    // if the loop completed successfully without encountering else condition
    cout << "No";
}

return 0;
}
```

**Output:**

Yes

Improved By : [Sakshi Parashar](#)

**Source**

<https://www.geeksforgeeks.org/check-given-array-of-size-n-can-represent-bst-of-n-levels-or-not/>

## Chapter 30

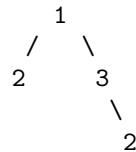
# Check if a Binary Tree (not BST) has duplicate values

Check if a Binary Tree (not BST) has duplicate values - GeeksforGeeks

Check if a Binary Tree (not BST) has duplicate values

Examples:

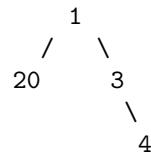
Input : Root of below tree



Output : Yes

Explanation : The duplicate value is 2.

Input : Root of below tree



Output : No

Explanation : There are no duplicates.

A simple solution is to store inorder traversal of given binary tree in an array. Then check if array has duplicates or not. We can avoid the use of array and solve the problem in O(n) time. The idea is to use hashing. We traverse the given tree, for every node, we check if it already exists in hash table. If exists, we return true (found duplicate). If it does not exist, we insert into hash table.

C++

```
// C++ Program to check duplicates
// in Binary Tree
#include <bits/stdc++.h>
using namespace std;

// A binary tree Node has data,
// pointer to left child
// and a pointer to right child
struct Node {
    int data;
    struct Node* left;
    struct Node* right;
};

// Helper function that allocates
// a new Node with the given data
// and NULL left and right pointers.
struct Node* newNode(int data)
{
    struct Node* node = new Node;
    node->data = data;
    node->left = node->right = NULL;
    return (node);
}

bool checkDupUtil(Node* root, unordered_set<int> &s)
{
    // If tree is empty, there are no
    // duplicates.
    if (root == NULL)
        return false;

    // If current node's data is already present.
    if (s.find(root->data) != s.end())
        return true;

    // Insert current node
    s.insert(root->data);

    // Recursively check in left and right
    // subtrees.
    return checkDupUtil(root->left, s) ||
           checkDupUtil(root->right, s);
}

// To check if tree has duplicates
```

```
bool checkDup(struct Node* root)
{
    unordered_set<int> s;
    return checkDupUtil(root, s);
}

// Driver program to test above functions
int main()
{
    struct Node* root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(2);
    root->left->left = newNode(3);
    if (checkDup(root))
        printf("Yes");
    else
        printf("No");

    return 0;
}
```

Output:

Yes

Improved By : [Aditya Sharma 3, Ankit.19](#)

## Source

<https://www.geeksforgeeks.org/check-binary-tree-not-bst-duplicate-values/>

## Chapter 31

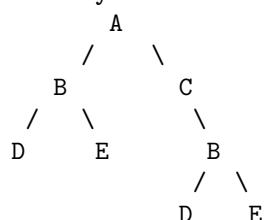
# Check if a Binary Tree contains duplicate subtrees of size 2 or more

Check if a Binary Tree contains duplicate subtrees of size 2 or more - GeeksforGeeks

Given a Binary Tree, check whether the Binary tree contains a duplicate sub-tree of size 2 or more.

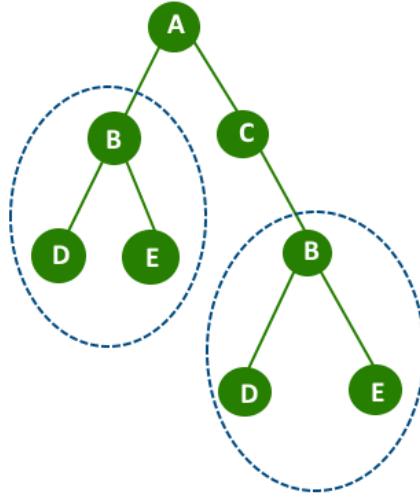
Note : Two same leaf nodes are not considered as subtree size of a leaf node is one.

Input : Binary Tree



Output : Yes

Asked in : Google Interview



Tree with duplicate Sub-Tree [ highlight by blue color ellipse ]

[ Method 1]

A simple solution is that, we pick every node of tree and try to find is any sub-tree of given tree is present in tree which is identical with that sub-tree. Here we can use below post to find if a subtree is present anywhere else in tree.

[Check if a binary tree is subtree of another binary tree](#)

[Method 2] ( Efficient solution )

An Efficient solution based on [tree serialization](#) and [hashing](#). The idea is to serialize subtrees as strings and store the strings in hash table. Once we find a serialized tree (which is not a leaf) already existing in hash-table, we return true.

Below c++ implementation of above idea.

```

// C++ program to find if there is a duplicate
// sub-tree of size 2 or more.
#include<bits/stdc++.h>
using namespace std;

// Separator node
const char MARKER = '$';

// Structure for a binary tree node
struct Node
{
    char key;
    Node *left, *right;
};

// A utility function to create a new node

```

```
Node* newNode(char key)
{
    Node* node = new Node;
    node->key = key;
    node->left = node->right = NULL;
    return node;
}

unordered_set<string> subtrees;

// This function returns empty string if tree
// contains a duplicate subtree of size 2 or more.
string dupSubUtil(Node *root)
{
    string s = "";

    // If current node is NULL, return marker
    if (root == NULL)
        return s + MARKER;

    // If left subtree has a duplicate subtree.
    string lStr = dupSubUtil(root->left);
    if (lStr.compare(s) == 0)
        return s;

    // Do same for right subtree
    string rStr = dupSubUtil(root->right);
    if (rStr.compare(s) == 0)
        return s;

    // Serialize current subtree
    s = s + root->key + lStr + rStr;

    // If current subtree already exists in hash
    // table. [Note that size of a serialized tree
    // with single node is 3 as it has two marker
    // nodes.
    if (s.length() > 3 &&
        subtrees.find(s) != subtrees.end())
        return "";

    subtrees.insert(s);

    return s;
}

// Driver program to test above functions
int main()
```

```
{  
    Node *root = newNode('A');  
    root->left = newNode('B');  
    root->right = newNode('C');  
    root->left->left = newNode('D');  
    root->left->right = newNode('E');  
    root->right->right = newNode('B');  
    root->right->right->right = newNode('E');  
    root->right->right->left= newNode('D');  
  
    string str = dupSubUtil(root);  
  
    (str.compare("") == 0) ? cout << " Yes ":"  
                           cout << " No " ;  
    return 0;  
}
```

Output:

Yes

### Source

<https://www.geeksforgeeks.org/check-binary-tree-contains-duplicate-subtrees-size-2/>

## Chapter 32

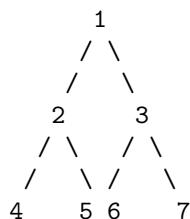
# Check if a binary tree is sorted level-wise or not

Check if a binary tree is sorted level-wise or not - GeeksforGeeks

Given a binary tree. The task is to check if the binary tree is sorted level-wise or not. A binary tree is level sorted if  $\max(i-1^{\text{th}} \text{ level}) < \min(i^{\text{th}} \text{ level})$ .

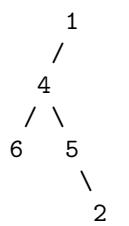
**Examples:**

Input :



Output : Sorted

Input:



Output: Not sorted

**Simple Solution:** A simple solution is to compare minimum and maximum value of each adjacent level  $i$  and  $i+1$ . Traverse to  $i^{\text{th}}$  and  $i+1^{\text{th}}$  level, compare the minimum value of  $i+1^{\text{th}}$  level with maximum value of  $i^{\text{th}}$  level and return the result.

Time complexity:  $O(n^2)$ .

**Efficient Solution:** An efficient solution is to do level order traversal and keep track of the minimum and maximum values of current level. Use a variable *prevMax* to store the maximum value of the previous level. Then compare the minimum value of current level with the maximum value of the previous level, *prevMax*. If minimum value is greater than the *prevMax*, then the given tree is sorted level-wise up to current level. For next level, *prevMax* is the equal to maximum value of current level. So update the *prevMax* with maximum value of current level. Repeat this until all levels of given tree are not traversed.

Below is the implementation of above approach:

```
// CPP program to determine whether
// binary tree is level sorted or not.

#include <bits/stdc++.h>
using namespace std;

// Structure of a tree node.
struct Node {
    int key;
    Node *left, *right;
};

// Function to create new tree node.
Node* newNode(int key)
{
    Node* temp = new Node;
    temp->key = key;
    temp->left = temp->right = NULL;
    return temp;
}

// Function to determine if
// given binary tree is level sorted
// or not.
int isSorted(Node* root)
{
    // to store maximum value of previous
    // level.
    int prevMax = INT_MIN;

    // to store minimum value of current
    // level.
    int minval;

    // to store maximum value of current
    // level.
    int maxval;
```

```
// to store number of nodes in current
// level.
int levelSize;

// queue to perform level order traversal.
queue<Node*> q;
q.push(root);

while (!q.empty()) {

    // find number of nodes in current
    // level.
    levelSize = q.size();

    minval = INT_MAX;
    maxval = INT_MIN;

    // traverse current level and find
    // minimum and maximum value of
    // this level.
    while (levelSize > 0) {
        root = q.front();
        q.pop();

        levelSize--;

        minval = min(minval, root->key);
        maxval = max(maxval, root->key);

        if (root->left)
            q.push(root->left);

        if (root->right)
            q.push(root->right);
    }

    // if minimum value of this level
    // is not greater than maximum
    // value of previous level then
    // given tree is not level sorted.
    if (minval <= prevMax)
        return 0;

    // maximum value of this level is
    // previous maximum value for
    // next level.
    prevMax = maxval;
}
```

```
        return 1;
}

// Driver program
int main()
{
    /*
        1
       /
      4
     \
      6
     / \
    8   9
   /   \
  12   10
*/
Node* root = newNode(1);
root->left = newNode(4);
root->left->right = newNode(6);
root->left->right->left = newNode(8);
root->left->right->right = newNode(9);
root->left->right->left->left = newNode(12);
root->left->right->right->right = newNode(10);

if (isSorted(root))
    cout << "Sorted";
else
    cout << "Not sorted";
return 0;
}
```

**Output:**

Sorted

**Time Complexity:** O(n)  
**Auxiliary Space:** O(n)

**Source**

<https://www.geeksforgeeks.org/check-if-a-binary-tree-is-sorted-level-wise-or-not/>

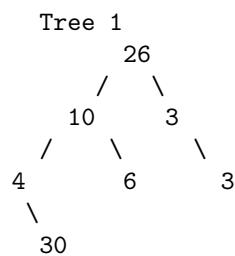
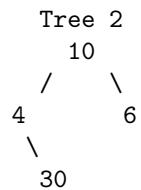
## Chapter 33

# Check if a binary tree is subtree of another binary tree | Set 1

Check if a binary tree is subtree of another binary tree | Set 1 - GeeksforGeeks

Given two binary trees, check if the first tree is subtree of the second one. A subtree of a tree T is a tree S consisting of a node in T and all of its descendants in T. The subtree corresponding to the root node is the entire tree; the subtree corresponding to any other node is called a proper subtree.

For example, in the following case, tree S is a subtree of tree T.



**Solution:** Traverse the tree T in preorder fashion. For every visited node in the traversal, see if the subtree rooted with this node is identical to S.

Following is the implementation for this.

C

```
#include <stdio.h>
#include <stdlib.h>

/* A binary tree node has data, left child and right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

/* A utility function to check whether trees with roots as root1 and
   root2 are identical or not */
bool areIdentical(struct node * root1, struct node *root2)
{
    /* base cases */
    if (root1 == NULL && root2 == NULL)
        return true;

    if (root1 == NULL || root2 == NULL)
        return false;

    /* Check if the data of both roots is same and data of left and right
       subtrees are also same */
    return (root1->data == root2->data    &&
            areIdentical(root1->left, root2->left) &&
            areIdentical(root1->right, root2->right) );
}

/* This function returns true if S is a subtree of T, otherwise false */
bool isSubtree(struct node *T, struct node *S)
{
    /* base cases */
    if (S == NULL)
        return true;

    if (T == NULL)
        return false;

    /* Check the tree with root as current node */
    if (areIdentical(T, S))
```

```

        return true;

    /* If the tree with root as current node doesn't match then
       try left and right subtrees one by one */
    return isSubtree(T->left, S) ||
           isSubtree(T->right, S);
}

/* Helper function that allocates a new node with the given data
   and NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* node =
        (struct node*)malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;
    return(node);
}

/* Driver program to test above function */
int main()
{
    // TREE 1
    /* Construct the following tree
        26
        /   \
      10     3
      / \   \
     4   6   3
        \
      30
    */
    struct node *T      = newNode(26);
    T->right          = newNode(3);
    T->right->right  = newNode(3);
    T->left           = newNode(10);
    T->left->left    = newNode(4);
    T->left->left->right = newNode(30);
    T->left->right   = newNode(6);

    // TREE 2
    /* Construct the following tree
        10
        /   \
      4     6
        \
      30
    */
}

```

```
30
*/
struct node *S      = newNode(10);
S->right         = newNode(6);
S->left          = newNode(4);
S->left->right  = newNode(30);

if (isSubtree(T, S))
    printf("Tree 2 is subtree of Tree 1");
else
    printf("Tree 2 is not a subtree of Tree 1");

getchar();
return 0;
}
```

**Java**

```
// Java program to check if binary tree is subtree of another binary tree

// A binary tree node
class Node
{
    int data;
    Node left, right, nextRight;

    Node(int item)
    {
        data = item;
        left = right = nextRight = null;
    }
}

class BinaryTree
{
    Node root1,root2;

    /* A utility function to check whether trees with roots as root1 and
       root2 are identical or not */
    boolean areIdentical(Node root1, Node root2)
    {

        /* base cases */
        if (root1 == null && root2 == null)
            return true;

        if (root1 == null || root2 == null)

```

```

        return false;

    /* Check if the data of both roots is same and data of left and right
       subtrees are also same */
    return (root1.data == root2.data
            && areIdentical(root1.left, root2.left)
            && areIdentical(root1.right, root2.right));
}

/* This function returns true if S is a subtree of T, otherwise false */
boolean isSubtree(Node T, Node S)
{
    /* base cases */
    if (S == null)
        return true;

    if (T == null)
        return false;

    /* Check the tree with root as current node */
    if (areIdentical(T, S))
        return true;

    /* If the tree with root as current node doesn't match then
       try left and right subtrees one by one */
    return isSubtree(T.left, S)
           || isSubtree(T.right, S);
}

public static void main(String args[])
{
    BinaryTree tree = new BinaryTree();

    // TREE 1
    /* Construct the following tree
       26
       /   \
      10   3
      / \   \
     4  6   3
        \
       30 */

    tree.root1 = new Node(26);
    tree.root1.right = new Node(3);
    tree.root1.right.right = new Node(3);
    tree.root1.left = new Node(10);
    tree.root1.left.left = new Node(4);
}

```

```
tree.root1.left.left.right = new Node(30);
tree.root1.left.right = new Node(6);

// TREE 2
/* Construct the following tree
   10
   /   \
  4     6
   \   /
  30  */

tree.root2 = new Node(10);
tree.root2.right = new Node(6);
tree.root2.left = new Node(4);
tree.root2.left.right = new Node(30);

if (tree.isSubtree(tree.root1, tree.root2))
    System.out.println("Tree 2 is subtree of Tree 1 ");
else
    System.out.println("Tree 2 is not a subtree of Tree 1");
}

}

// This code has been contributed by Mayank Jaiswal
```

### Python

```
# Python program to check binary tree is a subtree of
# another tree

# A binary tree node
class Node:

    # Constructor to create a new node
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

    # A utility function to check whether trees with roots
    # as root1 and root2 are identical or not
def areIdentical(root1, root2):

    # Base Case
    if root1 is None and root2 is None:
        return True
    if root1 is None or root2 is None:
        return False
```

```

# Check if the data of both roots is same and data of
# left and right subtrees are also same
return (root1.data == root2.data and
        areIdentical(root1.left , root2.left)and
        areIdentical(root1.right, root2.right)
       )

# This function returns True if S is a subtree of T,
# otherwise False
def isSubtree(T, S):

    # Base Case
    if S is None:
        return True

    if T is None:
        return False

    # Check the tree with root as current node
    if (areIdentical(T, S)):
        return True

    # IF the tree with root as current node doesn't match
    # then try left and right subtree one by one
    return isSubtree(T.left, S) or isSubtree(T.right, S)

# Driver program to test above function

""" TREE 1
Construct the following tree
      26
     /   \
    10   3
   / \   \
  4   6   3
 / \
30
"""

T = Node(26)
T.right = Node(3)
T.right.right = Node(3)
T.left = Node(10)
T.left.left = Node(4)
T.left.left.right = Node(30)
T.left.right = Node(6)

```

```
**** TREE 2
Construct the following tree
    10
   /   \
  4    6
  \
  30
"""
S = Node(10)
S.right = Node(6)
S.left = Node(4)
S.left.right = Node(30)

if isSubtree(T, S):
    print "Tree 2 is subtree of Tree 1"
else :
    print "Tree 2 is not a subtree of Tree 1"

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

Output:

```
Tree 2 is subtree of Tree 1
```

Time Complexity: Time worst case complexity of above solution is  $O(mn)$  where m and n are number of nodes in given two trees.

We can solve the above problem in  $O(n)$  time. Please refer [Check if a binary tree is subtree of another binary tree | Set 2](#) for  $O(n)$  solution.

**Improved By :** [SrivathsanAravamudan](#)

## Source

<https://www.geeksforgeeks.org/check-if-a-binary-tree-is-subtree-of-another-binary-tree/>

## Chapter 34

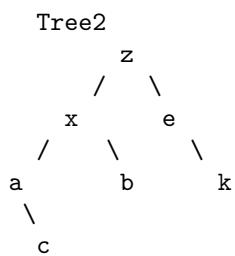
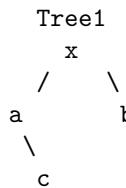
# Check if a binary tree is subtree of another binary tree | Set 2

Check if a binary tree is subtree of another binary tree | Set 2 - GeeksforGeeks

Given two binary trees, check if the first tree is subtree of the second one. A subtree of a tree T is a tree S consisting of a node in T and all of its descendants in T.

The subtree corresponding to the root node is the entire tree; the subtree corresponding to any other node is called a proper subtree.

For example, in the following case, Tree1 is a subtree of Tree2.



We have discussed a  $O(n^2)$  solution for this problem. In this post a  $O(n)$  solution is discussed. The idea is based on the fact that inorder and preorder/postorder uniquely identify a binary

[tree](#). Tree S is a subtree of T if both inorder and preorder traversals of S are substrings of inorder and preorder traversals of T respectively.

Following are detailed steps.

1) Find inorder and preorder traversals of T, store them in two auxiliary arrays inT[] and preT[].

2) Find inorder and preorder traversals of S, store them in two auxiliary arrays inS[] and preS[].

3) If inS[] is a subarray of inT[] and preS[] is a subarray of preT[], then S is a subtree of T. Else not.

We can also use postorder traversal in place of preorder in the above algorithm.

Let us consider the above example

Inorder and Preorder traversals of the big tree are.

```
inT[] = {a, c, x, b, z, e, k}
preT[] = {z, x, a, c, b, e, k}
```

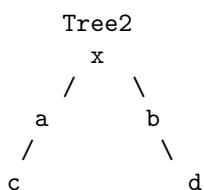
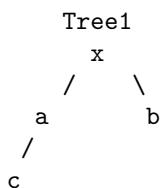
Inorder and Preorder traversals of small tree are

```
inS[] = {a, c, x, b}
preS[] = {x, a, c, b}
```

We can easily figure out that inS[] is a subarray of inT[] and preS[] is a subarray of preT[].

## EDIT

The above algorithm doesn't work for cases where a tree is present in another tree, but not as a subtree. Consider the following example.



Inorder and Preorder traversals of the big tree or Tree2 are.

Inorder and Preorder traversals of small tree or Tree1 are

The Tree2 is not a subtree of Tree1, but inS[] and preS[] are subarrays of inT[] and preT[] respectively.

The above algorithm can be extended to handle such cases by adding a special character whenever we encounter NULL in inorder and preorder traversals. Thanks to Shivam Goel for suggesting this extension.

Following is the implementation of above algorithm.

C

```
#include <iostream>
#include <cstring>
using namespace std;
#define MAX 100

// Structure of a tree node
struct Node
{
    char key;
    struct Node *left, *right;
};

// A utility function to create a new BST node
Node *newNode(char item)
{
    Node *temp = new Node;
    temp->key = item;
    temp->left = temp->right = NULL;
    return temp;
}

// A utility function to store inorder traversal of tree rooted
// with root in an array arr[]. Note that i is passed as reference
void storeInorder(Node *root, char arr[], int &i)
{
    if (root == NULL)
    {
        arr[i++] = '$';
        return;
    }
    storeInorder(root->left, arr, i);
    arr[i++] = root->key;
    storeInorder(root->right, arr, i);
}
```

```

}

// A utility function to store preorder traversal of tree rooted
// with root in an array arr[]. Note that i is passed as reference
void storePreOrder(Node *root, char arr[], int &i)
{
    if (root == NULL)
    {
        arr[i++] = '$';
        return;
    }
    arr[i++] = root->key;
    storePreOrder(root->left, arr, i);
    storePreOrder(root->right, arr, i);
}

/* This function returns true if S is a subtree of T, otherwise false */
bool isSubtree(Node *T, Node *S)
{
    /* base cases */
    if (S == NULL)  return true;
    if (T == NULL)  return false;

    // Store Inorder traversals of T and S in inT[0..m-1]
    // and inS[0..n-1] respectively
    int m = 0, n = 0;
    char inT[MAX], inS[MAX];
    storeInorder(T, inT, m);
    storeInorder(S, inS, n);
    inT[m] = '\0', inS[n] = '\0';

    // If inS[] is not a substring of preS[], return false
    if (strstr(inT, inS) == NULL)
        return false;

    // Store Preorder traversals of T and S in inT[0..m-1]
    // and inS[0..n-1] respectively
    m = 0, n = 0;
    char preT[MAX], preS[MAX];
    storePreOrder(T, preT, m);
    storePreOrder(S, preS, n);
    preT[m] = '\0', preS[n] = '\0';

    // If inS[] is not a substring of preS[], return false
    // Else return true
    return (strstr(preT, preS) != NULL);
}

```

```
// Driver program to test above function
int main()
{
    Node *T = newNode('a');
    T->left = newNode('b');
    T->right = newNode('d');
    T->left->left = newNode('c');
    T->right->right = newNode('e');

    Node *S = newNode('a');
    S->left = newNode('b');
    S->left->left = newNode('c');
    S->right = newNode('d');

    if (isSubtree(T, S))
        cout << "Yes: S is a subtree of T";
    else
        cout << "No: S is NOT a subtree of T";

    return 0;
}
```

### Java

```
// Java program to check if binary tree is subtree of another binary tree
class Node {

    char data;
    Node left, right;

    Node(char item) {
        data = item;
        left = right = null;
    }
}

class Passing {

    int i;
    int m = 0;
    int n = 0;
}

class BinaryTree {

    static Node root;
    Passing p = new Passing();
```

```

String strstr(String haystack, String needle) {
    if (haystack == null || needle == null) {
        return null;
    }
    int hLength = haystack.length();
    int nLength = needle.length();
    if (hLength < nLength) {
        return null;
    }
    if (nLength == 0) {
        return haystack;
    }
    for (int i = 0; i <= hLength - nLength; i++) {
        if (haystack.charAt(i) == needle.charAt(0)) {
            int j = 0;
            for (; j < nLength; j++) {
                if (haystack.charAt(i + j) != needle.charAt(j)) {
                    break;
                }
            }
            if (j == nLength) {
                return haystack.substring(i);
            }
        }
    }
    return null;
}

// A utility function to store inorder traversal of tree rooted
// with root in an array arr[]. Note that i is passed as reference
void storeInorder(Node node, char arr[], Passing i) {
    if (node == null) {
        arr[i.i++] = '$';
        return;
    }
    storeInorder(node.left, arr, i);
    arr[i.i++] = node.data;
    storeInorder(node.right, arr, i);
}

// A utility function to store preorder traversal of tree rooted
// with root in an array arr[]. Note that i is passed as reference
void storePreOrder(Node node, char arr[], Passing i) {
    if (node == null) {
        arr[i.i++] = '$';
        return;
    }
    arr[i.i++] = node.data;
}

```

```

        storePreOrder(node.left, arr, i);
        storePreOrder(node.right, arr, i);
    }

/* This function returns true if S is a subtree of T, otherwise false */
boolean isSubtree(Node T, Node S) {
    /* base cases */
    if (S == null) {
        return true;
    }
    if (T == null) {
        return false;
    }

    // Store Inorder traversals of T and S in inT[0..m-1]
    // and inS[0..n-1] respectively
    char inT[] = new char[100];
    String op1 = String.valueOf(inT);
    char inS[] = new char[100];
    String op2 = String.valueOf(inS);
    storeInorder(T, inT, p);
    storeInorder(S, inS, p);
    inT[p.m] = '\0';
    inS[p.m] = '\0';

    // If inS[] is not a substring of preS[], return false
    if (strstr(op1, op2) != null) {
        return false;
    }

    // Store Preorder traversals of T and S in inT[0..m-1]
    // and inS[0..n-1] respectively
    p.m = 0;
    p.n = 0;
    char preT[] = new char[100];
    char preS[] = new char[100];
    String op3 = String.valueOf(preT);
    String op4 = String.valueOf(preS);
    storePreOrder(T, preT, p);
    storePreOrder(S, preS, p);
    preT[p.m] = '\0';
    preS[p.n] = '\0';

    // If inS[] is not a substring of preS[], return false
    // Else return true
    return (strstr(op3, op4) != null);
}

```

```
//Driver program to test above functions
public static void main(String args[]) {
    BinaryTree tree = new BinaryTree();
    Node T = new Node('a');
    T.left = new Node('b');
    T.right = new Node('d');
    T.left.left = new Node('c');
    T.right.right = new Node('e');

    Node S = new Node('a');
    S.left = new Node('b');
    S.right = new Node('d');
    S.left.left = new Node('c');

    if (tree.isSubtree(T, S)) {
        System.out.println("Yes , S is a subtree of T");
    } else {
        System.out.println("No, S is not a subtree of T");
    }
}

// This code is contributed by Mayank Jaiswal
```

Output:

No: S is NOT a subtree of T

Time Complexity: Inorder and Preorder traversals of Binary Tree take  $O(n)$  time. The function [strstr\(\)](#) can also be implemented in  $O(n)$  time using [KMP string matching algorithm](#).

Auxiliary Space:  $O(n)$

Thanks to **Ashwini Singh** for suggesting this method. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## Source

<https://www.geeksforgeeks.org/check-binary-tree-subtree-another-binary-tree-set-2/>

## Chapter 35

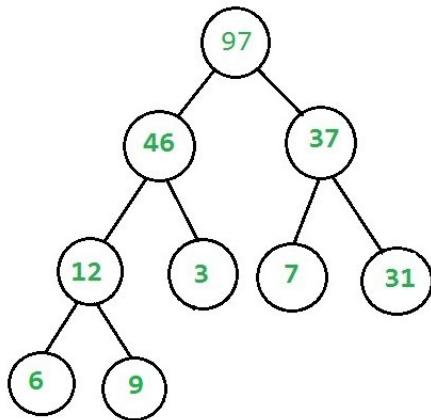
# Check if a given Binary Tree is Heap

Check if a given Binary Tree is Heap - GeeksforGeeks

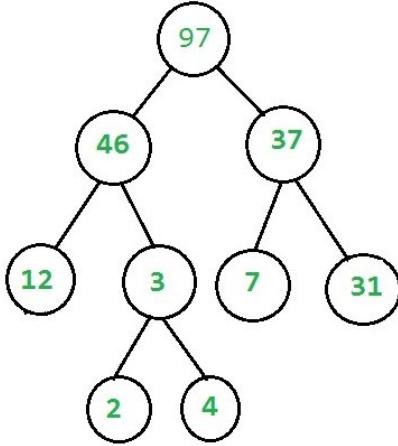
Given a binary tree we need to check it has heap property or not, Binary tree need to fulfill following two conditions for being a heap –

1. It should be a complete tree (i.e. all levels except last should be full).
2. Every node's value should be greater than or equal to its child node (considering max-heap).

For example this tree contains heap property –



While this doesn't –



We check each of the above condition separately, for checking completeness isComplete and for checking heap isHeapUtil function are written.

Detail about isComplete function can be found [here](#).

isHeapUtil function is written considering following things –

1. Every Node can have 2 children, 0 child (last level nodes) or 1 child (there can be at most one such node).
  2. If Node has No child then it's a leaf node and return true (Base case)
  3. If Node has one child (it must be left child because it is a complete tree) then we need to compare this node with its single child only.
  4. If Node has both child then check heap property at Node at recur for both subtrees.
- Complete code.

## Implementation

### C/C++

```

/* C program to checks if a binary tree is max heap ot not */
#include<stdio.h>
#include<stdlib.h>
#include<stdbool.h>

/* Tree node structure */
struct Node
{
    int key;
    struct Node *left;
    struct Node *right;
};

/* Helper function that allocates a new node */
  
```

```

struct Node *newNode(int k)
{
    struct Node *node = (struct Node*)malloc(sizeof(struct Node));
    node->key = k;
    node->right = node->left = NULL;
    return node;
}

/* This function counts the number of nodes in a binary tree */
unsigned int countNodes(struct Node* root)
{
    if (root == NULL)
        return (0);
    return (1 + countNodes(root->left) + countNodes(root->right));
}

/* This function checks if the binary tree is complete or not */
bool isCompleteUtil (struct Node* root, unsigned int index,
                     unsigned int number_nodes)
{
    // An empty tree is complete
    if (root == NULL)
        return (true);

    // If index assigned to current node is more than
    // number of nodes in tree, then tree is not complete
    if (index >= number_nodes)
        return (false);

    // Recur for left and right subtrees
    return (isCompleteUtil(root->left, 2*index + 1, number_nodes) &&
            isCompleteUtil(root->right, 2*index + 2, number_nodes));
}

// This Function checks the heap property in the tree.
bool isHeapUtil(struct Node* root)
{
    // Base case : single node satisfies property
    if (root->left == NULL && root->right == NULL)
        return (true);

    // node will be in second last level
    if (root->right == NULL)
    {
        // check heap property at Node
        // No recursive call , because no need to check last level
        return (root->key >= root->left->key);
    }
}

```

```
else
{
    // Check heap property at Node and
    // Recursive check heap property at left and right subtree
    if (root->key >= root->left->key &&
        root->key >= root->right->key)
        return ((isHeapUtil(root->left)) &&
                (isHeapUtil(root->right)));
    else
        return (false);
}

// Function to check binary tree is a Heap or Not.
bool isHeap(struct Node* root)
{
    // These two are used in isCompleteUtil()
    unsigned int node_count = countNodes(root);
    unsigned int index = 0;

    if (isCompleteUtil(root, index, node_count) && isHeapUtil(root))
        return true;
    return false;
}

// Driver program
int main()
{
    struct Node* root = NULL;
    root = newNode(10);
    root->left = newNode(9);
    root->right = newNode(8);
    root->left->left = newNode(7);
    root->left->right = newNode(6);
    root->right->left = newNode(5);
    root->right->right = newNode(4);
    root->left->left->left = newNode(3);
    root->left->left->right = newNode(2);
    root->left->right->left = newNode(1);

    if (isHeap(root))
        printf("Given binary tree is a Heap\n");
    else
        printf("Given binary tree is not a Heap\n");

    return 0;
}
```

**Java**

```
/* Java program to checks if a binary tree is max heap ot not */

// A Binary Tree node
class Node
{
    int key;
    Node left, right;

    Node(int k)
    {
        key = k;
        left = right = null;
    }
}

class Is_BinaryTree_MaxHeap
{
    /* This function counts the number of nodes in a binary tree */
    int countNodes(Node root)
    {
        if(root==null)
            return 0;
        return(1 + countNodes(root.left) + countNodes(root.right));
    }

    /* This function checks if the binary tree is complete or not */
    boolean isCompleteUtil(Node root, int index, int number_nodes)
    {
        // An empty tree is complete
        if(root == null)
            return true;

        // If index assigned to current node is more than
        // number of nodes in tree, then tree is not complete
        if(index >= number_nodes)
            return false;

        // Recur for left and right subtrees
        return isCompleteUtil(root.left, 2*index+1, number_nodes) &&
               isCompleteUtil(root.right, 2*index+2, number_nodes);
    }

    // This Function checks the heap property in the tree.
    boolean isHeapUtil(Node root)
    {
```

```
// Base case : single node satisfies property
if(root.left == null && root.right==null)
    return true;

// node will be in second last level
if(root.right == null)
{
    // check heap property at Node
    // No recursive call , because no need to check last level
    return root.key >= root.left.key;
}
else
{
    // Check heap property at Node and
    // Recursive check heap property at left and right subtree
    if(root.key >= root.left.key && root.key >= root.right.key)
        return isHeapUtil(root.left) && isHeapUtil(root.right);
    else
        return false;
}
}

// Function to check binary tree is a Heap or Not.
boolean isHeap(Node root)
{
    if(root == null)
        return true;

    // These two are used in isCompleteUtil()
    int node_count = countNodes(root);

    if(isCompleteUtil(root, 0 , node_count)==true && isHeapUtil(root)==true)
        return true;
    return false;
}

// driver function to test the above functions
public static void main(String args[])
{
    Is_BinaryTree_MaxHeap bt = new Is_BinaryTree_MaxHeap();

    Node root = new Node(10);
    root.left = new Node(9);
    root.right = new Node(8);
    root.left.left = new Node(7);
    root.left.right = new Node(6);
    root.right.left = new Node(5);
    root.right.right = new Node(4);
```

```
root.left.left.left = new Node(3);
root.left.left.right = new Node(2);
root.left.right.left = new Node(1);

if(bt.isHeap(root) == true)
    System.out.println("Given binary tree is a Heap");
else
    System.out.println("Given binary tree is not a Heap");
}

}

// This code has been contributed by Amit Khandelwal
```

### Python

```
# To check if a binary tree
# is a MAX Heap or not
class GFG:
    def __init__(self, value):
        self.key = value
        self.left = None
        self.right = None

    def count_nodes(self, root):
        if root is None:
            return 0
        else:
            return (1 + self.count_nodes(root.left) +
                    self.count_nodes(root.right))

    def heap_propert_util(self, root):

        if (root.left is None and
            root.right is None):
            return True

        if root.right is None:
            return root.key >= root.left.key
        else:
            if (root.key >= root.left.key and
                root.key >= root.right.key):
                return (self.heap_propert_util(root.left) and
                        self.heap_propert_util(root.right))
            else:
                return False

    def complete_tree_util(self, root,
                          index, node_count):
```

```
if root is None:
    return True
if index >= node_count:
    return False
return (self.complete_tree_util(root.left, 2 *
                               index + 1, node_count) and
        self.complete_tree_util(root.right, 2 *
                               index + 2, node_count))

def check_if_heap(self):
    node_count = self.count_nodes(self)
    if (self.complete_tree_util(self, 0, node_count) and
        self.heap_propert_util(self)):
        return True
    else:
        return False

# Driver Code
root = GFG(5)
root.left = GFG(2)
root.right = GFG(3)
root.left.left = GFG(1)

if root.check_if_heap():
    print("Given binary tree is a heap")
else:
    print("Given binary tree is not a Heap")

# This code has been
# contributed by Yash Agrawal
```

#### Output:

Given binary tree is a Heap

This article is contributed by Utkarsh Trivedi. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Improved By : [scorncr17](#)

#### Source

<https://www.geeksforgeeks.org/check-if-a-given-binary-tree-is-heap/>

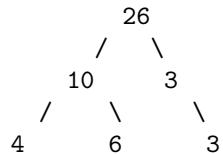
## Chapter 36

# Check if a given Binary Tree is SumTree

Check if a given Binary Tree is SumTree - GeeksforGeeks

Write a function that returns true if the given Binary Tree is SumTree else false. A SumTree is a Binary Tree where the value of a node is equal to sum of the nodes present in its left subtree and right subtree. An empty tree is SumTree and sum of an empty tree can be considered as 0. A leaf node is also considered as SumTree.

Following is an example of SumTree.



### Method 1 ( Simple )

Get the sum of nodes in left subtree and right subtree. Check if the sum calculated is equal to root's data. Also, recursively check if the left and right subtrees are SumTrees.

C

```
#include <stdio.h>
#include <stdlib.h>

/* A binary tree node has data, left child and right child */
struct node
{
    int data;
    struct node* left;
```

```
    struct node* right;
};

/* A utility function to get the sum of values in tree with root
   as root */
int sum(struct node *root)
{
    if(root == NULL)
        return 0;
    return sum(root->left) + root->data + sum(root->right);
}

/* returns 1 if sum property holds for the given
   node and both of its children */
int isSumTree(struct node* node)
{
    int ls, rs;

    /* If node is NULL or it's a leaf node then
       return true */
    if(node == NULL ||
       (node->left == NULL && node->right == NULL))
        return 1;

    /* Get sum of nodes in left and right subtrees */
    ls = sum(node->left);
    rs = sum(node->right);

    /* if the node and both of its children satisfy the
       property return 1 else 0*/
    if((node->data == ls + rs)&&
       isSumTree(node->left) &&
       isSumTree(node->right))
        return 1;

    return 0;
}

/*
Helper function that allocates a new node
with the given data and NULL left and right
pointers.
*/
struct node* newNode(int data)
{
    struct node* node =
        (struct node*)malloc(sizeof(struct node));
    node->data = data;
```

```
    node->left = NULL;
    node->right = NULL;
    return(node);
}

/* Driver program to test above function */
int main()
{
    struct node *root = newNode(26);
    root->left = newNode(10);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(6);
    root->right->right = newNode(3);
    if(isSumTree(root))
        printf("The given tree is a SumTree ");
    else
        printf("The given tree is not a SumTree ");

    getchar();
    return 0;
}
```

### Java

```
// Java program to check if Binary tree is sum tree or not

/* A binary tree node has data, left child and right child */
class Node
{
    int data;
    Node left, right, nextRight;

    Node(int item)
    {
        data = item;
        left = right = nextRight = null;
    }
}

class BinaryTree
{
    Node root;

    /* A utility function to get the sum of values in tree with root
       as root */
    int sum(Node node)
    {
```

```
if (node == null)
    return 0;
return sum(node.left) + node.data + sum(node.right);
}

/* returns 1 if sum property holds for the given
   node and both of its children */
int isSumTree(Node node)
{
    int ls, rs;

    /* If node is NULL or it's a leaf node then
       return true */
    if ((node == null) || (node.left == null && node.right == null))
        return 1;

    /* Get sum of nodes in left and right subtrees */
    ls = sum(node.left);
    rs = sum(node.right);

    /* if the node and both of its children satisfy the
       property return 1 else 0*/
    if ((node.data == ls + rs) && (isSumTree(node.left) != 0)
        && (isSumTree(node.right)) != 0)
        return 1;

    return 0;
}

/* Driver program to test above functions */
public static void main(String args[])
{
    BinaryTree tree = new BinaryTree();
    tree.root = new Node(26);
    tree.root.left = new Node(10);
    tree.root.right = new Node(3);
    tree.root.left.left = new Node(4);
    tree.root.left.right = new Node(6);
    tree.root.right.right = new Node(3);

    if (tree.isSumTree(tree.root) != 0)
        System.out.println("The given tree is a sum tree");
    else
        System.out.println("The given tree is not a sum tree");
}
}

// This code has been contributed by Mayank Jaiswal
```

Output:

```
The given tree is a SumTree
```

Time Complexity:  $O(n^2)$  in worst case. Worst case occurs for a skewed tree.

### Method 2 ( Tricky )

The Method 1 uses sum() to get the sum of nodes in left and right subtrees. The method 2 uses following rules to get the sum directly.

- 1) If the node is a leaf node then sum of subtree rooted with this node is equal to value of this node.
- 2) If the node is not a leaf node then sum of subtree rooted with this node is twice the value of this node (Assuming that the tree rooted with this node is SumTree).

C

```
#include <stdio.h>
#include <stdlib.h>

/* A binary tree node has data, left child and right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

/* Utility function to check if the given node is leaf or not */
int isLeaf(struct node *node)
{
    if(node == NULL)
        return 0;
    if(node->left == NULL && node->right == NULL)
        return 1;
    return 0;
}

/* returns 1 if SumTree property holds for the given
   tree */
int isSumTree(struct node* node)
{
    int ls; // for sum of nodes in left subtree
    int rs; // for sum of nodes in right subtree

    /* If node is NULL or it's a leaf node then
       return true */
    if(node == NULL || isLeaf(node))
```

```
    return 1;

if( isSumTree(node->left) && isSumTree(node->right))
{
    // Get the sum of nodes in left subtree
    if(node->left == NULL)
        ls = 0;
    else if(isLeaf(node->left))
        ls = node->left->data;
    else
        ls = 2*(node->left->data);

    // Get the sum of nodes in right subtree
    if(node->right == NULL)
        rs = 0;
    else if(isLeaf(node->right))
        rs = node->right->data;
    else
        rs = 2*(node->right->data);

    /* If root's data is equal to sum of nodes in left
       and right subtrees then return 1 else return 0*/
    return(node->data == ls + rs);
}

return 0;
}

/* Helper function that allocates a new node
   with the given data and NULL left and right
   pointers.
*/
struct node* newNode(int data)
{
    struct node* node =
        (struct node*)malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;
    return(node);
}

/* Driver program to test above function */
int main()
{
    struct node *root = newNode(26);
    root->left = newNode(10);
    root->right = newNode(3);
```

```
root->left->left    = newNode(4);
root->left->right   = newNode(6);
root->right->right  = newNode(3);
if(isSumTree(root))
    printf("The given tree is a SumTree ");
else
    printf("The given tree is not a SumTree ");

getchar();
return 0;
}
```

**Java**

```
// Java program to check if Binary tree is sum tree or not

/* A binary tree node has data, left child and right child */
class Node
{
    int data;
    Node left, right, nextRight;

    Node(int item)
    {
        data = item;
        left = right = nextRight = null;
    }
}

class BinaryTree
{
    Node root;

    /* Utility function to check if the given node is leaf or not */
    int isLeaf(Node node)
    {
        if (node == null)
            return 0;
        if (node.left == null && node.right == null)
            return 1;
        return 0;
    }

    /* returns 1 if SumTree property holds for the given
       tree */
    int isSumTree(Node node)
    {
```

```
int ls; // for sum of nodes in left subtree
int rs; // for sum of nodes in right subtree

/* If node is NULL or it's a leaf node then
   return true */
if (node == null || isLeaf(node) == 1)
    return 1;

if (isSumTree(node.left) != 0 && isSumTree(node.right) != 0)
{
    // Get the sum of nodes in left subtree
    if (node.left == null)
        ls = 0;
    else if (isLeaf(node.left) != 0)
        ls = node.left.data;
    else
        ls = 2 * (node.left.data);

    // Get the sum of nodes in right subtree
    if (node.right == null)
        rs = 0;
    else if (isLeaf(node.right) != 0)
        rs = node.right.data;
    else
        rs = 2 * (node.right.data);

    /* If root's data is equal to sum of nodes in left
       and right subtrees then return 1 else return 0*/
    if ((node.data == rs + ls))
        return 1;
    else
        return 0;
}

return 0;
}

/* Driver program to test above functions */
public static void main(String args[])
{
    BinaryTree tree = new BinaryTree();
    tree.root = new Node(26);
    tree.root.left = new Node(10);
    tree.root.right = new Node(3);
    tree.root.left.left = new Node(4);
    tree.root.left.right = new Node(6);
    tree.root.right.right = new Node(3);
```

```
if (tree.isSumTree(tree.root) != 0)
    System.out.println("The given tree is a sum tree");
else
    System.out.println("The given tree is not a sum tree");
}
}

// This code has been contributed by Mayank Jaiswal
```

Output:

```
The given tree is a sum tree
```

Time Complexity: O(n)

## Source

<https://www.geeksforgeeks.org/check-if-a-given-binary-tree-is-sumtree/>

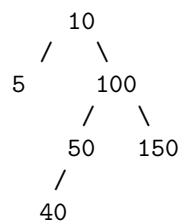
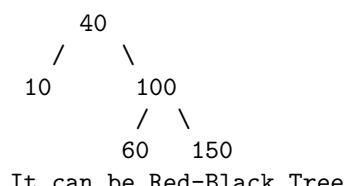
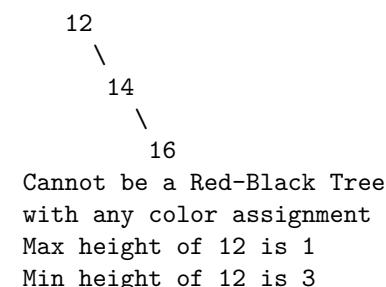
## Chapter 37

# Check if a given Binary Tree is height balanced like a Red-Black Tree

Check if a given Binary Tree is height balanced like a Red-Black Tree - GeeksforGeeks

In a [Red-Black Tree](#), the maximum height of a node is at most twice the minimum height ([The four Red-Black tree properties](#) make sure this is always followed). Given a Binary Search Tree, we need to check for following property.

*For every node, length of the longest leaf to node path has not more than twice the nodes on shortest path from node to leaf.*



It can also be Red-Black Tree

Expected time complexity is O(n). The tree should be traversed at-most once in the solution.

**We strongly recommend to minimize the browser and try this yourself first.**

For every node, we need to get the maximum and minimum heights and compare them. The idea is to traverse the tree and for every node check if it's balanced. We need to write a recursive function that returns three things, a boolean value to indicate the tree is balanced or not, minimum height and maximum height. To return multiple values, we can either use a structure or pass variables by reference. We have passed maxh and minh by reference so that the values can be used in parent calls.

```
/* Program to check if a given Binary Tree is balanced like a Red-Black Tree */
#include <iostream>
using namespace std;

struct Node
{
    int key;
    Node *left, *right;
};

/* utility that allocates a new Node with the given key */
Node* newNode(int key)
{
    Node* node = new Node;
    node->key = key;
    node->left = node->right = NULL;
    return (node);
}

// Returns returns tree if the Binary tree is balanced like a Red-Black
// tree. This function also sets value in maxh and minh (passed by
// reference). maxh and minh are set as maximum and minimum heights of root.
bool isBalancedUtil(Node *root, int &maxh, int &minh)
{
    // Base case
    if (root == NULL)
    {
        maxh = minh = 0;
        return true;
    }

    int lmxh, lmnh; // To store max and min heights of left subtree
    int rmxh, rmnh; // To store max and min heights of right subtree

    // Check if left subtree is balanced, also set lmxh and lmnh
    if (isBalancedUtil(root->left, lmxh, lmnh) == false)
```

```
    return false;

    // Check if right subtree is balanced, also set rmxh and rmnh
    if (isBalancedUtil(root->right, rmxh, rmnh) == false)
        return false;

    // Set the max and min heights of this node for the parent call
    maxh = max(lmxh, rmxh) + 1;
    minh = min(lmnh, rmnh) + 1;

    // See if this node is balanced
    if (maxh <= 2*minh)
        return true;

    return false;
}

// A wrapper over isBalancedUtil()
bool isBalanced(Node *root)
{
    int maxh, minh;
    return isBalancedUtil(root, maxh, minh);
}

/* Driver program to test above functions*/
int main()
{
    Node * root = newNode(10);
    root->left = newNode(5);
    root->right = newNode(100);
    root->right->left = newNode(50);
    root->right->right = newNode(150);
    root->right->left->left = newNode(40);
    isBalanced(root)? cout << "Balanced" : cout << "Not Balanced";

    return 0;
}
```

Output:

Balanced

Time Complexity: Time Complexity of above code is  $O(n)$  as the code does a simple tree traversal.

## Source

<https://www.geeksforgeeks.org/check-given-binary-tree-follows-height-property-red-black-tree/>

## Chapter 38

# Check if a given array can represent Preorder Traversal of Binary Search Tree

Check if a given array can represent Preorder Traversal of Binary Search Tree - Geeks-forGeeks

Given an array of numbers, return true if given array can represent preorder traversal of a Binary Search Tree, else return false. Expected time complexity is O(n).

Examples:

```
Input: pre[] = {2, 4, 3}
Output: true
Given array can represent preorder traversal
of below tree
    2
        4
        /
        3
```

```
Input: pre[] = {2, 4, 1}
Output: false
Given array cannot represent preorder traversal
of a Binary Search Tree.
```

```
Input: pre[] = {40, 30, 35, 80, 100}
Output: true
Given array can represent preorder traversal
of below tree
```

```
    40
   /
 30   80
   35   100
```

Input: pre[] = {40, 30, 35, 20, 80, 100}

Output: false

Given array cannot represent preorder traversal  
of a Binary Search Tree.

A **Simple Solution** is to do following for every node pre[i] starting from first one.

- 1) Find the first greater value on right side of current node.  
Let the index of this node be j. Return true if following  
conditions hold. Else return false
  - (i) All values after the above found greater value are  
greater than current node.
  - (ii) Recursive calls for the subarrays pre[i+1..j-1] and  
pre[j+1..n-1] also return true.

Time Complexity of the above solution is  $O(n^2)$

An **Efficient Solution** can solve this problem in  $O(n)$  time. The idea is to use a stack. This problem is similar to [Next \(or closest\) Greater Element problem](#). Here we find next greater element and after finding next greater, if we find a smaller element, then return false.

- 1) Create an empty stack.
- 2) Initialize root as INT\_MIN.
- 3) Do following for every element pre[i]
  - a) If pre[i] is smaller than current root, return false.
  - b) Keep removing elements from stack while pre[i] is greater  
than stack top. Make the last removed item as new root (to  
be compared next).  
At this point, pre[i] is greater than the removed root  
(That is why if we see a smaller element in step a), we  
return false)
  - c) push pre[i] to stack (All elements in stack are in decreasing  
order)

Below is implementation of above idea.

C++

```
// C++ program for an efficient solution to check if
// a given array can represent Preorder traversal of
// a Binary Search Tree
#include<bits/stdc++.h>
using namespace std;

bool canRepresentBST(int pre[], int n)
{
    // Create an empty stack
    stack<int> s;

    // Initialize current root as minimum possible
    // value
    int root = INT_MIN;

    // Traverse given array
    for (int i=0; i<n; i++)
    {
        // If we find a node who is on right side
        // and smaller than root, return false
        if (pre[i] < root)
            return false;

        // If pre[i] is in right subtree of stack top,
        // Keep removing items smaller than pre[i]
        // and make the last removed item as new
        // root.
        while (!s.empty() && s.top()<pre[i])
        {
            root = s.top();
            s.pop();
        }

        // At this point either stack is empty or
        // pre[i] is smaller than root, push pre[i]
        s.push(pre[i]);
    }
    return true;
}

// Driver program
int main()
{
    int pre1[] = {40, 30, 35, 80, 100};
    int n = sizeof(pre1)/sizeof(pre1[0]);
    canRepresentBST(pre1, n)? cout << "true": cout << "false";
}
```

```
int pre2[] = {40, 30, 35, 20, 80, 100};  
n = sizeof(pre2)/sizeof(pre2[0]);  
canRepresentBST(pre2, n)? cout << "truen":  
                           cout << "falsen";  
  
    return 0;  
}
```

**Java**

```
// Java program for an efficient solution to check if  
// a given array can represent Preorder traversal of  
// a Binary Search Tree  
import java.util.Stack;  
  
class BinarySearchTree {  
  
    boolean canRepresentBST(int pre[], int n) {  
        // Create an empty stack  
        Stack<Integer> s = new Stack<Integer>();  
  
        // Initialize current root as minimum possible  
        // value  
        int root = Integer.MIN_VALUE;  
  
        // Traverse given array  
        for (int i = 0; i < n; i++) {  
            // If we find a node who is on right side  
            // and smaller than root, return false  
            if (pre[i] < root) {  
                return false;  
            }  
  
            // If pre[i] is in right subtree of stack top,  
            // Keep removing items smaller than pre[i]  
            // and make the last removed item as new  
            // root.  
            while (!s.empty() && s.peek() < pre[i]) {  
                root = s.peek();  
                s.pop();  
            }  
  
            // At this point either stack is empty or  
            // pre[i] is smaller than root, push pre[i]  
            s.push(pre[i]);  
        }  
        return true;  
    }  
}
```

```
public static void main(String args[]) {  
    BinarySearchTree bst = new BinarySearchTree();  
    int[] pre1 = new int[]{40, 30, 35, 80, 100};  
    int n = pre1.length;  
    if (bst.canRepresentBST(pre1, n) == true) {  
        System.out.println("true");  
    } else {  
        System.out.println("false");  
    }  
    int[] pre2 = new int[]{40, 30, 35, 20, 80, 100};  
    int n1 = pre2.length;  
    if (bst.canRepresentBST(pre2, n) == true) {  
        System.out.println("true");  
    } else {  
        System.out.println("false");  
    }  
}  
}  
  
//This code is contributed by Mayank Jaiswal
```

### Python

```
# Python program for an efficient solution to check if  
# a given array can represent Preorder traversal of  
# a Binary Search Tree  
  
INT_MIN = -2**32  
  
def canRepresentBST(pre):  
  
    # Create an empty stack  
    s = []  
  
    # Initialize current root as minimum possible value  
    root = INT_MIN  
  
    # Traverse given array  
    for value in pre:  
        #NOTE:value is equal to pre[i] according to the  
        #given algo  
  
        # If we find a node who is on the right side  
        # and smaller than root, return False  
        if value < root :  
            return False
```

```
# If value(pre[i]) is in right subtree of stack top,  
# Keep removing items smaller than value  
# and make the last removed items as new root  
while(len(s) > 0 and s[-1] < value) :  
    root = s.pop()  
  
    # At this point either stack is empty or value  
    # is smaller than root, push value  
    s.append(value)  
  
return True  
  
# Driver Program  
pre1 = [40 , 30 , 35 , 80 , 100]  
print "true" if canRepresentBST(pre1) == True else "false"  
pre2 = [40 , 30 , 35 , 20 , 80 , 100]  
print "true" if canRepresentBST(pre2) == True else "false"  
  
# This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

Output:

```
true  
false
```

This article is contributed by **Romil Punetha**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## Source

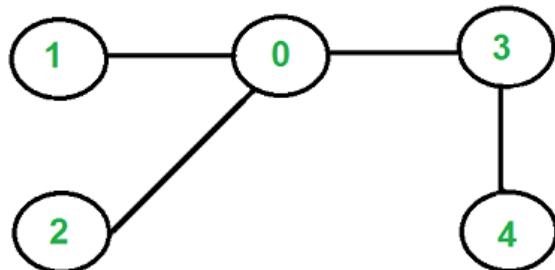
<https://www.geeksforgeeks.org/check-if-a-given-array-can-represent-preorder-traversal-of-binary-search-tree/>

## Chapter 39

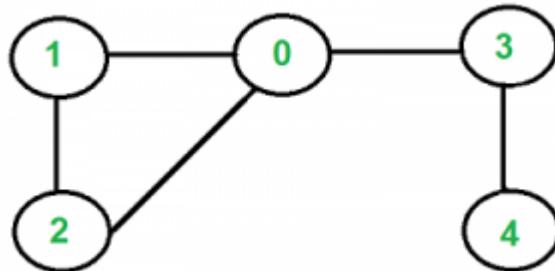
# Check if a given graph is tree or not

Check if a given graph is tree or not - GeeksforGeeks

Write a function that returns true if a given undirected graph is tree and false otherwise.  
For example, the following graph is a tree.



But the following graph is not a tree.



An undirected graph is tree if it has following properties.

- 1) There is no cycle.
- 2) The graph is connected.

For an undirected graph we can either use **BFS** or **DFS** to detect above two properties.

#### How to detect cycle in an undirected graph?

We can either use BFS or DFS. For every visited vertex ‘v’, if there is an adjacent ‘u’ such that u is already visited and u is not parent of v, then there is a cycle in graph. If we don’t find such an adjacent for any vertex, we say that there is no cycle (See [Detect cycle in an undirected graph](#) for more details).

#### How to check for connectivity?

Since the graph is undirected, we can start BFS or DFS from any vertex and check if all vertices are reachable or not. If all vertices are reachable, then graph is connected, otherwise not.

C++

```
// A C++ Program to check whether a graph is tree or not
#include<iostream>
#include <list>
#include <limits.h>
using namespace std;

// Class for an undirected graph
class Graph
{
    int V;      // No. of vertices
    list<int> *adj; // Pointer to an array for adjacency lists
    bool isCyclicUtil(int v, bool visited[], int parent);
public:
    Graph(int V);    // Constructor
    void addEdge(int v, int w);    // to add an edge to graph
    bool isTree();    // returns true if graph is tree
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w); // Add w to v's list.
    adj[w].push_back(v); // Add v to w's list.
}

// A recursive function that uses visited[] and parent to
// detect cycle in subgraph reachable from vertex v.
bool Graph::isCyclicUtil(int v, bool visited[], int parent)
{
```

```

// Mark the current node as visited
visited[v] = true;

// Recur for all the vertices adjacent to this vertex
list<int>::iterator i;
for (i = adj[v].begin(); i != adj[v].end(); ++i)
{
    // If an adjacent is not visited, then recur for
    // that adjacent
    if (!visited[*i])
    {
        if (isCyclicUtil(*i, visited, v))
            return true;
    }

    // If an adjacent is visited and not parent of current
    // vertex, then there is a cycle.
    else if (*i != parent)
        return true;
}
return false;
}

// Returns true if the graph is a tree, else false.
bool Graph::isTree()
{
    // Mark all the vertices as not visited and not part of
    // recursion stack
    bool *visited = new bool[V];
    for (int i = 0; i < V; i++)
        visited[i] = false;

    // The call to isCyclicUtil serves multiple purposes.
    // It returns true if graph reachable from vertex 0
    // is cyclic. It also marks all vertices reachable
    // from 0.
    if (isCyclicUtil(0, visited, -1))
        return false;

    // If we find a vertex which is not reachable from 0
    // (not marked by isCyclicUtil()), then we return false
    for (int u = 0; u < V; u++)
        if (!visited[u])
            return false;

    return true;
}

```

```
// Driver program to test above functions
int main()
{
    Graph g1(5);
    g1.addEdge(1, 0);
    g1.addEdge(0, 2);
    g1.addEdge(0, 3);
    g1.addEdge(3, 4);
    g1.isTree()? cout << "Graph is Tree\n":
                  cout << "Graph is not Tree\n";

    Graph g2(5);
    g2.addEdge(1, 0);
    g2.addEdge(0, 2);
    g2.addEdge(2, 1);
    g2.addEdge(0, 3);
    g2.addEdge(3, 4);
    g2.isTree()? cout << "Graph is Tree\n":
                  cout << "Graph is not Tree\n";

    return 0;
}
```

### Java

```
// A Java Program to check whether a graph is tree or not
import java.io.*;
import java.util.*;

// This class represents a directed graph using adjacency
// list representation
class Graph
{
    private int V;    // No. of vertices
    private LinkedList<Integer> adj[]; //Adjacency List

    // Constructor
    Graph(int v)
    {
        V = v;
        adj = new LinkedList[v];
        for (int i=0; i<v; ++i)
            adj[i] = new LinkedList();
    }

    // Function to add an edge into the graph
    void addEdge(int v,int w)
    {
```

```
        adj[v].add(w);
        adj[w].add(v);
    }

// A recursive function that uses visited[] and parent
// to detect cycle in subgraph reachable from vertex v.
Boolean isCyclicUtil(int v, Boolean visited[], int parent)
{
    // Mark the current node as visited
    visited[v] = true;
    Integer i;

    // Recur for all the vertices adjacent to this vertex
    Iterator<Integer> it = adj[v].iterator();
    while (it.hasNext())
    {
        i = it.next();

        // If an adjacent is not visited, then recur for
        // that adjacent
        if (!visited[i])
        {
            if (isCyclicUtil(i, visited, v))
                return true;
        }

        // If an adjacent is visited and not parent of
        // current vertex, then there is a cycle.
        else if (i != parent)
            return true;
    }
    return false;
}

// Returns true if the graph is a tree, else false.
Boolean isTree()
{
    // Mark all the vertices as not visited and not part
    // of recursion stack
    Boolean visited[] = new Boolean[V];
    for (int i = 0; i < V; i++)
        visited[i] = false;

    // The call to isCyclicUtil serves multiple purposes
    // It returns true if graph reachable from vertex 0
    // is cyclic. It also marks all vertices reachable
    // from 0.
    if (isCyclicUtil(0, visited, -1))
```

```
        return false;

        // If we find a vertex which is not reachable from 0
        // (not marked by isCyclicUtil(), then we return false
        for (int u = 0; u < V; u++)
            if (!visited[u])
                return false;

        return true;
    }

    // Driver method
    public static void main(String args[])
    {
        // Create a graph given in the above diagram
        Graph g1 = new Graph(5);
        g1.addEdge(1, 0);
        g1.addEdge(0, 2);
        g1.addEdge(0, 3);
        g1.addEdge(3, 4);
        if (g1.isTree())
            System.out.println("Graph is Tree");
        else
            System.out.println("Graph is not Tree");

        Graph g2 = new Graph(5);
        g2.addEdge(1, 0);
        g2.addEdge(0, 2);
        g2.addEdge(2, 1);
        g2.addEdge(0, 3);
        g2.addEdge(3, 4);

        if (g2.isTree())
            System.out.println("Graph is Tree");
        else
            System.out.println("Graph is not Tree");

    }
}
// This code is contributed by Aakash Hasija
```

### Python

```
# Python Program to check whether
# a graph is tree or not

from collections import defaultdict
```

```
class Graph():

    def __init__(self, V):
        self.V = V
        self.graph = defaultdict(list)

    def addEdge(self, v, w):
        # Add w to v list.
        self.graph[v].append(w)
        # Add v to w list.
        self.graph[w].append(v)

    # A recursive function that uses visited[]
    # and parent to detect cycle in subgraph
    # reachable from vertex v.
    def isCyclicUtil(self, v, visited, parent):

        # Mark current node as visited
        visited[v] = True

        # Recur for all the vertices adjacent
        # for this vertex
        for i in self.graph[v]:
            # If an adjacent is not visited,
            # then recur for that adjacent
            if visited[i] == False:
                if self.isCyclicUtil(i, visited, v) == True:
                    return True

            # If an adjacent is visited and not
            # parent of current vertex, then there
            # is a cycle.
            elif i != parent:
                return True

        return False

    # Returns true if the graph is a tree,
    # else false.
    def isTree(self):
        # Mark all the vertices as not visited
        # and not part of recursion stack
        visited = [False] * self.V

        # The call to isCyclicUtil serves multiple
        # purposes. It returns true if graph reachable
        # from vertex 0 is cyclcic. It also marks
        # all vertices reachable from 0.
```

```
if self.isCyclicUtil(0, visited, -1) == True:
    return False

# If we find a vertex which is not reachable
# from 0 (not marked by isCyclicUtil()),
# then we return false
for i in range(self.V):
    if visited[i] == False:
        return False

return True

# Driver program to test above functions
g1 = Graph(5)
g1.addEdge(1, 0)
g1.addEdge(0, 2)
g1.addEdge(0, 3)
g1.addEdge(3, 4)
print "Graph is a Tree" if g1.isTree() == True \
else "Graph is a not a Tree"

g2 = Graph(5)
g2.addEdge(1, 0)
g2.addEdge(0, 2)
g2.addEdge(2, 1)
g2.addEdge(0, 3)
g2.addEdge(3, 4)
print "Graph is a Tree" if g2.isTree() == True \
else "Graph is a not a Tree"

# This code is contributed by Divyanshu Mehta
```

Output:

```
Graph is Tree
Graph is not Tree
```

Thanks to **Vinit Verma** for suggesting this problem and initial solution. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## Source

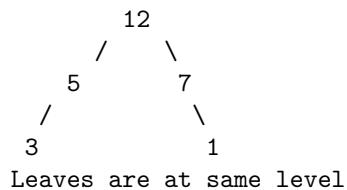
<https://www.geeksforgeeks.org/check-given-graph-tree/>

## Chapter 40

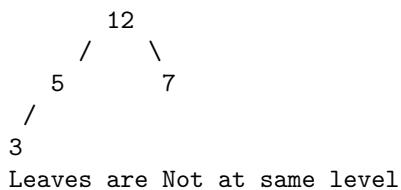
### Check if all leaves are at same level

Check if all leaves are at same level - GeeksforGeeks

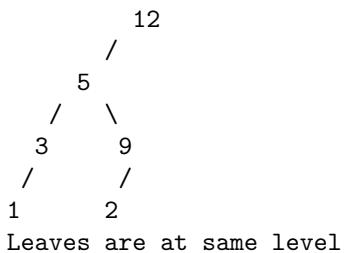
Given a Binary Tree, check if all leaves are at same level or not.



Leaves are at same level



Leaves are Not at same level



Leaves are at same level

**Method 1 (Recursive)**

The idea is to first find level of the leftmost leaf and store it in a variable leafLevel. Then compare level of all other leaves with leafLevel, if same, return true, else return false. We traverse the given Binary Tree in Preorder fashion. An argument leaflevel is passed to all calls. The value of leafLevel is initialized as 0 to indicate that the first leaf is not yet seen yet. The value is updated when we find first leaf. Level of subsequent leaves (in preorder) is compared with leafLevel.

C

```
// C program to check if all leaves are at same level
#include <stdio.h>
#include <stdlib.h>

// A binary tree node
struct Node
{
    int data;
    struct Node *left, *right;
};

// A utility function to allocate a new tree node
struct Node* newNode(int data)
{
    struct Node* node = (struct Node*) malloc(sizeof(struct Node));
    node->data = data;
    node->left = node->right = NULL;
    return node;
}

/* Recursive function which checks whether all leaves are at same level */
bool checkUtil(struct Node *root, int level, int *leafLevel)
{
    // Base case
    if (root == NULL)  return true;

    // If a leaf node is encountered
    if (root->left == NULL && root->right == NULL)
    {
        // When a leaf node is found first time
        if (*leafLevel == 0)
        {
            *leafLevel = level; // Set first found leaf's level
            return true;
        }

        // If this is not first leaf node, compare its level with
        // first leaf's level
        return (level == *leafLevel);
    }
}
```

```
// If this node is not leaf, recursively check left and right subtrees
return checkUtil(root->left, level+1, leafLevel) &&
       checkUtil(root->right, level+1, leafLevel);
}

/* The main function to check if all leafs are at same level.
   It mainly uses checkUtil() */
bool check(struct Node *root)
{
    int level = 0, leafLevel = 0;
    return checkUtil(root, level, &leafLevel);
}

// Driver program to test above function
int main()
{
    // Let us create tree shown in thirdt example
    struct Node *root = newNode(12);
    root->left = newNode(5);
    root->left->left = newNode(3);
    root->left->right = newNode(9);
    root->left->left->left = newNode(1);
    root->left->right->left = newNode(1);
    if (check(root))
        printf("Leaves are at same level\n");
    else
        printf("Leaves are not at same level\n");
    getchar();
    return 0;
}
```

### Java

```
// Java program to check if all leaves are at same level

// A binary tree node
class Node
{
    int data;
    Node left, right;

    Node(int item)
    {
        data = item;
        left = right = null;
    }
}
```

```
class Leaf
{
    int leaflevel=0;
}

class BinaryTree
{
    Node root;
    Leaf mylevel = new Leaf();

    /* Recursive function which checks whether all leaves are at same
       level */
    boolean checkUtil(Node node, int level, Leaf leafLevel)
    {
        // Base case
        if (node == null)
            return true;

        // If a leaf node is encountered
        if (node.left == null && node.right == null)
        {
            // When a leaf node is found first time
            if (leafLevel.leaflevel == 0)
            {
                // Set first found leaf's level
                leafLevel.leaflevel = level;
                return true;
            }

            // If this is not first leaf node, compare its level with
            // first leaf's level
            return (level == leafLevel.leaflevel);
        }

        // If this node is not leaf, recursively check left and right
        // subtrees
        return checkUtil(node.left, level + 1, leafLevel)
            && checkUtil(node.right, level + 1, leafLevel);
    }

    /* The main function to check if all leafs are at same level.
       It mainly uses checkUtil() */
    boolean check(Node node)
    {
        int level = 0;
        return checkUtil(node, level, mylevel);
    }
}
```

```
public static void main(String args[])
{
    // Let us create the tree as shown in the example
    BinaryTree tree = new BinaryTree();
    tree.root = new Node(12);
    tree.root.left = new Node(5);
    tree.root.left.left = new Node(3);
    tree.root.left.right = new Node(9);
    tree.root.left.left.left = new Node(1);
    tree.root.left.right.left = new Node(1);
    if (tree.check(tree.root))
        System.out.println("Leaves are at same level");
    else
        System.out.println("Leaves are not at same level");
}
}

// This code has been contributed by Mayank Jaiswal
```

### Python

```
# Python program to check if all leaves are at same level

# A binary tree node
class Node:

    # Constructor to create a new node
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

    # Recursive function which check whether all leaves are at
    # same level
    def checkUtil(root, level):

        # Base Case
        if root is None:
            return True

        # If a tree node is encountered
        if root.left is None and root.right is None:

            # When a leaf node is found first time
            if check.leafLevel == 0 :
                check.leafLevel = level # Set first leaf found
                return True
```

```
# If this is not first leaf node, compare its level
# with first leaf's level
return level == check.leafLevel

# If this is not first leaf node, compare its level
# with first leaf's level
return (checkUtil(root.left, level+1)and
       checkUtil(root.right, level+1))

def check(root):
    level = 0
    check.leafLevel = 0
    return (checkUtil(root, level))

# Driver program to test above function
root = Node(12)
root.left = Node(5)
root.left.left = Node(3)
root.left.right = Node(9)
root.left.left.left = Node(1)
root.left.right.left = Node(2)

if(check(root)):
    print "Leaves are at same level"
else:
    print "Leaves are not at same level"

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

Output:

```
Leaves are at same level
```

Time Complexity: The function does a simple traversal of the tree, so the complexity is  $O(n)$ .

### Method 1 (Iterative)

It can also be solved by iterative approach.

The idea is to iteratively traverse the tree, and when you encounter the first leaf node, store it's level in result variable, now whenever you encounter any leaf node, compare it's level with previously stored result, if they are same then proceed for rest of the tree, else return false.

```
// CPP program to check if all leaf nodes are at
```

```
// same level of binary tree
#include <bits/stdc++.h>
using namespace std;

// tree node
struct Node {
    int data;
    Node *left, *right;
};

// returns a new tree Node
Node* newNode(int data)
{
    Node* temp = new Node();
    temp->data = data;
    temp->left = temp->right = NULL;
    return temp;
}

// return true if all leaf nodes are
// at same level, else false
int checkLevelLeafNode(Node* root)
{
    if (!root)
        return 1;

    // create a queue for level order traversal
    queue<Node*> q;
    q.push(root);

    int result = INT_MAX;
    int level = 0;

    // traverse until the queue is empty
    while (!q.empty()) {
        int size = q.size();
        level += 1;

        // traverse for complete level
        while(size > 0){
            Node* temp = q.front();
            q.pop();

            // check for left child
            if (temp->left) {
                q.push(temp->left);
            }

            // if its leaf node
        }
    }

    // if all leaf nodes are at same level
    if (result == level)
        return 1;
    else
        return 0;
}
```

```
if(!temp->left->right && !temp->left->left){

    // if it's first leaf node, then update result
    if (result == INT_MAX)
        result = level;

    // if it's not first leaf node, then compare
    // the level with level of previous leaf node
    else if (result != level)
        return 0;
}
}

// check for right child
if (temp->right){
    q.push(temp->right);

    // if it's leaf node
    if (!temp->right->left && !temp->right->right)

        // if it's first leaf node till now,
        // then update the result
        if (result == INT_MAX)
            result = level;

        // if it is not the first leaf node,
        // then compare the level with level
        // of previous leaf node
        else if(result != level)
            return 0;

    }
    size -= 1;
}
}

return 1;
}

// driver program
int main()
{
    // construct a tree
    Node* root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->right = newNode(4);
    root->right->left = newNode(5);
```

```
root->right->right = newNode(6);

int result = checkLevelLeafNode(root);
if (result)
    cout << "All leaf nodes are at same level\n";
else
    cout << "Leaf nodes not at same level\n";
return 0;
}
```

Output:

```
All leaf nodes are at same level
```

Time Complexity : O(n)

This code is contributed by – [Mandeep Singh](#)

This article is contributed by [Chandra Prakash](#). Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Improved By : [msdeep14](#)

## Source

<https://www.geeksforgeeks.org/check-leaves-level/>

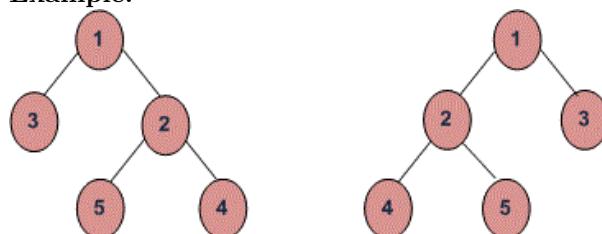
## Chapter 41

# Check if all levels of two trees are anagrams or not

Check if all levels of two trees are anagrams or not - GeeksforGeeks

Given two binary trees, we have to check if each of their levels are anagrams of each other or not.

**Example:**



**Tree 1:**

Level 0 : 1  
Level 1 : 3, 2  
Level 2 : 5, 4

**Tree 2:**

Level 0 : 1  
Level 1 : 2, 3  
Level 2 : 4, 5

As we can clearly see all the levels of above two binary trees are anagrams of each other, hence return true.

**Naive Approach:** Below is the step by step explanation of the naive approach to do this:

1. Write a recursive program for level order traversal of a tree.
2. Traverse each level of both the trees one by one and store the result of traversals in 2 different vectors, one for each tree.
3. Sort both the vectors and compare them iteratively for each level, if they are same for each level then return true else return false.

**Time Complexity:**  $O(n^2)$ , where n is the number of nodes.

**Efficient Approach:**

The idea is based on below article.

[Print level order traversal line by line | Set 1](#)

We traverse both trees simultaneously level by level. We store each level both trees in vectors (or array). To check if two vectors are anagram or not, we sort both and then compare.

**Time Complexity:**  $O(n)$ , where n is the number of nodes.

**C++**

```
/* Iterative program to check if two trees are level
   by level anagram. */
#include <bits/stdc++.h>
using namespace std;

// A Binary Tree Node
struct Node
{
    struct Node *left, *right;
    int data;
};

// Returns true if trees with root1 and root2
// are level by level anagram, else returns false.
bool areAnagrams(Node *root1, Node *root2)
{
    // Base Cases
    if (root1 == NULL && root2 == NULL)
        return true;
    if (root1 == NULL || root2 == NULL)
        return false;

    // start level order traversal of two trees
    // using two queues.
    queue<Node *> q1, q2;
    q1.push(root1);
    q2.push(root2);

    while (1)
    {
```

```
// n1 (queue size) indicates number of Nodes
// at current level in first tree and n2 indicates
// number of nodes in current level of second tree.
int n1 = q1.size(), n2 = q2.size();

// If n1 and n2 are different
if (n1 != n2)
    return false;

// If level order traversal is over
if (n1 == 0)
    break;

// Dequeue all Nodes of current level and
// Enqueue all Nodes of next level
vector<int> curr_level1, curr_level2;
while (n1 > 0)
{
    Node *node1 = q1.front();
    q1.pop();
    if (node1->left != NULL)
        q1.push(node1->left);
    if (node1->right != NULL)
        q1.push(node1->right);
    n1--;

    Node *node2 = q2.front();
    q2.pop();
    if (node2->left != NULL)
        q2.push(node2->left);
    if (node2->right != NULL)
        q2.push(node2->right);

    curr_level1.push_back(node1->data);
    curr_level2.push_back(node2->data);
}

// Check if nodes of current levels are
// anagrams or not.
sort(curr_level1.begin(), curr_level1.end());
sort(curr_level2.begin(), curr_level2.end());
if (curr_level1 != curr_level2)
    return false;
}

return true;
}
```

```
// Utility function to create a new tree Node
Node* newNode(int data)
{
    Node *temp = new Node;
    temp->data = data;
    temp->left = temp->right = NULL;
    return temp;
}

// Driver program to test above functions
int main()
{
    // Constructing both the trees.
    struct Node* root1 = newNode(1);
    root1->left = newNode(3);
    root1->right = newNode(2);
    root1->right->left = newNode(5);
    root1->right->right = newNode(4);

    struct Node* root2 = newNode(1);
    root2->left = newNode(2);
    root2->right = newNode(3);
    root2->left->left = newNode(4);
    root2->left->right = newNode(5);

    areAnagrams(root1, root2)? cout << "Yes" : cout << "No";
    return 0;
}
```

### Java

```
/* Iterative program to check if two trees
are level by level anagram. */
import java.util.ArrayList;
import java.util.Collections;
import java.util.LinkedList;
import java.util.Queue;

public class GFG
{
    // A Binary Tree Node
    static class Node
    {
        Node left, right;
        int data;
        Node(int data){
            this.data = data;
```

```
        left = null;
        right = null;
    }
}

// Returns true if trees with root1 and root2
// are level by level anagram, else returns false.
static boolean areAnagrams(Node root1, Node root2)
{
    // Base Cases
    if (root1 == null && root2 == null)
        return true;
    if (root1 == null || root2 == null)
        return false;

    // start level order traversal of two trees
    // using two queues.
    Queue<Node> q1 = new LinkedList<Node>();
    Queue<Node> q2 = new LinkedList<Node>();
    q1.add(root1);
    q2.add(root2);

    while (true)
    {
        // n1 (queue size) indicates number of
        // Nodes at current level in first tree
        // and n2 indicates number of nodes in
        // current level of second tree.
        int n1 = q1.size(), n2 = q2.size();

        // If n1 and n2 are different
        if (n1 != n2)
            return false;

        // If level order traversal is over
        if (n1 == 0)
            break;

        // Dequeue all Nodes of current level and
        // Enqueue all Nodes of next level
        ArrayList<Integer> curr_level1 = new
                                         ArrayList<>();
        ArrayList<Integer> curr_level2 = new
                                         ArrayList<>();
        while (n1 > 0)
        {
            Node node1 = q1.peek();
            q1.remove();
            if (node1.left != null)
                q2.add(node1.left);
            if (node1.right != null)
                q2.add(node1.right);
        }
    }
}
```

```
        if (node1.left != null)
            q1.add(node1.left);
        if (node1.right != null)
            q1.add(node1.right);
        n1--;

        Node node2 = q2.peek();
        q2.remove();
        if (node2.left != null)
            q2.add(node2.left);
        if (node2.right != null)
            q2.add(node2.right);

        curr_level1.add(node1.data);
        curr_level2.add(node2.data);
    }

    // Check if nodes of current levels are
    // anagrams or not.
    Collections.sort(curr_level1);
    Collections.sort(curr_level2);

    if (!curr_level1.equals(curr_level2))
        return false;
}

return true;
}

// Driver program to test above functions
public static void main(String args[])
{
    // Constructing both the trees.
    Node root1 = new Node(1);
    root1.left = new Node(3);
    root1.right = new Node(2);
    root1.right.left = new Node(5);
    root1.right.right = new Node(4);

    Node root2 = new Node(1);
    root2.left = new Node(2);
    root2.right = new Node(3);
    root2.left.left = new Node(4);
    root2.left.right = new Node(5);

    System.out.println(areAnagrams(root1, root2)?
        "Yes" : "No");
}
```

```
    }  
}  
// This code is contributed by Sumit Ghosh
```

Output:

**Yes**

**Note:** In the above program we are comparing the vectors storing each level of a tree directly using not equal to function ‘ != ‘ which compares the vectors first on the basis of their size and then on the basis of their content, hence saving our work of iteratively comparing the vectors.

### Source

<https://www.geeksforgeeks.org/check-if-all-levels-of-two-trees-are-anagrams-or-not/>

## Chapter 42

# Check if an array represents Inorder of Binary Search tree or not

Check if an array represents Inorder of Binary Search tree or not - GeeksforGeeks

Given an array of N element. The task is to check if it is Inorder traversal of any Binary Search Tree or not. Print “Yes” if it is Inorder traversal of any Binary Search Tree else print “No”.

Examples:

Input : arr[] = { 19, 23, 25, 30, 45 }  
Output : Yes

Input : arr[] = { 19, 23, 30, 25, 45 }  
Output : No

The idea is to use the fact that the inorder traversal of Binary Search Tree is sorted. So, just check if given array is sorted or not.

```
// C++ program to check if a given array is sorted
// or not.
#include<bits/stdc++.h>
using namespace std;

// Function that returns true if array is Inorder
// traversal of any Binary Search Tree or not.
bool isInorder(int arr[], int n)
{
```

```
// Array has one or no element
if (n == 0 || n == 1)
    return true;

for (int i = 1; i < n; i++)

    // Unsorted pair found
    if (arr[i-1] > arr[i])
        return false;

    // No unsorted pair found
    return true;
}

// Driver code
int main()
{
    int arr[] = { 19, 23, 25, 30, 45 };
    int n = sizeof(arr)/sizeof(arr[0]);

    if (isInorder(arr, n))
        cout << "Yes";
    else
        cout << "No";

    return 0;
}
```

Output:

Yes

## Source

<https://www.geeksforgeeks.org/check-array-represents-inorder-binary-search-tree-not/>

## Chapter 43

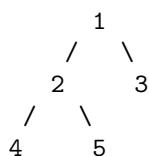
# Check if given Preorder, Inorder and Postorder traversals are of same tree

Check if given Preorder, Inorder and Postorder traversals are of same tree - GeeksforGeeks

Given [Preorder](#) , [Inorder](#) and [Postorder](#) traversals of some tree. Write a program to check if they all are of the same tree.

Examples:

```
Input : Inorder -> 4 2 5 1 3
        Preorder -> 1 2 4 5 3
        Postorder -> 4 5 2 3 1
Output : Yes
Explanation : All of the above three traversals are of
the same tree
```



```
Input : Inorder -> 4 2 5 1 3
        Preorder -> 1 5 4 2 3
        Postorder -> 4 1 2 3 5
Output : No
```

The most basic approach to solve this problem will be to first construct a tree using two of the three given traversals and then do the third traversal on this constructed tree and compare it with the given traversal. If both of the traversals are same then print Yes otherwise print

No. Here, we use Inorder and Preorder traversals to construct the tree. We may also use Inorder and Postorder traversal instead of Preorder traversal for tree construction. You may refer to [this](#) post on how to construct tree from given Inorder and Preorder traversal. After constructing the tree, we will obtain the Postorder traversal of this tree and compare it with the given Postorder traversal.

Below is the C++ implementation of above approach:

```
/* C++ program to check if all three given
   traversals are of the same tree */
#include <bits/stdc++.h>
using namespace std;

// A Binary Tree Node
struct Node
{
    int data;
    struct Node *left, *right;
};

// Utility function to create a new tree node
Node* newNode(int data)
{
    Node *temp = new Node;
    temp->data = data;
    temp->left = temp->right = NULL;
    return temp;
}

/* Function to find index of value in arr[start...end]
   The function assumes that value is present in in[] */
int search(int arr[], int strt, int end, int value)
{
    for (int i = strt; i <= end; i++)
    {
        if(arr[i] == value)
            return i;
    }
}

/* Recursive function to construct binary tree
   of size len from Inorder traversal in[] and
   Preorder traversal pre[]. Initial values
   of inStrt and inEnd should be 0 and len -1.
   The function doesn't do any error checking for
   cases where inorder and preorder do not form a
   tree */
Node* buildTree(int in[], int pre[], int inStrt,
                int inEnd)
```

```
{  
    static int preIndex = 0;  
  
    if(inStrt > inEnd)  
        return NULL;  
  
    /* Pick current node from Preorder traversal  
       using preIndex and increment preIndex */  
    Node *tNode = newNode(pre[preIndex++]);  
  
    /* If this node has no children then return */  
    if (inStrt == inEnd)  
        return tNode;  
  
    /* Else find the index of this node in  
       Inorder traversal */  
    int inIndex = search(in, inStrt, inEnd, tNode->data);  
  
    /* Using index in Inorder traversal,  
       construct left and right subtress */  
    tNode->left = buildTree(in, pre, inStrt, inIndex-1);  
    tNode->right = buildTree(in, pre, inIndex+1, inEnd);  
  
    return tNode;  
}  
  
/* function to compare Postorder traversal  
   on constructed tree and given Postorder */  
int checkPostorder(Node* node, int postOrder[], int index)  
{  
    if (node == NULL)  
        return index;  
  
    /* first recur on left child */  
    index = checkPostorder(node->left,postOrder,index);  
  
    /* now recur on right child */  
    index = checkPostorder(node->right,postOrder,index);  
  
    /* Compare if data at current index in  
       both Postorder traversals are same */  
    if (node->data == postOrder[index])  
        index++;  
    else  
        return -1;  
  
    return index;  
}
```

```
// Driver program to test above functions
int main()
{
    int inOrder[] = {4, 2, 5, 1, 3};
    int preOrder[] = {1, 2, 4, 5, 3};
    int postOrder[] = {4, 5, 2, 3, 1};

    int len = sizeof(inOrder)/sizeof(inOrder[0]);

    // build tree from given
    // Inorder and Preorder traversals
    Node *root = buildTree(inOrder, preOrder, 0, len - 1);

    // compare postorder traversal on constructed
    // tree with given Postorder traversal
    int index = checkPostorder(root,postOrder,0);

    // If both postorder traversals are same
    if (index == len)
        cout << "Yes";
    else
        cout << "No";

    return 0;
}
```

Output:

Yes

**Time Complexity :**  $O(n * n)$ , where n is number of nodes in the tree.

## Source

<https://www.geeksforgeeks.org/check-if-given-preorder-inorder-and-postorder-traversals-are-of-same-tree/>

## Chapter 44

# Check if leaf traversal of two Binary Trees is same?

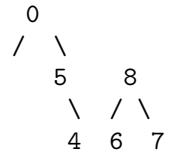
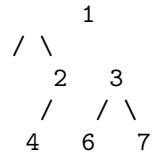
Check if leaf traversal of two Binary Trees is same? - GeeksforGeeks

Leaf traversal is sequence of leaves traversed from left to right. The problem is to check if leaf traversals of two given Binary Trees are same or not.

Expected time complexity  $O(n)$ . Expected auxiliary space  $O(h_1 + h_2)$  where  $h_1$  and  $h_2$  are heights of two Binary Trees.

Examples:

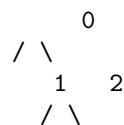
Input: Roots of below Binary Trees

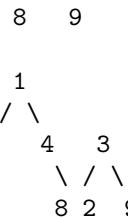


Output: same

Leaf order traversal of both trees is 4 6 7

Input: Roots of below Binary Trees





Output: Not Same  
Leaf traversals of two trees are different.  
For first, it is 8 9 2 and for second it is  
8 2 9

We strongly recommend you to minimize your browser and try this yourself first.

A **Simple Solution** is traverse first tree and store leaves from left and right in an array. Then traverse other tree and store leaves in another array. Finally compare two arrays. If both arrays are same, then return true.

The above solution requires  $O(m+n)$  extra space where m and n are nodes in first and second tree respectively.

**How to check with  $O(h_1 + h_2)$  space?**

The idea is use iterative traversal. Traverse both trees simultaneously, look for a leaf node in both trees and compare the found leaves. All leaves must match.

**Algorithm:**

1. Create empty stacks stack1 and stack2  
for iterative traversals of tree1 and tree2
2. insert (root of tree1) in stack1  
insert (root of tree2) in stack2
3. Stores current leaf nodes of tree1 and tree2  
temp1 = (root of tree1)  
temp2 = (root of tree2)
4. Traverse both trees using stacks  
while (stack1 and stack2 parent empty)  
{  
    // Means excess leaves in one tree  
    if (if one of the stacks are empty)  
        return false  
  
    // get next leaf node in tree1  
    temp1 = stack1.pop()  
    while (temp1 is not leaf node)

```
{  
    push right child to stack1  
    push left child to stack1  
}  
  
// get next leaf node in tree2  
temp2 = stack2.pop()  
while (temp2 is not leaf node)  
{  
    push right child to stack2  
    push left child to stack2  
}  
  
// If leaves do not match return false  
if (temp1 != temp2)  
    return false  
}  
  
5. If all leaves matched, return true
```

Below is Java implementation of above algorithm.

```
// Java program to check if two Leaf Traversal of  
// Two Binary Trees is same or not  
import java.util.*;  
import java.lang.*;  
import java.io.*;  
  
// Binary Tree node  
class Node  
{  
    int data;  
    Node left, right;  
    public Node(int x)  
    {  
        data = x;  
        left = right = null;  
    }  
    public boolean isLeaf()  
    {  
        return (left==null && right==null);  
    }  
}  
  
class LeafOrderTraversal  
{  
    // Returns true if leaf traversal of two trees is  
    // same, else false
```

```
public static boolean isSame(Node root1, Node root2)
{
    // Create empty stacks. These stacks are going
    // to be used for iterative traversals.
    Stack<Node> s1 = new Stack<Node>();
    Stack<Node> s2 = new Stack<Node>();

    s1.push(root1);
    s2.push(root2);

    // Loop until either of two stacks is not empty
    while (!s1.empty() || !s2.empty())
    {
        // If one of the stacks is empty means other
        // stack has extra leaves so return false
        if (s1.empty() || s2.empty())
            return false;

        Node temp1 = s1.pop();
        while (temp1!=null && !temp1.isLeaf())
        {
            // Push right and left children of temp1.
            // Note that right child is inserted
            // before left
            if (temp1.right != null)
                s1.push(temp1.right);
            if (temp1.left != null)
                s1.push(temp1.left);
            temp1 = s1.pop();
        }

        // same for tree2
        Node temp2 = s2.pop();
        while (temp2!=null && !temp2.isLeaf())
        {
            if (temp2.right != null)
                s2.push(temp2.right);
            if (temp2.left != null)
                s2.push(temp2.left);
            temp2 = s2.pop();
        }

        // If one is null and other is not, then
        // return false
        if (temp1==null && temp2!=null)
            return false;
        if (temp1!=null && temp2==null)
            return false;
    }
}
```

```
// If both are not null and data is not
// same return false
if (temp1!=null && temp2!=null)
{
    if (temp1.data != temp2.data)
        return false;
}
}

// If control reaches this point, all leaves
// are matched
return true;
}

// Driver program to test
public static void main(String[] args)
{
    // Let us create trees in above example 1
    Node root1 = new Node(1);
    root1.left = new Node(2);
    root1.right = new Node(3);
    root1.left.left = new Node(4);
    root1.right.left = new Node(6);
    root1.right.right = new Node(7);

    Node root2 = new Node(0);
    root2.left = new Node(1);
    root2.right = new Node(5);
    root2.left.right = new Node(4);
    root2.right.left = new Node(6);
    root2.right.right = new Node(7);

    if (isSame(root1, root2))
        System.out.println("Same");
    else
        System.out.println("Not Same");
}
}
```

Output:

Same

This article is contributed by **Kumar Gaurav**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

**Source**

<https://www.geeksforgeeks.org/check-if-leaf-traversal-of-two-binary-trees-is-same/>

## Chapter 45

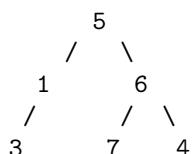
# Check if removing an edge can divide a Binary Tree in two halves

Check if removing an edge can divide a Binary Tree in two halves - GeeksforGeeks

Given a Binary Tree, find if there exist edge whose removal creates two trees of equal size.

Examples:

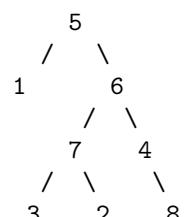
Input : root of following tree



Output : true

Removing edge 5-6 creates two trees of equal size

Input : root of following tree



Output : false

There is no edge whose removal creates two trees of equal size.

Source- Kshitij IIT KGP

We strongly recommend you to minimize your browser and try this yourself first.

### Method 1 (Simple)

First count number of nodes in whole tree. Let count of all nodes be n. Now traverse tree and for every node, find size of subtree rooted with this node. Let the subtree size be s. If n-s is equal to s, then return true, else false.

C++

```
// C++ program to check if there exist an edge whose
// removal creates two trees of same size
#include<bits/stdc++.h>
using namespace std;

struct Node
{
    int data;
    struct Node* left, *right;
};

// utility function to create a new node
struct Node* newNode(int x)
{
    struct Node* temp = new Node;
    temp->data = x;
    temp->left = temp->right = NULL;
    return temp;
}

// To calculate size of tree with given root
int count(Node* root)
{
    if (root==NULL)
        return 0;
    return count(root->left) + count(root->right) + 1;
}

// This function returns true if there is an edge
// whose removal can divide the tree in two halves
// n is size of tree
bool checkRec(Node* root, int n)
{
    // Base cases
    if (root ==NULL)
        return false;
```

```
// Check for root
if (count(root) == n-count(root))
    return true;

// Check for rest of the nodes
return checkRec(root->left, n) ||
       checkRec(root->right, n);
}

// This function mainly uses checkRec()
bool check(Node *root)
{
    // Count total nodes in given tree
    int n = count(root);

    // Now recursively check all nodes
    return checkRec(root, n);
}

// Driver code
int main()
{
    struct Node* root = newNode(5);
    root->left = newNode(1);
    root->right = newNode(6);
    root->left->left = newNode(3);
    root->right->left = newNode(7);
    root->right->right = newNode(4);

    check(root)? printf("YES") : printf("NO");

    return 0;
}
```

### Java

```
// Java program to check if there exist an edge whose
// removal creates two trees of same size

class Node
{
    int key;
    Node left, right;

    public Node(int key)
    {
        this.key = key;
```

```
        left = right = null;
    }
}

class BinaryTree
{
    Node root;

    // To calculate size of tree with given root
    int count(Node node)
    {
        if (node == null)
            return 0;

        return count(node.left) + count(node.right) + 1;
    }

    // This function returns true if there is an edge
    // whose removal can divide the tree in two halves
    // n is size of tree
    boolean checkRec(Node node, int n)
    {
        // Base cases
        if (node == null)
            return false;

        // Check for root
        if (count(node) == n - count(node))
            return true;

        // Check for rest of the nodes
        return checkRec(node.left, n)
               || checkRec(node.right, n);
    }

    // This function mainly uses checkRec()
    boolean check(Node node)
    {
        // Count total nodes in given tree
        int n = count(node);

        // Now recursively check all nodes
        return checkRec(node, n);
    }

    // Driver code
    public static void main(String[] args)
    {
```

```
BinaryTree tree = new BinaryTree();
tree.root = new Node(5);
tree.root.left = new Node(1);
tree.root.right = new Node(6);
tree.root.left.left = new Node(3);
tree.root.right.left = new Node(7);
tree.root.right.right = new Node(4);
if(tree.check(tree.root)==true)
    System.out.println("YES");
else
    System.out.println("NO");
}

}

// This code has been contributed by Mayank Jaiswal(mayank_24)
```

Output :

YES

Time complexity of above solution is  $O(n^2)$  where n is number of nodes in given Binary Tree.

### Method 2 (Efficient)

We can find the solution in  $O(n)$  time. The idea is to traverse tree in bottom up manner and while traversing keep updating size and keep checking if there is a node that follows the required property.

Below is C++ and implementation of above idea.

C++

```
// C++ program to check if there exist an edge whose
// removal creates two trees of same size
#include<bits/stdc++.h>
using namespace std;

struct Node
{
    int data;
    struct Node* left, *right;
};

// utility function to create a new node
struct Node* newNode(int x)
{
    struct Node* temp = new Node;
```

```
temp->data = x;
temp->left = temp->right = NULL;
return temp;
};

// To calculate size of tree with given root
int count(Node* root)
{
    if (root==NULL)
        return 0;
    return count(root->left) + count(root->right) + 1;
}

// This function returns size of tree rooted with given
// root. It also set "res" as true if there is an edge
// whose removal divides tree in two halves.
// n is size of tree
int checkRec(Node* root, int n, bool &res)
{
    // Base case
    if (root == NULL)
        return 0;

    // Compute sizes of left and right children
    int c = checkRec(root->left, n, res) + 1 +
            checkRec(root->right, n, res);

    // If required property is true for current node
    // set "res" as true
    if (c == n-c)
        res = true;

    // Return size
    return c;
}

// This function mainly uses checkRec()
bool check(Node *root)
{
    // Count total nodes in given tree
    int n = count(root);

    // Initialize result and recursively check all nodes
    bool res = false;
    checkRec(root, n, res);

    return res;
}
```

```
// Driver code
int main()
{
    struct Node* root = newNode(5);
    root->left = newNode(1);
    root->right = newNode(6);
    root->left->left = newNode(3);
    root->right->left = newNode(7);
    root->right->right = newNode(4);

    check(root)? printf("YES") : printf("NO");

    return 0;
}
```

**Java**

```
// Java program to check if there exist an edge whose
// removal creates two trees of same size

class Node
{
    int key;
    Node left, right;

    public Node(int key)
    {
        this.key = key;
        left = right = null;
    }
}

class Res
{
    boolean res = false;
}

class BinaryTree
{
    Node root;

    // To calculate size of tree with given root
    int count(Node node)
    {
        if (node == null)
            return 0;
```

```
    return count(node.left) + count(node.right) + 1;
}

// This function returns size of tree rooted with given
// root. It also set "res" as true if there is an edge
// whose removal divides tree in two halves.
// n is size of tree
int checkRec(Node root, int n, Res res)
{
    // Base case
    if (root == null)
        return 0;

    // Compute sizes of left and right children
    int c = checkRec(root.left, n, res) + 1
           + checkRec(root.right, n, res);

    // If required property is true for current node
    // set "res" as true
    if (c == n - c)
        res.res = true;

    // Return size
    return c;
}

// This function mainly uses checkRec()
boolean check(Node root)
{
    // Count total nodes in given tree
    int n = count(root);

    // Initialize result and recursively check all nodes
    Res res = new Res();
    checkRec(root, n, res);

    return res.res;
}

// Driver code
public static void main(String[] args)
{
    BinaryTree tree = new BinaryTree();
    tree.root = new Node(5);
    tree.root.left = new Node(1);
    tree.root.right = new Node(6);
    tree.root.left.left = new Node(3);
    tree.root.right.left = new Node(7);
```

```
tree.root.right.right = new Node(4);
if (tree.check(tree.root) == true)
    System.out.println("YES");
else
    System.out.println("NO");
}
}

// This code has been contributed by Mayank Jaiswal(mayank_24)
```

Output :

YES

This article is contributed by **Asaad Akram**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## Source

<https://www.geeksforgeeks.org/check-if-removing-an-edge-can-divide-a-binary-tree-in-two-halves/>

## Chapter 46

# Check if the given array can represent Level Order Traversal of Binary Search Tree

Check if the given array can represent Level Order Traversal of Binary Search Tree - Geeks-forGeeks

Given an array of size **n**. The problem is to check whether the given array can represent the level order traversal of a Binary Search Tree or not.

Examples:

```
Input : arr[] = {7, 4, 12, 3, 6, 8, 1, 5, 10}
Output : Yes
For the given arr[] the Binary Search Tree is:
    7
   /   \
  4   12
 / \   /
3   6  8
/   /   \
1   5   10
```

```
Input : arr[] = {11, 6, 13, 5, 12, 10}
Output : No
The given arr[] do not represent the level
order traversal of a BST.
```

The idea is to use a queue data structure. Every element of queue has a structure say **NodeDetails** which stores details of a tree node. The details are node's data, and two

variables **min** and **max** where **min** stores the lower limit for the node values which can be a part of the left subtree and **max** stores the upper limit for the node values which can be a part of the right subtree for the specified node in **NodeDetails** structure variable. For the 1st array value arr[0], create a **NodeDetails** structure having arr[0] as node's data and **min** = INT\_MIN and **max** = INT\_MAX. Add this structure variable to the queue. This Node will be the root of the tree. Move to 2nd element in arr[] and then perform the following steps:

1. Pop **NodeDetails** from the queue in **temp**.
2. Check whether the current array element can be a left child of the node in **temp** with the help of **min** and **temp.data** values. If it can, then create a new **NodeDetails** structure for this new array element value with its proper 'min' and 'max' values and push it to the queue, and move to next element in arr[].
3. Check whether the current array element can be a right child of the node in **temp** with the help of **max** and **temp.data** values. If it can, then create a new **NodeDetails** structure for this new array element value with its proper 'min' and 'max' values and push it to the queue, and move to next element in arr[].
4. Repeat steps 1, 2 and 3 until there are no more elements in arr[] or there are no more elements in the queue.

Finally, if all the elements of the array have been traversed then the array represents the level order traversal of a BST, else NOT.

```
// C++ implementation to check if the given array
// can represent Level Order Traversal of Binary
// Search Tree
#include <bits/stdc++.h>

using namespace std;

// to store details of a node like
// node's data, 'min' and 'max' to obtain the
// range of values where node's left and
// right child's should lie
struct NodeDetails
{
    int data;
    int min, max;
};

// function to check if the given array
// can represent Level Order Traversal
// of Binary Search Tree
bool levelOrderIsOfBST(int arr[], int n)
{
    // if tree is empty
    if (n == 0)
```

```
return true;

// queue to store NodeDetails
queue<NodeDetails> q;

// index variable to access array elements
int i=0;

// node details for the
// root of the BST
NodeDetails newNode;
newNode.data = arr[i++];
newNode.min = INT_MIN;
newNode.max = INT_MAX;
q.push(newNode);

// until there are no more elements
// in arr[] or queue is not empty
while (i != n && !q.empty())
{
    // extracting NodeDetails of a
    // node from the queue
    NodeDetails temp = q.front();
    q.pop();

    // check whether there are more elements
    // in the arr[] and arr[i] can be left child
    // of 'temp.data' or not
    if (i < n && (arr[i] < temp.data &&
                    arr[i] > temp.min))
    {
        // Create NodeDetails for newNode
        // and add it to the queue
        newNode.data = arr[i++];
        newNode.min = temp.min;
        newNode.max = temp.data;
        q.push(newNode);
    }

    // check whether there are more elements
    // in the arr[] and arr[i] can be right child
    // of 'temp.data' or not
    if (i < n && (arr[i] > temp.data &&
                    arr[i] < temp.max))
    {
        // Create NodeDetails for newNode
        // and add it to the queue
        newNode.data = arr[i++];
    }
}
```

```
        newNode.min = temp.data;
        newNode.max = temp.max;
        q.push(newNode);
    }
}

// given array represents level
// order traversal of BST
if (i == n)
    return true;

// given array do not represent
// level order traversal of BST
return false;
}

// Driver program to test above
int main()
{
    int arr[] = {7, 4, 12, 3, 6, 8, 1, 5, 10};
    int n = sizeof(arr) / sizeof(arr[0]);
    if (levelOrderIsOfBST(arr, n))
        cout << "Yes";
    else
        cout << "No";
    return 0;
}
```

Output:

Yes

Time Complexity: O(n)

## Source

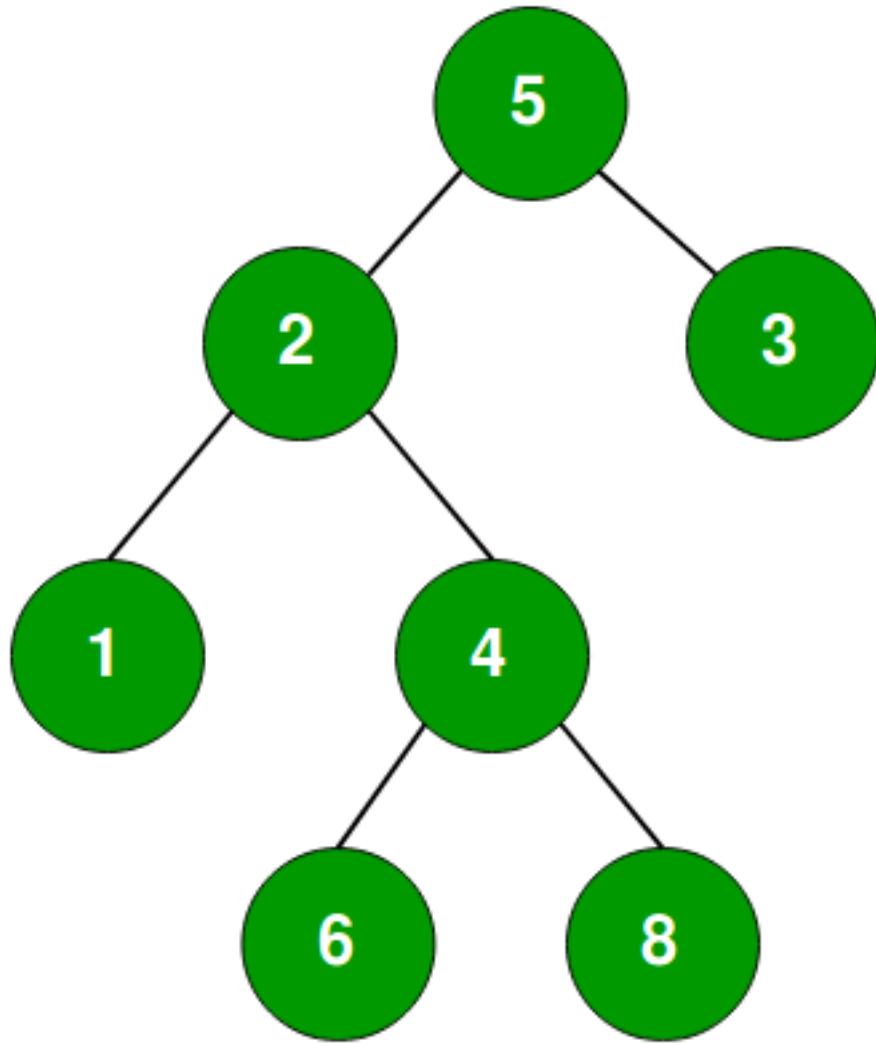
<https://www.geeksforgeeks.org/check-given-array-can-represent-level-order-traversal-binary-search-tree/>

## Chapter 47

### Check if there is a root to leaf path with given sequence

Check if there is a root to leaf path with given sequence - GeeksforGeeks

Given a binary tree and an array, the task is to find if the given array sequence is present as a root to leaf path in given tree.



Examples :

Input : arr[] = {2, 4, 8} for above tree  
Output: "Path Exist"

Input : arr[] = {5, 3, 4, 9} & above tree  
Output: "Path does not Exist"

A **simple solution** for this problem is to find all root to leaf paths in given tree and for each root to leaf path check that path and given sequence in array both are identical or not.

An **efficient solution** for this problem is to traverse the tree once and while traversing the tree we have to check that if path from root to current node is identical to the given sequence of root to leaf path. Here is the algorithm :

- Start traversing tree in **preorder** fashion.
- Whenever we moves down in tree then we also move by **one index** in given sequence of root to leaf path .
- If **current node** is equal to the **arr[index]** this means that till this level of tree path is identical.
- Now remaining path will either be in **left subtree** or in **right subtree**.
- If any node gets mismatched with **arr[index]** this means that current path is not identical to the given sequence of root to leaf path, so we return back and move in right subtree.
- Now when we are at **leaf node** and it is equal to **arr[index]** and there is no further element in given sequence of root to leaf path, this means that path exist in given tree.

#### Path Exists

Time complexity :  $O(n)$

#### Source

<https://www.geeksforgeeks.org/check-root-leaf-path-given-sequence/>

## Chapter 48

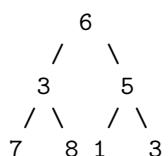
# Check if two nodes are cousins in a Binary Tree

Check if two nodes are cousins in a Binary Tree - GeeksforGeeks

Given the binary Tree and the two nodes say ‘a’ and ‘b’, determine whether the two nodes are cousins of each other or not.

Two nodes are cousins of each other if they are at same level and have different parents.

Example



Say two node be 7 and 1, result is TRUE.

Say two nodes are 3 and 5, result is FALSE.

Say two nodes are 7 and 5, result is FALSE.

The idea is to find level of one of the nodes. Using the found level, check if ‘a’ and ‘b’ are at this level. If ‘a’ and ‘b’ are at given level, then finally check if they are not children of same parent.

Following is the implementation of the above approach.

C

```
// C program to check if two Nodes in a binary tree are cousins
#include <stdio.h>
#include <stdlib.h>
```

```
// A Binary Tree Node
struct Node
{
    int data;
    struct Node *left, *right;
};

// A utility function to create a new Binary Tree Node
struct Node *newNode(int item)
{
    struct Node *temp = (struct Node *)malloc(sizeof(struct Node));
    temp->data = item;
    temp->left = temp->right = NULL;
    return temp;
}

// Recursive function to check if two Nodes are siblings
int isSibling(struct Node *root, struct Node *a, struct Node *b)
{
    // Base case
    if (root==NULL) return 0;

    return ((root->left==a && root->right==b) ||
            (root->left==b && root->right==a) ||
            isSibling(root->left, a, b) ||
            isSibling(root->right, a, b));
}

// Recursive function to find level of Node 'ptr' in a binary tree
int level(struct Node *root, struct Node *ptr, int lev)
{
    // base cases
    if (root == NULL) return 0;
    if (root == ptr) return lev;

    // Return level if Node is present in left subtree
    int l = level(root->left, ptr, lev+1);
    if (l != 0) return l;

    // Else search in right subtree
    return level(root->right, ptr, lev+1);
}

// Returns 1 if a and b are cousins, otherwise 0
int isCousin(struct Node *root, struct Node *a, struct Node *b)
{
    //1. The two Nodes should be on the same level in the binary tree.
```

```
//2. The two Nodes should not be siblings (means that they should
// not have the same parent Node).
if ((level(root,a,1) == level(root,b,1)) && !(isSibling(root,a,b)))
    return 1;
else return 0;
}

// Driver Program to test above functions
int main()
{
    struct Node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->left->right->right = newNode(15);
    root->right->left = newNode(6);
    root->right->right = newNode(7);
    root->right->left->right = newNode(8);

    struct Node *Node1,*Node2;
    Node1 = root->left->left;
    Node2 = root->right->right;

    isCousin(root,Node1,Node2)? puts("Yes"): puts("No");

    return 0;
}
```

### Java

```
// Java program to check if two binary tree are cousins
class Node
{
    int data;
    Node left, right;

    Node(int item)
    {
        data = item;
        left = right = null;
    }
}

class BinaryTree
{
    Node root;
```

```
// Recursive function to check if two Nodes are
// siblings
boolean isSibling(Node node, Node a, Node b)
{
    // Base case
    if (node == null)
        return false;

    return ((node.left == a && node.right == b) ||
            (node.left == b && node.right == a) ||
            isSibling(node.left, a, b) ||
            isSibling(node.right, a, b));
}

// Recursive function to find level of Node 'ptr' in
// a binary tree
int level(Node node, Node ptr, int lev)
{
    // base cases
    if (node == null)
        return 0;

    if (node == ptr)
        return lev;

    // Return level if Node is present in left subtree
    int l = level(node.left, ptr, lev + 1);
    if (l != 0)
        return l;

    // Else search in right subtree
    return level(node.right, ptr, lev + 1);
}

// Returns 1 if a and b are cousins, otherwise 0
boolean isCousin(Node node, Node a, Node b)
{
    // 1. The two Nodes should be on the same level
    //      in the binary tree.
    // 2. The two Nodes should not be siblings (means
    //      that they should not have the same parent
    //      Node).
    return ((level(node, a, 1) == level(node, b, 1)) &&
            (!isSibling(node, a, b)));
}

//Driver program to test above functions
public static void main(String args[])
```

```
{  
    BinaryTree tree = new BinaryTree();  
    tree.root = new Node(1);  
    tree.root.left = new Node(2);  
    tree.root.right = new Node(3);  
    tree.root.left.left = new Node(4);  
    tree.root.left.right = new Node(5);  
    tree.root.left.right.right = new Node(15);  
    tree.root.right.left = new Node(6);  
    tree.root.right.right = new Node(7);  
    tree.root.right.left.right = new Node(8);  
  
    Node Node1, Node2;  
    Node1 = tree.root.left.left;  
    Node2 = tree.root.right.right;  
    if (tree.isCousin(tree.root, Node1, Node2))  
        System.out.println("Yes");  
    else  
        System.out.println("No");  
}  
}  
  
// This code has been contributed by Mayank Jaiswal
```

### Python

```
# Python program to check if two nodes in a binary  
# tree are cousins  
  
# A Binary Tree Node  
class Node:  
  
    # Constructor to create a new Binary Tree  
    def __init__(self, data):  
        self.data = data  
        self.left = None  
        self.right = None  
  
def isSibling(root, a, b):  
  
    # Base Case  
    if root is None:  
        return 0  
  
    return ((root.left == a and root.right == b) or  
            (root.left == b and root.right == a) or  
            isSibling(root.left, a, b) or  
            isSibling(root.right, a, b))
```

```
# Recursive function to find level of Node 'ptr' in
# a binary tree
def level(root, ptr, lev):

    # Base Case
    if root is None :
        return 0
    if root == ptr:
        return lev

    # Return level if Node is present in left subtree
    l = level(root.left, ptr, lev+1)
    if l != 0:
        return l

    # Else search in right subtree
    return level(root.right, ptr, lev+1)

# Returns 1 if a and b are cousins, otherwise 0
def isCousin(root,a, b):

    # 1. The two nodes should be on the same level in
    # the binary tree
    # The two nodes should not be siblings(means that
    # they should not have the smae parent node

    if ((level(root,a,1) == level(root, b, 1)) and
        not (isSibling(root, a, b))):
        return 1
    else:
        return 0

# Driver program to test above function
root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)
root.left.right.right = Node(15)
root.right.left = Node(6)
root.right.right = Node(7)
root.right.left.right = Node(8)

node1 = root.left.right
node2 = root.right.right
```

```
print "Yes" if isCousin(root, node1, node2) == 1 else "No"  
# This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

Ouput:

Yes

Time Complexity of the above solution is  $O(n)$  as it does at most three traversals of binary tree.

[Check if two nodes are cousins in a Binary Tree | Set-2](#)

This article is contributed by **Ayush Srivastava**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## Source

<https://www.geeksforgeeks.org/check-two-nodes-cousins-binary-tree/>

## Chapter 49

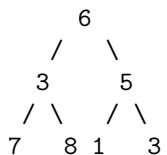
# Check if two nodes are cousins in a Binary Tree | Set-2

Check if two nodes are cousins in a Binary Tree | Set-2 - GeeksforGeeks

Given a binary tree and the two nodes say ‘a’ and ‘b’, determine whether two given nodes are cousins of each other or not.

Two nodes are cousins of each other if they are at same level and have different parents.

**Example:**



Say two node be 7 and 1, result is TRUE.

Say two nodes are 3 and 5, result is FALSE.

Say two nodes are 7 and 5, result is FALSE.

A solution in [Set-1](#) that finds whether given nodes are cousins or not by performing three traversals of binary tree has been discussed. The problem can be solved by performing level order traversal. The idea is to use a queue to perform level order traversal, in which each queue element is a pair of node and parent of that node. For each node visited in level order traversal, check if that node is either first given node or second given node. If any node is found store parent of that node. While performing level order traversal, one level is traversed at a time. If both nodes are found in given level, then their parent values are compared to check if they are siblings or not. If one node is found in given level and another is not found, then given nodes are not cousins.

Below is the implementation of above approach:

```
// CPP program to check if two Nodes in
// a binary tree are cousins
// using level-order traversals
#include <bits/stdc++.h>
using namespace std;

// A Binary Tree Node
struct Node {
    int data;
    struct Node *left, *right;
};

// A utility function to create a new
// Binary Tree Node
struct Node* newNode(int item)
{
    struct Node* temp = (struct Node*)malloc(sizeof(struct Node));
    temp->data = item;
    temp->left = temp->right = NULL;
    return temp;
}

// Returns true if a and b are cousins,
// otherwise false.
bool isCousin(Node* root, Node* a, Node* b)
{
    if (root == NULL)
        return false;

    // To store parent of node a.
    Node* parA = NULL;

    // To store parent of node b.
    Node* parB = NULL;

    // queue to perform level order
    // traversal. Each element of
    // queue is a pair of node and
    // its parent.
    queue<pair<Node*, Node*>> q;

    // Dummy node to act like parent
    // of root node.
    Node* tmp = newNode(-1);

    // To store front element of queue.
    pair<Node*, Node*> ele;
```

```
// Push root to queue.  
q.push(make_pair(root, tmp));  
int levSize;  
  
while (!q.empty()) {  
  
    // find number of elements in  
    // current level.  
    levSize = q.size();  
    while (levSize) {  
  
        ele = q.front();  
        q.pop();  
  
        // check if current node is node a  
        // or node b or not.  
        if (ele.first->data == a->data) {  
            parA = ele.second;  
        }  
  
        if (ele.first->data == b->data) {  
            parB = ele.second;  
        }  
  
        // push children of current node  
        // to queue.  
        if (ele.first->left) {  
            q.push(make_pair(ele.first->left, ele.first));  
        }  
  
        if (ele.first->right) {  
            q.push(make_pair(ele.first->right, ele.first));  
        }  
  
        levSize--;  
  
        // If both nodes are found in  
        // current level then no need  
        // to traverse current level further.  
        if (parA && parB)  
            break;  
    }  
  
    // Check if both nodes are siblings  
    // or not.  
    if (parA && parB) {  
        return parA != parB;  
    }  
}
```

```
// If one node is found in current level
// and another is not found, then
// both nodes are not cousins.
if ((parA && !parB) || (parB && !parA)) {
    return false;
}
}

return false;
}
// Driver Code
int main()
{
/*
          1
         / \
        2   3
       / \ / \
      4  5 6  7
         \ \
        15 8
*/
struct Node* root = newNode(1);
root->left = newNode(2);
root->right = newNode(3);
root->left->left = newNode(4);
root->left->right = newNode(5);
root->left->right->right = newNode(15);
root->right->left = newNode(6);
root->right->right = newNode(7);
root->right->left->right = newNode(8);

struct Node *Node1, *Node2;
Node1 = root->left->left;
Node2 = root->right->right;

isCousin(root, Node1, Node2) ? puts("Yes") : puts("No");

return 0;
}
```

**Output:**

Yes

**Time Complexity:**  $O(n)$

**Auxiliary Space:**  $O(n)$

### Source

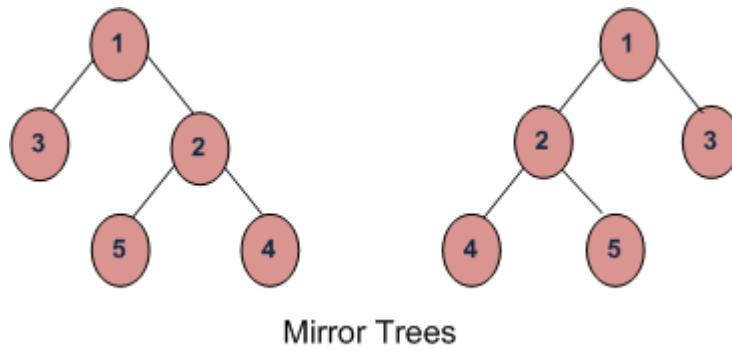
<https://www.geeksforgeeks.org/check-if-two-nodes-are-cousins-in-a-binary-tree-set-2/>

# Chapter 50

## Check if two trees are Mirror

Check if two trees are Mirror - GeeksforGeeks

Given two Binary Trees, write a function that returns true if two trees are mirror of each other, else false. For example, the function should return true for following input trees.



This problem is different from the problem discussed [here](#).

For two trees 'a' and 'b' to be mirror images, the following three conditions must be true:

1. Their root node's key must be same
2. Left subtree of root of 'a' and right subtree root of 'b' are mirror.
3. Right subtree of 'a' and left subtree of 'b' are mirror.

Below is implementation of above idea.

C++

```
// C++ program to check if two trees are mirror
// of each other
#include<bits/stdc++.h>
using namespace std;
```

```
/* A binary tree node has data, pointer to
   left child and a pointer to right child */
struct Node
{
    int data;
    Node* left, *right;
};

/* Given two trees, return true if they are
   mirror of each other */
int areMirror(Node* a, Node* b)
{
    /* Base case : Both empty */
    if (a==NULL && b==NULL)
        return true;

    // If only one is empty
    if (a==NULL || b == NULL)
        return false;

    /* Both non-empty, compare them recursively
       Note that in recursive calls, we pass left
       of one tree and right of other tree */
    return  a->data == b->data &&
            areMirror(a->left, b->right) &&
            areMirror(a->right, b->left);
}

/* Helper function that allocates a new node */
Node* newNode(int data)
{
    Node* node = new Node;
    node->data  = data;
    node->left   = node->right  = NULL;
    return(node);
}

/* Driver program to test areMirror() */
int main()
{
    Node *a = newNode(1);
    Node *b = newNode(1);
    a->left = newNode(2);
    a->right = newNode(3);
    a->left->left  = newNode(4);
    a->left->right = newNode(5);
```

```
b->left = newNode(3);
b->right = newNode(2);
b->right->left = newNode(5);
b->right->right = newNode(4);

areMirror(a, b)? cout << "Yes" : cout << "No";

return 0;
}
```

**Java**

```
// Java program to see if two trees
// are mirror of each other

// A binary tree node
class Node
{
    int data;
    Node left, right;

    public Node(int data)
    {
        this.data = data;
        left = right = null;
    }
}

class BinaryTree
{
    Node a, b;

    /* Given two trees, return true if they are
       mirror of each other */
    boolean areMirror(Node a, Node b)
    {
        /* Base case : Both empty */
        if (a == null && b == null)
            return true;

        // If only one is empty
        if (a == null || b == null)
            return false;

        /* Both non-empty, compare them recursively
           Note that in recursive calls, we pass left
           of one tree and right of other tree */
        return a.data == b.data
    }
}
```

```
        && areMirror(a.left, b.right)
        && areMirror(a.right, b.left);
    }

// Driver code to test above methods
public static void main(String[] args)
{
    BinaryTree tree = new BinaryTree();
    Node a = new Node(1);
    Node b = new Node(1);
    a.left = new Node(2);
    a.right = new Node(3);
    a.left.left = new Node(4);
    a.left.right = new Node(5);

    b.left = new Node(3);
    b.right = new Node(2);
    b.right.left = new Node(5);
    b.right.right = new Node(4);

    if (tree.areMirror(a, b) == true)
        System.out.println("Yes");
    else
        System.out.println("No");

}
}

// This code has been contributed by Mayank Jaiswal(mayank_24)
```

### Python3

```
# Python3 program to check if two
# trees are mirror of each other

# A binary tree node
class Node:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

# Given two trees, return true
# if they are mirror of each other
def areMirror(a, b):

    # Base case : Both empty
    if a is None and b is None:
```

```
    return True

# If only one is empty
if a is None or b is None:
    return False

# Both non-empty, compare them
# recursively. Note that in
# recursive calls, we pass left
# of one tree and right of other tree
return (a.data == b.data and
        areMirror(a.left, b.right) and
        areMirror(a.right , b.left))

# Driver code
root1 = Node(1)
root2 = Node(1)

root1.left = Node(2)
root1.right = Node(3)
root1.left.left = Node(4)
root1.left.right = Node(5)

root2.left = Node(3)
root2.right = Node(2)
root2.right.left = Node(5)
root2.right.right = Node(4)

if areMirror(root1, root2):
    print ("Yes")
else:
    print ("No")

# This code is contributed by AshishR
```

**Output :**

Yes

**Time Complexity :** O(n)

#### **Iterative method to check if two trees are mirror of each other**

This article is contributed by **Ashish Gupta**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

**Improved By :** [AshishR](#)

**Source**

<https://www.geeksforgeeks.org/check-if-two-trees-are-mirror/>

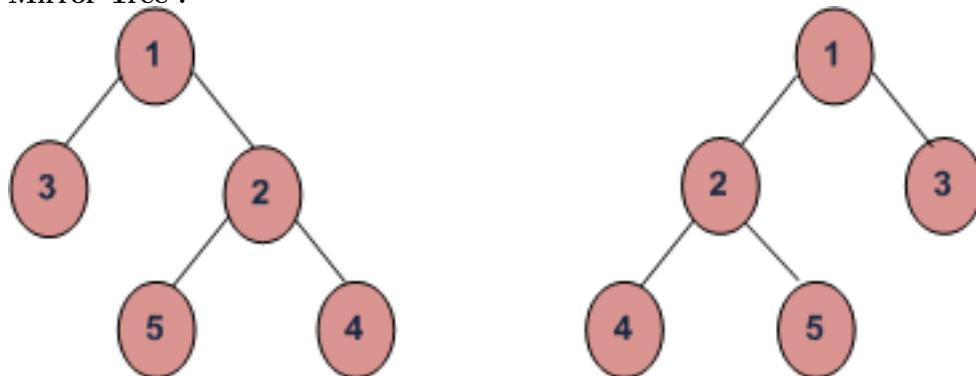
## Chapter 51

# Check if two trees are Mirror | Set 2

Check if two trees are Mirror | Set 2 - GeeksforGeeks

Given two Binary Trees, returns true if two trees are mirror of each other, else false.

**Mirror Tree :**



**Mirror Trees**

Previously discussed approach is [here](#).

**Approach :**

Find the inorder traversal of both the Binary Trees, and check whether one traversal is reverse of another or not. If they are reverse of each other then the trees are mirror of each other, else not.

**CPP**

```
// CPP code to check two binary trees are
```

```
// mirror.
#include<bits/stdc++.h>
using namespace std;

struct Node
{
    int data;
    Node* left, *right;
};

// inorder traversal of Binary Tree
void inorder(Node *n, vector<int> &v)
{
    if (n->left != NULL)
        inorder(n->left, v);
    v.push_back(n->data);
    if (n->right != NULL)
        inorder(n->right, v);
}

// Checking if binary tree is mirror
// of each other or not.
bool areMirror(Node* a, Node* b)
{
    if (a == NULL && b == NULL)
        return true;
    if (a == NULL || b== NULL)
        return false;

    // Storing inorder traversals of both
    // the trees.
    vector<int> v1, v2;
    inorder(a, v1);
    inorder(b, v2);

    if (v1.size() != v2.size())
        return false;

    // Comparing the two arrays, if they
    // are reverse then return 1, else 0
    for (int i=0, j=v2.size()-1; j >= 0;
         i++, j--)
        if (v1[i] != v2[j])
            return false;

    return true;
}
```

```
// Helper function to allocate a new node
Node* newNode(int data)
{
    Node* node = new Node;
    node->data = data;
    node->left = node->right = NULL;

    return(node);
}

// Driver code
int main()
{
    Node *a = newNode(1);
    Node *b = newNode(1);

    a -> left = newNode(2);
    a -> right = newNode(3);
    a -> left -> left = newNode(4);
    a -> left -> right = newNode(5);

    b -> left = newNode(3);
    b -> right = newNode(2);
    b -> right -> left = newNode(5);
    b -> right -> right = newNode(4);

    areMirror(a, b)? cout << "Yes" : cout << "No";

    return 0;
}
```

Output:

Yes

## Source

<https://www.geeksforgeeks.org/check-two-trees-mirror-set-2/>

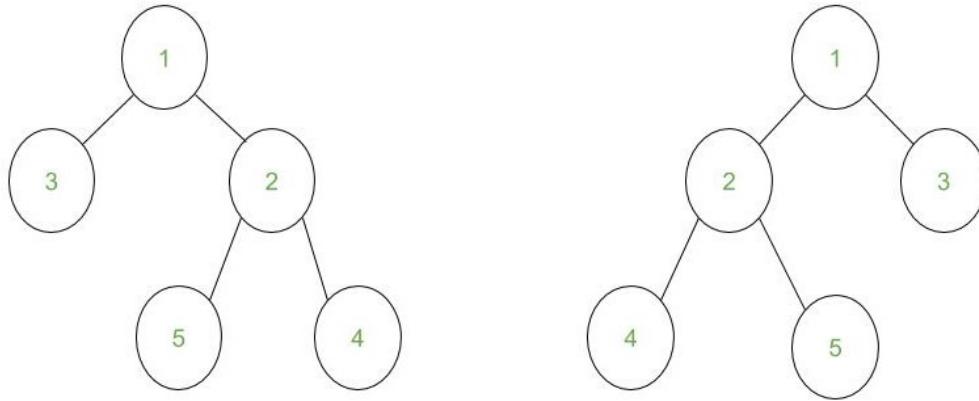
## Chapter 52

# Check if two trees are mirror of each other using level order traversal

Check if two trees are mirror of each other using level order traversal - GeeksforGeeks

Given two binary trees, the task is to check whether the two binary trees is a mirror of each other or not.

**Mirror of a Binary Tree:** Mirror of a Binary Tree T is another Binary Tree M(T) with left and right children of all non-leaf nodes interchanged.



### Mirror trees

Trees in the above figure are mirrors of each other.

A [recursive solution](#) and an [iterative method using inorder traversal](#) to check whether the two binary trees is a mirror of each other or not have been already discussed. In this post a solution using [level order traversal](#) has been discussed.

The idea is to use a queue in which two nodes of both the trees which needs to be checked for equality are present together. At each step of level order traversal, get two nodes from the queue, check for their equality and then insert next two children nodes of these nodes which need to be checked for equality. During insertion step, first left child of first tree node and right child of second tree node are inserted. After this right child of first tree node and left child of second tree node are inserted. If at any stage one node is NULL and other is not, then both trees are not a mirror of each other.

Below is the implementation of above approach:

```
// C++ implementation to check whether the two
// binary trees are mirrors of each other or not
#include <bits/stdc++.h>
using namespace std;

// Structure of a node in binary tree
struct Node {
    int data;
```

```
    struct Node *left, *right;
};

// Function to create and return
// a new node for a binary tree
struct Node* newNode(int data)
{
    struct Node* temp = new Node();
    temp->data = data;
    temp->left = temp->right = NULL;
    return temp;
}

// Function to check whether the two binary trees
// are mirrors of each other or not
string areMirrors(Node* a, Node* b)
{
    // If both are NULL, then are mirror.
    if (a == NULL && b == NULL)
        return "Yes";

    // If only one is NULL, then not
    // mirror.
    if (a == NULL || b == NULL)
        return "No";

    queue<Node*> q;

    // Push root of both trees in queue.
    q.push(a);
    q.push(b);

    while (!q.empty()) {

        // Pop two elements of queue, to
        // get two nodes and check if they
        // are symmetric.
        a = q.front();
        q.pop();

        b = q.front();
        q.pop();

        // If data value of both nodes is
        // not same, then not mirror.
        if (a->data != b->data)
            return "No";
    }
}
```

```
// Push left child of first tree node
// and right child of second tree node
// into queue if both are not NULL.
if (a->left && b->right) {
    q.push(a->left);
    q.push(b->right);
}

// If any one of the nodes is NULL and
// other is not NULL, then not mirror.
else if (a->left || b->right)
    return "No";

// Push right child of first tree node
// and left child of second tree node
// into queue if both are not NULL.
if (a->right && b->left) {
    q.push(a->right);
    q.push(b->left);
}

// If any one of the nodes is NULL and
// other is not NULL, then not mirror.
else if (a->right || b->left)
    return "No";
}

return "Yes";
}
// Driver Code
int main()
{
    // 1st binary tree formation
    /*
        1
       / \
      3   2
         / \
        5   4
    */
    Node* root1 = newNode(1);
    root1->left = newNode(3);
    root1->right = newNode(2);
    root1->right->left = newNode(5);
    root1->right->right = newNode(4);

    // 2nd binary tree formation
    /*
```

```
    1
   / \
  2   3
 / \
4   5
*/
Node* root2 = newNode(1);
root2->left = newNode(2);
root2->right = newNode(3);
root2->left->left = newNode(4);
root2->left->right = newNode(5);

cout << areMirrors(root1, root2);
return 0;
}
```

**Output:**

Yes

**Time complexity:** O(N)

**Source**

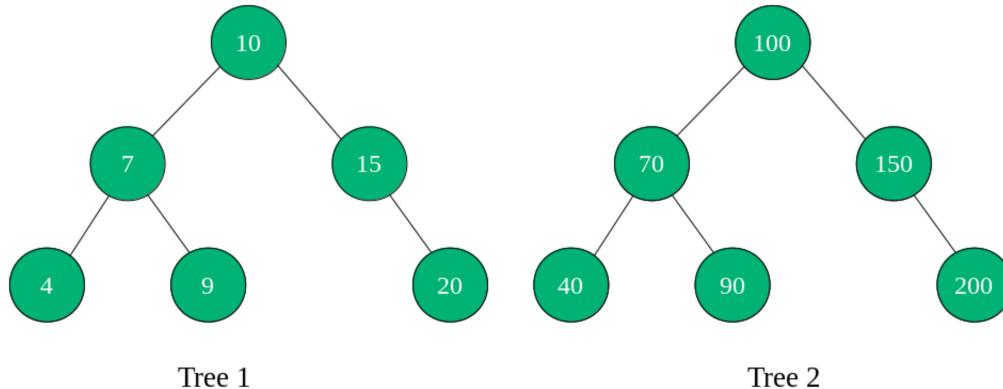
<https://www.geeksforgeeks.org/check-if-two-trees-are-mirror-of-each-other-using-level-order-traversal/>

## Chapter 53

# Check if two trees have same structure

Check if two trees have same structure - GeeksforGeeks

Given two binary trees. The task is to write a program to check if the two trees are identical in structure.



In the above figure both of the trees, Tree1 and Tree2 are identical in structure. That is, they have the same structure.

**Note:** This problem is different from [Check if two trees are identical](#) as here we need to compare only the structures of the two trees and not the values at their nodes.

The idea is to traverse both trees simultaneously following the same paths and keep checking if a node exists for both the trees or not.

### Algorithm:

1. If both trees are empty then return 1.

2. Else If both trees are non-empty:
  - Check left subtrees recursively i.e., call `isSameStructure(tree1->left_subtree, tree2->left_subtree)`
  - Check right subtrees recursively i.e., call `isSameStructure(tree1->right_subtree, tree2->right_subtree)`
  - If the value returned in above two steps are true then return 1.
3. Else return 0 (one is empty and other is not).

Below is the implementation of above algorithm:

```
// C++ program to check if two trees have
// same structure
#include <iostream>
using namespace std;

// A binary tree node has data, pointer to left child
// and a pointer to right child
struct Node
{
    int data;
    struct Node* left;
    struct Node* right;
};

// Helper function that allocates a new node with the
// given data and NULL left and right pointers.
Node* newNode(int data)
{
    Node* node = new Node;
    node->data = data;
    node->left = NULL;
    node->right = NULL;

    return(node);
}

// Function to check if two trees have same
// structure
int isSameStructure(Node* a, Node* b)
{
    // 1. both empty
    if (a==NULL && b==NULL)
        return 1;

    // 2. both non-empty -> compare them
    if (a!=NULL && b!=NULL)
    {
```

```
    return
    (
        isSameStructure(a->left, b->left) &&
        isSameStructure(a->right, b->right)
    );
}

// 3. one empty, one not -> false
return 0;
}

// Driver code
int main()
{
    Node *root1 = newNode(10);
    Node *root2 = newNode(100);
    root1->left = newNode(7);
    root1->right = newNode(15);
    root1->left->left = newNode(4);
    root1->left->right = newNode(9);
    root1->right->right = newNode(20);

    root2->left = newNode(70);
    root2->right = newNode(150);
    root2->left->left = newNode(40);
    root2->left->right = newNode(90);
    root2->right->right = newNode(200);

    if (isSameStructure(root1, root2))
        printf("Both trees have same structure");
    else
        printf("Trees do not have same structure");

    return 0;
}
```

**Output:**

```
Both trees have same structure
```

**Source**

<https://www.geeksforgeeks.org/check-if-two-trees-have-same-structure/>

## Chapter 54

### Check mirror in n-ary tree

Check mirror in n-ary tree - GeeksforGeeks

Given two n-ary trees, the task is to check if they are mirror of each other or not. Print “Yes” if they are mirror of each other else “No”.

Examples:

```
Input : Node = 3, Edges = 2
Edge 1 of first N-ary: 1 2
Edge 2 of first N-ary: 1 3
Edge 1 of second N-ary: 1 2
Edge 2 of second N-ary: 1 3
Output : Yes
```

```
Input : Node = 3, Edges = 2
Edge 1 of first N-ary: 1 2
Edge 2 of first N-ary: 1 3
Edge 1 of second N-ary: 1 2
Edge 2 of second N-ary: 1 3
Output : No
```

The idea is to use Queue and Stack to check if given N-ary tree are mirror of each other or not.

Let first n-ary tree be t1 and second n-ary tree is t2. For each node in t1, make stack and push its connected node in it. Now, for each node in t2, make queue and push its connected node in it.

Now, for each corresponding node do following:

```
While stack and Queue is not empty.
a = top element of stack;
```

```
b = front of stack;
if (a != b)
    return false;
pop element from stack and queue.

// C++ program to check if two n-ary trees are
// mirror.
#include <bits/stdc++.h>
using namespace std;

// First vector stores all nodes and adjacent of every
// node in a stack.
// Second vector stores all nodes and adjacent of every
// node in a queue.
bool mirrorUtil(vector<stack<int> >& tree1,
                 vector<queue<int> >& tree2)
{
    // Traversing each node in tree.
    for (int i = 1; i < tree1.size(); ++i) {
        stack<int>& s = tree1[i];
        queue<int>& q = tree2[i];

        // While stack is not empty && Queue is not empty
        while (!s.empty() && !q.empty()) {

            // checking top element of stack and front
            // of queue.
            if (s.top() != q.front())
                return false;

            s.pop();
            q.pop();
        }

        // If queue or stack is not empty, return false.
        if (!s.empty() || !q.empty())
            return false;
    }

    return true;
}

// Returns true if given two trees are mirrors.
// A tree is represented as two arrays to store
// all tree edges.
void areMirrors(int m, int n, int u1[], int v1[],
                int u2[], int v2[])
{
```

```
vector<stack<int>> tree1(m + 1);
vector<queue<int>> tree2(m + 1);

// Pushing node in the stack of first tree.
for (int i = 0; i < n; i++)
    tree1[u1[i]].push(v1[i]);

// Pushing node in the queue of second tree.
for (int i = 0; i < n; i++)
    tree2[u2[i]].push(v2[i]);

mirrorUtil(tree1, tree2) ? (cout << "Yes" << endl) :
                           (cout << "No" << endl);
}

// Driver code
int main()
{
    int M = 3, N = 2;

    int u1[] = { 1, 1 };
    int v1[] = { 2, 3 };

    int u2[] = { 1, 1 };
    int v2[] = { 3, 2 };

    areMirrors(M, N, u1, v1, u2, v2);

    return 0;
}
```

Output:

Yes

Reference: <https://practice.geeksforgeeks.org/problems/check-mirror-in-n-ary-tree/0>

## Source

<https://www.geeksforgeeks.org/check-mirror-n-ary-tree/>

## Chapter 55

# Check sum of Covered and Uncovered nodes of Binary Tree

Check sum of Covered and Uncovered nodes of Binary Tree - GeeksforGeeks

Given a binary tree, you need to check whether sum of all covered elements is equal to sum of all uncovered elements or not.

In a binary tree, a node is called Uncovered if it appears either on left boundary or right boundary. Rest of the nodes are called covered.

For example, consider below binary tree

In above binary tree,

Covered node: 6, 5, 7

Uncovered node: 9, 4, 3, 17, 22, 20

The output for this tree should be false as  
sum of covered and uncovered node is not same

**We strongly recommend you to minimize your browser and try this yourself first.**

For calculating sum of Uncovered nodes we will follow below steps:

1) Start from root, go to left and keep going until left child is available, if not go to right child and again follow same procedure until you reach a leaf node.

2) After step 1 sum of left boundary will be stored, now for right part again do the same procedure but now keep going to right until right child is available, if not then go to left child and follow same procedure until you reach a leaf node.

After above 2 steps sum of all Uncovered node will be stored, we can subtract it from total sum and get sum of covered elements and check for equines of binary tree.

C++

```
// C++ program to find sum of Covered and Uncovered node of
// binary tree
#include <bits/stdc++.h>
using namespace std;

/* A binary tree node has key, pointer to left
   child and a pointer to right child */
struct Node
{
    int key;
    struct Node* left, *right;
};

/* To create a newNode of tree and return pointer */
struct Node* newNode(int key)
{
    Node* temp = new Node;
    temp->key = key;
    temp->left = temp->right = NULL;
    return (temp);
}

/* Utility function to calculate sum of all node of tree */
int sum(Node* t)
{
    if (t == NULL)
        return 0;
    return t->key + sum(t->left) + sum(t->right);
}

/* Recursive function to calculate sum of left boundary
   elements */
int uncoveredSumLeft(Node* t)
{
    /* If leaf node, then just return its key value */
    if (t->left == NULL && t->right == NULL)
        return t->key;

    /* If left is available then go left otherwise go right */
    if (t->left != NULL)
        return t->key + uncoveredSumLeft(t->left);
    else
        return t->key + uncoveredSumLeft(t->right);
}

/* Recursive function to calculate sum of right boundary
   elements */
int uncoveredSumRight(Node* t)
```

```

{
    /* If leaf node, then just return its key value */
    if (t->left == NULL && t->right == NULL)
        return t->key;

    /* If right is available then go right otherwise go left */
    if (t->right != NULL)
        return t->key + uncoveredSumRight(t->right);
    else
        return t->key + uncoveredSumRight(t->left);
}

// Returns sum of uncovered elements
int uncoverSum(Node* t)
{
    /* Initializing with 0 in case we don't have
       left or right boundary */
    int lb = 0, rb = 0;

    if (t->left != NULL)
        lb = uncoveredSumLeft(t->left);
    if (t->right != NULL)
        rb = uncoveredSumRight(t->right);

    /* returning sum of root node, left boundry
       and right boundry*/
    return t->key + lb + rb;
}

// Returns true if sum of covered and uncovered elements
// is same.
bool isSumSame(Node *root)
{
    // Sum of uncovered elements
    int sumUC = uncoverSum(root);

    // Sum of all elements
    int sumT = sum(root);

    // Check if sum of covered and uncovered is same
    return (sumUC == (sumT - sumUC));
}

/* Helper function to print inorder traversal of
   binary tree */
void inorder(Node* root)
{
    if (root)

```

```
{  
    inorder(root->left);  
    printf("%d ", root->key);  
    inorder(root->right);  
}  
}  
  
// Driver program to test above functions  
int main()  
{  
    // Making above given diagram's binary tree  
    Node* root = newNode(8);  
    root->left = newNode(3);  
  
    root->left->left = newNode(1);  
    root->left->right = newNode(6);  
    root->left->right->left = newNode(4);  
    root->left->right->right = newNode(7);  
  
    root->right = newNode(10);  
    root->right->right = newNode(14);  
    root->right->right->left = newNode(13);  
  
    if (isSumSame(root))  
        printf("Sum of covered and uncovered is same\n");  
    else  
        printf("Sum of covered and uncovered is not same\n");  
}
```

### Java

```
// Java program to find sum of covered and uncovered nodes  
// of a binary tree  
  
/* A binary tree node has key, pointer to left child and  
   a pointer to right child */  
class Node  
{  
    int key;  
    Node left, right;  
  
    public Node(int key)  
    {  
        this.key = key;  
        left = right = null;  
    }  
}
```

```
class BinaryTree
{
    Node root;

    /* Utility function to calculate sum of all node of tree */
    int sum(Node t)
    {
        if (t == null)
            return 0;
        return t.key + sum(t.left) + sum(t.right);
    }

    /* Recursive function to calculate sum of left boundary
       elements */
    int uncoveredSumLeft(Node t)
    {
        /* If left node, then just return its key value */
        if (t.left == null && t.right == null)
            return t.key;

        /* If left is available then go left otherwise go right */
        if (t.left != null)
            return t.key + uncoveredSumLeft(t.left);
        else
            return t.key + uncoveredSumLeft(t.right);
    }

    /* Recursive function to calculate sum of right boundary
       elements */
    int uncoveredSumRight(Node t)
    {
        /* If left node, then just return its key value */
        if (t.left == null && t.right == null)
            return t.key;

        /* If right is available then go right otherwise go left */
        if (t.right != null)
            return t.key + uncoveredSumRight(t.right);
        else
            return t.key + uncoveredSumRight(t.left);
    }

    // Returns sum of uncovered elements
    int uncoverSum(Node t)
    {
        /* Initializing with 0 in case we don't have
           left or right boundary */
        int lb = 0, rb = 0;
```

```
if (t.left != null)
    lb = uncoveredSumLeft(t.left);
if (t.right != null)
    rb = uncoveredSumRight(t.right);

/* returning sum of root node, left boundry
   and right boundry*/
return t.key + lb + rb;
}

// Returns true if sum of covered and uncovered elements
// is same.
boolean isSumSame(Node root)
{
    // Sum of uncovered elements
    int sumUC = uncoverSum(root);

    // Sum of all elements
    int sumT = sum(root);

    // Check if sum of covered and uncovered is same
    return (sumUC == (sumT - sumUC));
}

/* Helper function to print inorder traversal of
   binary tree */
void inorder(Node root)
{
    if (root != null)
    {
        inorder(root.left);
        System.out.print(root.key + " ");
        inorder(root.right);
    }
}

// Driver program to test above functions
public static void main(String[] args)
{
    BinaryTree tree = new BinaryTree();

    // Making above given diagram's binary tree
    tree.root = new Node(8);
    tree.root.left = new Node(3);
    tree.root.left.left = new Node(1);
    tree.root.left.right = new Node(6);
```

```
tree.root.left.right.left = new Node(4);
tree.root.left.right.right = new Node(7);

tree.root.right = new Node(10);
tree.root.right.right = new Node(14);
tree.root.right.right.left = new Node(13);

if (tree.isSumSame(tree.root))
    System.out.println("Sum of covered and uncovered is same");
else
    System.out.println("Sum of covered and uncovered is not same");
}
}

// This code has been contributed by Mayank Jaiswal(mayank_24)
```

Output :

Sum of covered and uncovered is not same

This article is contributed by Utkarsh Trivedi. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

**Improved By :** [MayankShorey](#)

## Source

<https://www.geeksforgeeks.org/check-sum-covered-uncovered-nodes-binary-tree/>

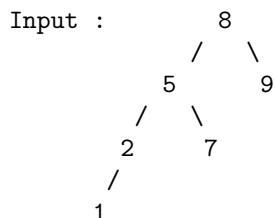
## Chapter 56

# Check whether BST contains Dead End or not

Check whether BST contains Dead End or not - GeeksforGeeks

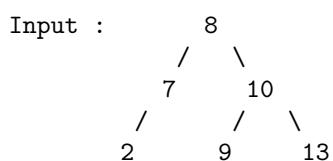
Given a [Binary search Tree](#) that contains positive integer values greater than 0. The task is to check whether the BST contains a dead end or not. Here Dead End means, we are not able to insert any element after that node.

Examples:



Output : Yes

Explanation : Node "1" is the dead End because  
after that we cant insert any element.



Output : Yes

Explanation : We can't insert any element at  
node 9.

If we take a closer look at problem, we can notice that we basically need to check if there is leaf node with value x such that  $x+1$  and  $x-1$  exist in BST with exception of  $x = 1$ . For  $x = 1$ , we can't insert 0 as problem statement says BST contains positive integers only.

To implement above idea we first traverse whole BST and store all nodes in a hash\_map. We also store all leaves in a separate hash to avoid re-traversal of BST. Finally we check for every leaf node x, if  $x-1$  and  $x+1$  are present in hash\_map or not.

Below is C++ implementation of above idea .

```
// C++ program check weather BST contains
// dead end or not
#include<bits/stdc++.h>
using namespace std;

// A BST node
struct Node
{
    int data;
    struct Node *left, *right;
};

// A utility function to create a new node
Node *newNode(int data)
{
    Node *temp = new Node;
    temp->data = data;
    temp->left = temp->right = NULL;
    return temp;
}

/* A utility function to insert a new Node
   with given key in BST */
struct Node* insert(struct Node* node, int key)
{
    /* If the tree is empty, return a new Node */
    if (node == NULL) return newNode(key);

    /* Otherwise, recur down the tree */
    if (key < node->data)
        node->left = insert(node->left, key);
    else if (key > node->data)
        node->right = insert(node->right, key);

    /* return the (unchanged) Node pointer */
    return node;
}

// Function to store all node of given binary search tree
```

```
void storeNodes(Node * root, unordered_set<int> &all_nodes,
                unordered_set<int> &leaf_nodes)
{
    if (root == NULL)
        return ;

    // store all node of binary search tree
    all_nodes.insert(root->data);

    // store leaf node in leaf_hash
    if (root->left==NULL && root->right==NULL)
    {
        leaf_nodes.insert(root->data);
        return ;
    }

    // recur call rest tree
    storeNodes(root->left, all_nodes, leaf_nodes);
    storeNodes(root->right, all_nodes, leaf_nodes);
}

// Returns true if there is a dead end in tree,
// else false.
bool isDeadEnd(Node *root)
{
    // Base case
    if (root == NULL)
        return false;

    // create two empty hash sets that store all
    // BST elements and leaf nodes respectively.
    unordered_set<int> all_nodes, leaf_nodes;

    // insert 0 in 'all_nodes' for handle case
    // if bst contain value 1
    all_nodes.insert(0);

    // Call storeNodes function to store all BST Node
    storeNodes(root, all_nodes, leaf_nodes);

    // Traversal leaf node and check Tree contain
    // continuous sequence of
    // size tree or Not
    for (auto i = leaf_nodes.begin() ; i != leaf_nodes.end(); i++)
    {
        int x = (*i);

        // Here we check first and last element of
```

```
// continuous sequence that are x-1 & x+1
if (all_nodes.find(x+1) != all_nodes.end() &&
    all_nodes.find(x-1) != all_nodes.end())
    return true;
}

return false ;
}

// Driver program
int main()
{
/*      8
     /   \
    5     11
   /   \
  2     7
  \
  3
  \
  4 */
Node *root = NULL;
root = insert(root, 8);
root = insert(root, 5);
root = insert(root, 2);
root = insert(root, 3);
root = insert(root, 7);
root = insert(root, 11);
root = insert(root, 4);
if (isDeadEnd(root) == true)
    cout << "Yes " << endl;
else
    cout << "No " << endl;
return 0;
}
```

Output:

Yes

Time Complexity : O(n)

Simple Recursive solution to check whether BST contains dead End

## Source

<https://www.geeksforgeeks.org/check-whether-bst-contains-dead-end-not/>

## Chapter 57

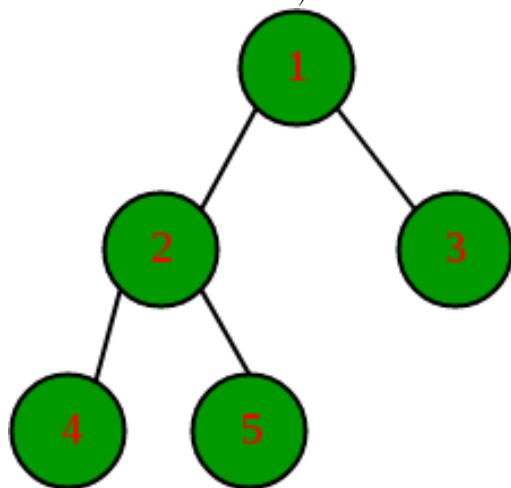
# Check whether a binary tree is a complete tree or not | Set 2 (Recursive Solution)

Check whether a binary tree is a complete tree or not | Set 2 (Recursive Solution) - Geeks-forGeeks

A complete binary tree is a binary tree whose all levels except the last level are completely filled and all the leaves in the last level are all to the left side. More information about complete binary trees can be found [here](#).

For Example:-

Below tree is a Complete Binary Tree (All nodes till the second last nodes are filled and all leaves are to the left side)



An iterative solution for this problem is discussed in below post.

[Check whether a given Binary Tree is Complete or not | Set 1 \(Using Level Order Traversal\)](#)

In this post a recursive solution is discussed.

In the array representation of a binary tree, if the parent node is assigned an index of 'i' and left child gets assigned an index of ' $2*i + 1$ ' while the right child is assigned an index of ' $2*i + 2$ '. If we represent the above binary tree as an array with the respective indices assigned to the different nodes of the tree above from top to down and left to right.

Hence we proceed in the following manner in order to check if the binary tree is complete binary tree.

1. Calculate the number of nodes (count) in the binary tree.
2. Start recursion of the binary tree from the root node of the binary tree with index (i) being set as 0 and the number of nodes in the binary (count).
3. If the current node under examination is NULL, then the tree is a complete binary tree. Return true.
4. If index (i) of the current node is greater than or equal to the number of nodes in the binary tree (count) i.e. ( $i \geq count$ ), then the tree is not a complete binary. Return false.
5. Recursively check the left and right sub-trees of the binary tree for same condition. For the left sub-tree use the index as ( $2*i + 1$ ) while for the right sub-tree use the index as ( $2*i + 2$ ).

The time complexity of the above algorithm is  $O(n)$ . Following is the code for checking if a binary tree is a complete binary tree.

**C**

```
/* C program to checks if a binary tree complete ot not */
#include<stdio.h>
#include<stdlib.h>
#include<stdbool.h>

/* Tree node structure */
struct Node
{
    int key;
    struct Node *left, *right;
};

/* Helper function that allocates a new node with the
   given key and NULL left and right pointer. */
struct Node *newNode(char k)
{
    struct Node *node = (struct Node*)malloc(sizeof(struct Node));
    node->key = k;
    node->right = node->left = NULL;
    return node;
}
```

```
/* This function counts the number of nodes in a binary tree */
unsigned int countNodes(struct Node* root)
{
    if (root == NULL)
        return (0);
    return (1 + countNodes(root->left) + countNodes(root->right));
}

/* This function checks if the binary tree is complete or not */
bool isComplete (struct Node* root, unsigned int index,
                 unsigned int number_nodes)
{
    // An empty tree is complete
    if (root == NULL)
        return (true);

    // If index assigned to current node is more than
    // number of nodes in tree, then tree is not complete
    if (index >= number_nodes)
        return (false);

    // Recur for left and right subtrees
    return (isComplete(root->left, 2*index + 1, number_nodes) &&
            isComplete(root->right, 2*index + 2, number_nodes));
}

// Driver program
int main()
{
    // Let us create tree in the last diagram above
    struct Node* root = NULL;
    root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->right = newNode(6);

    unsigned int node_count = countNodes(root);
    unsigned int index = 0;

    if (isComplete(root, index, node_count))
        printf("The Binary Tree is complete\n");
    else
        printf("The Binary Tree is not complete\n");
    return (0);
}
```

**Java**

```
// Java program to check if binay tree is complete or not

/* Tree node structure */
class Node
{
    int data;
    Node left, right;

    Node(int item) {
        data = item;
        left = right = null;
    }
}

class BinaryTree
{
    Node root;

    /* This function counts the number of nodes in a binary tree */
    int countNodes(Node root)
    {
        if (root == null)
            return (0);
        return (1 + countNodes(root.left) + countNodes(root.right));
    }

    /* This function checks if the binary tree is complete or not */
    boolean isComplete(Node root, int index, int number_nodes)
    {
        // An empty tree is complete
        if (root == null)
            return true;

        // If index assigned to current node is more than
        // number of nodes in tree, then tree is not complete
        if (index >= number_nodes)
            return false;

        // Recur for left and right subtrees
        return (isComplete(root.left, 2 * index + 1, number_nodes)
            && isComplete(root.right, 2 * index + 2, number_nodes));
    }

    // Driver program
    public static void main(String args[])
    {
```

```
{  
    BinaryTree tree = new BinaryTree();  
  
    // Let us create tree in the last diagram above  
    Node NewRoot = null;  
    tree.root = new Node(1);  
    tree.root.left = new Node(2);  
    tree.root.right = new Node(3);  
    tree.root.left.right = new Node(5);  
    tree.root.left.left = new Node(4);  
    tree.root.right.right = new Node(6);  
  
    int node_count = tree.countNodes(tree.root);  
    int index = 0;  
  
    if (tree.isComplete(tree.root, index, node_count))  
        System.out.print("The binary tree is complete");  
    else  
        System.out.print("The binary tree is not complete");  
}  
}  
  
// This code is contributed by Mayank Jaiswal
```

### Python

```
# Python program to check if a binary tree complete or not  
  
# Tree node structure  
class Node:  
  
    # Constructor to create a new node  
    def __init__(self, key):  
        self.key = key  
        self.left = None  
        self.right = None  
  
    # This function counts the number of nodes in a binary tree  
def countNodes(root):  
    if root is None:  
        return 0  
    return (1+ countNodes(root.left) + countNodes(root.right))  
  
# This function checks if binary tree is complete or not  
def isComplete(root, index, number_nodes):  
  
    # An empty is complete
```

```
if root is None:  
    return True  
  
# If index assigned to current nodes is more than  
# number of nodes in tree, then tree is not complete  
if index >= number_nodes :  
    return False  
  
# Recur for left and right subtress  
return (isComplete(root.left , 2*index+1 , number_nodes)  
       and isComplete(root.right, 2*index+2, number_nodes)  
       )  
  
# Driver Program  
  
root = Node(1)  
root.left = Node(2)  
root.right = Node(3)  
root.left.left = Node(4)  
root.left.right = Node(5)  
root.right.right = Node(6)  
  
node_count = countNodes(root)  
index = 0  
  
if isComplete(root, index, node_count):  
    print "The Binary Tree is complete"  
else:  
    print "The Binary Tree is not complete"  
  
# This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

Output:

The Binary Tree is not complete

This article is contributed by **Gaurav Gupta**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## Source

<https://www.geeksforgeeks.org/check-whether-binary-tree-complete-not-set-2-recursive-solution/>

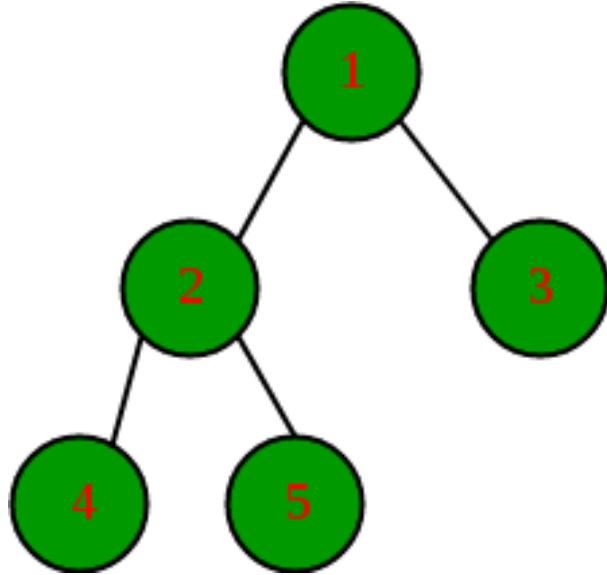
## Chapter 58

# Check whether a binary tree is a full binary tree or not

Check whether a binary tree is a full binary tree or not - GeeksforGeeks

A full binary tree is defined as a binary tree in which all nodes have either zero or two child nodes. Conversely, there is no node in a full binary tree, which has one child node. More information about full binary trees can be found [here](#).

For Example :



To check whether a binary tree is a full binary tree we need to test the following cases:-

- 1) If a binary tree node is NULL then it is a full binary tree.
- 2) If a binary tree node does have empty left and right sub-trees, then it is a full binary tree by definition.

- 3) If a binary tree node has left and right sub-trees, then it is a part of a full binary tree by definition. In this case recursively check if the left and right sub-trees are also binary trees themselves.
- 4) In all other combinations of right and left sub-trees, the binary tree is not a full binary tree.

Following is the implementation for checking if a binary tree is a full binary tree.

C

```
// C program to check whether a given Binary Tree is full or not
#include<stdio.h>
#include<stdlib.h>
#include<stdbool.h>

/* Tree node structure */
struct Node
{
    int key;
    struct Node *left, *right;
};

/* Helper function that allocates a new node with the
   given key and NULL left and right pointer. */
struct Node *newNode(char k)
{
    struct Node *node = (struct Node*)malloc(sizeof(struct Node));
    node->key = k;
    node->right = node->left = NULL;
    return node;
}

/* This function tests if a binary tree is a full binary tree. */
bool isFullTree (struct Node* root)
{
    // If empty tree
    if (root == NULL)
        return true;

    // If leaf node
    if (root->left == NULL && root->right == NULL)
        return true;

    // If both left and right are not NULL, and left & right subtrees
    // are full
    if ((root->left) && (root->right))
        return (isFullTree(root->left) && isFullTree(root->right));

    // We reach here when none of the above if conditions work
```

```
    return false;
}

// Driver Program
int main()
{
    struct Node* root = NULL;
    root = newNode(10);
    root->left = newNode(20);
    root->right = newNode(30);

    root->left->right = newNode(40);
    root->left->left = newNode(50);
    root->right->left = newNode(60);
    root->right->right = newNode(70);

    root->left->left->left = newNode(80);
    root->left->left->right = newNode(90);
    root->left->right->left = newNode(80);
    root->left->right->right = newNode(90);
    root->right->left->left = newNode(80);
    root->right->left->right = newNode(90);
    root->right->right->left = newNode(80);
    root->right->right->right = newNode(90);

    if (isFullTree(root))
        printf("The Binary Tree is full\n");
    else
        printf("The Binary Tree is not full\n");

    return(0);
}
```

### Java

```
// Java program to check if binay tree is full or not

/* Tree node structure */
class Node
{
    int data;
    Node left, right;

    Node(int item)
    {
        data = item;
        left = right = null;
    }
}
```

```
}  
  
class BinaryTree  
{  
    Node root;  
  
    /* this function checks if a binary tree is full or not */  
    boolean isFullTree(Node node)  
    {  
        // if empty tree  
        if(node == null)  
            return true;  
  
        // if leaf node  
        if(node.left == null && node.right == null )  
            return true;  
  
        // if both left and right subtrees are not null  
        // the are full  
        if((node.left!=null) && (node.right!=null))  
            return (isFullTree(node.left) && isFullTree(node.right));  
  
        // if none work  
        return false;  
    }  
  
    // Driver program  
    public static void main(String args[])  
    {  
        BinaryTree tree = new BinaryTree();  
        tree.root = new Node(10);  
        tree.root.left = new Node(20);  
        tree.root.right = new Node(30);  
        tree.root.left.right = new Node(40);  
        tree.root.left.left = new Node(50);  
        tree.root.right.left = new Node(60);  
        tree.root.left.left.left = new Node(80);  
        tree.root.right.right = new Node(70);  
        tree.root.left.left.right = new Node(90);  
        tree.root.left.right.left = new Node(80);  
        tree.root.left.right.right = new Node(90);  
        tree.root.right.left.left = new Node(80);  
        tree.root.right.left.right = new Node(90);  
        tree.root.right.right.left = new Node(80);  
        tree.root.right.right.right = new Node(90);  
  
        if(tree.isFullTree(tree.root))
```

```
        System.out.print("The binary tree is full");
    else
        System.out.print("The binary tree is not full");
    }
}

// This code is contributed by Mayank Jaiswal
```

### Python

```
# Python program to check whether given Binary tree is full or not

# Tree node structure
class Node:

    # Constructor of the node class for creating the node
    def __init__(self , key):
        self.key = key
        self.left = None
        self.right = None

    # Checks if the binary tree is full or not
    def isFullTree(root):

        # If empty tree
        if root is None:
            return True

        # If leaf node
        if root.left is None and root.right is None:
            return True

        # If both left and right subtress are not None and
        # left and right subtress are full
        if root.left is not None and root.right is not None:
            return (isFullTree(root.left) and isFullTree(root.right))

        # We reach here when none of the above if conditiions work
        return False

# Driver Program
root = Node(10);
root.left = Node(20);
root.right = Node(30);

root.left.right = Node(40);
root.left.left = Node(50);
root.right.left = Node(60);
```

```
root.right.right = Node(70);

root.left.left.left = Node(80);
root.left.left.right = Node(90);
root.left.right.left = Node(80);
root.left.right.right = Node(90);
root.right.left.left = Node(80);
root.right.left.right = Node(90);
root.right.right.left = Node(80);
root.right.right.right = Node(90);

if isFullTree(root):
    print "The Binary tree is full"
else:
    print "Binary tree is not full"

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

**Output:**

The Binary Tree is full

Time complexity of the above code is  $O(n)$  where  $n$  is number of nodes in given binary tree.

This article is contributed by **Gaurav Gupta**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

**Improved By :** [mahaveer2407](#)

**Source**

<https://www.geeksforgeeks.org/check-whether-binary-tree-full-binary-tree-not/>

## Chapter 59

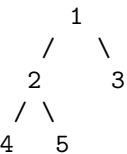
# Check whether a binary tree is a full binary tree or not | Iterative Approach

Check whether a binary tree is a full binary tree or not | Iterative Approach - GeeksforGeeks

Given a binary tree containing **n** nodes. The problem is to check whether the given binary tree is a full binary tree or not. A full binary tree is defined as a binary tree in which all nodes have either zero or two child nodes. Conversely, there is no node in a full binary tree, which has only one child node.

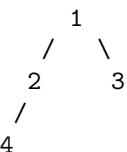
Examples:

Input :



Output : Yes

Input :



Output : No

**Approach:** In the [previous post](#) a recursive solution has been discussed. In this post an

iterative approach has been followed. Perform [iterative level order traversal](#) of the tree using queue. For each **node** encountered, follow the steps given below:

1. If (`node->left == NULL && node->right == NULL`), it is a leaf node. Discard it and start processing the next node from the queue.
2. If (`node->left == NULL || node->right == NULL`), then it means that only child of **node** is present. Return false as the binary tree is not a full binary tree.
3. Else, push the left and right child's of the **node** on to the queue.

If all the node's from the queue gets processed without returning false, then return true as the binary tree is a full binary tree.

```
// C++ implementation to check whether a binary
// tree is a full binary tree or not
#include <bits/stdc++.h>
using namespace std;

// structure of a node of binary tree
struct Node {
    int data;
    Node *left, *right;
};

// function to get a new node
Node* getNode(int data)
{
    // allocate space
    Node* newNode = (Node*)malloc(sizeof(Node));

    // put in the data
    newNode->data = data;
    newNode->left = newNode->right = NULL;
    return newNode;
}

// function to check whether a binary tree
// is a full binary tree or not
bool isFullBinaryTree(Node* root)
{
    // if tree is empty
    if (!root)
        return true;

    // queue used for level oder traversal
    queue<Node*> q;

    // push 'root' to 'q'
```

```
q.push(root);

// traverse all the nodes of the binary tree
// level by level until queue is empty
while (!q.empty()) {
    // get the pointer to 'node' at front
    // of queue
    Node* node = q.front();
    q.pop();

    // if it is a leaf node then continue
    if (node->left == NULL && node->right == NULL)
        continue;

    // if either of the child is not null and the
    // other one is null, then binary tree is not
    // a full binary tree
    if (node->left == NULL || node->right == NULL)
        return false;

    // push left and right childs of 'node'
    // on to the queue 'q'
    q.push(node->left);
    q.push(node->right);
}

// binary tree is a full binary tree
return true;
}

// Driver program to test above
int main()
{
    Node* root = getNode(1);
    root->left = getNode(2);
    root->right = getNode(3);
    root->left->left = getNode(4);
    root->left->right = getNode(5);

    if (isFullBinaryTree(root))
        cout << "Yes";
    else
        cout << "No";

    return 0;
}
```

Output:

**Yes**

Time Complexity:  $O(n)$ .

Auxiliary Space:  $O(\max)$ , where **max** is the maximum number of nodes at a particular level.

## **Source**

<https://www.geeksforgeeks.org/check-whether-binary-tree-full-binary-tree-not-iterative-approach/>

## Chapter 60

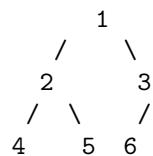
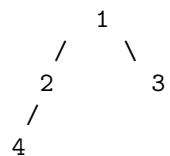
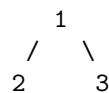
# Check whether a given Binary Tree is Complete or not | Set 1 (Iterative Solution)

Check whether a given Binary Tree is Complete or not | Set 1 (Iterative Solution) - Geeks-forGeeks

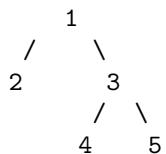
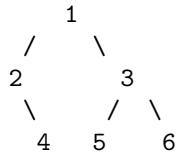
Given a Binary Tree, write a function to check whether the given Binary Tree is Complete Binary Tree or not.

A [complete binary tree](#) is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible. See following examples.

The following trees are examples of Complete Binary Trees



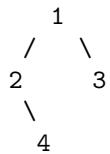
The following trees are examples of Non-Complete Binary Trees



The method 2 of [level order traversal post](#) can be easily modified to check whether a tree is Complete or not. To understand the approach, let us first define a term ‘Full Node’. A node is ‘Full Node’ if both left and right children are not empty (or not NULL).

The approach is to do a level order traversal starting from root. In the traversal, once a node is found which is NOT a Full Node, all the following nodes must be leaf nodes.

Also, one more thing needs to be checked to handle the below case: If a node has empty left child, then the right child must be empty.



Thanks to Guddu Sharma for suggesting this simple and efficient approach.

**C/C++**

```
// A program to check if a given binary tree is complete or not
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#define MAX_Q_SIZE 500

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
    int data;
    struct node* left;
```

```
    struct node* right;
};

/* function prototypes for functions needed for Queue data
   structure. A queue is needed for level order traversal */
struct node** createQueue(int *, int *);
void enQueue(struct node **, int *, struct node *);
struct node *deQueue(struct node **, int *);
bool isQueueEmpty(int *front, int *rear);

/* Given a binary tree, return true if the tree is complete
   else false */
bool isCompleteBT(struct node* root)
{
    // Base Case: An empty tree is complete Binary Tree
    if (root == NULL)
        return true;

    // Create an empty queue
    int rear, front;
    struct node **queue = createQueue(&front, &rear);

    // Create a flag variable which will be set true
    // when a non full node is seen
    bool flag = false;

    // Do level order traversal using queue.
    enQueue(queue, &rear, root);
    while(!isQueueEmpty(&front, &rear))
    {
        struct node *temp_node = deQueue(queue, &front);

        /* Check if left child is present*/
        if(temp_node->left)
        {
            // If we have seen a non full node, and we see a node
            // with non-empty left child, then the given tree is not
            // a complete Binary Tree
            if (flag == true)
                return false;

            enQueue(queue, &rear, temp_node->left); // Enqueue Left Child
        }
        else // If this a non-full node, set the flag as true
        flag = true;

        /* Check if right child is present*/
        if(temp_node->right)
```

```
{  
    // If we have seen a non full node, and we see a node  
    // with non-empty right child, then the given tree is not  
    // a complete Binary Tree  
    if(flag == true)  
        return false;  
  
    enQueue(queue, &rear, temp_node->right); // Enqueue Right Child  
}  
else // If this a non-full node, set the flag as true  
    flag = true;  
}  
  
// If we reach here, then the tree is complete Bianry Tree  
return true;  
}  
  
/*UTILITY FUNCTIONS*/  
struct node** createQueue(int *front, int *rear)  
{  
    struct node **queue =  
        (struct node **)malloc(sizeof(struct node*)*MAX_Q_SIZE);  
  
    *front = *rear = 0;  
    return queue;  
}  
  
void enQueue(struct node **queue, int *rear, struct node *new_node)  
{  
    queue[*rear] = new_node;  
    (*rear)++;  
}  
  
struct node *deQueue(struct node **queue, int *front)  
{  
    (*front)++;  
    return queue[*front - 1];  
}  
  
bool isQueueEmpty(int *front, int *rear)  
{  
    return (*rear == *front);  
}  
  
/* Helper function that allocates a new node with the  
   given data and NULL left and right pointers. */  
struct node* newNode(int data)
```

```
{  
    struct node* node = (struct node*)  
        malloc(sizeof(struct node));  
    node->data = data;  
    node->left = NULL;  
    node->right = NULL;  
  
    return(node);  
}  
  
/* Driver program to test above functions*/  
int main()  
{  
    /* Let us construct the following Binary Tree which  
       is not a complete Binary Tree  
           1  
          /   \  
         2     3  
        / \   \\  
       4   5   6  
    */  
  
    struct node *root = newNode(1);  
    root->left = newNode(2);  
    root->right = newNode(3);  
    root->left->left = newNode(4);  
    root->left->right = newNode(5);  
    root->right->right = newNode(6);  
  
    if (isCompleteBT(root) == true)  
        printf ("Complete Binary Tree");  
    else  
        printf ("NOT Complete Binary Tree");  
  
    return 0;  
}
```

### Java

```
//A Java program to check if a given binary tree is complete or not  
  
import java.util.LinkedList;  
import java.util.Queue;  
  
public class CompleteBTree  
{  
    /* A binary tree node has data, pointer to left child  
       and a pointer to right child */
```

```
static class Node
{
    int data;
    Node left;
    Node right;

    // Constructor
    Node(int d)
    {
        data = d;
        left = null;
        right = null;
    }
}

/* Given a binary tree, return true if the tree is complete
   else false */
static boolean isCompleteBT(Node root)
{
    // Base Case: An empty tree is complete Binary Tree
    if(root == null)
        return true;

    // Create an empty queue
    Queue<Node> queue =new LinkedList<>();

    // Create a flag variable which will be set true
    // when a non full node is seen
    boolean flag = false;

    // Do level order traversal using queue.
    queue.add(root);
    while(!queue.isEmpty())
    {
        Node temp_node = queue.remove();

        /* Check if left child is present*/
        if(temp_node.left != null)
        {
            // If we have seen a non full node, and we see a node
            // with non-empty left child, then the given tree is not
            // a complete Binary Tree
            if(flag == true)
                return false;

            // Enqueue Left Child
            queue.add(temp_node.left);
        }
    }
}
```

```
// If this a non-full node, set the flag as true
else
    flag = true;

/* Check if right child is present*/
if(temp_node.right != null)
{
    // If we have seen a non full node, and we see a node
    // with non-empty right child, then the given tree is not
    // a complete Binary Tree
    if(flag == true)
        return false;

    // Enqueue Right Child
    queue.add(temp_node.right);

}
// If this a non-full node, set the flag as true
else
    flag = true;
}
// If we reach here, then the tree is complete Bianry Tree
return true;
}

/* Driver program to test above functions*/
public static void main(String[] args)
{

/* Let us construct the following Binary Tree which
   is not a complete Binary Tree
      1
     / \
    2   3
   / \   \
  4   5   6
*/
Node root = new Node(1);
root.left = new Node(2);
root.right = new Node(3);
root.left.left = new Node(4);
root.left.right = new Node(5);
root.right.right = new Node(6);

if(isCompleteBT(root) == true)
    System.out.println("Complete Binary Tree");
else
```

```
        System.out.println("NOT Complete Binary Tree");
    }

}

//This code is contributed by Sumit Ghosh
```

### Python

```
# Check whether binary tree is complete or not

# A binary tree node
class Node:

    # Constructor to create a new node
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

    # Given a binary tree, return true if the tree is complete
    # else return false
    def isCompleteBT(root):

        # Base Case: An empty tree is complete Binary tree
        if root is None:
            return True

        # Create an empty queue
        queue = []

        # Create a flag variable which will be set True
        # when a non full node is seen
        flag = False

        # Do level order traversal using queue
        queue.append(root)
        while(len(queue) > 0):
            tempNode = queue.pop(0) # Dequeue

            # Check if left child is present
            if (tempNode.left):

                # If we have seen a non full node, and we see
                # a node with non-empty left child, then the
                # given tree is not a complete binary tree
                if flag == True :
                    return False

                flag = True
```

```
# Enqueue left child
queue.append(tempNode.left)

# If this a non-full node, set the flag as true
else:
    flag = True

# Check if right child is present
if(tempNode.right):

    # If we have seen a non full node, and we
    # see a node with non-empty right child, then
    # the given tree is not a complete BT
    if flag == True:
        return False

    # Enqueue right child
    queue.append(tempNode.right)

# If this is non-full node, set the flag as True
else:
    flag = True

# If we reach here, then the tree is complete BT
return True

# Driver program to test above function

""" Let us construct the following Binary Tree which
is not a complete Binary Tree
      1
     /   \
    2     3
   / \   /
  4   5   6
"""
root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)
root.right.right = Node(6)

if (isCompleteBT(root)):
    print "Complete Binary Tree"
else:
    print "NOT Complete Binary Tree"
```

```
# This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

Output:

```
NOT Complete Binary Tree
```

*Time Complexity:*  $O(n)$  where n is the number of nodes in given Binary Tree

*Auxiliary Space:*  $O(n)$  for queue.

## Source

<https://www.geeksforgeeks.org/check-if-a-given-binary-tree-is-complete-tree-or-not/>

## Chapter 61

# Check whether a given binary tree is perfect or not

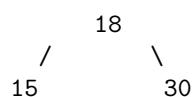
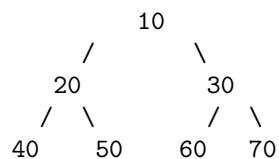
Check whether a given binary tree is perfect or not - GeeksforGeeks

Given a Binary Tree, write a function to check whether the given Binary Tree is a perfect Binary Tree or not.

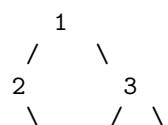
A Binary tree is [Perfect Binary Tree](#) in which all internal nodes have two children and all leaves are at same level.

Examples:

The following tree is a perfect binary tree



The following tree is **not** a perfect binary tree



4    5    6

A **Perfect Binary Tree** of height h (where height is number of nodes on path from root to leaf) has  $2^h - 1$  nodes.

Below is an idea to check whether a given Binary Tree is perfect or not.

1. Find depth of any node (in below tree we find depth of leftmost node). Let this depth be d.
2. Now recursively traverse the tree and check for following two conditions.
  - Every internal node should have both children non-empty
  - All leaves are at depth 'd'

```
// C program to check whether a given
// Binary Tree is Perfect or not
#include<bits/stdc++.h>

/* Tree node structure */
struct Node
{
    int key;
    struct Node *left, *right;
};

// Returns depth of leftmost leaf.
int findADepth(Node *node)
{
    int d = 0;
    while (node != NULL)
    {
        d++;
        node = node->left;
    }
    return d;
}

/* This function tests if a binary tree is perfect
   or not. It basically checks for two things :
   1) All leaves are at same level
   2) All internal nodes have two children */
bool isPerfectRec(struct Node* root, int d, int level = 0)
{
    // An empty tree is perfect
    if (root == NULL)
        return true;

    // If this is a leaf node and it is at
    // level 'd' then return true, else
    // return false
    if (root->left == NULL && root->right == NULL)
        return (level == d);

    // If this is an internal node and
    // both of its children are not NULL
    // and both of them are perfect
    // then return true, else return false
    if (root->left != NULL && root->right != NULL)
        return (isPerfectRec(root->left, d + 1, level + 1) && isPerfectRec(root->right, d + 1, level + 1));

    // If this is an internal node and
    // either one of its children is
    // not perfect or both are perfect
    // but not at same level then
    // return false
    return false;
}
```

```
// If leaf node, then its depth must be same as
// depth of all other leaves.
if (root->left == NULL && root->right == NULL)
    return (d == level+1);

// If internal node and one child is empty
if (root->left == NULL || root->right == NULL)
    return false;

// Left and right subtrees must be perfect.
return isPerfectRec(root->left, d, level+1) &&
       isPerfectRec(root->right, d, level+1);
}

// Wrapper over isPerfectRec()
bool isPerfect(Node *root)
{
    int d = findADepth(root);
    return isPerfectRec(root, d);
}

/* Helper function that allocates a new node with the
   given key and NULL left and right pointer. */
struct Node *newNode(int k)
{
    struct Node *node = new Node;
    node->key = k;
    node->right = node->left = NULL;
    return node;
}

// Driver Program
int main()
{
    struct Node* root = NULL;
    root = newNode(10);
    root->left = newNode(20);
    root->right = newNode(30);

    root->left->left = newNode(40);
    root->left->right = newNode(50);
    root->right->left = newNode(60);
    root->right->right = newNode(70);

    if (isPerfect(root))
        printf("Yes\n");
    else
        printf("No\n");
}
```

```
    return(0);  
}
```

Output:

**Yes**

Time complexity :  $O(n)$

### Source

<https://www.geeksforgeeks.org/check-weather-given-binary-tree-perfect-not/>

## Chapter 62

# Clone a Binary Tree with Random Pointers

Clone a Binary Tree with Random Pointers - GeeksforGeeks

Given a Binary Tree where every node has following structure.

```
struct node {  
    int key;  
    struct node *left,*right,*random;  
}
```

The random pointer points to any random node of the binary tree and can even point to NULL, clone the given binary tree.

### Method 1 (Use Hashing)

The idea is to store mapping from given tree nodes to clone tree node in hashtable. Following are detailed steps.

- 1) Recursively traverse the given Binary and copy key value, left pointer and right pointer to clone tree. While copying, store the mapping from given tree node to clone tree node in a hashtable. In the following pseudo code, ‘cloneNode’ is currently visited node of clone tree and ‘treeNode’ is currently visited node of given tree.

```
cloneNode->key = treeNode->key  
cloneNode->left = treeNode->left  
cloneNode->right = treeNode->right  
map[treeNode] = cloneNode
```

- 2) Recursively traverse both trees and set random pointers using entries from hash table.

```
cloneNode->random = map[treeNode->random]
```

Following is C++ implementation of above idea. The following implementation uses `map` from C++ STL. Note that `map` doesn't implement hash table, it actually is based on self-balancing binary search tree.

```
// A hashmap based C++ program to clone a binary tree with random pointers
#include<iostream>
#include<map>
using namespace std;

/* A binary tree node has data, pointer to left child, a pointer to right
   child and a pointer to random node*/
struct Node
{
    int key;
    struct Node* left, *right, *random;
};

/* Helper function that allocates a new Node with the
   given data and NULL left, right and random pointers. */
Node* newNode(int key)
{
    Node* temp = new Node;
    temp->key = key;
    temp->random = temp->right = temp->left = NULL;
    return (temp);
}

/* Given a binary tree, print its Nodes in inorder*/
void printInorder(Node* node)
{
    if (node == NULL)
        return;

    /* First recur on left subtree */
    printInorder(node->left);

    /* then print data of Node and its random */
    cout << "[" << node->key << " ";
    if (node->random == NULL)
        cout << "NULL], ";
    else
        cout << node->random->key << "], ";

    /* now recur on right subtree */
    printInorder(node->right);
```

```

}

// This function creates clone by copying key and left and right pointers
// This function also stores mapping from given tree node to clone.
Node* copyLeftRightNode(Node* treeNode, map<Node *, Node *> *mymap)
{
    if (treeNode == NULL)
        return NULL;
    Node* cloneNode = newNode(treeNode->key);
    (*mymap)[treeNode] = cloneNode;
    cloneNode->left = copyLeftRightNode(treeNode->left, mymap);
    cloneNode->right = copyLeftRightNode(treeNode->right, mymap);
    return cloneNode;
}

// This function copies random node by using the hashmap built by
// copyLeftRightNode()
void copyRandom(Node* treeNode, Node* cloneNode, map<Node *, Node *> *mymap)
{
    if (cloneNode == NULL)
        return;
    cloneNode->random = (*mymap)[treeNode->random];
    copyRandom(treeNode->left, cloneNode->left, mymap);
    copyRandom(treeNode->right, cloneNode->right, mymap);
}

// This function makes the clone of given tree. It mainly uses
// copyLeftRightNode() and copyRandom()
Node* cloneTree(Node* tree)
{
    if (tree == NULL)
        return NULL;
    map<Node *, Node *> *mymap = new map<Node *, Node *>;
    Node* newTree = copyLeftRightNode(tree, mymap);
    copyRandom(tree, newTree, mymap);
    return newTree;
}

/* Driver program to test above functions*/
int main()
{
    //Test No 1
    Node *tree = newNode(1);
    tree->left = newNode(2);
    tree->right = newNode(3);
    tree->left->left = newNode(4);
    tree->left->right = newNode(5);
    tree->random = tree->left->right;
}

```

```

tree->left->left->random = tree;
tree->left->right->random = tree->right;

// Test No 2
// tree = NULL;

// Test No 3
// tree = newNode(1);

// Test No 4
/* tree = newNode(1);
tree->left = newNode(2);
tree->right = newNode(3);
tree->random = tree->right;
tree->left->random = tree;
*/
cout << "Inorder traversal of original binary tree is: \n";
printInorder(tree);

Node *clone = cloneTree(tree);

cout << "\n\nInorder traversal of cloned binary tree is: \n";
printInorder(clone);

return 0;
}

```

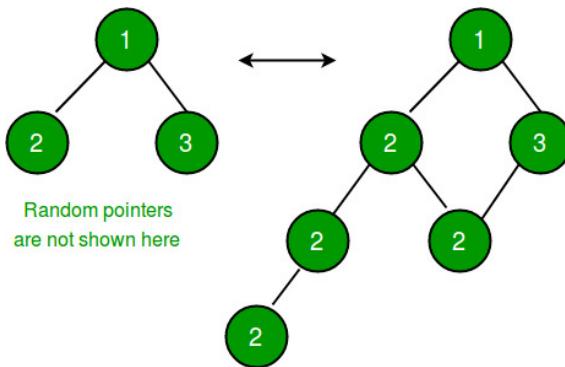
Output:

```
Inorder traversal of original binary tree is:
[4 1], [2 NULL], [5 3], [1 5], [3 NULL],
```

```
Inorder traversal of cloned binary tree is:
[4 1], [2 NULL], [5 3], [1 5], [3 NULL],
```

### Method 2 (Temporarily Modify the Given Binary Tree)

1. Create new nodes in cloned tree and insert each new node in original tree between the left pointer edge of corresponding node in the original tree (See the below image).  
i.e. if current node is A and it's left child is B ( A — >> B ), then new cloned node with key A will be created (say cA) and it will be put as A — >> cA — >> B (B can be a NULL or a non-NULL left child). Right child pointer will be set correctly i.e. if for current node A, right child is C in original tree (A — >> C) then corresponding cloned nodes cA and cC will like cA — >> cC



2. Set random pointer in cloned tree as per original tree  
i.e. if node A's random pointer points to node B, then in cloned tree, cA will point to cB (cA and cB are new node in cloned tree corresponding to node A and B in original tree)

3. Restore left pointers correctly in both original and cloned tree

Following is C++ implementation of above algorithm.

```
#include <iostream>
using namespace std;

/* A binary tree node has data, pointer to left child, a pointer to right
   child and a pointer to random node*/
struct Node
{
    int key;
    struct Node* left, *right, *random;
};

/* Helper function that allocates a new Node with the
   given data and NULL left, right and random pointers. */
Node* newNode(int key)
{
    Node* temp = new Node;
    temp->key = key;
    temp->random = temp->right = temp->left = NULL;
    return (temp);
}

/* Given a binary tree, print its Nodes in inorder*/
void printInorder(Node* node)
{
    if (node == NULL)
        return;

    /* First recur on left subtree */
    printInorder(node->left);

    /* Print this node */
    cout << node->key << " ";
    cout << node->random->key << endl;

    /* Recur on right subtree */
    printInorder(node->right);
}
```

```

/* then print data of Node and its random */
cout << "[" << node->key << " ";
if (node->random == NULL)
    cout << "NULL], ";
else
    cout << node->random->key << "], ";

/* now recur on right subtree */
printInorder(node->right);
}

// This function creates new nodes cloned tree and puts new cloned node
// in between current node and it's left child
// i.e. if current node is A and it's left child is B ( A --- >> B ),
//       then new cloned node with key A wil be created (say cA) and
//       it will be put as
//       A --- >> cA --- >> B
// Here B can be a NULL or a non-NULL left child
// Right child pointer will be set correctly
// i.e. if for current node A, right child is C in original tree
// (A --- >> C) then corresponding cloned nodes cA and cC will like
// cA ----- >> cC
Node* copyLeftRightNode(Node* treeNode)
{
    if (treeNode == NULL)
        return NULL;

    Node* left = treeNode->left;
    treeNode->left = newNode(treeNode->key);
    treeNode->left->left = left;
    if(left != NULL)
        left->left = copyLeftRightNode(left);

    treeNode->left->right = copyLeftRightNode(treeNode->right);
    return treeNode->left;
}

// This function sets random pointer in cloned tree as per original tree
// i.e. if node A's random pointer points to node B, then
// in cloned tree, cA wil point to cB (cA and cB are new node in cloned
// tree corresponding to node A and B in original tree)
void copyRandomNode(Node* treeNode, Node* cloneNode)
{
    if (treeNode == NULL)
        return;
    if(treeNode->random != NULL)
        cloneNode->random = treeNode->random->left;
}

```

```

        else
            cloneNode->random = NULL;

        if(treeNode->left != NULL && cloneNode->left != NULL)
            copyRandomNode(treeNode->left->left, cloneNode->left->left);
            copyRandomNode(treeNode->right, cloneNode->right);
    }

    // This function will restore left pointers correctly in
    // both original and cloned tree
    void restoreTreeLeftNode(Node* treeNode, Node* cloneNode)
    {
        if (treeNode == NULL)
            return;
        if (cloneNode->left != NULL)
        {
            Node* cloneLeft = cloneNode->left->left;
            treeNode->left = treeNode->left->left;
            cloneNode->left = cloneLeft;
        }
        else
            treeNode->left = NULL;

        restoreTreeLeftNode(treeNode->left, cloneNode->left);
        restoreTreeLeftNode(treeNode->right, cloneNode->right);
    }

    //This function makes the clone of given tree
    Node* cloneTree(Node* treeNode)
    {
        if (treeNode == NULL)
            return NULL;
        Node* cloneNode = copyLeftRightNode(treeNode);
        copyRandomNode(treeNode, cloneNode);
        restoreTreeLeftNode(treeNode, cloneNode);
        return cloneNode;
    }

    /* Driver program to test above functions*/
    int main()
    {
        /* //Test No 1
        Node *tree = newNode(1);
        tree->left = newNode(2);
        tree->right = newNode(3);
        tree->left->left = newNode(4);
        tree->left->right = newNode(5);
```

```
tree->random = tree->left->right;
tree->left->left->random = tree;
tree->left->right->random = tree->right;

// Test No 2
//     Node *tree = NULL;
/*
// Test No 3
Node *tree = newNode(1);

// Test No 4
Node *tree = newNode(1);
tree->left = newNode(2);
tree->right = newNode(3);
tree->random = tree->right;
tree->left->random = tree;

Test No 5
Node *tree = newNode(1);
tree->left = newNode(2);
tree->right = newNode(3);
tree->left->left = newNode(4);
tree->left->right = newNode(5);
tree->right->left = newNode(6);
tree->right->right = newNode(7);
tree->random = tree->left;
*/
// Test No 6
Node *tree = newNode(10);
Node *n2 = newNode(6);
Node *n3 = newNode(12);
Node *n4 = newNode(5);
Node *n5 = newNode(8);
Node *n6 = newNode(11);
Node *n7 = newNode(13);
Node *n8 = newNode(7);
Node *n9 = newNode(9);
tree->left = n2;
tree->right = n3;
tree->random = n2;
n2->left = n4;
n2->right = n5;
n2->random = n8;
n3->left = n6;
n3->right = n7;
n3->random = n5;
n4->random = n9;
n5->left = n8;
```

```
n5->right = n9;
n5->random = tree;
n6->random = n9;
n9->random = n8;

/*      Test No 7
Node *tree = newNode(1);
tree->left = newNode(2);
tree->right = newNode(3);
tree->left->random = tree;
tree->right->random = tree->left;
*/
cout << "Inorder traversal of original binary tree is: \n";
printInorder(tree);

Node *clone = cloneTree(tree);

cout << "\n\nInorder traversal of cloned binary tree is: \n";
printInorder(clone);

return 0;
}
```

Output:

```
Inorder traversal of original binary tree is:
[5 9], [6 7], [7 NULL], [8 10], [9 7], [10 6], [11 9], [12 8], [13 NULL], 

Inorder traversal of cloned binary tree is:
[5 9], [6 7], [7 NULL], [8 10], [9 7], [10 6], [11 9], [12 8], [13 NULL],
```

This article is contributed by **Anurag Singh**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## Source

<https://www.geeksforgeeks.org/clone-binary-tree-random-pointers/>

## Chapter 63

# Closest leaf to a given node in Binary Tree

Closest leaf to a given node in Binary Tree - GeeksforGeeks

Given a Binary Tree and a node  $x$  in it, find distance of the closest leaf to  $x$  in Binary Tree. If given node itself is a leaf, then distance is 0.

Examples:

```
Input: Root of below tree
      And x = pointer to node 13
          10
         /   \
        12   13
           /
          14
Output 1
Distance 1. Closest leaf is 14.
```

```
Input: Root of below tree
      And x = pointer to node 13
          10
         /   \
        12   13
           /   \
          14   15
         / \   / \
        21 22 23 24
       / \ / \
      1 2 3 4 5 6 7 8
```

```
Output 2
Closest leaf is 12 through 10.
```

We strongly recommend you to minimize your browser and try this yourself first.

The idea is to first traverse the subtree rooted with give node and find the closest leaf in this subtree. Store this distance. Now traverse tree starting from root. If given node x is in left subtree of root, then find the closest leaf in right subtree, else find the closest left in left subtree. Below is C++ implementation of this idea.

C++

```
/* Find closest leaf to the given node x in a tree */
#include<bits/stdc++.h>
using namespace std;

// A Tree node
struct Node
{
    int key;
    struct Node* left, *right;
};

// Utility function to create a new node
Node* newNode(int key)
{
    Node* temp = new Node;
    temp->key = key;
    temp->left = temp->right = NULL;
    return (temp);
}

// This function finds closest leaf to root. This distance
// is stored at *minDist.
void findLeafDown(Node *root, int lev, int *minDist)
{
    // base case
    if (root == NULL)
        return ;

    // If this is a leaf node, then check if it is closer
    // than the closest so far
    if (root->left == NULL && root->right == NULL)
    {
        if (lev < (*minDist))
            *minDist = lev;
    }
}
```

```
        return;
    }

    // Recur for left and right subtrees
    findLeafDown(root->left, lev+1, minDist);
    findLeafDown(root->right, lev+1, minDist);
}

// This function finds if there is closer leaf to x through
// parent node.
int findThroughParent(Node * root, Node *x, int *minDist)
{
    // Base cases
    if (root == NULL) return -1;
    if (root == x) return 0;

    // Search x in left subtree of root
    int l = findThroughParent(root->left, x, minDist);

    // If left subtree has x
    if (l != -1)
    {
        // Find closest leaf in right subtree
        findLeafDown(root->right, l+2, minDist);
        return l+1;
    }

    // Search x in right subtree of root
    int r = findThroughParent(root->right, x, minDist);

    // If right subtree has x
    if (r != -1)
    {
        // Find closest leaf in left subtree
        findLeafDown(root->left, r+2, minDist);
        return r+1;
    }

    return -1;
}

// Returns minimum distance of a leaf from given node x
int minimumDistance(Node *root, Node *x)
{
    // Initialize result (minimum distance from a leaf)
    int minDist = INT_MAX;

    // Find closest leaf down to x
```

```
findLeafDown(x, 0, &minDist);

// See if there is a closer leaf through parent
findThroughParent(root, x, &minDist);

return minDist;
}

// Driver program
int main ()
{
    // Let us create Binary Tree shown in above example
    Node *root = newNode(1);
    root->left = newNode(12);
    root->right = newNode(13);

    root->right->left = newNode(14);
    root->right->right = newNode(15);

    root->right->left->left = newNode(21);
    root->right->left->right = newNode(22);
    root->right->right->left = newNode(23);
    root->right->right->right = newNode(24);

    root->right->left->left->left = newNode(1);
    root->right->left->left->right = newNode(2);
    root->right->left->right->left = newNode(3);
    root->right->left->right->right = newNode(4);
    root->right->right->left->left = newNode(5);
    root->right->right->left->right = newNode(6);
    root->right->right->right->left = newNode(7);
    root->right->right->right->right = newNode(8);

    Node *x = root->right;

    cout << "The closest leaf to the node with value "
        << x->key << " is at a distance of "
        << minimumDistance(root, x) << endl;

    return 0;
}
```

### Java

```
// Java program to find closest leaf to given node x in a tree

// A binary tree node
class Node
```

```
{  
    int key;  
    Node left, right;  
  
    public Node(int key)  
    {  
        this.key = key;  
        left = right = null;  
    }  
}  
  
class Distance  
{  
    int minDis = Integer.MAX_VALUE;  
}  
  
class BinaryTree  
{  
    Node root;  
  
    // This function finds closest leaf to root. This distance  
    // is stored at *minDist.  
    void findLeafDown(Node root, int lev, Distance minDist)  
    {  
  
        // base case  
        if (root == null)  
            return;  
  
        // If this is a leaf node, then check if it is closer  
        // than the closest so far  
        if (root.left == null && root.right == null)  
        {  
            if (lev < (minDist.minDis))  
                minDist.minDis = lev;  
  
            return;  
        }  
  
        // Recur for left and right subtrees  
        findLeafDown(root.left, lev + 1, minDist);  
        findLeafDown(root.right, lev + 1, minDist);  
    }  
  
    // This function finds if there is closer leaf to x through  
    // parent node.  
    int findThroughParent(Node root, Node x, Distance minDist)  
    {
```

```
// Base cases
if (root == null)
    return -1;

if (root == x)
    return 0;

// Search x in left subtree of root
int l = findThroughParent(root.left, x, minDist);

// If left subtree has x
if (l != -1)
{
    // Find closest leaf in right subtree
    findLeafDown(root.right, l + 2, minDist);
    return l + 1;
}

// Search x in right subtree of root
int r = findThroughParent(root.right, x, minDist);

// If right subtree has x
if (r != -1)
{
    // Find closest leaf in left subtree
    findLeafDown(root.left, r + 2, minDist);
    return r + 1;
}

return -1;
}

// Returns minimum distance of a leaf from given node x
int minimumDistance(Node root, Node x)
{
    // Initialize result (minimum distance from a leaf)
    Distance d = new Distance();

    // Find closest leaf down to x
    findLeafDown(x, 0, d);

    // See if there is a closer leaf through parent
    findThroughParent(root, x, d);

    return d.minDis;
}

// Driver program
```

```
public static void main(String[] args)
{
    BinaryTree tree = new BinaryTree();

    // Let us create Binary Tree shown in above example
    tree.root = new Node(1);
    tree.root.left = new Node(12);
    tree.root.right = new Node(13);

    tree.root.right.left = new Node(14);
    tree.root.right.right = new Node(15);

    tree.root.right.left.left = new Node(21);
    tree.root.right.left.right = new Node(22);
    tree.root.right.right.left = new Node(23);
    tree.root.right.right.right = new Node(24);

    tree.root.right.left.left.left = new Node(1);
    tree.root.right.left.left.right = new Node(2);
    tree.root.right.left.right.left = new Node(3);
    tree.root.right.left.right.right = new Node(4);
    tree.root.right.right.left.left = new Node(5);
    tree.root.right.right.left.right = new Node(6);
    tree.root.right.right.right.left = new Node(7);
    tree.root.right.right.right.right = new Node(8);

    Node x = tree.root.right;

    System.out.println("The closest leaf to node with value "
        + x.key + " is at a distance of "
        + tree.minimumDistance(tree.root, x));
}

// This code has been contributed by mayank_24
```

Output:

```
The closest leaf to the node with value 13 is at a distance of 2
```

Time Complexity of this above solution is  $O(n)$  as it does at most two traversals of given Binary Tree.

This article is contributed by [Ekta Goel](#). Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

**Source**

<https://www.geeksforgeeks.org/closest-leaf-to-a-given-node-in-binary-tree/>

## Chapter 64

# Combinatorics on ordered trees

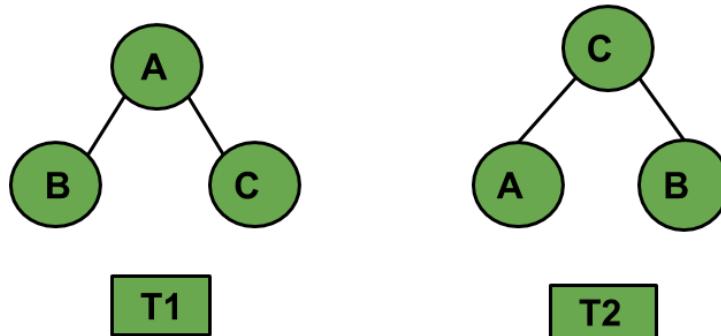
Combinatorics on ordered trees - GeeksforGeeks

An **ordered tree** is an oriented tree in which the children of a node are somehow ordered. It is a rooted tree in which an ordering is specified for the children of each vertex. This is called a “plane tree” because an ordering of the children is equivalent to an embedding of the tree in the plane, with the root at the top and the children of each vertex lower than that vertex.

Ordered tree can be further specified as labelled ordered tree and unlabelled ordered tree.

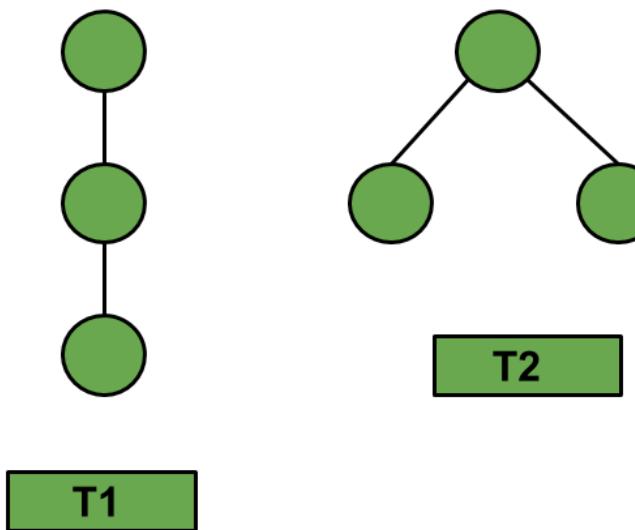
**Prerequisite :** [Catalan Numbers | Binomial Coefficient](#).

**Labelled ordered trees :** A labeled tree is a tree where each vertex is assigned a unique number from 1 to n.



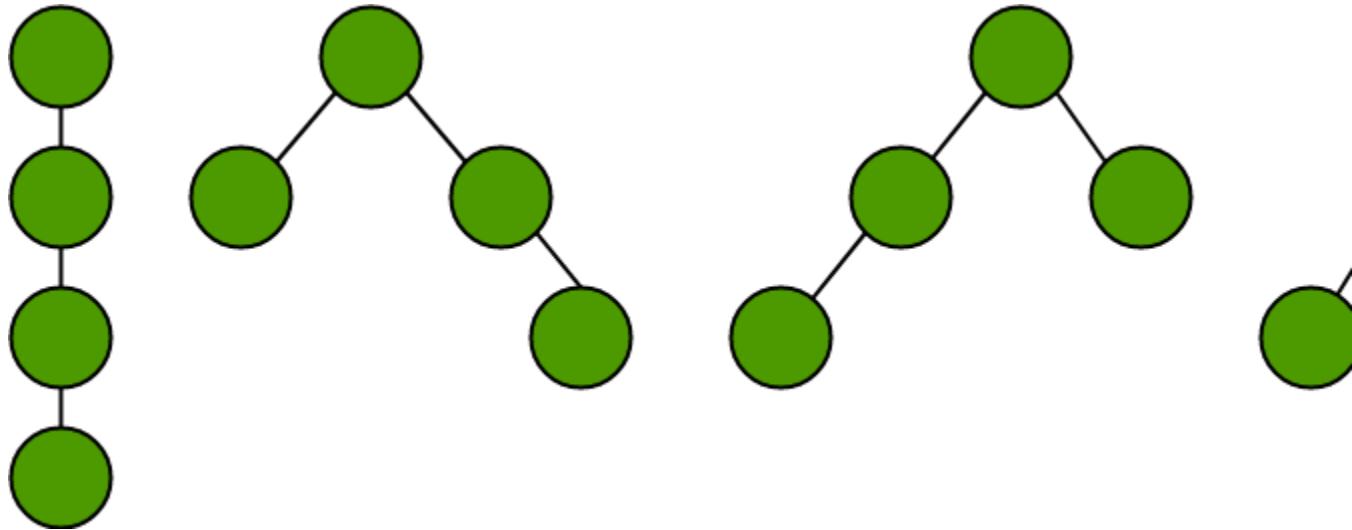
If T1 and T2 are ordered trees. Then,  $T1 \neq T2$  else  $T1 = T2$ .

**Unlabelled ordered trees :** An unlabelled tree is a tree where every vertex is unlabelled. Given below are the possible unlabelled ordered tree having 3 vertices.



The total number of unlabelled ordered trees having  $n$  nodes is equal to the  $(n - 1)$ -th [Catalan Number](#).

Given below are the possible unlabelled ordered trees having 4 nodes. This diagram will work as a reference example for the next few results.



#### 1. Number of trees with exactly $k$ leaves.

Let us consider, we have a ' $n$ ' edges . Then, the solution for the total possible ordered trees having ' $k$ ' leaves is given by :

$$L_n(k) = \frac{1}{n} \binom{n}{k} \binom{n}{k-1}$$

**2. Total number of nodes of degree d in these trees.**

Let us consider, we have a ‘n’ edges . Then, the solution for the total number of nodes having degree ‘d’ is given by :

$$D_n(d) = \binom{2n - 1 - d}{n - 1}$$

**3. Number of trees in which the root has degree r.**

Let us consider, we have a ‘n’ edges . Then, the solution for the total possible ordered trees whose root has degree ‘r’ is given by :

$$R_n(r) = \frac{r}{n} \binom{2n - 1 - r}{n - 1}$$

Below is the implementation of above combinatorics functions using Binomial Coefficient :

C++

```
// CPP code to find the number of ordered trees
// with given number of edges and leaves
#include <bits/stdc++.h>
using namespace std;

// Function returns value of
// Binomial Coefficient C(n, k)
int binomialCoeff(int n, int k)
{
    int C[n + 1][k + 1] = { 0 };
    for (int i = 0; i <= n; i++)
        C[i][0] = 1; // NCr(i, 0) is 1 for all i
```

```

int i, j;

// Calculate value of Binomial
// Coefficient in bottom up manner
for (i = 0; i <= n; i++) {
    for (j = 0; j <= min(i, k); j++) {

        // Base Cases
        if (j == 0 || j == i)
            C[i][j] = 1;

        // Calculate value using
        // previously stored values
        else
            C[i][j] = C[i - 1][j - 1] + C[i - 1][j];
    }
}

return C[n][k];
}

// Function to calculate the number
// of trees with exactly k leaves.
int k_Leaves(int n, int k)
{
    int ans = (binomialCoeff(n, k) * binomialCoeff(n, k - 1)) / n;
    cout << "Number of trees having 4 edges"
        << " and exactly 2 leaves : " << ans << endl;
    return 0;
}

// Function to calculate total number of
// nodes of degree d in these trees.
int number0fNodes(int n, int d)
{
    int ans = binomialCoeff(2 * n - 1 - d, n - 1);
    cout << "Number of nodes of degree 1 in"
        << " a tree having 4 edges : " << ans << endl;
    return 0;
}

// Function to calculate the number of
// trees in which the root has degree r.
int rootDegreeR(int n, int r)
{
    int ans = r * binomialCoeff(2 * n - 1 - r, n - 1);
    ans = ans / n;
    cout << "Number of trees having 4 edges"

```

```
    << " where root has degree 2 : " << ans << endl;
    return 0;
}

// Driver program to test above functions
int main()
{
    // Number of trees having 3
    // edges and exactly 2 leaves
    k_Leaves(3, 2);

    // Number of nodes of degree
    // 3 in a tree having 4 edges
    numberOfNodes(3, 1);

    // Number of trees having 3
    // edges where root has degree 2
    rootDegreeR(3, 2);

    return 0;
}
```

### Java

```
// java code to find the number of ordered
// trees with given number of edges and
// leaves
import java.io.*;

class GFG {

    // Function returns value of
    // Binomial Coefficient C(n, k)
    static int binomialCoeff(int n, int k)
    {
        int [][]C = new int[n+1] [k+1];
        int i, j;

        // Calculate value of Binomial
        // Coefficient in bottom up manner
        for (i = 0; i <= n; i++) {
            for (j = 0; j <= Math.min(i, k); j++)
            {

                // Base Cases
                if (j == 0 || j == i)
                    C[i][j] = 1;
```

```

// Calculate value using
// previously stored values
else
    C[i][j] = C[i - 1][j - 1]
        + C[i - 1][j];
}
}

return C[n][k];
}

// Function to calculate the number
// of trees with exactly k leaves.
static int k_Leaves(int n, int k)
{
    int ans = (binomialCoeff(n, k) *
               binomialCoeff(n, k - 1)) / n;
    System.out.println("Number of trees "
        + "having 4 edges and exactly 2 "
        + "leaves : " + ans);
    return 0;
}

// Function to calculate total number of
// nodes of degree d in these trees.
static int numberOfNodes(int n, int d)
{
    int ans = binomialCoeff(2 * n - 1 - d,
                           n - 1);
    System.out.println("Number of nodes "
        + "of degree 1 in a tree having 4 "
        + "edges : " + ans);
    return 0;
}

// Function to calculate the number of
// trees in which the root has degree r.
static int rootDegreeR(int n, int r)
{
    int ans = r * binomialCoeff(2 * n
        - 1 - r, n - 1);
    ans = ans / n;
    System.out.println("Number of trees "
        + "having 4 edges where root has"
        + " degree 2 : " + ans);
    return 0;
}

```

```
// Driver program to test above functions

public static void main (String[] args)
{
    // Number of trees having 3
    // edges and exactly 2 leaves
    k_Leaves(3, 2);

    // Number of nodes of degree
    // 3 in a tree having 4 edges
    number0fNodes(3, 1);

    // Number of trees having 3
    // edges where root has degree 2
    rootDegreeR(3, 2);
}
}

// This code is contributed by anuj_67.
```

### C#

```
// C# code to find the number of ordered
// trees with given number of edges and
// leaves
using System;

class GFG {

    // Function returns value of
    // Binomial Coefficient C(n, k)
    static int binomialCoeff(int n, int k)
    {
        int [,]C = new int[n+1,k+1];
        int i, j;

        // Calculate value of Binomial
        // Coefficient in bottom up manner
        for (i = 0; i <= n; i++) {
            for (j = 0; j <= Math.Min(i, k); j++)
            {

                // Base Cases
                if (j == 0 || j == i)
                    C[i,j] = 1;

                // Calculate value using
```

```

        // previously stored values
        else
            C[i,j] = C[i - 1,j - 1]
                      + C[i - 1,j];
    }

}

return C[n,k];
}

// Function to calculate the number
// of trees with exactly k leaves.
static int k_Leaves(int n, int k)
{
    int ans = (binomialCoeff(n, k) *
               binomialCoeff(n, k - 1)) / n;
    Console.WriteLine( "Number of trees "
                      + "having 4 edges and exactly 2 "
                      + "leaves : " + ans) ;
    return 0;
}

// Function to calculate total number of
// nodes of degree d in these trees.
static int numberOfWorkNodes(int n, int d)
{
    int ans = binomialCoeff(2 * n - 1 - d,
                           n - 1);
    Console.WriteLine("Number of nodes "
                      +"of degree 1 in a tree having 4 "
                      + "edges : " + ans);
    return 0;
}

// Function to calculate the number of
// trees in which the root has degree r.
static int rootDegreeR(int n, int r)
{
    int ans = r * binomialCoeff(2 * n
                               - 1 - r, n - 1);
    ans = ans / n;
    Console.WriteLine("Number of trees "
                      + "having 4 edges where root has"
                      + " degree 2 : " + ans);
    return 0;
}

// Driver program to test above functions

```

```
public static void Main ()  
{  
  
    // Number of trees having 3  
    // edges and exactly 2 leaves  
    k_Leaves(3, 2);  
  
    // Number of nodes of degree  
    // 3 in a tree having 4 edges  
    numberOfNodes(3, 1);  
  
    // Number of trees having 3  
    // edges where root has degree 2  
    rootDegreeR(3, 2);  
}  
}  
  
// This code is contributed by anuj_67.
```

### PHP

```
<?php  
// PHP code to find the number of ordered  
// trees with given number of edges and  
// leaves  
  
// Function returns value of Binomial  
// Coefficient C(n, k)  
function binomialCoeff($n, $k)  
{  
    $C = array(array());  
    $i; $j;  
  
    // Calculate value of Binomial  
    // Coefficient in bottom up manner  
    for ($i = 0; $i <= $n; $i++) {  
        for ($j = 0; $j <= min($i, $k); $j++)  
        {  
  
            // Base Cases  
            if ($j == 0 or $j == $i)  
                $C[$i][$j] = 1;  
  
            // Calculate value using  
            // previously stored values  
            else  
                $C[$i][$j] = $C[$i - 1][$j - 1]
```

```

        + $C[$i - 1][$j];
    }
}

return $C[$n][$k];
}

// Function to calculate the number
// of trees with exactly k leaves.
function k_Leaves( $n, $k)
{
    $ans = (binomialCoeff($n, $k) *
            binomialCoeff($n, $k - 1)) / $n;

    echo "Number of trees having 4 edges and ",
         "exactly 2 leaves : " , $ans ,"\n";

    return 0;
}

// Function to calculate total number of
// nodes of degree d in these trees.
function number0fNodes( $n, $d)
{
    $ans = binomialCoeff(2 * $n - 1 - $d, $n - 1);
    echo "Number of nodes of degree 1 in"
         , " a tree having 4 edges : " , $ans,"\n" ;
    return 0;
}

// Function to calculate the number of
// trees in which the root has degree r.
function rootDegreeR( $n, $r)
{
    $ans = $r * binomialCoeff(2 * $n - 1 - $r,
                               $n - 1);
    $ans = $ans / $n;
    echo "Number of trees having 4 edges"
         , " where root has degree 2 : " , $ans ;
    return 0;
}

// Driver program to test above functions
// Number of trees having 3
// edges and exactly 2 leaves
k_Leaves(3, 2);

// Number of nodes of degree

```

```
// 3 in a tree having 4 edges  
numberOfNodes(3, 1);  
  
// Number of trees having 3  
// edges where root has degree 2  
rootDegreeR(3, 2);  
  
// This code is contributed by anuj_67.  
?>
```

**Output:**

```
Number of trees having 4 edges and exactly 2 leaves : 3  
Number of nodes of degree 1 in a tree having 4 edges : 6  
Number of trees having 4 edges where root has degree 2 : 2
```

**Time Complexity :**  $O(n^k)$ .  
**Auxiliary Space :**  $O(n^k)$ .

**Improved By :** [vt\\_m](#)

**Source**

<https://www.geeksforgeeks.org/combinatorics-ordered-trees/>

## Chapter 65

# Complexity of different operations in Binary tree, Binary Search Tree and AVL tree

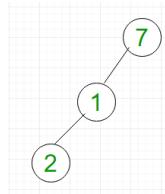
Complexity of different operations in Binary tree, Binary Search Tree and AVL tree - GeeksforGeeks

In this article, we will discuss complexity of different operations in binary trees including BST and AVL trees. Before understanding this article, you should have basic idea about: [Binary Tree](#), [Binary Search Tree](#) and [AVL Tree](#).

The main operations in binary tree are: search, insert and delete. We will see the worst case time complexity of these operations in binary trees.

### Binary Tree –

In a binary tree, a node can have maximum two children. Consider the left skewed binary tree shown in Figure 1.

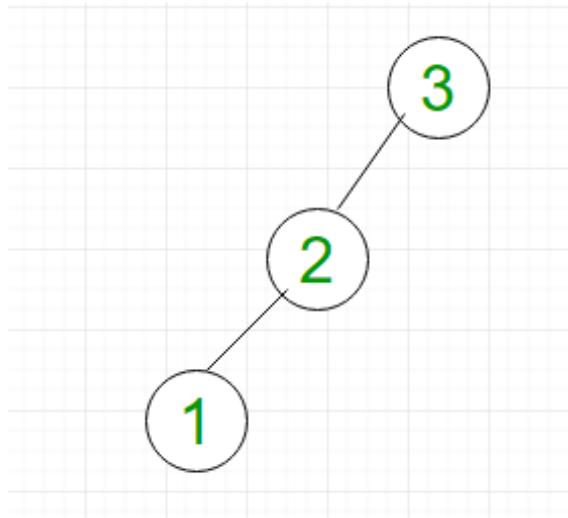


- **Searching:** For searching element 2, we have to traverse all elements (assuming we do breadth first traversal). Therefore, searching in binary tree has worst case complexity of  $O(n)$ .
- **Insertion:** For inserting element as left child of 2, we have to traverse all elements. Therefore, insertion in binary tree has worst case complexity of  $O(n)$ .

- **Deletion:** For deletion of element 2, we have to traverse all elements to find 2 (assuming we do breadth first traversal). Therefore, deletion in binary tree has worst case complexity of  $O(n)$ .

#### Binary Search Tree (BST) –

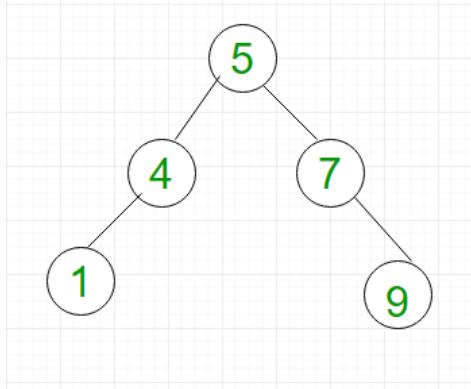
BST is a special type of binary tree in which left child of a node has value less than the parent and right child has value greater than parent. Consider the left skewed BST shown in Figure 2.



- **Searching:** For searching element 1, we have to traverse all elements (in order 3, 2, 1). Therefore, searching in binary search tree has worst case complexity of  $O(n)$ . In general, time complexity is  $O(h)$  where  $h$  is height of BST.
- **Insertion:** For inserting element 0, it must be inserted as left child of 1. Therefore, we need to traverse all elements (in order 3, 2, 1) to insert 0 which has worst case complexity of  $O(n)$ . In general, time complexity is  $O(h)$ .
- **Deletion:** For deletion of element 1, we have to traverse all elements to find 1 (in order 3, 2, 1). Therefore, deletion in binary tree has worst case complexity of  $O(n)$ . In general, time complexity is  $O(h)$ .

#### AVL/ Height Balanced Tree –

AVL tree is binary search tree with additional property that difference between height of left sub-tree and right sub-tree of any node can't be more than 1. For example, BST shown in Figure 2 is not AVL as difference between left sub-tree and right sub-tree of node 3 is 2. However, BST shown in Figure 3 is AVL tree.



- **Searching:** For searching element 1, we have to traverse elements (in order 5, 7, 9) = 3 =  $\log_2 n$ . Therefore, searching in AVL tree has worst case complexity of  $O(\log_2 n)$ .
- **Insertion:** For inserting element 12, it must be inserted as right child of 9. Therefore, we need to traverse elements (in order 5, 7, 9) to insert 12 which has worst case complexity of  $O(\log_2 n)$ .
- **Deletion:** For deletion of element 9, we have to traverse elements to find 9 (in order 5, 7, 9). Therefore, deletion in binary tree has worst case complexity of  $O(\log_2 n)$ .

We will discuss questions based on complexities of binary tree operations.

**Que-1.** What is the worst case time complexity for search, insert and delete operations in a general Binary Search Tree?

- (A)  $O(n)$  for all
- (B)  $O(\log n)$  for all
- (C)  $O(\log n)$  for search and insert, and  $O(n)$  for delete
- (D)  $O(\log n)$  for search, and  $O(n)$  for insert and delete

**Solution:** As discussed, all operations in BST have worst case time complexity of  $O(n)$ . So, the correct option is (A).

**Que-2.** What are the worst case time complexities of searching in binary tree, BST and AVL tree respectively?

- (A)  $O(n)$  for all
- (B)  $O(\log n)$  for all
- (C)  $O(n)$  for binary tree, and  $O(\log n)$  for others
- (D)  $O(n)$  for binary tree and BST, and  $O(\log n)$  for AVL

**Solution:** As discussed, search operation in binary tree and BST have worst case time complexity of  $O(n)$ . However, AVL tree has worst case time complexity of  $O(\log n)$ . So, the correct option is (D).

## Source

<https://www.geeksforgeeks.org/complexity-different-operations-binary-tree-binary-search-tree-avl-tree/>

## Chapter 66

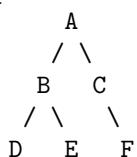
# Connect Nodes at same Level (Level Order Traversal)

Connect Nodes at same Level (Level Order Traversal) - GeeksforGeeks

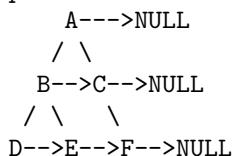
Write a function to connect all the adjacent nodes at the same level in a binary tree.

Example:

Input Tree



Output Tree



We have already discussed  $O(n^2)$  time and  $O$  approach in [Connect nodes at same level](#) as morris traversal in worst case can be  $O(n)$  and calling it to set right pointer can result in  $O(n^2)$  time complexity.

In this post, We have discussed [Level Order Traversal](#) with NULL markers which are needed to mark levels in tree.

C++

```
// Connect nodes at same level using level order
// traversal.
#include <bits/stdc++.h>
using namespace std;

struct Node {
    int data;
    struct Node* left, *right, *nextRight;
};

// Sets nextRight of all nodes of a tree
void connect(struct Node* root)
{
    queue<Node*> q;
    q.push(root);

    // null marker to represent end of current level
    q.push(NULL);

    // Do Level order of tree using NULL markers
    while (!q.empty()) {
        Node *p = q.front();
        q.pop();
        if (p != NULL) {

            // next element in queue represents next
            // node at current Level
            p->nextRight = q.front();

            // push left and right children of current
            // node
            if (p->left)
                q.push(p->left);
            if (p->right)
                q.push(p->right);
        }

        // if queue is not empty, push NULL to mark
        // nodes at this level are visited
        else if (!q.empty())
            q.push(NULL);
    }
}

/* UTILITY FUNCTIONS */
/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct Node* newnode(int data)
```

```
{  
    struct Node* node = (struct Node*)  
        malloc(sizeof(struct Node));  
    node->data = data;  
    node->left = node->right = node->nextRight = NULL;  
    return (node);  
}  
  
/* Driver program to test above functions*/  
int main()  
{  
  
    /* Constructed binary tree is  
       10  
      /   \  
     8     2  
    /       \  
   3       90  
 */  
    struct Node* root = newnode(10);  
    root->left = newnode(8);  
    root->right = newnode(2);  
    root->left->left = newnode(3);  
    root->right->right = newnode(90);  
  
    // Populates nextRight pointer in all nodes  
    connect(root);  
  
    // Let us check the values of nextRight pointers  
    printf("Following are populated nextRight pointers in \n"  
        "the tree (-1 is printed if there is no nextRight) \n");  
    printf("nextRight of %d is %d \n", root->data,  
        root->nextRight ? root->nextRight->data : -1);  
    printf("nextRight of %d is %d \n", root->left->data,  
        root->left->nextRight ? root->left->nextRight->data : -1);  
    printf("nextRight of %d is %d \n", root->right->data,  
        root->right->nextRight ? root->right->nextRight->data : -1);  
    printf("nextRight of %d is %d \n", root->left->left->data,  
        root->left->left->nextRight ? root->left->left->nextRight->data : -1);  
    printf("nextRight of %d is %d \n", root->right->right->data,  
        root->right->right->nextRight ? root->right->right->nextRight->data : -1);  
    return 0;  
}
```

### Java

```
// Connect nodes at same level using level order  
// traversal.
```

```
import java.util.LinkedList;
import java.util.Queue;
public class Connect_node_same_level {

    // Node class
    static class Node {
        int data;
        Node left, right, nextRight;
        Node(int data){
            this.data = data;
            left = null;
            right = null;
            nextRight = null;
        }
    };
    // Sets nextRight of all nodes of a tree
    static void connect(Node root)
    {
        Queue<Node> q = new LinkedList<Node>();
        q.add(root);

        // null marker to represent end of current level
        q.add(null);

        // Do Level order of tree using NULL markers
        while (!q.isEmpty()) {
            Node p = q.peek();
            q.remove();
            if (p != null) {

                // next element in queue represents next
                // node at current Level
                p.nextRight = q.peek();

                // push left and right children of current
                // node
                if (p.left != null)
                    q.add(p.left);
                if (p.right != null)
                    q.add(p.right);
            }

            // if queue is not empty, push NULL to mark
            // nodes at this level are visited
            else if (!q.isEmpty())
                q.add(null);
        }
    }
}
```

```
}

/* Driver program to test above functions*/
public static void main(String args[])
{
    /* Constructed binary tree is
        10
       /   \
      8     2
     /   \
    3     90
    */
    Node root = new Node(10);
    root.left = new Node(8);
    root.right = new Node(2);
    root.left.left = new Node(3);
    root.right.right = new Node(90);

    // Populates nextRight pointer in all nodes
    connect(root);

    // Let us check the values of nextRight pointers
    System.out.println("Following are populated nextRight pointers in \n" +
    "the tree (-1 is printed if there is no nextRight)");
    System.out.println("nextRight of "+ root.data +" is "+
    ((root.nextRight != null) ? root.nextRight.data : -1));
    System.out.println("nextRight of "+ root.left.data+" is "+
    ((root.left.nextRight != null) ? root.left.nextRight.data : -1));
    System.out.println("nextRight of "+ root.right.data+" is "+
    ((root.right.nextRight != null) ? root.right.nextRight.data : -1));
    System.out.println("nextRight of "+ root.left.left.data+" is "+
    ((root.left.left.nextRight != null) ? root.left.left.nextRight.data : -1));
    System.out.println("nextRight of "+ root.right.right.data+" is "+
    ((root.right.right.nextRight != null) ? root.right.right.nextRight.data : -1));
}

// This code is contributed by Sumit Ghosh
```

Output:

```
Following are populated nextRight pointers in
the tree (-1 is printed if there is no nextRight)
nextRight of 10 is -1
nextRight of 8 is 2
nextRight of 2 is -1
nextRight of 3 is 90
```

nextRight of 90 is -1

Time complexity : $O(n)$  where n is the number of nodes

**Alternate Implementation:**

We can also follow the implementation discussed in [Print level order traversal line by line | Set 1](#). We keep connecting nodes of same level by keeping track of prev visited node of same level.

Implementation : <https://ide.geeksforgeeks.org/gV1Oc2>

Thanks to Akilan Sengottaiyan for suggesting this alternate implementation.

**Source**

<https://www.geeksforgeeks.org/connect-nodes-level-level-order-traversal/>

# Chapter 67

## Connect nodes at same level

Connect nodes at same level - GeeksforGeeks

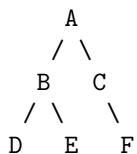
Write a function to connect all the adjacent nodes at the same level in a binary tree. Structure of the given Binary Tree node is like following.

```
struct node{  
    int data;  
    struct node* left;  
    struct node* right;  
    struct node* nextRight;  
}
```

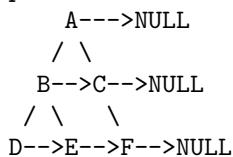
Initially, all the nextRight pointers point to garbage values. Your function should set these pointers to point next right for each node.

Example

Input Tree



Output Tree



### Method 1 (Extend Level Order Traversal or BFS)

Consider the method 2 of [Level Order Traversal](#). The method 2 can easily be extended to connect nodes of same level. We can augment queue entries to contain level of nodes also which is 0 for root, 1 for root's children and so on. So a queue node will now contain a pointer to a tree node and an integer level. When we enqueue a node, we make sure that correct level value for node is being set in queue. To set nextRight, for every node N, we dequeue the next node from queue, if the level number of next node is same, we set the nextRight of N as address of the dequeued node, otherwise we set nextRight of N as NULL.

Please refer [connect Nodes at same Level \(Level Order Traversal\)](#) for implementation.

Time Complexity: O(n)

### Method 2 (Extend Pre Order Traversal)

This approach works only for [Complete Binary Trees](#). In this method we set nextRight in Pre Order fashion to make sure that the nextRight of parent is set before its children. When we are at node p, we set the nextRight of its left and right children. Since the tree is complete tree, nextRight of p's left child ( $p->left->nextRight$ ) will always be p's right child, and nextRight of p's right child ( $p->right->nextRight$ ) will always be left child of p's nextRight (if p is not the rightmost node at its level). If p is the rightmost node, then nextRight of p's right child will be NULL.

**C/C++**

```
#include <stdio.h>
#include <stdlib.h>

struct node
{
    int data;
    struct node *left;
    struct node *right;
    struct node *nextRight;
};

void connectRecur(struct node* p);

// Sets the nextRight of root and calls connectRecur()
// for other nodes
void connect (struct node *p)
{
    // Set the nextRight for root
    p->nextRight = NULL;

    // Set the next right for rest of the nodes
    // (other than root)
    connectRecur(p);
```

```

}

/* Set next right of all descendants of p.
   Assumption: p is a compete binary tree */
void connectRecur(struct node* p)
{
    // Base case
    if (!p)
        return;

    // Set the nextRight pointer for p's left child
    if (p->left)
        p->left->nextRight = p->right;

    // Set the nextRight pointer for p's right child
    // p->nextRight will be NULL if p is the right
    // most child at its level
    if (p->right)
        p->right->nextRight = (p->nextRight)? p->nextRight->left: NULL;

    // Set nextRight for other nodes in pre order fashion
    connectRecur(p->left);
    connectRecur(p->right);
}

/* UTILITY FUNCTIONS */
/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newnode(int data)
{
    struct node* node = (struct node*)
                           malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;
    node->nextRight = NULL;

    return(node);
}

/* Driver program to test above functions*/
int main()
{

    /* Constructed binary tree is
           10
         /   \
        8     2
    */
}

```

```
    /
3
*/
struct node *root = newnode(10);
root->left      = newnode(8);
root->right     = newnode(2);
root->left->left = newnode(3);

// Populates nextRight pointer in all nodes
connect(root);

// Let us check the values of nextRight pointers
printf("Following are populated nextRight pointers in the tree "
"(-1 is printed if there is no nextRight) \n");
printf("nextRight of %d is %d \n", root->data,
root->nextRight? root->nextRight->data: -1);
printf("nextRight of %d is %d \n", root->left->data,
root->left->nextRight? root->left->nextRight->data: -1);
printf("nextRight of %d is %d \n", root->right->data,
root->right->nextRight? root->right->nextRight->data: -1);
printf("nextRight of %d is %d \n", root->left->left->data,
root->left->left->nextRight? root->left->left->nextRight->data: -1);
return 0;
}
```

### Java

```
// Java program to connect nodes at same level using extended
// pre-order traversal

// A binary tree node
class Node
{
    int data;
    Node left, right, nextRight;

    Node(int item)
    {
        data = item;
        left = right = nextRight = null;
    }
}

class BinaryTree
{
    Node root;

    // Sets the nextRight of root and calls connectRecur()
}
```

```

// for other nodes
void connect(Node p)
{
    // Set the nextRight for root
    p.nextRight = null;

    // Set the next right for rest of the nodes (other
    // than root)
    connectRecur(p);
}

/* Set next right of all descendants of p.
   Assumption: p is a complete binary tree */
void connectRecur(Node p)
{
    // Base case
    if (p == null)
        return;

    // Set the nextRight pointer for p's left child
    if (p.left != null)
        p.left.nextRight = p.right;

    // Set the nextRight pointer for p's right child
    // p->nextRight will be NULL if p is the right most child
    // at its level
    if (p.right != null)
        p.right.nextRight = (p.nextRight != null) ?
            p.nextRight.left : null;

    // Set nextRight for other nodes in pre order fashion
    connectRecur(p.left);
    connectRecur(p.right);
}

// Driver program to test above functions
public static void main(String args[])
{
    BinaryTree tree = new BinaryTree();

    /* Constructed binary tree is
       10
       / \
      8   2
      /
     3
    */
}

```

```

tree.root = new Node(10);
tree.root.left = new Node(8);
tree.root.right = new Node(2);
tree.root.left.left = new Node(3);

// Populates nextRight pointer in all nodes
tree.connect(tree.root);

// Let us check the values of nextRight pointers
System.out.println("Following are populated nextRight pointers in "
    + "the tree" + "(-1 is printed if there is no nextRight)");
int a = tree.root.nextRight != null ? tree.root.nextRight.data : -1;
System.out.println("nextRight of " + tree.root.data + " is "
    + a);
int b = tree.root.left.nextRight != null ?
    tree.root.left.nextRight.data : -1;
System.out.println("nextRight of " + tree.root.left.data + " is "
    + b);
int c = tree.root.right.nextRight != null ?
    tree.root.right.nextRight.data : -1;
System.out.println("nextRight of " + tree.root.right.data + " is "
    + c);
int d = tree.root.left.left.nextRight != null ?
    tree.root.left.left.nextRight.data : -1;
System.out.println("nextRight of " + tree.root.left.left.data + " is "
    + d);
}

}

// This code has been contributed by Mayank Jaiswal

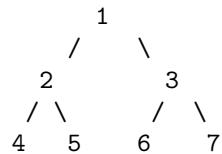
```

Thanks to Dhanya for suggesting this approach.

Time Complexity: O(n)

#### ***Why doesn't method 2 work for trees which are not Complete Binary Trees?***

Let us consider following tree as an example. In Method 2, we set the nextRight pointer in pre order fashion. When we are at node 4, we set the nextRight of its children which are 8 and 9 (the nextRight of 4 is already set as node 5). nextRight of 8 will simply be set as 9, but nextRight of 9 will be set as NULL which is incorrect. We can't set the correct nextRight, because when we set nextRight of 9, we only have nextRight of node 4 and ancestors of node 4, we don't have nextRight of nodes in right subtree of root.





See **Connect nodes at same level using constant extra space** for more solutions.

### Source

<https://www.geeksforgeeks.org/connect-nodes-at-same-level/>

## Chapter 68

# Connect nodes at same level using constant extra space

Connect nodes at same level using constant extra space - GeeksforGeeks

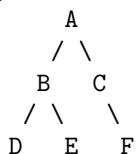
Write a function to connect all the adjacent nodes at the same level in a binary tree. Structure of the given Binary Tree node is like following.

```
struct node {  
    int data;  
    struct node* left;  
    struct node* right;  
    struct node* nextRight;  
}
```

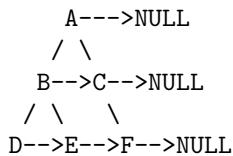
Initially, all the nextRight pointers point to garbage values. Your function should set these pointers to point next right for each node. You can use only constant extra space.

Example

Input Tree



Output Tree

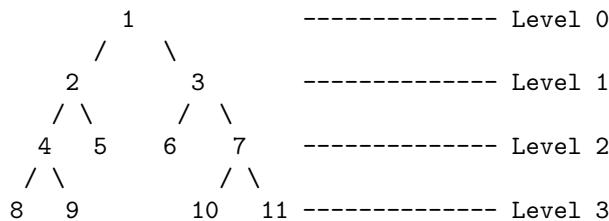


We discussed two different approaches to do it in the [previous post](#). The auxiliary space required in both of those approaches is not constant. Also, the method 2 discussed there only works for complete Binary Tree.

In this post, we will first modify the method 2 to make it work for all kind of trees. After that, we will remove recursion from this method so that the extra space becomes constant.

### A Recursive Solution

In the method 2 of previous post, we traversed the nodes in pre order fashion. Instead of traversing in Pre Order fashion (root, left, right), if we traverse the nextRight node before the left and right children (root, nextRight, left), then we can make sure that all nodes at level  $i$  have the nextRight set, before the level  $i+1$  nodes. Let us consider the following example (same example as [previous post](#)). The method 2 fails for right child of node 4. In this method, we make sure that all nodes at the 4's level (level 2) have nextRight set, before we try to set the nextRight of 9. So when we set the nextRight of 9, we search for a nonleaf node on right side of node 4 (getNextRight() does this for us).



C

```

void connectRecur(struct node* p);
struct node *getNextRight(struct node *p);

// Sets the nextRight of root and calls connectRecur() for other nodes
void connect (struct node *p)
{
    // Set the nextRight for root
    p->nextRight = NULL;

    // Set the next right for rest of the nodes (other than root)
    connectRecur(p);
}

/* Set next right of all descendants of p. This function makes sure that
nextRight of nodes ar level i is set before level i+1 nodes. */
void connectRecur(struct node* p)
{
    // Base case
    if (!p)
        return;
}
  
```

```

/* Before setting nextRight of left and right children, set nextRight
of children of other nodes at same level (because we can access
children of other nodes using p's nextRight only) */
if (p->nextRight != NULL)
    connectRecur(p->nextRight);

/* Set the nextRight pointer for p's left child */
if (p->left)
{
    if (p->right)
    {
        p->left->nextRight = p->right;
        p->right->nextRight = getNextRight(p);
    }
    else
        p->left->nextRight = getNextRight(p);

    /* Recursively call for next level nodes. Note that we call only
    for left child. The call for left child will call for right child */
    connectRecur(p->left);
}

/* If left child is NULL then first node of next level will either be
   p->right or getNextRight(p) */
else if (p->right)
{
    p->right->nextRight = getNextRight(p);
    connectRecur(p->right);
}
else
    connectRecur(getNextRight(p));
}

/* This function returns the leftmost child of nodes at the same level as p.
This function is used to getNExt right of p's right child
If right child of p is NULL then this can also be used for the left child */
struct node *getNextRight(struct node *p)
{
    struct node *temp = p->nextRight;

    /* Traverse nodes at p's level and find and return
       the first node's first child */
    while(temp != NULL)
    {
        if(temp->left != NULL)
            return temp->left;
        if(temp->right != NULL)

```

```
        return temp->right;
    temp = temp->nextRight;
}

// If all the nodes at p's level are leaf nodes then return NULL
return NULL;
}
```

**Java**

```
// Recursive Java program to connect nodes at same level
// using constant extra space

// A binary tree node
class Node
{
    int data;
    Node left, right, nextRight;

    Node(int item)
    {
        data = item;
        left = right = nextRight = null;
    }
}

class BinaryTree
{
    Node root;

    /* Set next right of all descendants of p. This function makes sure that
       nextRight of nodes at level i is set before level i+1 nodes. */
    void connectRecur(Node p)
    {
        // Base case
        if (p == null)
            return;

        /* Before setting nextRight of left and right children, set nextRight
           of children of other nodes at same level (because we can access
           children of other nodes using p's nextRight only) */
        if (p.nextRight != null)
            connectRecur(p.nextRight);

        /* Set the nextRight pointer for p's left child */
        if (p.left != null)
        {
            if (p.right != null)
```

```

    {
        p.left.nextRight = p.right;
        p.right.nextRight = getNextRight(p);
    }
    else
        p.left.nextRight = getNextRight(p);

    /* Recursively call for next level nodes. Note that we call only
       for left child. The call for left child will call for right child */
    connectRecur(p.left);
}

/* If left child is NULL then first node of next level will either be
   p->right or getNextRight(p) */
else if (p.right != null)
{
    p.right.nextRight = getNextRight(p);
    connectRecur(p.right);
}
else
    connectRecur(getNextRight(p));
}

/* This function returns the leftmost child of nodes at the same
   level as p. This function is used to getNExt right of p's right child
   If right child of p is NULL then this can also be used for
   the left child */
Node getNextRight(Node p)
{
    Node temp = p.nextRight;

    /* Traverse nodes at p's level and find and return
       the first node's first child */
    while (temp != null)
    {
        if (temp.left != null)
            return temp.left;
        if (temp.right != null)
            return temp.right;
        temp = temp.nextRight;
    }

    // If all the nodes at p's level are leaf nodes then return NULL
    return null;
}

/* Driver program to test the above functions */
public static void main(String args[])

```

```
{  
    BinaryTree tree = new BinaryTree();  
    tree.root = new Node(10);  
    tree.root.left = new Node(8);  
    tree.root.right = new Node(2);  
    tree.root.left.left = new Node(3);  
    tree.root.right.right = new Node(90);  
  
    // Populates nextRight pointer in all nodes  
    tree.connectRecur(tree.root);  
  
    // Let us check the values of nextRight pointers  
    int a = tree.root.nextRight != null ?  
        tree.root.nextRight.data : -1;  
    int b = tree.root.left.nextRight != null ?  
        tree.root.left.nextRight.data : -1;  
    int c = tree.root.right.nextRight != null ?  
        tree.root.right.nextRight.data : -1;  
    int d = tree.root.left.left.nextRight != null ?  
        tree.root.left.left.nextRight.data : -1;  
    int e = tree.root.right.right.nextRight != null ?  
        tree.root.right.right.nextRight.data : -1;  
  
    // Now lets print the values  
    System.out.println("Following are populated nextRight pointers in "  
        + " the tree(-1 is printed if there is no nextRight)");  
    System.out.println("nextRight of " + tree.root.data + " is " + a);  
    System.out.println("nextRight of " + tree.root.left.data + " is " + b);  
    System.out.println("nextRight of " + tree.root.right.data + " is " + c);  
    System.out.println("nextRight of " + tree.root.left.left.data +  
        " is " + d);  
    System.out.println("nextRight of " + tree.root.right.right.data +  
        " is " + e);  
}  
}  
  
// This code has been contributed by Mayank Jaiswal
```

### An Iterative Solution

The recursive approach discussed above can be easily converted to iterative. In the iterative version, we use nested loop. The outer loop, goes through all the levels and the inner loop goes through all the nodes at every level. This solution uses constant space.

C

```
#include <stdio.h>  
#include <stdlib.h>
```

```
struct node
{
    int data;
    struct node *left;
    struct node *right;
    struct node *nextRight;
};

/* This function returns the leftmost child of nodes at the same level as p.
   This function is used to getNExt right of p's right child
   If right child of is NULL then this can also be sued for the left child */
struct node *getNextRight(struct node *p)
{
    struct node *temp = p->nextRight;

    /* Traverse nodes at p's level and find and return
       the first node's first child */
    while (temp != NULL)
    {
        if (temp->left != NULL)
            return temp->left;
        if (temp->right != NULL)
            return temp->right;
        temp = temp->nextRight;
    }

    // If all the nodes at p's level are leaf nodes then return NULL
    return NULL;
}

/* Sets nextRight of all nodes of a tree with root as p */
void connect(struct node* p)
{
    struct node *temp;

    if (!p)
        return;

    // Set nextRight for root
    p->nextRight = NULL;

    // set nextRight of all levels one by one
    while (p != NULL)
    {
        struct node *q = p;

        /* Connect all children nodes of p and children nodes of all other nodes
           at same level as p */

```

```
while (q != NULL)
{
    // Set the nextRight pointer for p's left child
    if (q->left)
    {
        // If q has right child, then right child is nextRight of
        // p and we also need to set nextRight of right child
        if (q->right)
            q->left->nextRight = q->right;
        else
            q->left->nextRight = getNextRight(q);
    }

    if (q->right)
        q->right->nextRight = getNextRight(q);

    // Set nextRight for other nodes in pre order fashion
    q = q->nextRight;
}

// start from the first node of next level
if (p->left)
    p = p->left;
else if (p->right)
    p = p->right;
else
    p = getNextRight(p);
}

/* UTILITY FUNCTIONS */
/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newnode(int data)
{
    struct node* node = (struct node*)
                           malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;
    node->nextRight = NULL;

    return(node);
}

/* Driver program to test above functions*/
int main()
{
```

```
/* Constructed binary tree is
      10
     /   \
    8     2
   /       \
  3         90
*/
struct node *root = newnode(10);
root->left      = newnode(8);
root->right     = newnode(2);
root->left->left = newnode(3);
root->right->right = newnode(90);

// Populates nextRight pointer in all nodes
connect(root);

// Let us check the values of nextRight pointers
printf("Following are populated nextRight pointers in the tree "
       "(-1 is printed if there is no nextRight) \n");
printf("nextRight of %d is %d \n", root->data,
       root->nextRight? root->nextRight->data: -1);
printf("nextRight of %d is %d \n", root->left->data,
       root->left->nextRight? root->left->nextRight->data: -1);
printf("nextRight of %d is %d \n", root->right->data,
       root->right->nextRight? root->right->nextRight->data: -1);
printf("nextRight of %d is %d \n", root->left->left->data,
       root->left->left->nextRight? root->left->left->nextRight->data: -1);
printf("nextRight of %d is %d \n", root->right->right->data,
       root->right->right->nextRight? root->right->right->nextRight->data: -1);

getchar();
return 0;
}
```

### Java

```
// Iterative Java program to connect nodes at same level
// using constant extra space

// A binary tree node
class Node
{
    int data;
    Node left, right, nextRight;

    Node(int item)
    {
```

```
        data = item;
        left = right = nextRight = null;
    }
}

class BinaryTree
{
    Node root;

    /* This function returns the leftmost child of nodes at the same level
       as p. This function is used to getNExt right of p's right child
       If right child of is NULL then this can also be sued for the
       left child */
    Node getNextRight(Node p)
    {
        Node temp = p.nextRight;

        /* Traverse nodes at p's level and find and return
           the first node's first child */
        while (temp != null)
        {
            if (temp.left != null)
                return temp.left;
            if (temp.right != null)
                return temp.right;
            temp = temp.nextRight;
        }

        // If all the nodes at p's level are leaf nodes then return NULL
        return null;
    }

    /* Sets nextRight of all nodes of a tree with root as p */
    void connect(Node p) {
        Node temp = null;

        if (p == null)
            return;

        // Set nextRight for root
        p.nextRight = null;

        // set nextRight of all levels one by one
        while (p != null)
        {
            Node q = p;

            /* Connect all childrem nodes of p and children nodes of all other
```

```
        nodes at same level as p */
while (q != null)
{
    // Set the nextRight pointer for p's left child
    if (q.left != null)
    {

        // If q has right child, then right child is nextRight of
        // p and we also need to set nextRight of right child
        if (q.right != null)
            q.left.nextRight = q.right;
        else
            q.left.nextRight = getNextRight(q);
    }

    if (q.right != null)
        q.right.nextRight = getNextRight(q);

    // Set nextRight for other nodes in pre order fashion
    q = q.nextRight;
}

// start from the first node of next level
if (p.left != null)
    p = p.left;
else if (p.right != null)
    p = p.right;
else
    p = getNextRight(p);
}

/*
 * Driver program to test above functions */
public static void main(String args[])
{
    /* Constructed binary tree is
       10
      /   \
     8     2
    /       \
   3         90
*/
    BinaryTree tree = new BinaryTree();
    tree.root = new Node(10);
    tree.root.left = new Node(8);
    tree.root.right = new Node(2);
    tree.root.left.left = new Node(3);
    tree.root.right.right = new Node(90);
```

```
// Populates nextRight pointer in all nodes
tree.connect(tree.root);

// Let us check the values of nextRight pointers
int a = tree.root.nextRight != null ?
        tree.root.nextRight.data : -1;
int b = tree.root.left.nextRight != null ?
        tree.root.left.nextRight.data : -1;
int c = tree.root.right.nextRight != null ?
        tree.root.right.nextRight.data : -1;
int d = tree.root.left.left.nextRight != null ?
        tree.root.left.left.nextRight.data : -1;
int e = tree.root.right.right.nextRight != null ?
        tree.root.right.right.nextRight.data : -1;

// Now lets print the values
System.out.println("Following are populated nextRight pointers in "
    + " the tree(-1 is printed if there is no nextRight)");
System.out.println("nextRight of " + tree.root.data + " is " + a);
System.out.println("nextRight of " + tree.root.left.data
    + " is " + b);
System.out.println("nextRight of " + tree.root.right.data +
    " is " + c);
System.out.println("nextRight of " + tree.root.left.left.data +
    " is " + d);
System.out.println("nextRight of " + tree.root.right.right.data +
    " is " + e);
}

}

// This code has been contributed by Mayank Jaiswal
```

Output:

```
Following are populated nextRight pointers in the tree (-1 is printed if
there is no nextRight)
nextRight of 10 is -1
nextRight of 8 is 2
nextRight of 2 is -1
nextRight of 3 is 90
nextRight of 90 is -1
```

## Source

<https://www.geeksforgeeks.org/connect-nodes-at-same-level-with-o1-extra-space/>

## Chapter 69

# Construct Ancestor Matrix from a Given Binary Tree

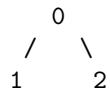
Construct Ancestor Matrix from a Given Binary Tree - GeeksforGeeks

Given a Binary Tree where all values are from **0** to **n-1**. Construct an ancestor matrix **mat[n][n]**. Ancestor matrix is defined as below.

```
mat[i][j] = 1 if i is ancestor of j  
mat[i][j] = 0, otherwise
```

Examples:

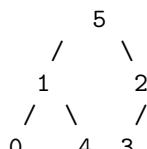
Input: Root of below Binary Tree.



Output: 0 1 1

```
    0 0 0  
    0 0 0
```

Input: Root of below Binary Tree.



Output: 0 0 0 0 0 0

```
  1 0 0 0 1 0  
  0 0 0 1 0 0
```

```
0 0 0 0 0 0  
0 0 0 0 0 0  
1 1 1 1 1 0
```

We strongly recommend you to minimize your browser and try this yourself first.

The idea is to traverse the tree. While traversing, keep track of ancestors in an array. When we visit a node, we add it to ancestor array and consider corresponding row in adjacency matrix. We mark all ancestors in its row as 1. Once a node and all its children are processed, we remove the node from ancestor array.

Below is C++ implementation of above idea.

```
// C++ program to construct ancestor matrix for  
// given tree.  
#include<bits/stdc++.h>  
using namespace std;  
#define MAX 100  
  
/* A binary tree node */  
struct Node  
{  
    int data;  
    Node *left, *right;  
};  
  
// Creating a global boolean matrix for simplicity  
bool mat[MAX][MAX];  
  
// anc[] stores all ancestors of current node. This  
// function fills ancestors for all nodes.  
// It also returns size of tree. Size of tree is  
// used to print ancestor matrix.  
int ancestorMatrixRec(Node *root, vector<int> &anc)  
{  
    /* base case */  
    if (root == NULL) return 0;;  
  
    // Update all ancestors of current node  
    int data = root->data;  
    for (int i=0; i<anc.size(); i++)  
        mat[anc[i]][data] = true;  
  
    // Push data to list of ancestors  
    anc.push_back(data);  
  
    // Traverse left and right subtrees  
    int l = ancestorMatrixRec(root->left, anc);
```

```

int r = ancestorMatrixRec(root->right, anc);

// Remove data from list the list of ancestors
// as all descendants of it are processed now.
anc.pop_back();

return l+r+1;
}

// This function mainly calls ancestorMatrixRec()
void ancestorMatrix(Node *root)
{
    // Create an empty ancestor array
    vector<int> anc;

    // Fill ancestor matrix and find size of
    // tree.
    int n = ancestorMatrixRec(root, anc);

    // Print the filled values
    for (int i=0; i<n; i++)
    {
        for (int j=0; j<n; j++)
            cout << mat[i][j] << " ";
        cout << endl;
    }
}

/* Helper function to create a new node */
Node* newnode(int data)
{
    Node* node = new Node;
    node->data = data;
    node->left = node->right = NULL;
    return (node);
}

/* Driver program to test above functions*/
int main()
{
    /* Construct the following binary tree
           5
         /   \
        1     2
       / \   /
      0   4  3  */
    Node *root      = newnode(5);
    root->left     = newnode(1);
}

```

```
root->right      = newnode(2);
root->left->left  = newnode(0);
root->left->right = newnode(4);
root->right->left = newnode(3);

ancestorMatrix(root);

return 0;
}
```

Output:

```
0 0 0 0 0 0
1 0 0 0 1 0
0 0 0 1 0 0
0 0 0 0 0 0
0 0 0 0 0 0
1 1 1 1 1 0
```

Time complexity of above solution is  $O(n^2)$ .

**How to do reverse – construct tree from ancestor matrix?**

[Construct tree from ancestor matrix](#)

This article is contributed by **Dheeraj Gupta**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## Source

<https://www.geeksforgeeks.org/construct-ancestor-matrix-from-a-given-binary-tree/>

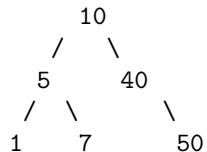
## Chapter 70

# Construct BST from given preorder traversal | Set 2

Construct BST from given preorder traversal | Set 2 - GeeksforGeeks

Given preorder traversal of a binary search tree, construct the BST.

For example, if the given traversal is {10, 5, 1, 7, 40, 50}, then the output should be root of following tree.



We have discussed  $O(n^2)$  and  $O(n)$  recursive solutions in the [previous post](#). Following is a stack based iterative solution that works in  $O(n)$  time.

1. Create an empty stack.
2. Make the first value as root. Push it to the stack.
3. Keep on popping while the stack is not empty and the next value is greater than stack's top value. Make this value as the right child of the last popped node. Push the new node to the stack.
4. If the next value is less than the stack's top value, make this value as the left child of the stack's top node. Push the new node to the stack.
5. Repeat steps 2 and 3 until there are items remaining in pre[].

C

```
/* A O(n) iterative program for construction of BST from preorder traversal */
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
typedef struct Node
{
    int data;
    struct Node *left, *right;
} Node;

// A Stack has array of Nodes, capacity, and top
typedef struct Stack
{
    int top;
    int capacity;
    Node* *array;
} Stack;

// A utility function to create a new tree node
Node* newNode( int data )
{
    Node* temp = (Node *)malloc( sizeof( Node ) );
    temp->data = data;
    temp->left = temp->right = NULL;
    return temp;
}

// A utility function to create a stack of given capacity
Stack* createStack( int capacity )
{
    Stack* stack = (Stack *)malloc( sizeof( Stack ) );
    stack->top = -1;
    stack->capacity = capacity;
    stack->array = (Node **)malloc( stack->capacity * sizeof( Node* ) );
    return stack;
}

// A utility function to check if stack is full
int isFull( Stack* stack )
{
    return stack->top == stack->capacity - 1;
}

// A utility function to check if stack is empty
int isEmpty( Stack* stack )
```

```
{  
    return stack->top == -1;  
}  
  
// A utility function to push an item to stack  
void push( Stack* stack, Node* item )  
{  
    if( isFull( stack ) )  
        return;  
    stack->array[ ++stack->top ] = item;  
}  
  
// A utility function to remove an item from stack  
Node* pop( Stack* stack )  
{  
    if( isEmpty( stack ) )  
        return NULL;  
    return stack->array[ stack->top-- ];  
}  
  
// A utility function to get top node of stack  
Node* peek( Stack* stack )  
{  
    return stack->array[ stack->top ];  
}  
  
// The main function that constructs BST from pre[]  
Node* constructTree ( int pre[], int size )  
{  
    // Create a stack of capacity equal to size  
    Stack* stack = createStack( size );  
  
    // The first element of pre[] is always root  
    Node* root = newNode( pre[0] );  
  
    // Push root  
    push( stack, root );  
  
    int i;  
    Node* temp;  
  
    // Iterate through rest of the size-1 items of given preorder array  
    for ( i = 1; i < size; ++i )  
    {  
        temp = NULL;  
  
        /* Keep on popping while the next value is greater than  
         stack's top value. */
```

```
while ( !isEmpty( stack ) && pre[i] > peek( stack )->data )
    temp = pop( stack );

// Make this greater value as the right child and push it to the stack
if ( temp != NULL)
{
    temp->right = newNode( pre[i] );
    push( stack, temp->right );
}

// If the next value is less than the stack's top value, make this value
// as the left child of the stack's top node. Push the new node to stack
else
{
    peek( stack )->left = newNode( pre[i] );
    push( stack, peek( stack )->left );
}

return root;
}

// A utility function to print inorder traversal of a Binary Tree
void printInorder (Node* node)
{
    if (node == NULL)
        return;
    printInorder(node->left);
    printf("%d ", node->data);
    printInorder(node->right);
}

// Driver program to test above functions
int main ()
{
    int pre[] = {10, 5, 1, 7, 40, 50};
    int size = sizeof( pre ) / sizeof( pre[0] );

    Node *root = constructTree(pre, size);

    printf("Inorder traversal of the constructed tree: \n");
    printInorder(root);

    return 0;
}
```

Java

```
// Java program to construct BST from given preorder traversal

import java.util.*;

// A binary tree node
class Node {

    int data;
    Node left, right;

    Node(int d) {
        data = d;
        left = right = null;
    }
}

class BinaryTree {

    // The main function that constructs BST from pre[]
    Node constructTree(int pre[], int size) {

        // The first element of pre[] is always root
        Node root = new Node(pre[0]);

        Stack<Node> s = new Stack<Node>();

        // Push root
        s.push(root);

        // Iterate through rest of the size-1 items of given preorder array
        for (int i = 1; i < size; ++i) {
            Node temp = null;

            /* Keep on popping while the next value is greater than
               stack's top value. */
            while (!s.isEmpty() && pre[i] > s.peek().data) {
                temp = s.pop();
            }

            // Make this greater value as the right child and push it to the stack
            if (temp != null) {
                temp.right = new Node(pre[i]);
                s.push(temp.right);
            }

            // If the next value is less than the stack's top value, make this value
            // as the left child of the stack's top node. Push the new node to stack
            else {

```

```
        temp = s.peek();
        temp.left = new Node(pre[i]);
        s.push(temp.left);
    }
}

return root;
}

// A utility function to print inorder traversal of a Binary Tree
void printInorder(Node node) {
    if (node == null) {
        return;
    }
    printInorder(node.left);
    System.out.print(node.data + " ");
    printInorder(node.right);
}

// Driver program to test above functions
public static void main(String[] args) {
    BinaryTree tree = new BinaryTree();
    int pre[] = new int[]{10, 5, 1, 7, 40, 50};
    int size = pre.length;
    Node root = tree.constructTree(pre, size);
    System.out.println("Inorder traversal of the constructed tree is ");
    tree.printInorder(root);
}
}

// This code has been contributed by Mayank Jaiswal
```

Output:

```
1 5 7 10 40 50
```

Time Complexity:  $O(n)$ . The complexity looks more from first look. If we take a closer look, we can observe that every item is pushed and popped only once. So at most  $2n$  push/pop operations are performed in the main loops of `constructTree()`. Therefore, time complexity is  $O(n)$ .

## Source

<https://www.geeksforgeeks.org/construct-bst-from-given-preorder-traversal-set-2/>

## Chapter 71

# Construct BST from its given level order traversal

Construct BST from its given level order traversal - GeeksforGeeks

Construct the BST (Binary Search Tree) from its given level order traversal.

Examples:

```
Input : arr[] = {7, 4, 12, 3, 6, 8, 1, 5, 10}
Output : BST:
```

```
      7
     / \
    4   12
   / \ /
  3   6 8
 /   / \
1     5   10
```

The idea is to use a queue to construct tree. Every element of queue has a structure say **NodeDetails** which stores details of a tree node. The details are pointer to the node, and two variables **min** and **max** where **min** stores the lower limit for the node values which can be a part of the left subtree and **max** stores the upper limit for the node values which can be a part of the right subtree for the specified node in **NodeDetails** structure variable. For the 1st array value arr[0], create a node and then create a **NodeDetails** structure having pointer to this node and min = INT\_MIN and max = INT\_MAX. Add this structure variable to a queue. This Node will be the root of the tree. Move to 2nd element in arr[] and then perform the following steps:

1. Pop **NodeDetails** from the queue in **temp**.

2. Check whether the current array element can be a left child of the node in **temp** with the help of **min** and **temp.node** data values. If it can, then create a new **NodeDetails** structure for this new array element value with its proper ‘min’ and ‘max’ values and push it to the queue, make this new node as the left child of temp’s node and move to next element in arr[].
3. Check whether the current array element can be a right child of the node in **temp** with the help of **max** and **temp.node** data values. If it can, then create a new **NodeDetails** structure for this new array element value with its proper ‘min’ and ‘max’ values and push it to the queue, make this new node as the right child of temp’s node and move to next element in arr[].
4. Repeat steps 1, 2 and 3 until there are no more elements in arr[].

For a left child node, its **min** value will be its parent’s ‘min’ value and **max** value will be the data value of its parent node. For a right child node, its **min** value will be the data value of its parent node and **max** value will be its parent’s ‘max’ value.

Below is C++ implementation of above approach:

```
// C++ implementation to construct a BST
// from its level order traversal
#include <bits/stdc++.h>

using namespace std;

// node of a BST
struct Node
{
    int data;
    Node *left, *right;
};

// to store details of a node like
// pointer to the node, 'min' and 'max'
// to obtain the range of values where
// node's left and right child's could lie
struct NodeDetails
{
    Node *ptr;
    int min, max;
};

// function to get a new node
Node* getNode(int data)
{
    // Allocate memory
    Node *newNode =
        (Node*)malloc(sizeof(Node));
```

```
// put in the data
newNode->data = data;
newNode->left = newNode->right = NULL;
return newNode;
}

// function to construct a BST from
// its level order traversal
Node* constructBst(int arr[], int n)
{
    // if tree is empty
    if (n == 0)
        return NULL;

    Node *root;

    // queue to store NodeDetails
    queue<NodeDetails> q;

    // index variable to access array elements
    int i=0;

    // node details for the
    // root of the BST
    NodeDetails newNode;
    newNode.ptr = getNode(arr[i++]);
    newNode.min = INT_MIN;
    newNode.max = INT_MAX;
    q.push(newNode);

    // getting the root of the BST
    root = newNode.ptr;

    // until there are no more elements
    // in arr[]
    while (i != n)
    {
        // extracting NodeDetails of a
        // node from the queue
        NodeDetails temp = q.front();
        q.pop();

        // check whether there are more elements
        // in the arr[] and arr[i] can be left child
        // of 'temp.ptr' or not
        if (i < n && (arr[i] < temp.ptr->data &&
                        arr[i] > temp.min))
        {
```

```
// Create NodeDetails for newNode
/// and add it to the queue
newNode.ptr = getNode(arr[i++]);
newNode.min = temp.min;
newNode.max = temp.ptr->data;
q.push(newNode);

// make this 'newNode' as left child
// of 'temp.ptr'
temp.ptr->left = newNode.ptr;
}

// check whether there are more elements
// in the arr[] and arr[i] can be right child
// of 'temp.ptr' or not
if (i < n && (arr[i] > temp.ptr->data &&
                arr[i] < temp.max))
{
    // Create NodeDetails for newNode
    /// and add it to the queue
    newNode.ptr = getNode(arr[i++]);
    newNode.min = temp.ptr->data;
    newNode.max = temp.max;
    q.push(newNode);

    // make this 'newNode' as right child
    // of 'temp.ptr'
    temp.ptr->right = newNode.ptr;
}
}

// root of the required BST
return root;
}

// function to print the inorder traversal
void inorderTraversal(Node* root)
{
    if (!root)
        return;

    inorderTraversal(root->left);
    cout << root->data << " ";
    inorderTraversal(root->right);
}

// Driver program to test above
int main()
```

```
{  
    int arr[] = {7, 4, 12, 3, 6, 8, 1, 5, 10};  
    int n = sizeof(arr) / sizeof(arr[0]);  
  
    Node *root = constructBst(arr, n);  
  
    cout << "Inorder Traversal: ";  
    inorderTraversal(root);  
    return 0;  
}
```

Output:

Inorder Traversal: 1 3 4 5 6 7 8 10 12

**Time Complexity :**  $O(n)$

### Source

<https://www.geeksforgeeks.org/construct-bst-given-level-order-traversal/>

## Chapter 72

# Construct Binary Tree from String with bracket representation

Construct Binary Tree from String with bracket representation - GeeksforGeeks

Construct a binary tree from a string consisting of parenthesis and integers. The whole input represents a binary tree. It contains an integer followed by zero, one or two pairs of parenthesis. The integer represents the root's value and a pair of parenthesis contains a child binary tree with the same structure. Always start to construct the left child node of the parent first if it exists.

Examples:

Input : "1(2)(3)"

Output : 1 2 3

Explanation :

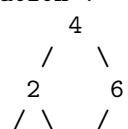


Explanation: first pair of parenthesis contains left subtree and second one contains the right subtree. Preorder of above tree is "1 2 3".

Input : "4(2(3)(1))(6(5))"

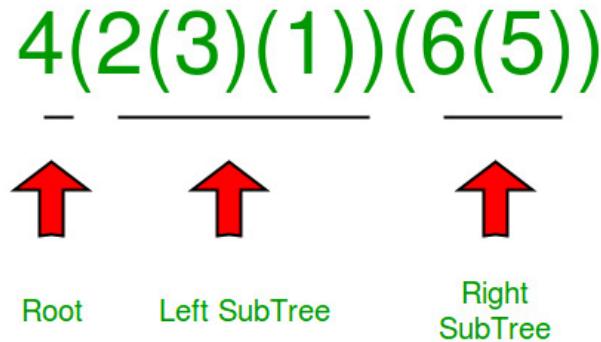
Output : 4 2 3 1 6 5

Explanation :



3 1 5

We know first character in string is root. Substring inside the first adjacent pair of parenthesis is for left subtree and substring inside second pair of parenthesis is for right subtree as in the below diagram.



We need to find the substring corresponding to left subtree and substring corresponding to right subtree and then recursively call on both of the substrings.

For this first find the index of starting index and end index of each substring.

To find the index of closing parenthesis of left subtree substring, use a stack. Lets the found index is stored in index variable.

```
/* C++ program to construct a binary tree from
   the given string */
#include <bits/stdc++.h>
using namespace std;

/* A binary tree node has data, pointer to left
   child and a pointer to right child */
struct Node {
    int data;
    Node *left, *right;
};

/* Helper function that allocates a new node */
Node* newNode(int data)
{
    Node* node = (Node*)malloc(sizeof(Node));
    node->data = data;
    node->left = node->right = NULL;
    return (node);
}

/* This function is here just to test */
void preOrder(Node* node)
{
    if (node == NULL)
```

```

        return;
    printf("%d ", node->data);
    preOrder(node->left);
    preOrder(node->right);
}

// function to return the index of close parenthesis
int findIndex(string str, int si, int ei)
{
    if (si > ei)
        return -1;

    // Inbuilt stack
    stack<char> s;

    for (int i = si; i <= ei; i++) {

        // if open parenthesis, push it
        if (str[i] == '(')
            s.push(str[i]);

        // if close parenthesis
        else if (str[i] == ')') {
            if (s.top() == '(') {
                s.pop();

                // if stack is empty, this is
                // the required index
                if (s.empty())
                    return i;
            }
        }
    }
    // if not found return -1
    return -1;
}

// function to construct tree from string
Node* treeFromString(string str, int si, int ei)
{
    // Base case
    if (si > ei)
        return NULL;

    // new root
    Node* root = newNode(str[si] - '0');
    int index = -1;
}

```

```
// if next char is '(' find the index of
// its complement ')'
if (si + 1 <= ei && str[si + 1] == '(')
    index = findIndex(str, si + 1, ei);

// if index found
if (index != -1) {

    // call for left subtree
    root->left = treeFromString(str, si + 2, index - 1);

    // call for right subtree
    root->right = treeFromString(str, index + 2, ei - 1);
}
return root;
}

// Driver Code
int main()
{
    string str = "4(2(3)(1))(6(5))";
    Node* root = treeFromString(str, 0, str.length() - 1);
    preOrder(root);
}
```

Output:

```
4 2 3 1 6 5
```

## Source

<https://www.geeksforgeeks.org/construct-binary-tree-string-bracket-representation/>

## Chapter 73

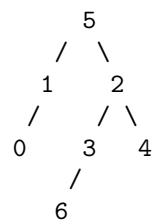
# Construct Binary Tree from given Parent Array representation

Construct Binary Tree from given Parent Array representation - GeeksforGeeks

Given an array that represents a tree in such a way that array indexes are values in tree nodes and array values give the parent node of that particular index (or node). The value of the root node index would always be -1 as there is no parent for root. Construct the standard linked representation of given Binary Tree from this given representation.

Input: parent[] = {1, 5, 5, 2, 2, -1, 3}

Output: root of below tree



Explanation:

Index of -1 is 5. So 5 is root.

5 is present at indexes 1 and 2. So 1 and 2 are children of 5.

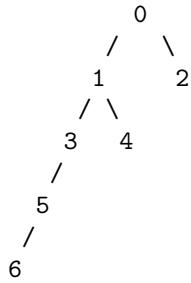
1 is present at index 0, so 0 is child of 1.

2 is present at indexes 3 and 4. So 3 and 4 are children of 2.

3 is present at index 6, so 6 is child of 3.

Input: parent[] = {-1, 0, 0, 1, 1, 3, 5};

Output: root of below tree



Expected time complexity is O(n) where n is number of elements in given array.

**We strongly recommend to minimize your browser and try this yourself first.**

A **Simple Solution** to recursively construct by first searching the current root, then recurring for the found indexes (there can be at most two indexes) and making them left and right subtrees of root. This solution takes  $O(n^2)$  as we have to linearly search for every node.

An **Efficient Solution** can solve the above problem in  $O(n)$  time. The idea is to use extra space. An array created[0..n-1] is used to keep track of created nodes.

*createTree(parent[], n)*

1. Create an array of pointers say created[0..n-1]. The value of created[i] is NULL if node for index i is not created, else value is pointer to the created node.
2. Do following for every index i of given array  
*createNode(parent, i, created)*

*createNode(parent[], i, created[])*

1. If created[i] is not NULL, then node is already created. So return.
2. Create a new node with value 'i'.
3. If parent[i] is -1 (i is root), make created node as root and return.
4. Check if parent of 'i' is created (We can check this by checking if created[parent[i]] is NULL or not).
5. If parent is not created, recur for parent and create the parent first.
6. Let the pointer to parent be p. If p->left is NULL, then make the new node as left child. Else make the new node as right child of parent.

Following is C++ implementation of above idea.

C++

```
// C++ program to construct a Binary Tree from parent array
#include<bits/stdc++.h>
using namespace std;

// A tree node
struct Node
{
    int key;
    struct Node *left, *right;
};

// Utility function to create new Node
Node *newNode(int key)
{
    Node *temp = new Node;
    temp->key = key;
    temp->left = temp->right = NULL;
    return (temp);
}

// Creates a node with key as 'i'. If i is root, then it changes
// // root. If parent of i is not created, then it creates parent first
void createNode(int parent[], int i, Node *created[], Node **root)
{
    // If this node is already created
    if (created[i] != NULL)
        return;

    // Create a new node and set created[i]
    created[i] = newNode(i);

    // If 'i' is root, change root pointer and return
    if (parent[i] == -1)
    {
        *root = created[i];
        return;
    }

    // If parent is not created, then create parent first
    if (created[parent[i]] == NULL)
        createNode(parent, parent[i], created, root);

    // Find parent pointer
    Node *p = created[parent[i]];

    // If this is first child of parent
    if (p->left == NULL)
        p->left = created[i];
}
```

```
    else // If second child
        p->right = created[i];
}

// Creates tree from parent[0..n-1] and returns root of the created tree
Node *createTree(int parent[], int n)
{
    // Create an array created[] to keep track
    // of created nodes, initialize all entries
    // as NULL
    Node *created[n];
    for (int i=0; i<n; i++)
        created[i] = NULL;

    Node *root = NULL;
    for (int i=0; i<n; i++)
        createNode(parent, i, created, &root);

    return root;
}

//For adding new line in a program
inline void newLine(){
    cout << "\n";
}

// Utility function to do inorder traversal
void inorder(Node *root)
{
    if (root != NULL)
    {
        inorder(root->left);
        cout << root->key << " ";
        inorder(root->right);
    }
}

// Driver method
int main()
{
    int parent[] = {-1, 0, 0, 1, 1, 3, 5};
    int n = sizeof parent / sizeof parent[0];
    Node *root = createTree(parent, n);
    cout << "Inorder Traversal of constructed tree\n";
    inorder(root);
    newLine();
}
```

**Java**

```
// Java program to construct a binary tree from parent array

// A binary tree node
class Node
{
    int key;
    Node left, right;

    public Node(int key)
    {
        this.key = key;
        left = right = null;
    }
}

class BinaryTree
{
    Node root;

    // Creates a node with key as 'i'. If i is root, then it changes
    // // root. If parent of i is not created, then it creates parent first
    void createNode(int parent[], int i, Node created[])
    {
        // If this node is already created
        if (created[i] != null)
            return;

        // Create a new node and set created[i]
        created[i] = new Node(i);

        // If 'i' is root, change root pointer and return
        if (parent[i] == -1)
        {
            root = created[i];
            return;
        }

        // If parent is not created, then create parent first
        if (created[parent[i]] == null)
            createNode(parent, parent[i], created);

        // Find parent pointer
        Node p = created[parent[i]];

        // If this is first child of parent
        if (p.left == null)
```

```
p.left = created[i];
else // If second child

    p.right = created[i];
}

/* Creates tree from parent[0..n-1] and returns root of
   the created tree */
Node createTree(int parent[], int n)
{
    // Create an array created[] to keep track
    // of created nodes, initialize all entries
    // as NULL
    Node[] created = new Node[n];
    for (int i = 0; i < n; i++)
        created[i] = null;

    for (int i = 0; i < n; i++)
        createNode(parent, i, created);

    return root;
}

//For adding new line in a program
void newLine()
{
    System.out.println("");
}

// Utility function to do inorder traversal
void inorder(Node node)
{
    if (node != null)
    {
        inorder(node.left);
        System.out.print(node.key + " ");
        inorder(node.right);
    }
}

// Driver method
public static void main(String[] args)
{

    BinaryTree tree = new BinaryTree();
    int parent[] = new int[]{-1, 0, 0, 1, 1, 3, 5};
    int n = parent.length;
    Node node = tree.createTree(parent, n);
```

```
        System.out.println("Inorder traversal of constructed tree ");
        tree.inorder(node);
        tree.newLine();
    }
}

// This code has been contributed by Mayank Jaiswal(mayank_24)
```

### Python

```
# Python implementation to construct a Binary Tree from
# parent array

# A node structure
class Node:
    # A utility function to create a new node
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None

    """ Creates a node with key as 'i'. If i is root,then
        it changes root. If parent of i is not created, then
        it creates parent first
    """
    def createNode(parent, i, created, root):

        # If this node is already created
        if created[i] is not None:
            return

        # Create a new node and set created[i]
        created[i] = Node(i)

        # If 'i' is root, change root pointer and return
        if parent[i] == -1:
            root[0] = created[i] # root[0] denotes root of the tree
            return

        # If parent is not created, then create parent first
        if created[parent[i]] is None:
            createNode(parent, parent[i], created, root )

        # Find parent pointer
        p = created[parent[i]]

        # If this is first child of parent
        if p.left is None:
```

```
p.left = created[i]
# If second child
else:
    p.right = created[i]

# Creates tree from parent[0..n-1] and returns root of the
# created tree
def createTree(parent):
    n = len(parent)

    # Create an array created[] to keep track
    # of created nodes, initialize all entries as None
    created = [None for i in range(n+1)]

    root = [None]
    for i in range(n):
        createNode(parent, i, created, root)

    return root[0]

# Inorder traversal of tree
def inorder(root):
    if root is not None:
        inorder(root.left)
        print root.key,
        inorder(root.right)

# Driver Method
parent = [-1, 0, 0, 1, 1, 3, 5]
root = createTree(parent)
print "Inorder Traversal of constructed tree"
inorder(root)

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

Output:

```
Inorder Traversal of constructed tree
6 5 3 1 4 0 2
```

< Similar Problem: [Find Height of Binary Tree represented by Parent array](#)

## Source

<https://www.geeksforgeeks.org/construct-a-binary-tree-from-parent-array-representation/>

## Chapter 74

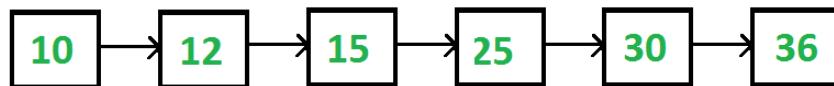
# Construct Complete Binary Tree from its Linked List Representation

Construct Complete Binary Tree from its Linked List Representation - GeeksforGeeks

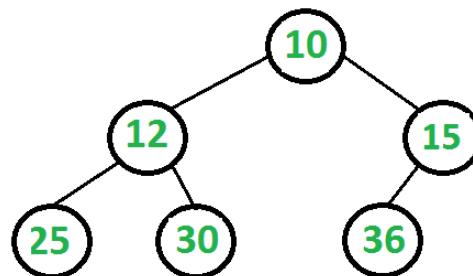
Given Linked List Representation of Complete Binary Tree, construct the Binary tree. A complete binary tree can be represented in an array in the following approach.

If root node is stored at index i, its left, and right children are stored at indices  $2*i+1$ ,  $2*i+2$  respectively.

Suppose tree is represented by a linked list in same way, how do we convert this into normal linked representation of binary tree where every node has data, left and right pointers? In the linked list representation, we cannot directly access the children of the current node unless we traverse the list.



The above linked list represents following binary tree



We are mainly given level order traversal in sequential access form. We know head of linked list is always root of the tree. We take the first node as root and we also know that the

next two nodes are left and right children of root. So we know partial Binary Tree. The idea is to do Level order traversal of the partially built Binary Tree using queue and traverse the linked list at the same time. At every step, we take the parent node from queue, make next two nodes of linked list as children of the parent node, and enqueue the next two nodes to queue.

1. Create an empty queue.
2. Make the first node of the list as root, and enqueue it to the queue.
3. Until we reach the end of the list, do the following.
  - .....a. Dequeue one node from the queue. This is the current parent.
  - .....b. Traverse two nodes in the list, add them as children of the current parent.
  - .....c. Enqueue the two nodes into the queue.

Below is the code which implements the same in C++.

C++

```
// C++ program to create a Complete Binary tree from its Linked List
// Representation
#include <iostream>
#include <string>
#include <queue>
using namespace std;

// Linked list node
struct ListNode
{
    int data;
    ListNode* next;
};

// Binary tree node structure
struct BinaryTreeNode
{
    int data;
    BinaryTreeNode *left, *right;
};

// Function to insert a node at the beginning of the Linked List
void push(struct ListNode** head_ref, int new_data)
{
    // allocate node and assign data
    struct ListNode* new_node = new ListNode;
    new_node->data = new_data;

    // link the old list off the new node
    new_node->next = (*head_ref);

    // move the head to point to the new node
}
```

```
(*head_ref)      = new_node;
}

// method to create a new binary tree node from the given data
BinaryTreeNode* newBinaryTreeNode(int data)
{
    BinaryTreeNode *temp = new BinaryTreeNode;
    temp->data = data;
    temp->left = temp->right = NULL;
    return temp;
}

// converts a given linked list representing a complete binary tree into the
// linked representation of binary tree.
void convertList2Binary(ListNode *head, BinaryTreeNode* &root)
{
    // queue to store the parent nodes
    queue<BinaryTreeNode *> q;

    // Base Case
    if (head == NULL)
    {
        root = NULL; // Note that root is passed by reference
        return;
    }

    // 1.) The first node is always the root node, and add it to the queue
    root = newBinaryTreeNode(head->data);
    q.push(root);

    // advance the pointer to the next node
    head = head->next;

    // until the end of linked list is reached, do the following steps
    while (head)
    {
        // 2.a) take the parent node from the q and remove it from q
        BinaryTreeNode* parent = q.front();
        q.pop();

        // 2.c) take next two nodes from the linked list. We will add
        // them as children of the current parent node in step 2.b. Push them
        // into the queue so that they will be parents to the future nodes
        BinaryTreeNode *leftChild = NULL, *rightChild = NULL;
        leftChild = newBinaryTreeNode(head->data);
        q.push(leftChild);
        head = head->next;
        if (head)
```

```
{  
    rightChild = newBinaryTreeNode(head->data);  
    q.push(rightChild);  
    head = head->next;  
}  
  
// 2.b) assign the left and right children of parent  
parent->left = leftChild;  
parent->right = rightChild;  
}  
}  
  
// Utility function to traverse the binary tree after conversion  
void inorderTraversal(BinaryTreeNode* root)  
{  
    if (root)  
    {  
        inorderTraversal( root->left );  
        cout << root->data << " ";  
        inorderTraversal( root->right );  
    }  
}  
  
// Driver program to test above functions  
int main()  
{  
    // create a linked list shown in above diagram  
    struct ListNode* head = NULL;  
    push(&head, 36); /* Last node of Linked List */  
    push(&head, 30);  
    push(&head, 25);  
    push(&head, 15);  
    push(&head, 12);  
    push(&head, 10); /* First node of Linked List */  
  
    BinaryTreeNode *root;  
    convertList2Binary(head, root);  
  
    cout << "Inorder Traversal of the constructed Binary Tree is: \n";  
    inorderTraversal(root);  
    return 0;  
}
```

**Java**

```
// Java program to create complete Binary Tree from its Linked List  
// representation
```

```
// importing necessary classes
import java.util.*;

// A linked list node
class ListNode
{
    int data;
    ListNode next;
    ListNode(int d)
    {
        data = d;
        next = null;
    }
}

// A binary tree node
class BinaryTreeNode
{
    int data;
    BinaryTreeNode left, right = null;
    BinaryTreeNode(int data)
    {
        this.data = data;
        left = right = null;
    }
}

class BinaryTree
{
    ListNode head;
    BinaryTreeNode root;

    // Function to insert a node at the beginning of
    // the Linked List
    void push(int new_data)
    {
        // allocate node and assign data
        ListNode new_node = new ListNode(new_data);

        // link the old list off the new node
        new_node.next = head;

        // move the head to point to the new node
        head = new_node;
    }

    // converts a given linked list representing a
    // complete binary tree into the linked
```

```
// representation of binary tree.
BinaryTreeNode convertList2Binary(BinaryTreeNode node)
{
    // queue to store the parent nodes
    Queue<BinaryTreeNode> q =
        new LinkedList<BinaryTreeNode>();

    // Base Case
    if (head == null)
    {
        node = null;
        return null;
    }

    // 1.) The first node is always the root node, and
    //      add it to the queue
    node = new BinaryTreeNode(head.data);
    q.add(node);

    // advance the pointer to the next node
    head = head.next;

    // until the end of linked list is reached, do the
    // following steps
    while (head != null)
    {
        // 2.a) take the parent node from the q and
        //       remove it from q
        BinaryTreeNode parent = q.peek();
        BinaryTreeNode pp = q.poll();

        // 2.c) take next two nodes from the linked list.
        // We will add them as children of the current
        // parent node in step 2.b. Push them into the
        // queue so that they will be parents to the
        // future nodes
        BinaryTreeNode leftChild = null, rightChild = null;
        leftChild = new BinaryTreeNode(head.data);
        q.add(leftChild);
        head = head.next;
        if (head != null)
        {
            rightChild = new BinaryTreeNode(head.data);
            q.add(rightChild);
            head = head.next;
        }

        // 2.b) assign the left and right children of
    }
}
```

```
//      parent
parent.left = leftChild;
parent.right = rightChild;
}

return node;
}

// Utility function to traverse the binary tree
// after conversion
void inorderTraversal(BinaryTreeNode node)
{
    if (node != null)
    {
        inorderTraversal(node.left);
        System.out.print(node.data + " ");
        inorderTraversal(node.right);
    }
}

// Driver program to test above functions
public static void main(String[] args)
{
    BinaryTree tree = new BinaryTree();
    tree.push(36); /* Last node of Linked List */
    tree.push(30);
    tree.push(25);
    tree.push(15);
    tree.push(12);
    tree.push(10); /* First node of Linked List */
    BinaryTreeNode node = tree.convertList2Binary(tree.root);

    System.out.println("Inorder Traversal of the"+
                      " constructed Binary Tree is:");
    tree.inorderTraversal(node);
}
}

// This code has been contributed by Mayank Jaiswal
```

### Python

```
# Python program to create a Complete Binary Tree from
# its linked list representation

# Linked List node
class ListNode:

    # Constructor to create a new node
```

```
def __init__(self, data):
    self.data = data
    self.next = None

# Binary Tree Node structure
class BinaryTreeNode:

    # Constructor to create a new node
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

# Class to convert the linked list to Binary Tree
class Conversion:

    # Constructor for storing head of linked list
    # and root for the Binary Tree
    def __init__(self, data = None):
        self.head = None
        self.root = None

    def push(self, new_data):

        # Creating a new linked list node and storing data
        new_node = ListNode(new_data)

        # Make next of new node as head
        new_node.next = self.head

        # Move the head to point to new node
        self.head = new_node

    def convertList2Binary(self):

        # Queue to store the parent nodes
        q = []

        # Base Case
        if self.head is None:
            self.root = None
            return

        # 1.) The first node is always the root node,
        # and add it to the queue
        self.root = BinaryTreeNode(self.head.data)
        q.append(self.root)

        ... (remaining code for the conversion algorithm)
```

```
# Advance the pointer to the next node
self.head = self.head.next

# Until th end of linked list is reached, do:
while(self.head):

    # 2.a) Take the parent node from the q and
    # and remove it from q
    parent = q.pop(0) # Front of queue

    # 2.c) Take next two nodes from the linked list.
    # We will add them as children of the current
    # parent node in step 2.b.
    # Push them into the queue so that they will be
    # parent to the future node
    leftChild= None
    rightChild = None

    leftChild = BinaryTreeNode(self.head.data)
    q.append(leftChild)
    self.head = self.head.next
    if(self.head):
        rightChild = BinaryTreeNode(self.head.data)
        q.append(rightChild)
        self.head = self.head.next

    #2.b) Assign the left and right children of parent
    parent.left = leftChild
    parent.right = rightChild

def inorderTraversal(self, root):
    if(root):
        self.inorderTraversal(root.left)
        print root.data,
        self.inorderTraversal(root.right)

# Driver Program to test above function

# Object of conversion class
conv = Conversion()
conv.push(36)
conv.push(30)
conv.push(25)
conv.push(15)
conv.push(12)
conv.push(10)

conv.convertList2Binary()
```

```
print "Inorder Traversal of the contructed Binary Tree is:"  
conv.inorderTraversal(conv.root)  
  
# This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

Output:

```
Inorder Traversal of the constructed Binary Tree is:  
25 12 30 10 36 15
```

**Time Complexity:** Time complexity of the above solution is  $O(n)$  where  $n$  is the number of nodes.

This article is compiled by **Ravi Chandra Enaganti**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## Source

<https://www.geeksforgeeks.org/given-linked-list-representation-of-complete-tree-convert-it-to-linked-representation/>

## Chapter 75

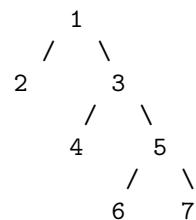
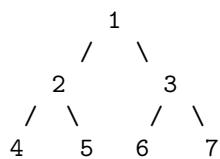
# Construct Full Binary Tree from given preorder and postorder traversals

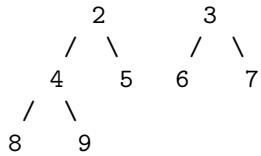
Construct Full Binary Tree from given preorder and postorder traversals - GeeksforGeeks

Given two arrays that represent preorder and postorder traversals of a full binary tree, construct the binary tree.

A **Full Binary Tree** is a binary tree where every node has either 0 or 2 children

Following are examples of Full Trees.

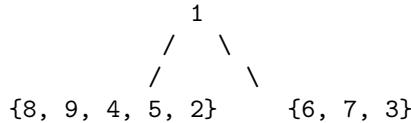




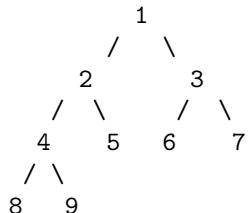
It is not possible to construct a general Binary Tree from preorder and postorder traversals (See [this](#)). But if know that the Binary Tree is Full, we can construct the tree without ambiguity. Let us understand this with the help of following example.

Let us consider the two given arrays as  $\text{pre}[] = \{1, 2, 4, 8, 9, 5, 3, 6, 7\}$  and  $\text{post}[] = \{8, 9, 4, 5, 2, 6, 7, 3, 1\}$ ;

In  $\text{pre}[]$ , the leftmost element is root of tree. Since the tree is full and array size is more than 1. The value next to 1 in  $\text{pre}[]$ , must be left child of root. So we know 1 is root and 2 is left child. How to find the all nodes in left subtree? We know 2 is root of all nodes in left subtree. All nodes before 2 in  $\text{post}[]$  must be in left subtree. Now we know 1 is root, elements  $\{8, 9, 4, 5, 2\}$  are in left subtree, and the elements  $\{6, 7, 3\}$  are in right subtree.



We recursively follow the above approach and get the following tree.



C

```

/* program for construction of full binary tree */
#include <stdio.h>
#include <stdlib.h>

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
  
```

```
{  
    int data;  
    struct node *left;  
    struct node *right;  
};  
  
// A utility function to create a node  
struct node* newNode (int data)  
{  
    struct node* temp = (struct node *) malloc( sizeof(struct node) );  
  
    temp->data = data;  
    temp->left = temp->right = NULL;  
  
    return temp;  
}  
  
// A recursive function to construct Full from pre[] and post[].  
// preIndex is used to keep track of index in pre[].  
// l is low index and h is high index for the current subarray in post[]  
struct node* constructTreeUtil (int pre[], int post[], int* preIndex,  
                                int l, int h, int size)  
{  
    // Base case  
    if (*preIndex >= size || l > h)  
        return NULL;  
  
    // The first node in preorder traversal is root. So take the node at  
    // preIndex from preorder and make it root, and increment preIndex  
    struct node* root = newNode ( pre[*preIndex] );  
    ++*preIndex;  
  
    // If the current subarry has only one element, no need to recur  
    if (l == h)  
        return root;  
  
    // Search the next element of pre[] in post[]  
    int i;  
    for (i = l; i <= h; ++i)  
        if (pre[*preIndex] == post[i])  
            break;  
  
    // Use the index of element found in postorder to divide postorder array in  
    // two parts. Left subtree and right subtree  
    if (i <= h)  
    {  
        root->left = constructTreeUtil (pre, post, preIndex, l, i, size);  
        root->right = constructTreeUtil (pre, post, preIndex, i + 1, h, size);  
    }  
}
```

```
    }

    return root;
}

// The main function to construct Full Binary Tree from given preorder and
// postorder traversals. This function mainly uses constructTreeUtil()
struct node *constructTree (int pre[], int post[], int size)
{
    int preIndex = 0;
    return constructTreeUtil (pre, post, &preIndex, 0, size - 1, size);
}

// A utility function to print inorder traversal of a Binary Tree
void printInorder (struct node* node)
{
    if (node == NULL)
        return;
    printInorder(node->left);
    printf("%d ", node->data);
    printInorder(node->right);
}

// Driver program to test above functions
int main ()
{
    int pre[] = {1, 2, 4, 8, 9, 5, 3, 6, 7};
    int post[] = {8, 9, 4, 5, 2, 6, 7, 3, 1};
    int size = sizeof( pre ) / sizeof( pre[0] );

    struct node *root = constructTree(pre, post, size);

    printf("Inorder traversal of the constructed tree: \n");
    printInorder(root);

    return 0;
}
```

### Java

```
// Java program for construction
// of full binary tree
public class fullbinarytreepostpre
{
    // variable to hold index in pre[] array
    static int preindex;

    static class node
```

```
{  
    int data;  
    node left, right;  
  
    public node(int data)  
    {  
        this.data = data;  
    }  
}  
  
// A recursive function to construct Full  
// from pre[] and post[]. preIndex is used  
// to keep track of index in pre[]. l is  
// low index and h is high index for the  
// current subarray in post[]  
static node constructTreeUtil(int pre[], int post[], int l,  
                             int h, int size)  
{  
  
    // Base case  
    if (preindex >= size || l > h)  
        return null;  
  
    // The first node in preorder traversal is  
    // root. So take the node at preIndex from  
    // preorder and make it root, and increment  
    // preIndex  
    node root = new node(pre[preindex]);  
    preindex++;  
  
    // If the current subarry has only one  
    // element, no need to recur or  
    // preIndex > size after incrementing  
    if (l == h || preindex >= size)  
        return root;  
    int i;  
  
    // Search the next element of pre[] in post[]  
    for (i = l; i <= h; i++)  
    {  
        if (post[i] == pre[preindex])  
            break;  
    }  
    // Use the index of element found in  
    // postorder to divide postorder array  
    // in two parts. Left subtree and right subtree  
    if (i <= h)  
    {
```

```
        root.left = constructTreeUtil(pre, post, l, i, size);
        root.right = constructTreeUtil(pre, post, i + 1, h, size);
    }
    return root;
}

// The main function to construct Full
// Binary Tree from given preorder and
// postorder traversals. This function
// mainly uses constructTreeUtil()
static node constructTree(int pre[], int post[], int size)
{
    preindex = 0;
    return constructTreeUtil(pre, post, 0, size - 1, size);
}

static void printInorder(node root)
{
    if (root == null)
        return;
    printInorder(root.left);
    System.out.print(root.data + " ");
    printInorder(root.right);
}

public static void main(String[] args)
{
    int pre[] = { 1, 2, 4, 8, 9, 5, 3, 6, 7 };
    int post[] = { 8, 9, 4, 5, 2, 6, 7, 3, 1 };

    int size = pre.length;
    node root = constructTree(pre, post, size);

    System.out.println("Inorder traversal of the constructed tree:");
    printInorder(root);
}
}

// This code is contributed by Rishabh Mahrsee
```

Output:

```
Inorder traversal of the constructed tree:
8 4 9 2 5 1 6 3 7
```

## Source

<https://www.geeksforgeeks.org/full-and-complete-binary-tree-from-given-preorder-and-postorder-traversals/>

## Chapter 76

# Construct Full Binary Tree using its Preorder traversal and Preorder traversal of its mirror tree

Construct Full Binary Tree using its Preorder traversal and Preorder traversal of its mirror tree - GeeksforGeeks

Given two arrays that represent Preorder traversals of a full binary tree and its mirror tree, we need to write a program to construct the binary tree using these two Preorder traversals.

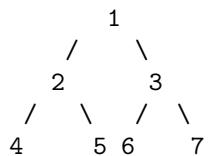
A **Full Binary Tree** is a binary tree where every node has either 0 or 2 children.

**Note:** It is not possible to construct a general binary tree using these two traversals. But we can create a full binary tree using the above traversals without any ambiguity. For more details refer to [this article](#).

Examples:

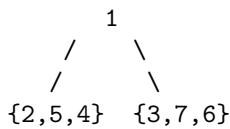
Input :    preOrder[] = {1,2,4,5,3,6,7}  
              preOrderMirror[] = {1,3,7,6,2,5,4}

Output :

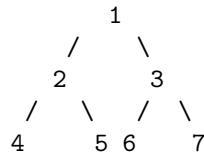


- **Method 1:** Let us consider the two given arrays as preOrder[] = {1, 2, 4, 5, 3, 6, 7} and preOrderMirror[] = {1,3,7,6,2,5,4}.

In both `preOrder[]` and `preOrderMirror[]`, the leftmost element is root of tree. Since the tree is full and array size is more than 1. The value next to 1 in `preOrder[]`, must be left child of root and value next to 1 in `preOrderMirror[]` must be right child of root. So we know 1 is root and 2 is left child and 3 is the right child. How to find the all nodes in left subtree? We know 2 is root of all nodes in left subtree and 3 is root of all nodes in right subtree. All nodes from and 2 in `preOrderMirror[]` must be in left subtree of root node 1 and all node after 3 and before 2 in `preOrderMirror[]` must be in right subtree of root node 1. Now we know 1 is root, elements {2, 5, 4} are in left subtree, and the elements {3, 7, 6} are in the right subtree.



We will recursively follow the above approach and get the below tree:



Below is the implementation of above approach:

```

// C++ program to construct full binary tree
// using its preorder traversal and preorder
// traversal of its mirror tree

#include<bits/stdc++.h>
using namespace std;

// A Binary Tree Node
struct Node
{
    int data;
    struct Node *left, *right;
};

// Utility function to create a new tree node
Node* newNode(int data)
{
    Node *temp = new Node;
    temp->data = data;
    temp->left = temp->right = NULL;
    return temp;
}

```

```
}

// A utility function to print inorder traversal
// of a Binary Tree
void printInorder(Node* node)
{
    if (node == NULL)
        return;

    printInorder(node->left);
    printf("%d ", node->data);
    printInorder(node->right);
}

// A recursive function to construct Full binary tree
// from pre[] and preM[]. preIndex is used to keep
// track of index in pre[]. l is low index and h is high
// index for the current subarray in preM[]
Node* constructBinaryTreeUtil(int pre[], int preM[],
                               int &preIndex, int l,int h,int size)
{
    // Base case
    if (preIndex >= size || l > h)
        return NULL;

    // The first node in preorder traversal is root.
    // So take the node at preIndex from preorder and
    // make it root, and increment preIndex
    Node* root = newNode(pre[preIndex]);
    ++(preIndex);

    // If the current subarry has only one element,
    // no need to recur
    if (l == h)
        return root;

    // Search the next element of pre[] in preM[]
    int i;
    for (i = l; i <= h; ++i)
        if (pre[preIndex] == preM[i])
            break;

    // construct left and right subtrees recursively
    if (i <= h)
    {
        root->left = constructBinaryTreeUtil (pre, preM,
                                              preIndex, i, h, size);
        root->right = constructBinaryTreeUtil (pre, preM,
```

```
                                preIndex, l+1, i-1, size);
}

// return root
return root;
}

// function to construct full binary tree
// using its preorder traversal and preorder
// traversal of its mirror tree
void constructBinaryTree(Node* root,int pre[],
                         int preMirror[], int size)
{
    int preIndex = 0;
    int preMIndex = 0;

    root = constructBinaryTreeUtil(pre,preMirror,
                                   preIndex,0,size-1,size);

    printInorder(root);
}

// Driver program to test above functions
int main()
{
    int preOrder[] = {1,2,4,5,3,6,7};
    int preOrderMirror[] = {1,3,7,6,2,5,4};

    int size = sizeof(preOrder)/sizeof(preOrder[0]);

    Node* root = new Node;

    constructBinaryTree(root,preOrder,preOrderMirror,size);

    return 0;
}
```

Output:

4 2 5 1 6 3 7

- **Method 2:** If we observe carefully, then reverse of the Preorder traversal of mirror tree will be the Postorder traversal of original tree. We can construct the tree from given Preorder and Postorder traversals in a similar manner as above. You can refer to [this](#) article on how to Construct Full Binary Tree from given preorder and postorder traversals.

## Source

<https://www.geeksforgeeks.org/construct-full-binary-tree-using-preorder-traversal-preorder-traversal-mirror-tree/>

## Chapter 77

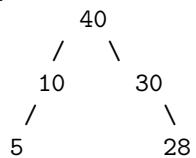
# Construct Special Binary Tree from given Inorder traversal

Construct Special Binary Tree from given Inorder traversal - GeeksforGeeks

Given Inorder Traversal of a Special Binary Tree in which key of every node is greater than keys in left and right children, construct the Binary Tree and return root.

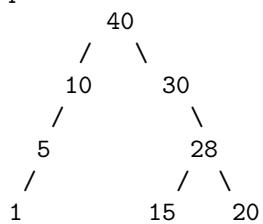
Examples:

Input: `inorder[] = {5, 10, 40, 30, 28}`  
Output: root of following tree

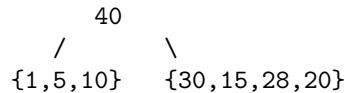


Input: `inorder[] = {1, 5, 10, 40, 30, 15, 28, 20}`

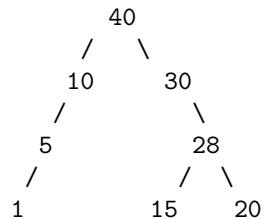
Output: root of following tree



The idea used in [Construction of Tree from given Inorder and Preorder traversals](#) can be used here. Let the given array is `{1, 5, 10, 40, 30, 15, 28, 20}`. The maximum element in given array must be root. The elements on left side of the maximum element are in left subtree and elements on right side are in right subtree.



We recursively follow above step for left and right subtrees, and finally get the following tree.



**Algorithm:** buildTree()

- 1) Find index of the maximum element in array. The maximum element must be root of Binary Tree.
- 2) Create a new tree node ‘root’ with the data as the maximum value found in step 1.
- 3) Call buildTree for elements before the maximum element and make the built tree as left subtree of ‘root’.
- 5) Call buildTree for elements after the maximum element and make the built tree as right subtree of ‘root’.
- 6) return ‘root’.

**Implementation:** Following is the implementation of the above algorithm.

C/C++

```

/* program to construct tree from inorder traversal */
#include<stdio.h>
#include<stdlib.h>

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

/* Prototypes of a utility function to get the maximum
   value in inorder[start..end] */
int max(int inorder[], int strt, int end);

```

```
/* A utility function to allocate memory for a node */
struct node* newNode(int data);

/* Recursive function to construct binary of size len from
   Inorder traversal inorder[]. Initial values of start and end
   should be 0 and len -1. */
struct node* buildTree (int inorder[], int start, int end)
{
    if (start > end)
        return NULL;

    /* Find index of the maximum element from Binary Tree */
    int i = max (inorder, start, end);

    /* Pick the maximum value and make it root */
    struct node *root = newNode(inorder[i]);

    /* If this is the only element in inorder[start..end],
       then return it */
    if (start == end)
        return root;

    /* Using index in Inorder traversal, construct left and
       right subtress */
    root->left = buildTree (inorder, start, i-1);
    root->right = buildTree (inorder, i+1, end);

    return root;
}

/* UTILITY FUNCTIONS */
/* Function to find index of the maximum value in arr[start...end] */
int max (int arr[], int strt, int end)
{
    int i, max = arr[strt], maxind = strt;
    for(i = strt+1; i <= end; i++)
    {
        if(arr[i] > max)
        {
            max = arr[i];
            maxind = i;
        }
    }
    return maxind;
}

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
```

```
struct node* newNode (int data)
{
    struct node* node = (struct node*)malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;

    return node;
}

/* This function is here just to test buildTree() */
void printInorder (struct node* node)
{
    if (node == NULL)
        return;

    /* first recur on left child */
    printInorder (node->left);

    /* then print the data of node */
    printf("%d ", node->data);

    /* now recur on right child */
    printInorder (node->right);
}

/* Driver program to test above functions */
int main()
{
    /* Assume that inorder traversal of following tree is given
       40
      /   \
     10   30
    /       \
   5       28 */

    int inorder[] = {5, 10, 40, 30, 28};
    int len = sizeof(inorder)/sizeof(inorder[0]);
    struct node *root = buildTree(inorder, 0, len - 1);

    /* Let us test the built tree by printing Inorder traversal */
    printf("\n Inorder traversal of the constructed tree is \n");
    printInorder(root);
    return 0;
}
```

Java

```
// Java program to construct tree from inorder traversal

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
class Node
{
    int data;
    Node left, right;

    Node(int item)
    {
        data = item;
        left = right = null;
    }
}

class BinaryTree
{
    Node root;

    /* Recursive function to construct binary of size len from
       Inorder traversal inorder[]. Initial values of start and end
       should be 0 and len -1. */
    Node buildTree(int inorder[], int start, int end, Node node)
    {
        if (start > end)
            return null;

        /* Find index of the maximum element from Binary Tree */
        int i = max(inorder, start, end);

        /* Pick the maximum value and make it root */
        node = new Node(inorder[i]);

        /* If this is the only element in inorder[start..end],
           then return it */
        if (start == end)
            return node;

        /* Using index in Inorder traversal, construct left and
           right subtress */
        node.left = buildTree(inorder, start, i - 1, node.left);
        node.right = buildTree(inorder, i + 1, end, node.right);

        return node;
    }

    /* UTILITY FUNCTIONS */
```

```
/* Function to find index of the maximum value in arr[start...end] */
int max(int arr[], int strt, int end)
{
    int i, max = arr[strt], maxind = strt;
    for (i = strt + 1; i <= end; i++)
    {
        if (arr[i] > max)
        {
            max = arr[i];
            maxind = i;
        }
    }
    return maxind;
}

/* This function is here just to test buildTree() */
void printInorder(Node node)
{
    if (node == null)
        return;

    /* first recur on left child */
    printInorder(node.left);

    /* then print the data of node */
    System.out.print(node.data + " ");

    /* now recur on right child */
    printInorder(node.right);
}

public static void main(String args[])
{
    BinaryTree tree = new BinaryTree();

    /* Assume that inorder traversal of following tree is given
       40
       /   \
      10   30
      /       \
     5       28 */
    int inorder[] = new int[]{5, 10, 40, 30, 28};
    int len = inorder.length;
    Node mynode = tree.buildTree(inorder, 0, len - 1, tree.root);

    /* Let us test the built tree by printing Inorder traversal */
    System.out.println("Inorder traversal of the constructed tree is ");
}
```

```
        tree.printInorder(mynode);
    }
}

// This code has been contributed by Mayank Jaiswal
```

Output:

```
Inorder traversal of the constructed tree is
5 10 40 30 28
```

Time Complexity:  $O(n^2)$

## Source

<https://www.geeksforgeeks.org/construct-binary-tree-from-inorder-traversal/>

## Chapter 78

# Construct Tree from given Inorder and Preorder traversals

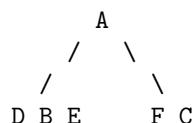
Construct Tree from given Inorder and Preorder traversals - GeeksforGeeks

Let us consider the below traversals:

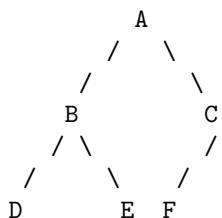
Inorder sequence: D B E A F C

Preorder sequence: A B D E C F

In a Preorder sequence, leftmost element is the root of the tree. So we know 'A' is root for given sequences. By searching 'A' in Inorder sequence, we can find out all elements on left side of 'A' are in left subtree and elements on right are in right subtree. So we know below structure now.



We recursively follow above steps and get the following tree.



Algorithm: buildTree()

- 1) Pick an element from Preorder. Increment a Preorder Index Variable (preIndex in below

- code) to pick next element in next recursive call.
- 2) Create a new tree node tNode with the data as picked element.
  - 3) Find the picked element's index in Inorder. Let the index be inIndex.
  - 4) Call buildTree for elements before inIndex and make the built tree as left subtree of tNode.
  - 5) Call buildTree for elements after inIndex and make the built tree as right subtree of tNode.
  - 6) return tNode.

Thanks to Rohini and Tushar for suggesting the code.

## C

```
/* program to construct tree using inorder and preorder traversals */
#include <stdio.h>
#include <stdlib.h>

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node {
    char data;
    struct node* left;
    struct node* right;
};

/* Prototypes for utility functions */
int search(char arr[], int strt, int end, char value);
struct node* newNode(char data);

/* Recursive function to construct binary of size len from
   Inorder traversal in[] and Preorder traversal pre[]. Initial values
   of inStrt and inEnd should be 0 and len -1. The function doesn't
   do any error checking for cases where inorder and preorder
   do not form a tree */
struct node* buildTree(char in[], char pre[], int inStrt, int inEnd)
{
    static int preIndex = 0;

    if (inStrt > inEnd)
        return NULL;

    /* Pick current node from Preorder traversal using preIndex
       and increment preIndex */
    struct node* tNode = newNode(pre[preIndex++]);

    /* If this node has no children then return */
    if (inStrt == inEnd)
        return tNode;
```

```
/* Else find the index of this node in Inorder traversal */
int inIndex = search(in, inStrt, inEnd, tNode->data);

/* Using index in Inorder traversal, construct left and
right subtress */
tNode->left = buildTree(in, pre, inStrt, inIndex - 1);
tNode->right = buildTree(in, pre, inIndex + 1, inEnd);

return tNode;
}

/* UTILITY FUNCTIONS */
/* Function to find index of value in arr[start...end]
The function assumes that value is present in in[] */
int search(char arr[], int strt, int end, char value)
{
    int i;
    for (i = strt; i <= end; i++) {
        if (arr[i] == value)
            return i;
    }
}

/* Helper function that allocates a new node with the
given data and NULL left and right pointers. */
struct node* newNode(char data)
{
    struct node* node = (struct node*)malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;

    return (node);
}

/* This funtcion is here just to test buildTree() */
void printInorder(struct node* node)
{
    if (node == NULL)
        return;

    /* first recur on left child */
    printInorder(node->left);

    /* then print the data of node */
    printf("%c ", node->data);

    /* now recur on right child */
}
```

```
    printInorder(node->right);
}

/* Driver program to test above functions */
int main()
{
    char in[] = { 'D', 'B', 'E', 'A', 'F', 'C' };
    char pre[] = { 'A', 'B', 'D', 'E', 'C', 'F' };
    int len = sizeof(in) / sizeof(in[0]);
    struct node* root = buildTree(in, pre, 0, len - 1);

    /* Let us test the built tree by printing Inorder traversal */
    printf("Inorder traversal of the constructed tree is \n");
    printInorder(root);
    getchar();
}
```

### Java

```
// Java program to construct a tree using inorder and preorder traversal

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
class Node {
    char data;
    Node left, right;

    Node(char item)
    {
        data = item;
        left = right = null;
    }
}

class BinaryTree {
    Node root;
    static int preIndex = 0;

    /* Recursive function to construct binary of size len from
       Inorder traversal in[] and Preorder traversal pre[].
       Initial values of inStrt and inEnd should be 0 and len -1.
       The function doesn't do any error checking for cases where
       inorder and preorder do not form a tree */
    Node buildTree(char in[], char pre[], int inStrt, int inEnd)
    {
        if (inStrt > inEnd)
            return null;
```

```
/* Pick current node from Preorder traversal using preIndex
   and increment preIndex */
Node tNode = new Node(pre[preIndex++]);

/* If this node has no children then return */
if (inStrt == inEnd)
    return tNode;

/* Else find the index of this node in Inorder traversal */
int inIndex = search(in, inStrt, inEnd, tNode.data);

/* Using index in Inorder traversal, construct left and
   right subtress */
tNode.left = buildTree(in, pre, inStrt, inIndex - 1);
tNode.right = buildTree(in, pre, inIndex + 1, inEnd);

return tNode;
}

/* UTILITY FUNCTIONS */

/* Function to find index of value in arr[start...end]
   The function assumes that value is present in in[] */
int search(char arr[], int strt, int end, char value)
{
    int i;
    for (i = strt; i <= end; i++) {
        if (arr[i] == value)
            return i;
    }
    return i;
}

/* This funtcion is here just to test buildTree() */
void printInorder(Node node)
{
    if (node == null)
        return;

    /* first recur on left child */
    printInorder(node.left);

    /* then print the data of node */
    System.out.print(node.data + " ");

    /* now recur on right child */
    printInorder(node.right);
}
```

```
// driver program to test above functions
public static void main(String args[])
{
    BinaryTree tree = new BinaryTree();
    char in[] = new char[] { 'D', 'B', 'E', 'A', 'F', 'C' };
    char pre[] = new char[] { 'A', 'B', 'D', 'E', 'C', 'F' };
    int len = in.length;
    Node root = tree.buildTree(in, pre, 0, len - 1);

    // building the tree by printing inorder traversal
    System.out.println("Inorder traversal of constructed tree is : ");
    tree.printInorder(root);
}
}

// This code has been contributed by Mayank Jaiswal
```

### Python

```
# Python program to construct tree using inorder and
# preorder traversals

# A binary tree node
class Node:

    # Constructor to create a new node
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

    """Recursive function to construct binary of size len from
    Inorder traversal in[] and Preorder traversal pre[]. Initial values
    of inStrt and inEnd should be 0 and len -1. The function doesn't
    do any error checking for cases where inorder and preorder
    do not form a tree """
def buildTree(inOrder, preOrder, inStrt, inEnd):

    if (inStrt > inEnd):
        return None

    # Pick current node from Preorder traversal using
    # preIndex and increment preIndex
    tNode = Node(preOrder[buildTree.preIndex])
    buildTree.preIndex += 1

    # If this node has no children then return
```

```
if inStrt == inEnd :
    return tNode

# Else find the index of this node in Inorder traversal
inIndex = search(inOrder, inStrt, inEnd, tNode.data)

# Using index in Inorder Traversal, construct left
# and right subtrees
tNode.left = buildTree(inOrder, preOrder, inStrt, inIndex-1)
tNode.right = buildTree(inOrder, preOrder, inIndex + 1, inEnd)

return tNode

# UTILITY FUNCTIONS
# Function to find index of value in arr[start...end]
# The function assumes that value is present in inOrder[]
def search(arr, start, end, value):
    for i in range(start, end + 1):
        if arr[i] == value:
            return i

def printInorder(node):
    if node is None:
        return

    # first recur on left child
    printInorder(node.left)

    # then print the data of node
    print node.data,

    # now recur on right child
    printInorder(node.right)

# Driver program to test above function
inOrder = ['D', 'B', 'E', 'A', 'F', 'C']
preOrder = ['A', 'B', 'D', 'E', 'C', 'F']
# Static variable preIndex
buildTree.preIndex = 0
root = buildTree(inOrder, preOrder, 0, len(inOrder)-1)

# Let us test the build tree by printing Inorder traversal
print "Inorder traversal of the constructed tree is"
printInorder(root)

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

Output :

Inorder traversal of constructed tree is :  
D B E A F C

Time Complexity:  $O(n^2)$ . Worst case occurs when tree is left skewed. Example Preorder and Inorder traversals for worst case are {A, B, C, D} and {D, C, B, A}.

**Efficient Approach :**

We can optimize the above solution using hashing (unordered\_map in C++ or HashMap in Java). We store indexes of inorder traversal in a hash table. So that search can be done  $O(1)$  time.

C++

```
/* C++ program to construct tree using inorder
   and preorder traversals */
#include <bits/stdc++.h>
using namespace std;

/* A binary tree node has data, pointer to left child
and a pointer to right child */
struct Node {
    char data;
    struct Node* left;
    struct Node* right;
};

struct Node* newNode(char data)
{
    struct Node* node = new Node;
    node->data = data;
    node->left = node->right = NULL;
    return (node);
}

/* Recursive function to construct binary of size
len from Inorder traversal in[] and Preorder traversal
pre[]. Initial values of inStrt and inEnd should be
0 and len -1. The function doesn't do any error
checking for cases where inorder and preorder
do not form a tree */
struct Node* buildTree(char in[], char pre[], int inStrt,
                      int inEnd, unordered_map<char, int>& mp)
{
    static int preIndex = 0;
```

```
if (inStrt > inEnd)
    return NULL;

/* Pick current node from Preorder traversal using preIndex
and increment preIndex */
int curr = pre[preIndex++];
struct Node* tNode = newNode(curr);

/* If this node has no children then return */
if (inStrt == inEnd)
    return tNode;

/* Else find the index of this node in Inorder traversal */
int inIndex = mp[curr];

/* Using index in Inorder traversal, construct left and
right subtress */
tNode->left = buildTree(in, pre, inStrt, inIndex - 1, mp);
tNode->right = buildTree(in, pre, inIndex + 1, inEnd, mp);

return tNode;
}

// This function mainly creates an unordered_map, then
// calls buildTree()
struct Node* buildTreeWrap(char in[], char pre[], int len)
{
    // Store indexes of all items so that we
    // we can quickly find later
    unordered_map<char, int> mp;
    for (int i = 0; i < len; i++)
        mp[in[i]] = i;

    return buildTree(in, pre, 0, len - 1, mp);
}

/* This function is here just to test buildTree() */
void printInorder(struct Node* node)
{
    if (node == NULL)
        return;
    printInorder(node->left);
    printf("%c ", node->data);
    printInorder(node->right);
}

/* Driver program to test above functions */
```

```
int main()
{
    char in[] = { 'D', 'B', 'E', 'A', 'F', 'C' };
    char pre[] = { 'A', 'B', 'D', 'E', 'C', 'F' };
    int len = sizeof(in) / sizeof(in[0]);

    struct Node* root = buildTreeWrap(in, pre, len);

    /* Let us test the built tree by printing
       Inorder traversal */
    printf("Inorder traversal of the constructed tree is \n");
    printInorder(root);
}
```

Time Complexity : O(n)

**Another approach :**

Use the fact that InOrder traversal is Left-Root-Right and PreOrder traversal is Root-Left-Right. Also, first node in the PreOrder traversal is always the root node and the first node in the InOrder traversal is the leftmost node in the tree.

Maintain two data-structures : Stack (to store the path we visited while traversing PreOrder array) and Set (to maintain the node in which the next right subtree is expected).

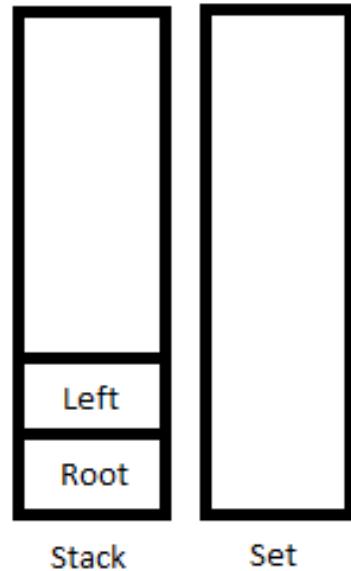
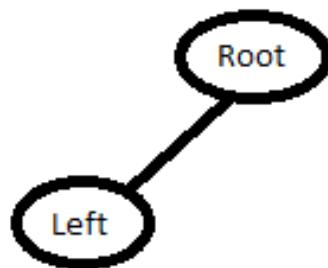
1. Do below until you reach the leftmost node.

Keep creating the nodes from PreOrder traversal

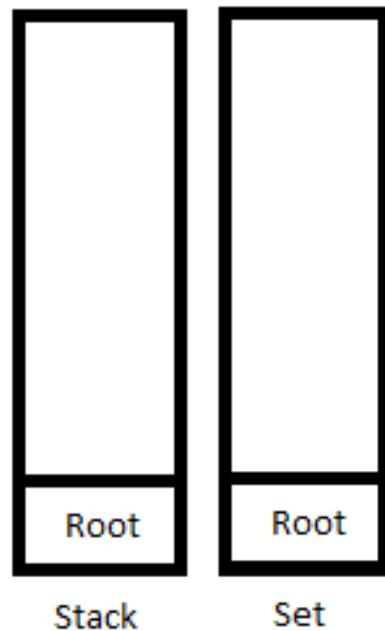
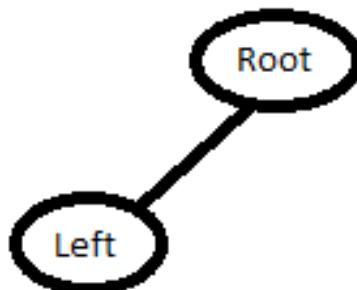
If the stack's topmost element is not in the set, link the created node to the left child of stack's topmost element (if any), without popping the element.

Else link the created node to the right child of stack's topmost element. Remove the stack's topmost element from the set and the stack.

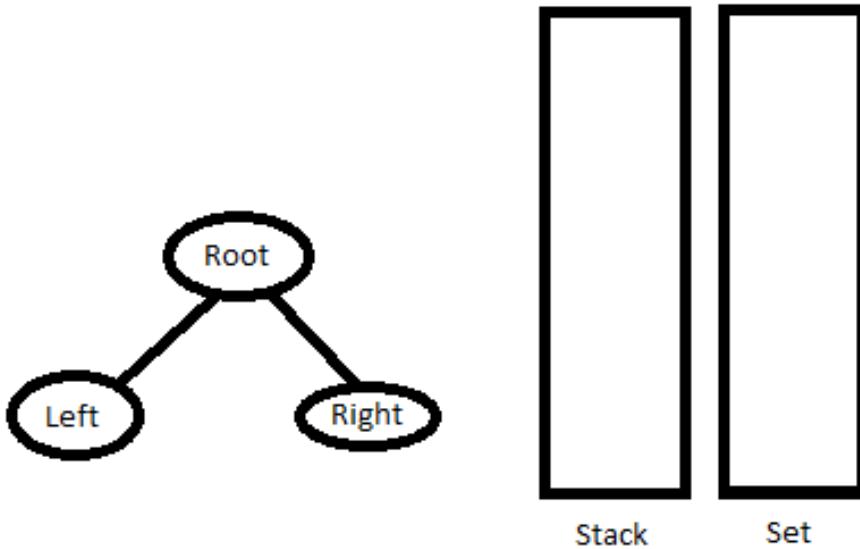
Push the node to a stack.



2. Keep popping the nodes from the stack until either the stack is empty, or the topmost element of stack compares to the current element of InOrder traversal. Once the loop is over, push the last node back into the stack and into the set.



3. Goto Step 1.



```
// Java program to construct a tree using inorder and preorder traversal
import java.util.*;

public class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;
    TreeNode(int x) { val = x; }
}

class BinaryTree {
    static Set<TreeNode> set = new HashSet<>();
    static Stack<TreeNode> stack = new Stack<>();

    // Function to build tree using given traversal
    public TreeNode buildTree(int[] preorder, int[] inorder)
    {

        TreeNode root = null;
        for (int pre = 0, in = 0; pre < preorder.length;) {

            TreeNode node = null;
            do {
                node = new TreeNode(preorder[pre]);
                if (root == null) {
                    root = node;
                }
                if (!stack.isEmpty()) {

```

```

        if (set.contains(stack.peek())) {
            set.remove(stack.peek());
            stack.pop().right = node;
        }
        else {
            stack.peek().left = node;
        }
    }
    stack.push(node);
} while (preorder[pre++] != inorder[in] && pre < preorder.length);

node = null;
while (!stack.isEmpty() && in < inorder.length && stack.peek().val == inorder[in]) {
    node = stack.pop();
    in++;
}

if (node != null) {
    set.add(node);
    stack.push(node);
}
}

return root;
}

// Function to print tree in Inorder
void printInorder(TreeNode node)
{
    if (node == null)
        return;

    /* first recur on left child */
    printInorder(node.left);

    /* then print the data of node */
    System.out.print(node.val + " ");

    /* now recur on right child */
    printInorder(node.right);
}

// driver program to test above functions
public static void main(String args[])
{
    BinaryTree tree = new BinaryTree();

    int in[] = new int[] { 9, 8, 4, 2, 10, 5, 10, 1, 6, 3, 13, 12, 7 };

```

```
int pre[] = new int[] { 1, 2, 4, 8, 9, 5, 10, 10, 3, 6, 7, 12, 13 };
int len = in.length;

TreeNode root = tree.buildTree(pre, in);

tree.printInorder(root);
}

}
```

Output :

```
9 8 4 2 10 5 10 1 6 3 13 12 7
```

Thanks Hardik Agarwal for suggesting this approach.

[Construct a Binary Tree from Postorder and Inorder](#)

## Source

<https://www.geeksforgeeks.org/construct-tree-from-given-inorder-and-preorder-traversal/>

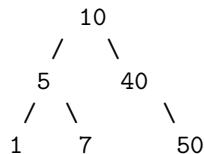
## Chapter 79

# Construct a Binary Search Tree from given postorder

Construct a Binary Search Tree from given postorder - GeeksforGeeks

Given postorder traversal of a binary search tree, construct the BST.

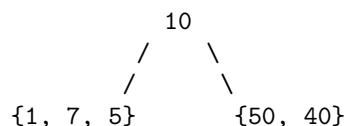
For example, if the given traversal is {1, 7, 5, 50, 40, 10}, then following tree should be constructed and root of the tree should be returned.



### Method 1 ( O(n^2) time complexity )

The last element of postorder traversal is always root. We first construct the root. Then we find the index of last element which is smaller than root. Let the index be ‘i’. The values between 0 and ‘i’ are part of left subtree, and the values between ‘i+1’ and ‘n-2’ are part of right subtree. Divide given post[] at index “i” and recur for left and right sub-trees.

For example in {1, 7, 5, 40, 50, 10}, 10 is the last element, so we make it root. Now we look for the last element smaller than 10, we find 5. So we know the structure of BST is as following.



We recursively follow above steps for subarrays {1, 7, 5} and {40, 50}, and get the complete tree.

### Method 2 ( O(n) time complexity )

The trick is to set a range {min .. max} for every node. Initialize the range as {INT\_MIN .. INT\_MAX}. The last node will definitely be in range, so create root node. To construct the left subtree, set the range as {INT\_MIN ...root->data}. If a values is in the range {INT\_MIN .. root->data}, the values is part part of left subtree. To construct the right subtree, set the range as {root->data .. INT\_MAX}.

Following code is used to generate the exact Binary Search Tree of a given post order traversal.

C

```
/* A O(n) program for construction of BST from
   postorder traversal */

#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
    int data;
    struct node *left, *right;
};

// A utility function to create a node
struct node* newNode (int data)
{
    struct node* temp =
        (struct node *) malloc( sizeof(struct node));

    temp->data = data;
    temp->left = temp->right = NULL;

    return temp;
}

// A recursive function to construct BST from post[].
// postIndex is used to keep track of index in post[].
struct node* constructTreeUtil(int post[], int* postIndex,
                               int key, int min, int max, int size)
{
    // Base case
    if (*postIndex < 0)
        return NULL;
```

```
struct node* root = NULL;

// If current element of post[] is in range, then
// only it is part of current subtree
if (key > min && key < max)
{
    // Allocate memory for root of this subtree and decrement
    // *postIndex
    root = newNode(key);
    *postIndex = *postIndex - 1;

    if (*postIndex >= 0)
    {

        // All nodes which are in range {key..max} will go in right
        // subtree, and first such node will be root of right subtree.
        root->right = constructTreeUtil(post, postIndex, post[*postIndex],
                                         max, size );

        // Construct the subtree under root
        // All nodes which are in range {min .. key} will go in left
        // subtree, and first such node will be root of left subtree.
        root->left = constructTreeUtil(post, postIndex, post[*postIndex],
                                         min, key, size );
    }
}
return root;
}

// The main function to construct BST from given postorder
// traversal. This function mainly uses constructTreeUtil()
struct node *constructTree (int post[], int size)
{
    int postIndex = size-1;
    return constructTreeUtil(post, &postIndex, post[postIndex],
                            INT_MIN, INT_MAX, size);
}

// A utility function to print inorder traversal of a Binary Tree
void printInorder (struct node* node)
{
    if (node == NULL)
        return;
    printInorder(node->left);
    printf("%d ", node->data);
    printInorder(node->right);
}
```

```
// Driver program to test above functions
int main ()
{
    int post[] = {1, 7, 5, 50, 40, 10};
    int size = sizeof(post) / sizeof(post[0]);

    struct node *root = constructTree(post, size);

    printf("Inorder traversal of the constructed tree: \n");
    printInorder(root);

    return 0;
}
```

**Java**

```
/* A O(n) program for construction of BST from
   postorder traversal */

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
class Node
{
    int data;
    Node left, right;

    Node(int data)
    {
        this.data = data;
        left = right = null;
    }
}

// Class containing variable that keeps a track of overall
// calculated postindex
class Index
{
    int postindex = 0;
}

class BinaryTree
{
    // A recursive function to construct BST from post[].
    // postIndex is used to keep track of index in post[].
    Node constructTreeUtil(int post[], Index postIndex,
                          int key, int min, int max, int size)
{
```

```

// Base case
if (postIndex.postindex < 0)
    return null;

Node root = null;

// If current element of post[] is in range, then
// only it is part of current subtree
if (key > min && key < max)
{
    // Allocate memory for root of this subtree and decrement
    // *postIndex
    root = new Node(key);
    postIndex.postindex = postIndex.postindex - 1;

    if (postIndex.postindex > 0)
    {
        // All nodes which are in range {key..max} will go in
        // right subtree, and first such node will be root of right
        // subtree
        root.right = constructTreeUtil(post, postIndex,
                                         post[postIndex.postindex],key, max, size);

        // Construct the subtree under root
        // All nodes which are in range {min .. key} will go in left
        // subtree, and first such node will be root of left subtree.
        root.left = constructTreeUtil(post, postIndex,
                                       post[postIndex.postindex],min, key, size);
    }
}
return root;
}

// The main function to construct BST from given postorder
// traversal. This function mainly uses constructTreeUtil()
Node constructTree(int post[], int size)
{
    Index index = new Index();
    index.postindex = size - 1;
    return constructTreeUtil(post, index, post[index.postindex],
                            Integer.MIN_VALUE, Integer.MAX_VALUE, size);
}

// A utility function to print inorder traversal of a Binary Tree
void printInorder(Node node)
{
    if (node == null)
        return;

```

```
        printInorder(node.left);
        System.out.print(node.data + " ");
        printInorder(node.right);
    }

// Driver program to test above functions
public static void main(String[] args)
{
    BinaryTree tree = new BinaryTree();
    int post[] = new int[]{1, 7, 5, 50, 40, 10};
    int size = post.length;

    Node root = tree.constructTree(post, size);

    System.out.println("Inorder traversal of the constructed tree:");
    tree.printInorder(root);
}
}

// This code has been contributed by Mayank Jaiswal
```

Output:

```
Inorder traversal of the constructed tree:
1 5 7 10 40 50
```

*Note that the output to the program will always be a sorted sequence as we are printing the inorder traversal of a Binary Search Tree.*

## Source

<https://www.geeksforgeeks.org/construct-a-binary-search-tree-from-given-postorder/>

## Chapter 80

# Construct a Binary Tree from Postorder and Inorder

Construct a Binary Tree from Postorder and Inorder - GeeksforGeeks

Given Postorder and Inorder traversals, construct the tree.

Examples:

```
Input :  
in[] = {2, 1, 3}  
post[] = {2, 3, 1}
```

```
Output : Root of below tree  
         1  
       /   \br/>      2     3
```

```
Input :  
in[] = {4, 8, 2, 5, 1, 6, 3, 7}  
post[] = {8, 4, 5, 2, 6, 7, 3, 1}
```

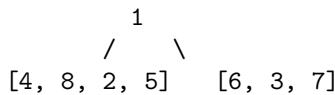
```
Output : Root of below tree  
         1  
       /   \br/>      2     3  
     /   \   /   \br/>    4     5   6   7  
           \br/>          8
```

We have already discussed [construction of tree from given Inorder and Preorder traversals](#). The idea is similar.

Let us see the process of constructing tree from  $\text{in}[] = \{4, 8, 2, 5, 1, 6, 3, 7\}$  and  $\text{post}[] = \{8, 4, 5, 2, 6, 7, 3, 1\}$

1) We first find the last node in  $\text{post}[]$ . The last node is “1”, we know this value is root as root always appear in the end of postorder traversal.

2) We search “1” in  $\text{in}[]$  to find left and right subtrees of root. Everything on left of “1” in  $\text{in}[]$  is in left subtree and everything on right is in right subtree.



3) We recur the above process for following two.

....b) Recur for  $\text{in}[] = \{6, 3, 7\}$  and  $\text{post}[] = \{6, 7, 3\}$

.....Make the created tree as right child of root.

....a) Recur for  $\text{in}[] = \{4, 8, 2, 5\}$  and  $\text{post}[] = \{8, 4, 5, 2\}$ .

.....Make the created tree as left child of root.

Below is C++ implementation of above idea. One important observation is, we recursively call for right subtree before left subtree as we decrease index of postorder index whenever we create a new node.

C++

```

/* C++ program to construct tree using inorder and
   postorder traversals */
#include <bits/stdc++.h>
using namespace std;

/* A binary tree node has data, pointer to left
   child and a pointer to right child */
struct Node {
    int data;
    Node *left, *right;
};

// Utility function to create a new node
Node* newNode(int data);

/* Prototypes for utility functions */
int search(int arr[], int strt, int end, int value);

/* Recursive function to construct binary of size n
   from Inorder traversal in[] and Postorder traversal
   post[] */
  
```

```

post[] . Initial values of inStrt and inEnd should
be 0 and n -1. The function doesn't do any error
checking for cases where inorder and postorder
do not form a tree */
Node* buildUtil(int in[], int post[], int inStrt,
                int inEnd, int* pIndex)
{
    // Base case
    if (inStrt > inEnd)
        return NULL;

    /* Pick current node from Postorder traversal using
       postIndex and decrement postIndex */
    Node* node = newNode(post[*pIndex]);
    (*pIndex)--;

    /* If this node has no children then return */
    if (inStrt == inEnd)
        return node;

    /* Else find the index of this node in Inorder
       traversal */
    int iIndex = search(in, inStrt, inEnd, node->data);

    /* Using index in Inorder traversal, construct left and
       right subtress */
    node->right = buildUtil(in, post, iIndex + 1, inEnd, pIndex);
    node->left = buildUtil(in, post, inStrt, iIndex - 1, pIndex);

    return node;
}

// This function mainly initializes index of root
// and calls buildUtil()
Node* buildTree(int in[], int post[], int n)
{
    int pIndex = n - 1;
    return buildUtil(in, post, 0, n - 1, &pIndex);
}

/* Function to find index of value in arr[start...end]
   The function assumes that value is postsent in in[] */
int search(int arr[], int strt, int end, int value)
{
    int i;
    for (i = strt; i <= end; i++) {
        if (arr[i] == value)
            break;
}

```

```
        }
        return i;
    }

/* Helper function that allocates a new node */
Node* newNode(int data)
{
    Node* node = (Node*)malloc(sizeof(Node));
    node->data = data;
    node->left = node->right = NULL;
    return (node);
}

/* This function is here just to test */
void preOrder(Node* node)
{
    if (node == NULL)
        return;
    printf("%d ", node->data);
    preOrder(node->left);
    preOrder(node->right);
}

// Driver code
int main()
{
    int in[] = { 4, 8, 2, 5, 1, 6, 3, 7 };
    int post[] = { 8, 4, 5, 2, 6, 7, 3, 1 };
    int n = sizeof(in) / sizeof(in[0]);

    Node* root = buildTree(in, post, n);

    cout << "Preorder of the constructed tree : \n";
    preOrder(root);

    return 0;
}
```

**Java**

```
// Java program to construct a tree using inorder
// and postorder traversals

/* A binary tree node has data, pointer to left
   child and a pointer to right child */
class Node {
    int data;
    Node left, right;
```

```
public Node(int data)
{
    this.data = data;
    left = right = null;
}
}

// Class Index created to implement pass by reference of Index
class Index {
    int index;
}

class BinaryTree {
    /* Recursive function to construct binary of size n
       from Inorder traversal in[] and Postorder traversal
       post[]. Initial values of inStrt and inEnd should
       be 0 and n -1. The function doesn't do any error
       checking for cases where inorder and postorder
       do not form a tree */
    Node buildUtil(int in[], int post[], int inStrt,
                  int inEnd, Index pIndex)
    {
        // Base case
        if (inStrt > inEnd)
            return null;

        /* Pick current node from Postorder traversal using
           postIndex and decrement postIndex */
        Node node = new Node(post[pIndex.index]);
        (pIndex.index)--;

        /* If this node has no children then return */
        if (inStrt == inEnd)
            return node;

        /* Else find the index of this node in Inorder
           traversal */
        int iIndex = search(in, inStrt, inEnd, node.data);

        /* Using index in Inorder traversal, construct left and
           right subtress */
        node.right = buildUtil(in, post, iIndex + 1, inEnd, pIndex);
        node.left = buildUtil(in, post, inStrt, iIndex - 1, pIndex);

        return node;
    }
}
```

```
// This function mainly initializes index of root
// and calls buildUtil()
Node buildTree(int in[], int post[], int n)
{
    Index pIndex = new Index();
    pIndex.index = n - 1;
    return buildUtil(in, post, 0, n - 1, pIndex);
}

/* Function to find index of value in arr[start...end]
   The function assumes that value is postsent in in[] */
int search(int arr[], int strt, int end, int value)
{
    int i;
    for (i = strt; i <= end; i++) {
        if (arr[i] == value)
            break;
    }
    return i;
}

/* This funtcion is here just to test  */
void preOrder(Node node)
{
    if (node == null)
        return;
    System.out.print(node.data + " ");
    preOrder(node.left);
    preOrder(node.right);
}

public static void main(String[] args)
{
    BinaryTree tree = new BinaryTree();
    int in[] = new int[] { 4, 8, 2, 5, 1, 6, 3, 7 };
    int post[] = new int[] { 8, 4, 5, 2, 6, 7, 3, 1 };
    int n = in.length;
    Node root = tree.buildTree(in, post, n);
    System.out.println("Preorder of the constructed tree : ");
    tree.preOrder(root);
}

// This code has been contributed by Mayank Jaiswal(mayank_24)
```

Output :

Preorder of the constructed tree :

1 2 4 8 5 3 6 7

Time Complexity :  $O(n^2)$

**Optimized approach:** We can optimize the above solution using hashing (unordered\_map in C++ or HashMap in Java). We store indexes of inorder traversal in a hash table. So that search can be done  $O(1)$  time.

```
/* C++ program to construct tree using inorder and
postorder traversals */
#include <bits/stdc++.h>
using namespace std;

/* A binary tree node has data, pointer to left
child and a pointer to right child */
struct Node {
    int data;
    Node *left, *right;
};

// Utility function to create a new node
Node* newNode(int data);

/* Recursive function to construct binary of size n
from Inorder traversal in[] and Postorder traversal
post[]. Initial values of inStrt and inEnd should
be 0 and n -1. The function doesn't do any error
checking for cases where inorder and postorder
do not form a tree */
Node* buildUtil(int in[], int post[], int inStrt,
                int inEnd, int* pIndex, unordered_map<int, int>& mp)
{
    // Base case
    if (inStrt > inEnd)
        return NULL;

    /* Pick current node from Postorder traversal
    using postIndex and decrement postIndex */
    int curr = post[*pIndex];
    Node* node = newNode(curr);
    (*pIndex)--;

    /* If this node has no children then return */
    if (inStrt == inEnd)
        return node;

    /* Else find the index of this node in Inorder
    */
```

```

traversal */
int iIndex = mp[curr];

/* Using index in Inorder traversal, construct
left and right subtress */
node->right = buildUtil(in, post, iIndex + 1,
                         inEnd, pIndex, mp);
node->left = buildUtil(in, post, inStrt,
                        iIndex - 1, pIndex, mp);

return node;
}

// This function mainly creates an unordered_map, then
// calls buildTreeUtil()
struct Node* buildTree(int in[], int post[], int len)
{
    // Store indexes of all items so that we
    // we can quickly find later
    unordered_map<int, int> mp;
    for (int i = 0; i < len; i++)
        mp[in[i]] = i;

    int index = len - 1; // Index in postorder
    return buildUtil(in, post, 0, len - 1,
                     &index, mp);
}

/* Helper function that allocates a new node */
Node* newNode(int data)
{
    Node* node = (Node*)malloc(sizeof(Node));
    node->data = data;
    node->left = node->right = NULL;
    return (node);
}

/* This function is here just to test */
void preOrder(Node* node)
{
    if (node == NULL)
        return;
    printf("%d ", node->data);
    preOrder(node->left);
    preOrder(node->right);
}

// Driver code

```

```
int main()
{
    int in[] = { 4, 8, 2, 5, 1, 6, 3, 7 };
    int post[] = { 8, 4, 5, 2, 6, 7, 3, 1 };
    int n = sizeof(in) / sizeof(in[0]);

    Node* root = buildTree(in, post, n);

    cout << "Preorder of the constructed tree : \n";
    preOrder(root);

    return 0;
}
```

## Source

<https://www.geeksforgeeks.org/construct-a-binary-tree-from-postorder-and-inorder/>

Time Complexity : O(n)

This article is contributed by **Rishi**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## Chapter 81

# Construct a complete binary tree from given array in level order fashion

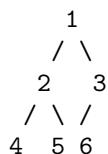
Construct a complete binary tree from given array in level order fashion - GeeksforGeeks

Given an array of elements, our task is to construct a complete binary tree from this array in level order fashion. That is, elements from left in the array will be filled in the tree level wise starting from level 0.

Examples:

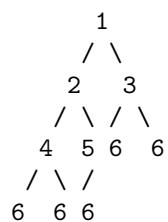
Input : arr[] = {1, 2, 3, 4, 5, 6}

Output : Root of the following tree



Input: arr[] = {1, 2, 3, 4, 5, 6, 6, 6, 6, 6}

Output: Root of the following tree



If we observe carefully we can see that if parent node is at index  $i$  in the array then the left child of that node is at index  $(2*i + 1)$  and right child is at index  $(2*i + 2)$  in the array. Using this concept, we can easily insert the left and right nodes by choosing its parent node. We will insert the first element present in the array as the root node at level 0 in the tree and start traversing the array and for every node  $i$  we will insert its both childs left and right in the tree.

Below is the recursive program to do this:

C++

```
// CPP program to construct binary
// tree from given array in level
// order fashion Tree Node
#include <bits/stdc++.h>
using namespace std;

/* A binary tree node has data,
pointer to left child and a
pointer to right child */
struct Node
{
    int data;
    Node* left, * right;
};

/* Helper function that allocates a
new node */
Node* newNode(int data)
{
    Node* node = (Node*)malloc(sizeof(Node));
    node->data = data;
    node->left = node->right = NULL;
    return (node);
}

// Function to insert nodes in level order
Node* insertLevelOrder(int arr[], Node* root,
                      int i, int n)
{
    // Base case for recursion
    if (i < n)
    {
        Node* temp = newNode(arr[i]);
        root = temp;

        // insert left child
        root->left = insertLevelOrder(arr,
                                       root->left, 2 * i + 1, n);
    }
}
```

```
// insert right child
root->right = insertLevelOrder(arr,
                                root->right, 2 * i + 2, n);
}
return root;
}

// Function to print tree nodes in
// InOrder fashion
void inOrder(Node* root)
{
    if (root != NULL)
    {
        inOrder(root->left);
        cout << root->data << " ";
        inOrder(root->right);
    }
}

// Driver program to test above function
int main()
{
    int arr[] = { 1, 2, 3, 4, 5, 6, 6, 6 };
    int n = sizeof(arr)/sizeof(arr[0]);
    Node* root = insertLevelOrder(arr, root, 0, n);
    inOrder(root);
}

// This code is contributed by Chhavi
```

### Java

```
// Java program to construct binary tree from
// given array in level order fashion

public class Tree {
    Node root;

    // Tree Node
    static class Node {
        int data;
        Node left, right;
        Node(int data)
        {
            this.data = data;
            this.left = null;
            this.right = null;
```

```
}

}

// Function to insert nodes in level order
public Node insertLevelOrder(int[] arr, Node root,
                             int i)
{
    // Base case for recursion
    if (i < arr.length) {
        Node temp = new Node(arr[i]);
        root = temp;

        // insert left child
        root.left = insertLevelOrder(arr, root.left,
                                     2 * i + 1);

        // insert right child
        root.right = insertLevelOrder(arr, root.right,
                                     2 * i + 2);
    }
    return root;
}

// Function to print tree nodes in InOrder fashion
public void inOrder(Node root)
{
    if (root != null) {
        inOrder(root.left);
        System.out.print(root.data + " ");
        inOrder(root.right);
    }
}

// Driver program to test above function
public static void main(String args[])
{
    Tree t2 = new Tree();
    int arr[] = { 1, 2, 3, 4, 5, 6, 6, 6, 6 };
    t2.root = t2.insertLevelOrder(arr, t2.root, 0);
    t2.inOrder(t2.root);
}
```

Output:

6 4 6 2 5 1 6 3 6

**Time Complexity:**  $O(n)$ , where  $n$  is the total number of nodes in the tree.

## Source

<https://www.geeksforgeeks.org/construct-complete-binary-tree-given-array/>

## Chapter 82

# Construct a special tree from given preorder traversal

Construct a special tree from given preorder traversal - GeeksforGeeks

Given an array ‘pre[]’ that represents Preorder traversal of a spacial binary tree where every node has either 0 or 2 children. One more array ‘preLN[]’ is given which has only two possible values ‘L’ and ‘N’. The value ‘L’ in ‘preLN[]’ indicates that the corresponding node in Binary Tree is a leaf node and value ‘N’ indicates that the corresponding node is non-leaf node. Write a function to construct the tree from the given two arrays.

Source: Amazon Interview Question

Example:

Input: pre[] = {10, 30, 20, 5, 15}, preLN[] = {'N', 'N', 'L', 'L', 'L'}  
Output: Root of following tree  

```
      10
      / \
     30   15
    /   \
   20   5
```

The first element in pre[] will always be root. So we can easily figure out root. If left subtree is empty, the right subtree must also be empty and preLN[] entry for root must be ‘L’. We can simply create a node and return it. If left and right subtrees are not empty, then recursively call for left and right subtrees and link the returned nodes to root.

C

```
/* A program to construct Binary Tree from preorder traversal */
#include<stdio.h>
```

```
/* A binary tree node structure */
struct node
{
    int data;
    struct node *left;
    struct node *right;
};

/* Utility function to create a new Binary Tree node */
struct node* newNode (int data)
{
    struct node *temp = new struct node;
    temp->data = data;
    temp->left = NULL;
    temp->right = NULL;
    return temp;
}

/* A recursive function to create a Binary Tree from given pre[]
   preLN[] arrays. The function returns root of tree. index_ptr is used
   to update index values in recursive calls. index must be initially
   passed as 0 */
struct node *constructTreeUtil(int pre[], char preLN[], int *index_ptr, int n)
{
    int index = *index_ptr; // store the current value of index in pre[]

    // Base Case: All nodes are constructed
    if (index == n)
        return NULL;

    // Allocate memory for this node and increment index for
    // subsequent recursive calls
    struct node *temp = newNode ( pre[index] );
    (*index_ptr)++;

    // If this is an internal node, construct left and right subtrees and link the subtrees
    if (preLN[index] == 'N')
    {
        temp->left = constructTreeUtil(pre, preLN, index_ptr, n);
        temp->right = constructTreeUtil(pre, preLN, index_ptr, n);
    }

    return temp;
}

// A wrapper over constructTreeUtil()
struct node *constructTree(int pre[], char preLN[], int n)
{
```

```
// Initialize index as 0. Value of index is used in recursion to maintain
// the current index in pre[] and preLN[] arrays.
int index = 0;

    return constructTreeUtil (pre, preLN, &index, n);
}

/* This function is used only for testing */
void printInorder (struct node* node)
{
    if (node == NULL)
        return;

    /* first recur on left child */
    printInorder (node->left);

    /* then print the data of node */
    printf("%d ", node->data);

    /* now recur on right child */
    printInorder (node->right);
}

/* Driver function to test above functions */
int main()
{
    struct node *root = NULL;

    /* Constructing tree given in the above figure
       10
      / \
     30   15
    / \
   20   5 */
    int pre[] = {10, 30, 20, 5, 15};
    char preLN[] = {'N', 'N', 'L', 'L', 'L'};
    int n = sizeof(pre)/sizeof(pre[0]);

    // construct the above tree
    root = constructTree (pre, preLN, n);

    // Test the constructed tree
    printf("Following is Inorder Traversal of the Constructed Binary Tree: \n");
    printInorder (root);

    return 0;
}
```

**Java**

```
// Java program to construct a binary tree from preorder traversal

// A Binary Tree node
class Node
{
    int data;
    Node left, right;

    Node(int item)
    {
        data = item;
        left = right = null;
    }
}

class Index
{
    int index = 0;
}

class BinaryTree
{
    Node root;
    Index myindex = new Index();

    /* A recursive function to create a Binary Tree from given pre[]
       preLN[] arrays. The function returns root of tree. index_ptr is used
       to update index values in recursive calls. index must be initially
       passed as 0 */
    Node constructTreeUtil(int pre[], char preLN[], Index index_ptr,
                           int n, Node temp)
    {
        // store the current value of index in pre[]
        int index = index_ptr.index;

        // Base Case: All nodes are constructed
        if (index == n)
            return null;

        // Allocate memory for this node and increment index for
        // subsequent recursive calls
        temp = new Node(pre[index]);
        (index_ptr.index)++;
    }

    // If this is an internal node, construct left and right subtrees
    // and link the subtrees
```

```
if (preLN[index] == 'N')
{
    temp.left = constructTreeUtil(pre, preLN, index_ptr, n,
                                    temp.left);
    temp.right = constructTreeUtil(pre, preLN, index_ptr, n,
                                   temp.right);
}

return temp;
}

// A wrapper over constructTreeUtil()
Node constructTree(int pre[], char preLN[], int n, Node node)
{
    // Initialize index as 0. Value of index is used in recursion to
    // maintain the current index in pre[] and preLN[] arrays.
    int index = 0;

    return constructTreeUtil(pre, preLN, myindex, n, node);
}

/* This function is used only for testing */
void printInorder(Node node)
{
    if (node == null)
        return;

    /* first recur on left child */
    printInorder(node.left);

    /* then print the data of node */
    System.out.print(node.data + " ");

    /* now recur on right child */
    printInorder(node.right);
}

// driver function to test the above functions
public static void main(String args[])
{
    BinaryTree tree = new BinaryTree();
    int pre[] = new int[]{10, 30, 20, 5, 15};
    char preLN[] = new char[]{'N', 'N', 'L', 'L', 'L'};
    int n = pre.length;

    // construct the above tree
    Node mynode = tree.constructTree(pre, preLN, n, tree.root);
}
```

```
// Test the constructed tree
System.out.println("Following is Inorder Traversal of the"
+ "Constructed Binary Tree: ");
tree.printInorder(mynode);
}

// This code has been contributed by Mayank Jaiswal
```

Output:

```
Following is Inorder Traversal of the Constructed Binary Tree:
20 30 5 10 15
```

Time Complexity: O(n)

[Construct the full k-ary tree from its preorder traversal](#)

## Source

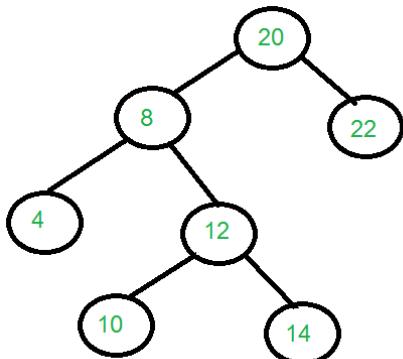
<https://www.geeksforgeeks.org/construct-a-special-tree-from-given-preorder-traversal/>

## Chapter 83

# Construct a tree from Inorder and Level order traversals | Set 1

Construct a tree from Inorder and Level order traversals | Set 1 - GeeksforGeeks

Given inorder and level-order traversals of a Binary Tree, construct the Binary Tree. Following is an example to illustrate the problem.



Input: Two arrays that represent Inorder and level order traversals of a Binary Tree  
in[] = {4, 8, 10, 12, 14, 20, 22};  
level[] = {20, 8, 22, 4, 12, 10, 14};

Output: Construct the tree represented by the two arrays.

For the above two arrays, the constructed tree is shown in the diagram on right side

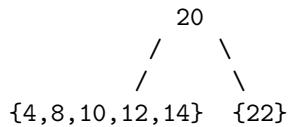
The following post can be considered as a prerequisite for this.

[Construct Tree from given Inorder and Preorder traversals](#)

Let us consider the above example.

```
in[] = {4, 8, 10, 12, 14, 20, 22};  
level[] = {20, 8, 22, 4, 12, 10, 14};
```

In a Levelorder sequence, the first element is the root of the tree. So we know '20' is root for given sequences. By searching '20' in Inorder sequence, we can find out all elements on left side of '20' are in left subtree and elements on right are in right subtree. So we know below structure now.



Let us call  $\{4,8,10,12,14\}$  as left subarray in Inorder traversal and  $\{22\}$  as right subarray in Inorder traversal.

In level order traversal, keys of left and right subtrees are not consecutive. So we extract all nodes from level order traversal which are in left subarray of Inorder traversal. To construct the left subtree of root, we recur for the extracted elements from level order traversal and left subarray of inorder traversal. In the above example, we recur for following two arrays.

```
// Recur for following arrays to construct the left subtree  
In[]    = {4, 8, 10, 12, 14}  
level[] = {8, 4, 12, 10, 14}
```

Similarly, we recur for following two arrays and construct the right subtree.

```
// Recur for following arrays to construct the right subtree  
In[]    = {22}  
level[] = {22}
```

Following is the implementation of the above approach.

C

```

/* program to construct tree using inorder and levelorder traversals */
#include <iostream>
using namespace std;

/* A binary tree node */
struct Node
{
    int key;
    struct Node* left, *right;
};

/* Function to find index of value in arr[start...end] */
int search(int arr[], int strt, int end, int value)
{
    for (int i = strt; i <= end; i++)
        if (arr[i] == value)
            return i;
    return -1;
}

// n is size of level[], m is size of in[] and m < n. This
// function extracts keys from level[] which are present in
// in[]. The order of extracted keys must be maintained
int *extractKeys(int in[], int level[], int m, int n)
{
    int *newlevel = new int[m], j = 0;
    for (int i = 0; i < n; i++)
        if (search(in, 0, m-1, level[i]) != -1)
            newlevel[j] = level[i], j++;
    return newlevel;
}

/* function that allocates a new node with the given key */
Node* newNode(int key)
{
    Node *node = new Node;
    node->key = key;
    node->left = node->right = NULL;
    return (node);
}

/* Recursive function to construct binary tree of size n from
   Inorder traversal in[] and Level Order traversal level[].
   inSrt and inEnd are start and end indexes of array in[]
   Initial values of inStrt and inEnd should be 0 and n -1.
   The function doesn't do any error checking for cases
   where inorder and levelorder do not form a tree */
Node* buildTree(int in[], int level[], int inStrt, int inEnd, int n)

```

```
{

    // If start index is more than the end index
    if (inStrt > inEnd)
        return NULL;

    /* The first node in level order traversal is root */
    Node *root = newNode(level[0]);

    /* If this node has no children then return */
    if (inStrt == inEnd)
        return root;

    /* Else find the index of this node in Inorder traversal */
    int inIndex = search(in, inStrt, inEnd, root->key);

    // Extract left subtree keys from level order traversal
    int *llevel = extractKeys(in, level, inIndex, n);

    // Extract right subtree keys from level order traversal
    int *rlevel = extractKeys(in + inIndex + 1, level, n-inIndex-1, n);

    /* construct left and right subtress */
    root->left = buildTree(in, llevel, inStrt, inIndex-1, n);
    root->right = buildTree(in, rlevel, inIndex+1, inEnd, n);

    // Free memory to avoid memory leak
    delete [] llevel;
    delete [] rlevel;

    return root;
}

/* Utility function to print inorder traversal of binary tree */
void printInorder(Node* node)
{
    if (node == NULL)
        return;
    printInorder(node->left);
    cout << node->key << " ";
    printInorder(node->right);
}

/* Driver program to test above functions */
int main()
{
    int in[] = {4, 8, 10, 12, 14, 20, 22};
    int level[] = {20, 8, 22, 4, 12, 10, 14};
```

```
int n = sizeof(in)/sizeof(in[0]);
Node *root = buildTree(in, level, 0, n - 1, n);

/* Let us test the built tree by printing Insorder traversal */
cout << "Inorder traversal of the constructed tree is \n";
printInorder(root);

return 0;
}
```

**Java**

```
// Java program to construct a tree from level order and
// and inorder traversal

// A binary tree node
class Node
{
    int data;
    Node left, right;

    Node(int item)
    {
        data = item;
        left = right = null;
    }

    public void setLeft(Node left)
    {
        this.left = left;
    }

    public void setRight(Node right)
    {
        this.right = right;
    }
}

class Tree
{
    Node root;

    Node buildTree(int in[], int level[])
    {
        Node startnode = null;
        return constructTree(startnode, level, in, 0, in.length - 1);
    }
}
```

```
Node constructTree(Node startNode, int[] levelOrder, int[] inOrder,
                   int inStart, int inEnd)
{
    // if start index is more than end index
    if (inStart > inEnd)
        return null;

    if (inStart == inEnd)
        return new Node(inOrder[inStart]);

    boolean found = false;
    int index = 0;

    // it represents the index in inOrder array of element that
    // appear first in levelOrder array.
    for (int i = 0; i < levelOrder.length - 1; i++)
    {
        int data = levelOrder[i];
        for (int j = inStart; j < inEnd; j++)
        {
            if (data == inOrder[j])
            {
                startNode = new Node(data);
                index = j;
                found = true;
                break;
            }
        }
        if (found == true)
            break;
    }

    //elements present before index are part of left child of startNode.
    //elements present after index are part of right child of startNode.
    startNode.setLeft(constructTree(startNode, levelOrder, inOrder,
                                    inStart, index - 1));
    startNode.setRight(constructTree(startNode, levelOrder, inOrder,
                                    index + 1, inEnd));

    return startNode;
}

/* Utility function to print inorder traversal of binary tree */
void printInorder(Node node)
{
    if (node == null)
        return;
```

```
    printInorder(node.left);
    System.out.print(node.data + " ");
    printInorder(node.right);
}

// Driver program to test the above functions
public static void main(String args[])
{
    Tree tree = new Tree();
    int in[] = new int[]{4, 8, 10, 12, 14, 20, 22};
    int level[] = new int[]{20, 8, 22, 4, 12, 10, 14};
    int n = in.length;
    Node node = tree.buildTree(in, level);

    /* Let us test the built tree by printing Inorder traversal */
    System.out.print("Inorder traversal of the constructed tree is ");
    tree.printInorder(node);
}
}

// This code has been contributed by Mayank Jaiswal
```

### Python3

```
# Python program to construct tree using
# inorder and level order traversals

# A binary tree node
class Node:

    # Constructor to create a new node
    def __init__(self, key):
        self.data = key
        self.left = None
        self.right = None

    """Recursive function to construct binary tree of size n from
    Inorder traversal ino[] and Level Order traversal level[].
    The function doesn't do any error checking for cases
    where inorder and levelorder do not form a tree """
def buildTree(level, ino):

    # If ino array is not empty
    if ino :

        # Check if that element exist in level order
        for i in range(0, len(level)):
```

```
if level[i] in ino:

    # Create a new node with
    # the matched element
    node = Node(level[i])

    # Get the index of the matched element
    # in level order array
    io_index = ino.index(level[i])
    break

# If inorder array is empty return node
if not ino:
    return node

# Construct left and right subtree
node.left = buildTree(level, ino[0:io_index])
node.right = buildTree(level, ino[io_index + 1:len(ino)])
return node

def printInorder(node):
    if node is None:
        return

    # first recur on left child
    printInorder(node.left)

    # then print the data of node
    print(node.data, end=" ")

    # now recur on right child
    printInorder(node.right)

# Driver code

levelorder = [20, 8, 22, 4, 12, 10, 14]
inorder = [4, 8, 10, 12, 14, 20, 22]

ino_len = len(inorder)
root = buildTree(levelorder, inorder)

# Let us test the build tree by
# printing Inorder traversal
print ("Inorder traversal of the constructed tree is")
printInorder(root)

# This code is contributed by 'Vaibhav Kumar'
```

Output:

```
Inorder traversal of the constructed tree is  
4 8 10 12 14 20 22
```

An upper bound on time complexity of above method is  $O(n^3)$ . In the main recursive function, extractNodes() is called which takes  $O(n^2)$  time.

The code can be optimized in many ways and there may be better solutions.

[Construct a tree from Inorder and Level order traversals | Set 2](#)

This article is contributed by **Abhay Rathi**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## Source

<https://www.geeksforgeeks.org/construct-tree-inorder-level-order-traversals/>

## Chapter 84

# Construct a tree from Inorder and Level order traversals | Set 2

Construct a tree from Inorder and Level order traversals | Set 2 - GeeksforGeeks

Given inorder and level-order traversals of a Binary Tree, construct the Binary Tree. Following is an example to illustrate the problem.

Examples:

Input: Two arrays that represent Inorder and level order traversals of a Binary Tree  
in[] = {4, 8, 10, 12, 14, 20, 22};  
level[] = {20, 8, 22, 4, 12, 10, 14};

Output: Construct the tree represented by the two arrays.  
For the above two arrays, the constructed tree is shown.

We have discussed a solution in below post that works in  $O(N^3)$   
[Construct a tree from Inorder and Level order traversals | Set 1](#)

**Approach :** Following algorithm uses  $O(N^2)$  time complexity to solve the above problem using the `unordered_set` data structure in c++ (basically making a hash-table) to put the values of left subtree of the current root and later and we will check in  $O(1)$  complexity to find if the current levelOrder node is part of left subtree or not.

If it is the part of left subtree then add in one lLevel array for left other wise add it to rLevel array for right subtree.

Below is the c++ implementation with the above idea

```
/* program to construct tree using inorder
   and levelorder traversals */
#include <iostream>
#include<set>
using namespace std;

/* A binary tree node */
struct Node
{
    int key;
    struct Node* left, *right;
};

Node* makeNode(int data){
    Node* newNode = new Node();
    newNode->key = data;
    newNode->right = newNode->right = NULL;
    return newNode;
}

// Function to build tree from given
// levelorder and inorder
Node* buildTree(int inorder[], int levelOrder[],
                int iStart, int iEnd, int n)
{
    if (n <= 0) return NULL;

    // First node of level order is root
    Node* root = makeNode(levelOrder[0]);

    // Search root in inorder
    int index = -1;
    for (int i=iStart; i<=iEnd; i++){
        if (levelOrder[0] == inorder[i]){
            index = i;
            break;
        }
    }

    // Insert all left nodes in hash table
    unordered_set<int> s;
    for (int i=iStart;i<index;i++)
        s.insert(inorder[i]);
```

```
// Separate level order traversals
// of left and right subtrees.
int lLevel[s.size()]; // Left
int rLevel[iEnd-iStart-s.size()]; // Right
int li = 0, ri = 0;
for (int i=1;i<n;i++) {
    if (s.find(levelOrder[i]) != s.end())
        lLevel[li++] = levelOrder[i];
    else
        rLevel[ri++] = levelOrder[i];
}

// Recursively build left and right
// subtrees and return root.
root->left = buildTree(inorder, lLevel,
                        iStart, index-1, index-iStart);
root->right = buildTree(inorder, rLevel,
                        index+1, iEnd, iEnd-index);
return root;

}

/* Utility function to print inorder
traversal of binary tree */
void printInorder(Node* node)
{
    if (node == NULL)
        return;
    printInorder(node->left);
    cout << node->key << " ";
    printInorder(node->right);
}

// Driver Code
int main()
{
    int in[] = {4, 8, 10, 12, 14, 20, 22};
    int level[] = {20, 8, 22, 4, 12, 10, 14};
    int n = sizeof(in)/sizeof(in[0]);
    Node *root = buildTree(in, level, 0,
                           n - 1, n);

    /* Let us test the built tree by
       printing Inorder traversal */
    cout << "Inorder traversal of the "
        "constructed tree is \n";
    printInorder(root);
```

```
    return 0;  
}
```

Output :

```
Inorder traversal of the  
constructed tree is 4 8 10 12 14  
20 22
```

Time Complexity:  $O(N^2)$

## Source

<https://www.geeksforgeeks.org/construct-tree-inorder-level-order-traversals-set-2/>

## Chapter 85

# Construct the full k-ary tree from its preorder traversal

Construct the full k-ary tree from its preorder traversal - GeeksforGeeks

Given an array which contains the preorder traversal of full k-ary tree, construct the full k-ary tree and print its postorder traversal. A full k-ary tree is a tree where each node has either 0 or k children.

Examples:

```
Input : preorder[] = {1, 2, 5, 6, 7,
                     3, 8, 9, 10, 4}
        k = 3
Output : Postorder traversal of constructed
         full k-ary tree is: 5 6 7 2 8 9 10
                           3 4 1
         Tree formed is:
                           1
                           /   |   \
                         2     3     4
                         / \   / \
                       5   6   8   9   10
```

```
Input : preorder[] = {1, 2, 5, 6, 7, 3, 4}
        k = 3
Output : Postorder traversal of constructed
         full k-ary tree is: 5 6 7 2 3 4 1
         Tree formed is:
                           1
                           /   |   \
                         2     3     4
                         / \
                       5   6   7
```

We have discussed this problem for Binary tree in below post.

[Construct a special tree from given preorder traversal](#)

In this post, solution for a k-ary tree is discussed.

In [Preorder traversal](#), first root node is processed then followed by the left subtree and right subtree. Because of this, to construct a full k-ary tree, we just need to keep on creating the nodes without bothering about the previous constructed nodes. We can use this to build the tree recursively.

Following are the steps to solve the problem:

1. Find the height of the tree.
2. Traverse the preorder array and recursively add each node

```
// C++ program to build full k-ary tree from
// its preorder traversal and to print the
// postorder traversal of the tree.
#include <bits/stdc++.h>
using namespace std;

// Structure of a node of an n-ary tree
struct Node {
    int key;
    vector<Node*> child;
};

// Utility function to create a new tree
// node with k children
Node* newNode(int value)
{
    Node* nNode = new Node;
    nNode->key = value;
    return nNode;
}

// Function to build full k-ary tree
Node* BuildKaryTree(int A[], int n, int k, int h, int& ind)
{
    // For null tree
    if (n <= 0)
        return NULL;

    Node* nNode = newNode(A[ind]);
    if (nNode == NULL) {
        cout << "Memory error" << endl;
        return NULL;
    }

    // For adding k children to a node
    for (int i = 0; i < k; i++) {
```

```
// Check if ind is in range of array
// Check if height of the tree is greater than 1
if (ind < n - 1 && h > 1) {
    ind++;

    // Recursively add each child
    nNode->child.push_back(BuildKaryTree(A, n, k, h - 1, ind));
} else {
    nNode->child.push_back(NULL);
}
}
return nNode;
}

// Function to find the height of the tree
Node* BuildKaryTree(int* A, int n, int k, int ind)
{
    int height = (int)ceil(log((double)n * (k - 1) + 1)
                           / log((double)k));
    return BuildKaryTree(A, n, k, height, ind);
}

// Function to print postorder traversal of the tree
void postord(Node* root, int k)
{
    if (root == NULL)
        return;
    for (int i = 0; i < k; i++)
        postord(root->child[i], k);
    cout << root->key << " ";
}

// Driver program to implement full k-ary tree
int main()
{
    int ind = 0;
    int k = 3, n = 10;
    int preorder[] = { 1, 2, 5, 6, 7, 3, 8, 9, 10, 4 };
    Node* root = BuildKaryTree(preorder, n, k, ind);
    cout << "Postorder traversal of constructed"
         " full k-ary tree is: ";
    postord(root, k);
    cout << endl;
    return 0;
}
```

Output:

Postorder traversal of constructed full k-ary tree is: 5 6 7 2 8 9 10 3 4 1

## Source

<https://www.geeksforgeeks.org/construct-full-k-ary-tree-preorder-traversal/>

## Chapter 86

# Construct tree from ancestor matrix

Construct tree from ancestor matrix - GeeksforGeeks

Given an ancestor matrix mat[n][n] where Ancestor matrix is defined as below.

```
mat[i][j] = 1 if i is ancestor of j  
mat[i][j] = 0, otherwise
```

Construct a Binary Tree from given ancestor matrix where all its values of nodes are from 0 to n-1.

1. It may be assumed that the input provided the program is valid and tree can be constructed out of it.
2. Many Binary trees can be constructed from one input. The program will construct any one of them.

Examples:

Input: 0 1 1  
0 0 0  
0 0 0  
Output: Root of one of the below trees.  
0  
/ \ OR / \  
1 2 2 1

Input: 0 0 0 0 0 0

```

1 0 0 0 1 0
0 0 0 1 0 0
0 0 0 0 0 0
0 0 0 0 0 0
1 1 1 1 1 0

Output: Root of one of the below trees.

      5           5           5
     / \         / \         / \
    1   2       OR   2   1     OR   1   2   OR ...
   / \   /       /   / \     / \   /
  0   4   3       3   0   4     4   0   3

There are different possible outputs because ancestor
matrix doesn't store that which child is left and which
is right.

```

This problem is mainly reverse of below problem.

[Construct Ancestor Matrix from a Given Binary Tree](#)

We strongly recommend you to minimize your browser and try this yourself first.

Observations used in the solution:

1. The rows that correspond to leaves have all 0's
2. The row that corresponds to root has maximum number of 1's.
3. Count of 1's in i'th row indicates number of descendants of node i.

The idea is to construct the tree in **bottom up manner**.

- 1) Create an array of node pointers node[].
- 2) Store row numbers that correspond to a given count. We have used [multimap](#) for this purpose.
- 3) Process all entries of multimap from smallest count to largest (Note that entries in map and multimap can be traversed in sorted order). Do following for every entry.
  - .....a) Create a new node for current row number.
  - .....b) If this node is not a leaf node, consider all those descendants of it whose parent is not set, make current node as its parent.
- 4) The last processed node (node with maximum sum) is root of tree.

Below is C++ implementation of above idea. Following are steps.

```

// Given an ancestor matrix for binary tree, construct
// the tree.
#include <bits/stdc++.h>
using namespace std;

#define N 6

```

```
/* A binary tree node */
struct Node
{
    int data;
    Node *left, *right;
};

/* Helper function to create a new node */
Node* newNode(int data)
{
    Node* node = new Node;
    node->data = data;
    node->left = node->right = NULL;
    return (node);
}

// Constructs tree from ancestor matrix
Node* ancestorTree(int mat[][] [N])
{
    // Binary array to determine weather
    // parent is set for node i or not
    int parent[N] = {0};

    // Root will store the root of the constructed tree
    Node* root = NULL;

    // Create a multimap, sum is used as key and row
    // numbers are used as values
    multimap<int, int> mm;

    for (int i = 0; i < N; i++)
    {
        int sum = 0; // Initialize sum of this row
        for (int j = 0; j < N; j++)
            sum += mat[i][j];

        // insert(sum, i) pairs into the multimap
        mm.insert(pair<int, int>(sum, i));
    }

    // node[i] will store node for i in constructed tree
    Node* node[N];

    // Traverse all entries of multimap. Note that values
    // are accessed in increasing order of sum
    for (auto it = mm.begin(); it != mm.end(); ++it)
    {
        // create a new node for every value
```

```

node[it->second] = newNode(it->second);

// To store last processed node. This node will be
// root after loop terminates
root = node[it->second];

// if non-leaf node
if (it->first != 0)
{
    // traverse row 'it->second' in the matrix
    for (int i = 0; i < N; i++)
    {
        // if parent is not set and ancestor exists
        if (!parent[i] && mat[it->second][i])
        {
            // check for unoccupied left/right node
            // and set parent of node i
            if (!node[it->second]->left)
                node[it->second]->left = node[i];
            else
                node[it->second]->right = node[i];

            parent[i] = 1;
        }
    }
}
return root;
}

/* Given a binary tree, print its nodes in inorder */
void printInorder(Node* node)
{
    if (node == NULL)
        return;
    printInorder(node->left);
    printf("%d ", node->data);
    printInorder(node->right);
}

// Driver program
int main()
{
    int mat[N][N] = {{ 0, 0, 0, 0, 0, 0, 0 },
                     { 1, 0, 0, 0, 1, 0 },
                     { 0, 0, 0, 1, 0, 0 },
                     { 0, 0, 0, 0, 0, 0 },
                     { 0, 0, 0, 0, 0, 0 },
                     { 0, 0, 0, 0, 0, 0 },
                     { 0, 0, 0, 0, 0, 0 }};
}

```

```
{ 1, 1, 1, 1, 1, 0 }  
};  
  
Node* root = ancestorTree(mat);  
  
cout << "Inorder traversal of tree is \n";  
printInorder(root);  
  
return 0;  
}
```

Output:

```
Inorder traversal of tree is  
0 1 4 5 3 2
```

Note that we can also use an array of vectors in place of multimap. We have used multimap for simplicity. Array of vectors would improve performance as inserting and accessing elements would take O(1) time.

This article is contributed by **Aditya Goel**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## Source

<https://www.geeksforgeeks.org/construct-tree-from-ancestor-matrix/>

# Chapter 87

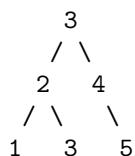
## Continuous Tree

Continuous Tree - GeeksforGeeks

A tree is Continuous tree if in each root to leaf path, absolute difference between keys of two adjacent is 1. We are given a binary tree, we need to check if tree is continuous or not.

Examples:

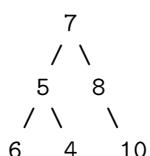
Input :



Output: "Yes"

```
// 3->2->1 every two adjacent node's absolute difference is 1
// 3->2->3 every two adjacent node's absolute difference is 1
// 3->4->5 every two adjacent node's absolute difference is 1
```

Input :



Output: "No"

```
// 7->5->6 here absolute difference of 7 and 5 is not 1.
// 7->5->4 here absolute difference of 7 and 5 is not 1.
// 7->8->10 here absolute difference of 8 and 10 is not 1.
```

The solution requires a traversal of tree. The important things to check are to make sure that all corner cases are handled. The corner cases include, empty tree, single node tree, a node with only left child and a node with only right child.

In tree traversal, we recursively check if left and right subtree are continuous. We also check if difference between keys of current node's key and its children keys is 1. Below is C++ implementation of the idea.

```
// C++ program to check if a tree is continuous or not
#include<bits/stdc++.h>
using namespace std;

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct Node
{
    int data;
    struct Node* left, * right;
};

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct Node* newNode(int data)
{
    struct Node* node = new Node;
    node->data = data;
    node->left = node->right = NULL;
    return(node);
}

// Function to check tree is continuous or not
bool treeContinuous(struct Node *ptr)
{
    // if next node is empty then return true
    if (ptr == NULL)
        return true;

    // if current node is leaf node then return true
    // because it is end of root to leaf path
    if (ptr->left == NULL && ptr->right == NULL)
        return true;

    // If left subtree is empty, then only check right
    if (ptr->left == NULL)
        return (abs(ptr->data - ptr->right->data) == 1) &&
            treeContinuous(ptr->right);

    // If right subtree is empty, then only check left
    if (ptr->right == NULL)
        return (abs(ptr->data - ptr->left->data) == 1) &&
            treeContinuous(ptr->left);
```

```
// If both left and right subtrees are not empty, check
// everything
return abs(ptr->data - ptr->left->data)==1 &&
       abs(ptr->data - ptr->right->data)==1 &&
       treeContinuous(ptr->left) &&
       treeContinuous(ptr->right);
}

/* Driver program to test mirror() */
int main()
{
    struct Node *root = newNode(3);
    root->left      = newNode(2);
    root->right     = newNode(4);
    root->left->left = newNode(1);
    root->left->right = newNode(3);
    root->right->right = newNode(5);
    treeContinuous(root)? cout << "Yes" : cout << "No";
    return 0;
}
```

Output:

Yes

## Source

<https://www.geeksforgeeks.org/continuous-tree/>

## Chapter 88

# Convert Ternary Expression to a Binary Tree

Convert Ternary Expression to a Binary Tree - GeeksforGeeks

Given a string that contains ternary expressions. The expressions may be nested, task is convert the given ternary expression to a binary Tree.

**Examples:**

Input : string expression = a?b:c  
Output :        a  
              / \  
            b   c

Input : expression = a?b?c:d:e  
Output :        a  
              / \  
            b   e  
          / \  
         c   d

Asked In : **Facebook Interview**

Idea is that we traverse a string make first character as root and do following step recursively

- 1. If we see Symbol ‘?’  
..... then we add next character as the left child of root.
- 2. If we see Symbol ‘:’  
..... then we add it as the right child of current root.  
do this process until we traverse all element of “String”.

Below c++ implementation of above idea

C++

```
// C++ program to convert a ternary expression to
// a tree.
#include<bits/stdc++.h>
using namespace std;

// tree structure
struct Node
{
    char data;
    Node *left, *right;
};

// function create a new node
Node *newNode(char Data)
{
    Node *new_node = new Node;
    new_node->data = Data;
    new_node->left = new_node->right = NULL;
    return new_node;
}

// Function to convert Ternary Expression to a Binary
// Tree. It return the root of tree
Node *convertExpression(string expression, int i)
{

    // Base case
    if (i >= expression.size())
        return NULL;

    // store current character of expression_string
    // [ 'a' to 'z']
    Node *root = newNode(expression[i]);

    // Move ahead in str
    ++i;

    // if current character of ternary expression is '?'
    // then we add next character as a left child of
    // current node
    if (i < expression.size() && expression.at(i)=='?')
        root->left = convertExpression(expression, i+1);

    // else we have to add it as a right child of
    // current node expression.at(0) == ':'
    else if (i < expression.size())
        root->right = convertExpression(expression, i+1);
}
```

```
    root->right = convertExpression(expression, i+1);

    return root;
}

// function print tree
void printTree( Node *root)
{
    if (!root)
        return ;
    cout << root->data << " ";
    printTree(root->left);
    printTree(root->right);
}

// Driver program to test above function
int main()
{
    string expression = "a?b?c:d:e";
    Node *root = convertExpression(expression, 0);
    printTree(root) ;
    return 0;
}
```

**Java**

```
// Java program to convert a ternary
// expression to a tree.
import java.util.Queue;
import java.util.LinkedList;

// Class to represent Tree node
class Node
{
    char data;
    Node left, right;

    public Node(char item)
    {
        data = item;
        left = null;
        right = null;
    }
}

// Class to convert a ternary expression to a Tree
class BinaryTree
{
```

```
// Function to convert Ternary Expression to a Binary
// Tree. It return the root of tree
Node convertExpression(char[] expression, int i)
{
    // Base case
    if (i >= expression.length)
        return null;

    // store current character of expression_string
    // [ 'a' to 'z']
    Node root = new Node(expression[i]);

    // Move ahead in str
    ++i;

    // if current character of ternary expression is '?'
    // then we add next character as a left child of
    // current node
    if (i < expression.length && expression[i]=='?')
        root.left = convertExpression(expression, i+1);

    // else we have to add it as a right child of
    // current node expression.at(0) == ':'
    else if (i < expression.length)
        root.right = convertExpression(expression, i+1);

    return root;
}

// function print tree
public void printTree( Node root)
{
    if (root == null)
        return;

    System.out.print(root.data + " ");
    printTree(root.left);
    printTree(root.right);
}

// Driver program to test above function
public static void main(String args[])
{
    String exp = "a?b?c:d:e";
    BinaryTree tree = new BinaryTree();
    char[] expression=exp.toCharArray();
    Node root = tree.convertExpression(expression, 0);
    tree.printTree(root) ;
```

```
}
```

```
/* This code is contributed by Mr. Somesh Awasthi */
```

### Python3

```
# Class to define a node
# structure of the tree
class Node:
    def __init__(self, key):
        self.data = key
        self.left = None
        self.right = None

# Function to convert ternary
# expression to a Binary tree
# It returns the root node
# of the tree
def convert_expression(expression, i):
    if i >= len(expression):
        return None

    # Create a new node object
    # for the expression at
    # ith index
    root = Node(expression[i])

    i += 1

    # if current character of
    # ternary expression is '?'
    # then we add next character
    # as a left child of
    # current node
    if (i < len(expression) and
        expression[i] is "?"):
        root.left = convert_expression(expression, i + 1)

    # else we have to add it
    # as a right child of
    # current node expression[0] == ':'
    elif i < len(expression):
        root.right = convert_expression(expression, i + 1)
    return root

# Function to print the tree
# in a pre-order traversal pattern
```

```
def print_tree(root):
    if not root:
        return
    print(root.data, end=' ')
    print_tree(root.left)
    print_tree(root.right)

# Driver Code
if __name__ == "__main__":
    string_expression = "a?b:c:d:e"
    root_node = convert_expression(string_expression, 0)
    print_tree(root_node)

# This code is contributed
# by Kanav Malhotra
```

**Output :**

a b c d e

**Time Complexity :** O(n) [ here n is length of String ]

**Improved By :** [kanavMalhotra](#)

## Source

<https://www.geeksforgeeks.org/convert-ternary-expression-binary-tree/>

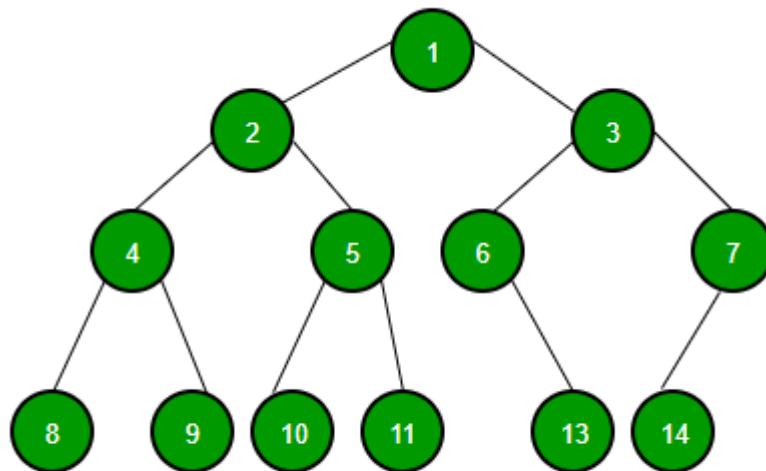
## Chapter 89

# Convert a Binary Tree into Doubly Linked List in spiral fashion

Convert a Binary Tree into Doubly Linked List in spiral fashion - GeeksforGeeks

Given a Binary Tree, convert it into Doubly Linked List where the nodes are represented Spirally. The left pointer of the binary tree node should act as a previous node for created DLL and right pointer should act as next node.

The solution should not allocate extra memory for DLL nodes. It should use binary tree nodes for creating DLL i.e. only change of pointers is allowed



For example, for the tree on left side, Doubly Linked List can be,

**1 2 3 7 6 5 4 8 9 10 11 13 14 or**

**1 3 2 4 5 6 7 14 13 11 10 9 8.**

**We strongly recommend you to minimize your browser and try this yourself first.**

We can do this by doing a spiral order traversal in  $O(n)$  time and  $O(n)$  extra space. The idea is to use deque (Double-ended queue) that can be expanded or contracted on both ends (either its front or its back). We do something similar to level order traversal but to maintain spiral order, for every odd level, we dequeue node from the front and inserts its left and right children in the back of the deque data structure. And for each even level, we dequeue node from the back and inserts its right and left children in the front of deque. We also maintain a stack to store Binary Tree nodes. Whenever we pop nodes from deque, we push that node into stack. Later, we pop all nodes from stack and push the nodes in the beginning of the list. We can avoid use of stack if we maintain a tail pointer that always points to last node of DLL and inserts nodes in  $O(1)$  time in the end.

Below is C++ implementation of above idea

C++

```
/* c++ program to convert Binary Tree into Doubly
   Linked List where the nodes are represented
   spirally. */
#include <bits/stdc++.h>
using namespace std;

// A Binary Tree Node
struct Node
{
    int data;
    struct Node *left, *right;
};

/* Given a reference to the head of a list and a node,
   inserts the node on the front of the list. */
void push(Node** head_ref, Node* node)
{
    // Make right of given node as head and left as
    // NULL
    node->right = (*head_ref);
    node->left = NULL;

    // change left of head node to given node
    if ((*head_ref) != NULL)
        (*head_ref)->left = node;
    (*head_ref) = node;
}
```

```
(*head_ref)->left = node ;  
  
// move the head to point to the given node  
(*head_ref) = node;  
}  
  
// Function to prints contents of DLL  
void printList(Node *node)  
{  
    while (node != NULL)  
    {  
        cout << node->data << " ";  
        node = node->right;  
    }  
}  
  
/* Function to print corner node at each level */  
void spiralLevelOrder(Node *root)  
{  
    // Base Case  
    if (root == NULL)  
        return;  
  
    // Create an empty deque for doing spiral  
    // level order traversal and enqueue root  
    deque<Node*> q;  
    q.push_front(root);  
  
    // create a stack to store Binary Tree nodes  
    // to insert into DLL later  
    stack<Node*> stk;  
  
    int level = 0;  
    while (!q.empty())  
    {  
        // nodeCount indicates number of Nodes  
        // at current level.  
        int nodeCount = q.size();  
  
        // Dequeue all Nodes of current level and  
        // Enqueue all Nodes of next level  
        if (level&1)      //odd level  
        {  
            while (nodeCount > 0)  
            {  
                // dequeue node from front & push it to  
                // stack  
                Node *node = q.front();  
                q.pop_front();  
                stk.push(node);  
                nodeCount--;  
            }  
        }  
        else // even level  
        {  
            while (nodeCount > 0)  
            {  
                // enqueue node from back to front  
                // stack  
                Node *node = q.back();  
                q.pop_back();  
                stk.push(node);  
                nodeCount--;  
            }  
        }  
        level++;  
    }  
    // Insert all nodes of stack into DLL  
    while (!stk.empty())  
    {  
        Node *node = stk.top();  
        stk.pop();  
        (*head_ref)->left = node ;  
        (*head_ref) = node;  
    }  
}
```

```
q.pop_front();
stk.push(node);

// insert its left and right children
// in the back of the deque
if (node->left != NULL)
    q.push_back(node->left);
if (node->right != NULL)
    q.push_back(node->right);

nodeCount--;
}

}

else      //even level
{
    while (nodeCount > 0)
    {
        // dequeue node from the back & push it
        // to stack
        Node *node = q.back();
        q.pop_back();
        stk.push(node);

        // inserts its right and left children
        // in the front of the deque
        if (node->right != NULL)
            q.push_front(node->right);
        if (node->left != NULL)
            q.push_front(node->left);
        nodeCount--;
    }
}

level++;
}

// head pointer for DLL
Node* head = NULL;

// pop all nodes from stack and
// push them in the beginning of the list
while (!stk.empty())
{
    push(&head, stk.top());
    stk.pop();
}

cout << "Created DLL is:\n";
printList(head);
```

```
}

// Utility function to create a new tree Node
Node* newNode(int data)
{
    Node *temp = new Node;
    temp->data = data;
    temp->left = temp->right = NULL;

    return temp;
}

// Driver program to test above functions
int main()
{
    // Let us create binary tree shown in above diagram
    Node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->left = newNode(6);
    root->right->right = newNode(7);

    root->left->left->left = newNode(8);
    root->left->left->right = newNode(9);
    root->left->right->left = newNode(10);
    root->left->right->right = newNode(11);
    //root->right->left->left = newNode(12);
    root->right->left->right = newNode(13);
    root->right->right->left = newNode(14);
    //root->right->right->right = newNode(15);

    spiralLevelOrder(root);

    return 0;
}
```

### Java

```
/* Java program to convert Binary Tree into Doubly Linked List
   where the nodes are represented spirally */

import java.util.*;

// A binary tree node
class Node
{
```

```
int data;
Node left, right;

public Node(int data)
{
    this.data = data;
    left = right = null;
}
}

class BinaryTree
{
    Node root;
    Node head;

    /* Given a reference to a node,
       inserts the node on the front of the list. */
    void push(Node node)
    {
        // Make right of given node as head and left as
        // NULL
        node.right = head;
        node.left = null;

        // change left of head node to given node
        if (head != null)
            head.left = node;

        // move the head to point to the given node
        head = node;
    }

    // Function to prints contents of DLL
    void printList(Node node)
    {
        while (node != null)
        {
            System.out.print(node.data + " ");
            node = node.right;
        }
    }

    /* Function to print corner node at each level */
    void spiralLevelOrder(Node root)
    {
        // Base Case
        if (root == null)
            return;
```

```
// Create an empty deque for doing spiral
// level order traversal and enqueue root
Deque<Node> q = new LinkedList<Node>();
q.addFirst(root);

// create a stack to store Binary Tree nodes
// to insert into DLL later
Stack<Node> stk = new Stack<Node>();

int level = 0;
while (!q.isEmpty())
{
    // nodeCount indicates number of Nodes
    // at current level.
    int nodeCount = q.size();

    // Dequeue all Nodes of current level and
    // Enqueue all Nodes of next level
    if ((level & 1) %2 != 0) //odd level
    {
        while (nodeCount > 0)
        {
            // dequeue node from front & push it to
            // stack
            Node node = q.peekFirst();
            q.pollFirst();
            stk.push(node);

            // insert its left and right children
            // in the back of the deque
            if (node.left != null)
                q.addLast(node.left);
            if (node.right != null)
                q.addLast(node.right);

            nodeCount--;
        }
    }
    else //even level
    {
        while (nodeCount > 0)
        {
            // dequeue node from the back & push it
            // to stack
            Node node = q.peekLast();
            q.pollLast();
            stk.push(node);
```

```
// inserts its right and left children
// in the front of the deque
if (node.right != null)
    q.addFirst(node.right);
if (node.left != null)
    q.addFirst(node.left);
nodeCount--;
}
}
level++;
}

// pop all nodes from stack and
// push them in the beginning of the list
while (!stk.empty())
{
    push(stk.peek());
    stk.pop();
}

System.out.println("Created DLL is : ");
printList(head);
}

// Driver program to test above functions
public static void main(String[] args)
{
    // Let us create binary tree as shown in above diagram
    BinaryTree tree = new BinaryTree();
    tree.root = new Node(1);
    tree.root.left = new Node(2);
    tree.root.right = new Node(3);
    tree.root.left.left = new Node(4);
    tree.root.left.right = new Node(5);
    tree.root.right.left = new Node(6);
    tree.root.right.right = new Node(7);

    tree.root.left.left.left = new Node(8);
    tree.root.left.left.right = new Node(9);
    tree.root.left.right.left = new Node(10);
    tree.root.left.right.right = new Node(11);
    // tree.root.right.left.left = new Node(12);
    tree.root.right.left.right = new Node(13);
    tree.root.right.right.left = new Node(14);
    // tree.root.right.right.right = new Node(15);

    tree.spiralLevelOrder(tree.root);
```

```
    }  
}  
  
// This code has been contributed by Mayank Jaiswal(mayank_24)
```

Output :

```
Created DLL is:  
1 2 3 7 6 5 4 8 9 10 11 13 14
```

### Source

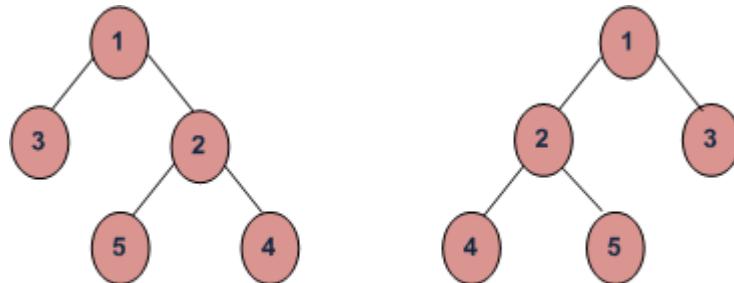
<https://www.geeksforgeeks.org/convert-a-binary-tree-into-doubly-linked-list-in-spiral-fashion/>

## Chapter 90

# Convert a Binary Tree into its Mirror Tree

Convert a Binary Tree into its Mirror Tree - GeeksforGeeks

Mirror of a Tree: Mirror of a Binary Tree T is another Binary Tree M(T) with left and right children of all non-leaf nodes interchanged.



**Mirror Trees**

Trees in the above figure are mirror of each other

Method 1 (Recursive)

Algorithm – Mirror(tree):

- (1) Call Mirror for left-subtree i.e., Mirror(left-subtree)
- (2) Call Mirror for right-subtree i.e., Mirror(right-subtree)
- (3) Swap left and right subtrees.

```
temp = left-subtree
left-subtree = right-subtree
right-subtree = temp
```

C

```
// C program to convert a binary tree
// to its mirror
#include<stdio.h>
#include<stdlib.h>

/* A binary tree node has data, pointer
   to left child and a pointer to right child */
struct Node
{
    int data;
    struct Node* left;
    struct Node* right;
};

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct Node* newNode(int data)

{
    struct Node* node = (struct Node*)
                           malloc(sizeof(struct Node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;

    return(node);
}

/* Change a tree so that the roles of the left and
   right pointers are swapped at every node.

So the tree...
      4
     / \
    2   5
   / \
  1   3

is changed to...
      4
     / \
    5   2
   / \
  3   1
*/
```

```
void mirror(struct Node* node)
{
    if (node==NULL)
        return;
    else
    {
        struct Node* temp;

        /* do the subtrees */
        mirror(node->left);
        mirror(node->right);

        /* swap the pointers in this node */
        temp      = node->left;
        node->left  = node->right;
        node->right = temp;
    }
}

/* Helper function to print Inorder traversal.*/
void inOrder(struct Node* node)
{
    if (node == NULL)
        return;

    inOrder(node->left);
    printf("%d ", node->data);
    inOrder(node->right);
}

/* Driver program to test mirror() */
int main()
{
    struct Node *root = newNode(1);
    root->left      = newNode(2);
    root->right     = newNode(3);
    root->left->left  = newNode(4);
    root->left->right = newNode(5);

    /* Print inorder traversal of the input tree */
    printf("Inorder traversal of the constructed"
          " tree is \n");
    inOrder(root);

    /* Convert tree to its mirror */
    mirror(root);
}
```

```
/* Print inorder traversal of the mirror tree */
printf("\nInorder traversal of the mirror tree"
      " is \n");
inOrder(root);

return 0;
}
```

**Java**

```
// Java program to convert binary tree into its mirror

/* Class containing left and right child of current
   node and key value*/
class Node
{
    int data;
    Node left, right;

    public Node(int item)
    {
        data = item;
        left = right = null;
    }
}

class BinaryTree
{
    Node root;

    void mirror()
    {
        root = mirror(root);
    }

    Node mirror(Node node)
    {
        if (node == null)
            return node;

        /* do the subtrees */
        Node left = mirror(node.left);
        Node right = mirror(node.right);

        /* swap the left and right pointers */
        node.left = right;
        node.right = left;
    }
}
```

```
        return node;
    }

    void inOrder()
    {
        inOrder(root);
    }

    /* Helper function to test mirror(). Given a binary
       search tree, print out its data elements in
       increasing sorted order.*/
    void inOrder(Node node)
    {
        if (node == null)
            return;

        inOrder(node.left);
        System.out.print(node.data + " ");

        inOrder(node.right);
    }

    /* testing for example nodes */
    public static void main(String args[])
    {
        /* creating a binary tree and entering the nodes */
        BinaryTree tree = new BinaryTree();
        tree.root = new Node(1);
        tree.root.left = new Node(2);
        tree.root.right = new Node(3);
        tree.root.left.left = new Node(4);
        tree.root.left.right = new Node(5);

        /* print inorder traversal of the input tree */
        System.out.println("Inorder traversal of input tree is :");
        tree.inOrder();
        System.out.println("");

        /* convert tree to its mirror */
        tree.mirror();

        /* print inorder traversal of the minor tree */
        System.out.println("Inorder traversal of binary tree is : ");
        tree.inOrder();
    }
}
```

Output :

```
Inorder traversal of the constructed tree is
4 2 5 1 3
Inorder traversal of the mirror tree is
3 1 5 2 4
```

**Time & Space Complexities:** This program is similar to traversal of tree space and time complexities will be same as Tree traversal (Please see our [Tree Traversal](#) post for details)

Method 2 (Iterative)

The idea is to do queue based level order traversal. While doing traversal, swap left and right children of every node.

```
// Iterative CPP program to convert a Binary
// Tree to its mirror
#include<bits/stdc++.h>
using namespace std;

/* A binary tree node has data, pointer to
   left child and a pointer to right child */
struct Node
{
    int data;
    struct Node* left;
    struct Node* right;
};

/* Helper function that allocates a new node
   with the given data and NULL left and right
   pointers. */
struct Node* newNode(int data)

{
    struct Node* node = new Node;
    node->data = data;
    node->left = node->right = NULL;
    return(node);
}

/* Change a tree so that the roles of the left and
   right pointers are swapped at every node.
   So the tree...
        4
```

```

      / \
    2   5
   / \
  1   3

is changed to...
      4
     / \
    5   2
   / \
  3   1
*/
void mirror(Node* root)
{
    if (root == NULL)
        return;

    queue<Node*> q;
    q.push(root);

    // Do BFS. While doing BFS, keep swapping
    // left and right children
    while (!q.empty())
    {
        // pop top node from queue
        Node* curr = q.front();
        q.pop();

        // swap left child with right child
        swap(curr->left, curr->right);

        // push left and right children
        if (curr->left)
            q.push(curr->left);
        if (curr->right)
            q.push(curr->right);
    }
}

/* Helper function to print Inorder traversal.*/
void inOrder(struct Node* node)
{
    if (node == NULL)
        return;
    inOrder(node->left);
    cout << node->data << " ";
    inOrder(node->right);
}

```

```
}

/* Driver program to test mirror() */
int main()
{
    struct Node *root = newNode(1);
    root->left      = newNode(2);
    root->right     = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);

    /* Print inorder traversal of the input tree */
    cout << "\n Inorder traversal of the"
        " constructed tree is \n";
    inOrder(root);

    /* Convert tree to its mirror */
    mirror(root);

    /* Print inorder traversal of the mirror tree */
    cout << "\n Inorder traversal of the "
        "mirror tree is \n";
    inOrder(root);

    return 0;
}
```

## Source

<https://www.geeksforgeeks.org/write-an-efficient-c-function-to-convert-a-tree-into-its-mirror-tree/>

## Chapter 91

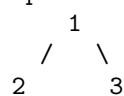
# Convert a Binary Tree such that every node stores the sum of all nodes in its right subtree

Convert a Binary Tree such that every node stores the sum of all nodes in its right subtree  
- GeeksforGeeks

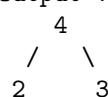
Given a binary tree, change the value in each node to sum of all the values in the nodes in the right subtree including its own.

**Examples:**

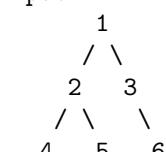
**Input :**



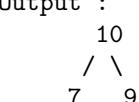
**Output :**



**Input :**



**Output :**



```
    / \   \
   4   5   6
```

**Approach :** The idea is to traverse the given binary tree in **bottom up** manner. Recursively compute the sum of nodes in right and left subtrees. Accumulate sum of nodes in the right subtree to the current node and return sum of nodes under current subtree.

Below is the implementation of above approach.

```
// C++ program to store sum of nodes in
// right subtree in every node
#include <bits/stdc++.h>
using namespace std;

// Node of tree
struct Node {
    int data;
    Node *left, *right;
};

// Function to create a new node
struct Node* createNode(int item)
{
    Node* temp = new Node;
    temp->data = item;
    temp->left = NULL;
    temp->right = NULL;

    return temp;
}

// Function to build new tree with
// all nodes having the sum of all
// nodes in its right subtree
int updateBTree(Node* root)
{
    // Base cases
    if (!root)
        return 0;
    if (root->left == NULL && root->right == NULL)
        return root->data;

    // Update right and left subtrees
    int rightsum = updateBTree(root->right);
    int leftsum = updateBTree(root->left);

    // Add rightsum to current node
    root->data += rightsum;
```

```
// Return sum of values under root
    return root->data + leftsum;
}

// Function to traverse tree in inorder way
void inorder(struct Node* node)
{
    if (node == NULL)
        return;
    inorder(node->left);
    cout << node->data << " ";
    inorder(node->right);
}

// Driver code
int main()
{
    /* Let us construct a binary tree
       1
      / \
     2   3
    / \   \
   4   5   6      */
    struct Node* root = NULL;
    root = createNode(1);
    root->left = createNode(2);
    root->right = createNode(3);
    root->left->left = createNode(4);
    root->left->right = createNode(5);
    root->right->right = createNode(6);

    // new tree construction
    updateBTree(root);

    cout << "Inorder traversal of the modified tree is \n";
    inorder(root);

    return 0;
}
```

**Output:**

```
Inorder traversal of the modified tree is
4 7 5 10 9 6
```

**Time Complexity:** O(n)

*Chapter 91. Convert a Binary Tree such that every node stores the sum of all nodes in its right subtree*

---

## Source

<https://www.geeksforgeeks.org/convert-a-binary-tree-such-that-every-node-stores-the-sum-of-all-nodes-in-its-right-subtree/>

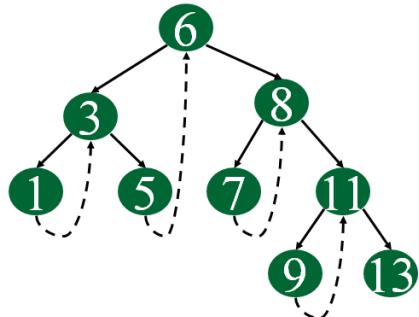
## Chapter 92

# Convert a Binary Tree to Threaded binary tree | Set 1 (Using Queue)

Convert a Binary Tree to Threaded binary tree | Set 1 (Using Queue) - GeeksforGeeks

We have discussed [Threaded Binary Tree](#). The idea of threaded binary trees is to make inorder traversal faster and do it without stack and without recursion. In a simple threaded binary tree, the NULL right pointers are used to store inorder successor. Where-ever a right pointer is NULL, it is used to store inorder successor.

Following diagram shows an example Single Threaded Binary Tree. The dotted lines represent threads.



Following is structure of single threaded binary tree.

```
struct Node
{
    int key;
    Node *left, *right;

    // Used to indicate whether the right pointer is a normal right
```

```
// pointer or a pointer to inorder successor.  
bool isThreaded;  
};
```

### **How to convert a Given Binary Tree to Threaded Binary Tree?**

We basically need to set NULL right pointers to inorder successor. We first do an inorder traversal of the tree and store it in a queue (we can use a simple array also) so that the inorder successor becomes the next node. We again do an inorder traversal and whenever we find a node whose right is NULL, we take the front item from queue and make it the right of current node. We also set isThreaded to true to indicate that the right pointer is a threaded link.

Following is the implementation of the above idea.

C++

```
/* C++ program to convert a Binary Tree to Threaded Tree */  
#include <iostream>  
#include <queue>  
using namespace std;  
  
/* Structure of a node in threaded binary tree */  
struct Node  
{  
    int key;  
    Node *left, *right;  
  
    // Used to indicate whether the right pointer is a normal  
    // right pointer or a pointer to inorder successor.  
    bool isThreaded;  
};  
  
// Helper function to put the Nodes in inorder into queue  
void populateQueue(Node *root, std::queue <Node *> *q)  
{  
    if (root == NULL) return;  
    if (root->left)  
        populateQueue(root->left, q);  
    q->push(root);  
    if (root->right)  
        populateQueue(root->right, q);  
}  
  
// Function to traverse queue, and make tree threaded  
void createThreadedUtil(Node *root, std::queue <Node *> *q)  
{  
    if (root == NULL) return;  
  
    if (root->left)
```

```
        createThreadedUtil(root->left, q);
        q->pop();

        if (root->right)
            createThreadedUtil(root->right, q);

        // If right pointer is NULL, link it to the
        // inorder successor and set 'isThreaded' bit.
        else
        {
            root->right = q->front();
            root->isThreaded = true;
        }
    }

    // This function uses populateQueue() and
    // createThreadedUtil() to convert a given binary tree
    // to threaded tree.
    void createThreaded(Node *root)
    {
        // Create a queue to store inorder traversal
        std::queue <Node *> q;

        // Store inorder traversal in queue
        populateQueue(root, &q);

        // Link NULL right pointers to inorder successor
        createThreadedUtil(root, &q);
    }

    // A utility function to find leftmost node in a binary
    // tree rooted with 'root'. This function is used in inOrder()
    Node *leftMost(Node *root)
    {
        while (root != NULL && root->left != NULL)
            root = root->left;
        return root;
    }

    // Function to do inorder traversal of a threaded binary tree
    void inOrder(Node *root)
    {
        if (root == NULL) return;

        // Find the leftmost node in Binary Tree
        Node *cur = leftMost(root);

        while (cur != NULL)
```

```
{  
    cout << cur->key << " ";  
  
    // If this Node is a thread Node, then go to  
    // inorder successor  
    if (cur->isThreaded)  
        cur = cur->right;  
  
    else // Else go to the leftmost child in right subtree  
        cur = leftMost(cur->right);  
}  
}  
  
// A utility function to create a new node  
Node *newNode(int key)  
{  
    Node *temp = new Node;  
    temp->left = temp->right = NULL;  
    temp->key = key;  
    return temp;  
}  
  
// Driver program to test above functions  
int main()  
{  
    /*      1  
         / \  
        2   3  
       / \ / \  
      4  5 6  7 */  
    Node *root = newNode(1);  
    root->left = newNode(2);  
    root->right = newNode(3);  
    root->left->left = newNode(4);  
    root->left->right = newNode(5);  
    root->right->left = newNode(6);  
    root->right->right = newNode(7);  
  
    createThreaded(root);  
  
    cout << "Inorder traversal of created threaded tree is\n";  
    inOrder(root);  
    return 0;  
}
```

### Java

```
// Java program to convert binary tree to threaded tree
```

```
import java.util.LinkedList;
import java.util.Queue;

/* Class containing left and right child of current
   node and key value*/
class Node
{
    int data;
    Node left, right;

    // Used to indicate whether the right pointer is a normal
    // right pointer or a pointer to inorder successor.
    boolean isThreaded;

    public Node(int item)
    {
        data = item;
        left = right = null;
    }
}

class BinaryTree
{
    Node root;

    // Helper function to put the Nodes in inorder into queue
    void populateQueue(Node node, Queue<Node> q)
    {
        if (node == null)
            return;
        if (node.left != null)
            populateQueue(node.left, q);
        q.add(node);
        if (node.right != null)
            populateQueue(node.right, q);
    }

    // Function to traverse queue, and make tree threaded
    void createThreadedUtil(Node node, Queue<Node> q)
    {
        if (node == null)
            return;

        if (node.left != null)
            createThreadedUtil(node.left, q);
        q.remove();

        if (node.right != null)
```

```
        createThreadedUtil(node.right, q);

        // If right pointer is NULL, link it to the
        // inorder successor and set 'isThreaded' bit.
        else
        {
            node.right = q.peek();
            node.isThreaded = true;
        }
    }

    // This function uses populateQueue() and
    // createThreadedUtil() to convert a given binary tree
    // to threaded tree.
    void createThreaded(Node node)
    {
        // Create a queue to store inorder traversal
        Queue<Node> q = new LinkedList<Node>();

        // Store inorder traversal in queue
        populateQueue(node, q);

        // Link NULL right pointers to inorder successor
        createThreadedUtil(node, q);
    }

    // A utility function to find leftmost node in a binary
    // tree rooted with 'root'. This function is used in inOrder()
    Node leftMost(Node node)
    {
        while (node != null && node.left != null)
            node = node.left;
        return node;
    }

    // Function to do inorder traversal of a threadded binary tree
    void inOrder(Node node)
    {
        if (node == null)
            return;

        // Find the leftmost node in Binary Tree
        Node cur = leftMost(node);

        while (cur != null)
        {
            System.out.print(" " + cur.data + " ");
            cur = cur.right;
        }
    }
}
```

```
// If this Node is a thread Node, then go to
// inorder successor
if (cur.isThreaded == true)
    cur = cur.right;
else // Else go to the leftmost child in right subtree
    cur = leftMost(cur.right);
}
}

// Driver program to test for above functions
public static void main(String args[])
{
    BinaryTree tree = new BinaryTree();
    tree.root = new Node(1);
    tree.root.left = new Node(2);
    tree.root.right = new Node(3);

    tree.root.left.left = new Node(4);
    tree.root.left.right = new Node(5);
    tree.root.right.left = new Node(6);
    tree.root.right.right = new Node(7);

    tree.createThreaded(tree.root);
    System.out.println("Inorder traversal of created threaded tree");
    tree.inOrder(tree.root);
}
}

// This code has been contributed by Mayank Jaiswal
```

Output:

```
Inorder traversal of creeated threaded tree is
4 2 5 1 6 3 7
```

### Convert a Binary Tree to Threaded binary tree | Set 2 (Efficient)

This article is contributed by **Minhaz**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

#### Source

<https://www.geeksforgeeks.org/convert-binary-tree-threaded-binary-tree-2/>

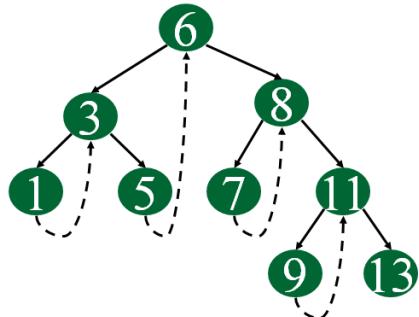
## Chapter 93

# Convert a Binary Tree to Threaded binary tree | Set 2 (Efficient)

Convert a Binary Tree to Threaded binary tree | Set 2 (Efficient) - GeeksforGeeks

Idea of [Threaded Binary Tree](#) is to make inorder traversal faster and do it without stack and without recursion. In a simple threaded binary tree, the NULL right pointers are used to store inorder successor. Where-ever a right pointer is NULL, it is used to store inorder successor.

Following diagram shows an example Single Threaded Binary Tree. The dotted lines represent threads.



Following is structure of single threaded binary tree.

```
struct Node
{
    int key;
    Node *left, *right;

    // Used to indicate whether the right pointer is a normal right
```

```
// pointer or a pointer to inorder successor.  
bool isThreaded;  
};
```

### How to convert a Given Binary Tree to Threaded Binary Tree?

We have discussed a Queue based solution [here](#). In this post, a space efficient solution is discussed that doesn't require queue.

The idea is based on the fact that we link from inorder predecessor to a node. We link those inorder predecessor which lie in subtree of node. So we find inorder predecessor of a node if its left is not NULL. Inorder predecessor of a node (whose left is NULL) is rightmost node in left child. Once we find the predecessor, we link a thread from it to current node. Following is the implementation of the above idea.

```
/* C++ program to convert a Binary Tree to  
   Threaded Tree */  
#include <iostream>  
#include <queue>  
using namespace std;  
  
/* Structure of a node in threaded binary tree */  
struct Node  
{  
    int key;  
    Node *left, *right;  
  
    // Used to indicate whether the right pointer  
    // is a normal right pointer or a pointer  
    // to inorder successor.  
    bool isThreaded;  
};  
  
// Converts tree with given root to threaded  
// binary tree.  
// This function returns rightmost child of  
// root.  
Node *createThreaded(Node *root)  
{  
    // Base cases : Tree is empty or has single  
    //               node  
    if (root == NULL)  
        return NULL;  
    if (root->left == NULL &&  
        root->right == NULL)  
        return root;  
  
    // Find predecessor if it exists  
    if (root->left != NULL)
```

```
{  
    // Find predecessor of root (Rightmost  
    // child in left subtree)  
    Node* l = createThreaded(root->left);  
  
    // Link a thread from predecessor to  
    // root.  
    l->right = root;  
    l->isThreaded = true;  
}  
  
// If current node is rightmost child  
if (root->right == NULL)  
    return root;  
  
// Recur for right subtree.  
return createThreaded(root->right);  
}  
  
// A utility function to find leftmost node  
// in a binary tree rooted with 'root'.  
// This function is used in inOrder()  
Node *leftMost(Node *root)  
{  
    while (root != NULL && root->left != NULL)  
        root = root->left;  
    return root;  
}  
  
// Function to do inorder traversal of a threaded  
// binary tree  
void inOrder(Node *root)  
{  
    if (root == NULL) return;  
  
    // Find the leftmost node in Binary Tree  
    Node *cur = leftMost(root);  
  
    while (cur != NULL)  
    {  
        cout << cur->key << " ";  
  
        // If this Node is a thread Node, then go to  
        // inorder successor  
        if (cur->isThreaded)  
            cur = cur->right;  
  
        else // Else go to the leftmost child in right subtree
    }
}
```

```
        cur = leftMost(cur->right);
    }
}

// A utility function to create a new node
Node *newNode(int key)
{
    Node *temp = new Node;
    temp->left = temp->right = NULL;
    temp->key = key;
    return temp;
}

// Driver program to test above functions
int main()
{
    /*      1
           / \
          2   3
         / \ / \
        4  5 6  7 */
    Node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->left = newNode(6);
    root->right->right = newNode(7);

    createThreaded(root);

    cout << "Inorder traversal of created "
         "threaded tree is\n";
    inOrder(root);
    return 0;
}
```

Output:

```
Inorder traversal of created threaded tree is
4 2 5 1 6 3 7
```

This algorithm works in  $O(n)$  time complexity and  $O(1)$  space other than function call stack.

## Source

<https://www.geeksforgeeks.org/convert-binary-tree-threaded-binary-tree-set-2-efficient/>

## Chapter 94

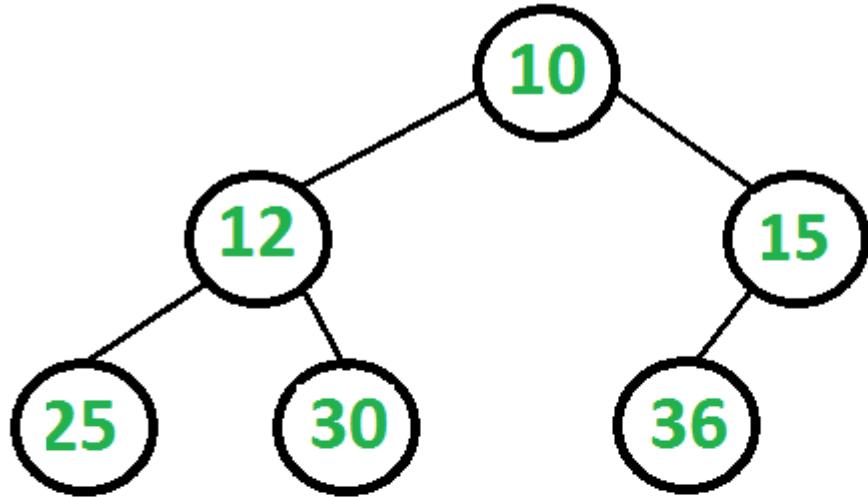
# Convert a Binary Tree to a Circular Doubly Link List

Convert a Binary Tree to a Circular Doubly Link List - GeeksforGeeks

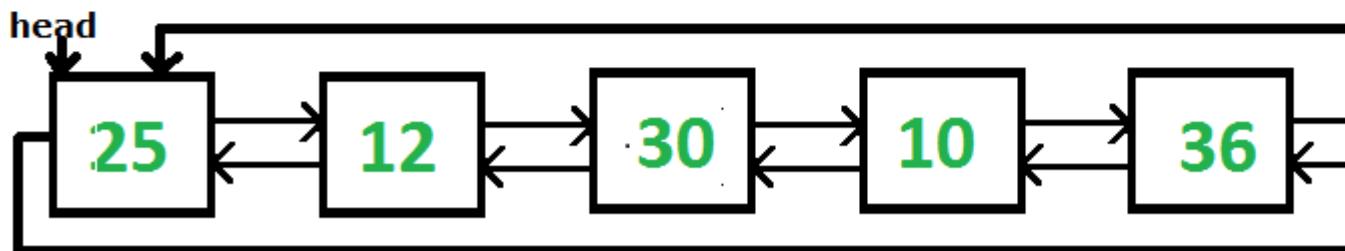
Given a Binary Tree, convert it to a Circular Doubly Linked List (In-Place).

- The left and right pointers in nodes are to be used as previous and next pointers respectively in converted Circular Linked List.
- The order of nodes in List must be same as Inorder of the given Binary Tree.
- The first node of Inorder traversal must be head node of the Circular List.

**Example:**



**The above tree should be in-place converted to following Circular Doubly Linked List**



The idea can be described using below steps.

- 1) Write a general purpose function that concatenates two given circular doubly lists (This function is explained below).
- 2) Now traverse the given tree
  - ....a) Recursively convert left subtree to a circular DLL. Let the converted list be leftList.
  - ....a) Recursively convert right subtree to a circular DLL. Let the converted list be rightList.
  - ....c) Make a circular linked list of root of the tree, make left and right of root to point to itself.
  - ....d) Concatenate leftList with list of single root node.
  - ....e) Concatenate the list produced in step above (d) with rightList.

Note that the above code traverses tree in Postorder fashion. We can traverse in inorder fashion also. We can first concatenate left subtree and root, then recur for right subtree and concatenate the result with left-root concatenation.

### How to Concatenate two circular DLLs?

- Get the last node of the left list. Retrieving the last node is an O(1) operation, since the prev pointer of the head points to the last node of the list.
- Connect it with the first node of the right list
- Get the last node of the second list
- Connect it with the head of the list.

Below are implementations of above idea.

C++

```
// C++ Program to convert a Binary Tree
// to a Circular Doubly Linked List
#include<iostream>
using namespace std;

// To represents a node of a Binary Tree
struct Node
{
    struct Node *left, *right;
    int data;
};

// A function that appends rightList at the end
// of leftList.
Node *concatenate(Node *leftList, Node *rightList)
{
    // If either of the list is empty
    // then return the other list
    if (leftList == NULL)
        return rightList;
    if (rightList == NULL)
        return leftList;

    // Store the last Node of left List
    Node *leftLast = leftList->left;

    // Store the last Node of right List
    Node *rightLast = rightList->left;

    // Connect the last node of Left List
    // with the first Node of the right List
    leftLast->right = rightList;
    rightList->left = leftLast;

    // Left of first node points to
    // the last node in the list
```

```
leftList->left = rightLast;

// Right of last node refers to the first
// node of the List
rightLast->right = leftList;

return leftList;
}

// Function converts a tree to a circular Linked List
// and then returns the head of the Linked List
Node *bTreeToCList(Node *root)
{
    if (root == NULL)
        return NULL;

    // Recursively convert left and right subtrees
    Node *left = bTreeToCList(root->left);
    Node *right = bTreeToCList(root->right);

    // Make a circular linked list of single node
    // (or root). To do so, make the right and
    // left pointers of this node point to itself
    root->left = root->right = root;

    // Step 1 (concatenate the left list with the list
    //          with single node, i.e., current node)
    // Step 2 (concatenate the returned list with the
    //          right List)
    return concatenate(concatenate(left, root), right);
}

// Display Circular Link List
void displayCList(Node *head)
{
    cout << "Circular Linked List is :\n";
    Node *itr = head;
    do
    {
        cout << itr->data << " ";
        itr = itr->right;
    } while (head!=itr);
    cout << "\n";
}

// Create a new Node and return its address
Node *newNode(int data)
```

```
{  
    Node *temp = new Node();  
    temp->data = data;  
    temp->left = temp->right = NULL;  
    return temp;  
}  
  
// Driver Program to test above function  
int main()  
{  
    Node *root = newNode(10);  
    root->left = newNode(12);  
    root->right = newNode(15);  
    root->left->left = newNode(25);  
    root->left->right = newNode(30);  
    root->right->left = newNode(36);  
  
    Node *head = bTreeToCList(root);  
    displayCList(head);  
  
    return 0;  
}
```

**Java**

```
// Java Program to convert a Binary Tree to a  
// Circular Doubly Linked List  
  
// Node class represents a Node of a Tree  
class Node  
{  
    int val;  
    Node left,right;  
  
    public Node(int val)  
    {  
        this.val = val;  
        left = right = null;  
    }  
}  
  
// A class to represent a tree  
class Tree  
{  
    Node root;  
    public Tree()  
    {  
        root = null;
```

```
}

// concatenate both the lists and returns the head
// of the List
public Node concatenate(Node leftList,Node rightList)
{
    // If either of the list is empty, then
    // return the other list
    if (leftList == null)
        return rightList;
    if (rightList == null)
        return leftList;

    // Store the last Node of left List
    Node leftLast = leftList.left;

    // Store the last Node of right List
    Node rightLast = rightList.left;

    // Connect the last node of Left List
    // with the first Node of the right List
    leftLast.right = rightList;
    rightList.left = leftLast;

    // left of first node refers to
    // the last node in the list
    leftList.left = rightLast;

    // Right of last node refers to the first
    // node of the List
    rightLast.right = leftList;

    // Return the Head of the List
    return leftList;
}

// Method converts a tree to a circular
// Link List and then returns the head
// of the Link List
public Node bTreeToCList(Node root)
{
    if (root == null)
        return null;

    // Recursively convert left and right subtrees
    Node left = bTreeToCList(root.left);
    Node right = bTreeToCList(root.right);
```

```
// Make a circular linked list of single node
// (or root). To do so, make the right and
// left pointers of this node point to itself
root.left = root.right = root;

// Step 1 (concatenate the left list with the list
//           with single node, i.e., current node)
// Step 2 (concatenate the returned list with the
//           right List)
return concatenate(concatenate(left, root), right);
}

// Display Circular Link List
public void display(Node head)
{
    System.out.println("Circular Linked List is :");
    Node itr = head;
    do
    {
        System.out.print(itr.val+ " ");
        itr = itr.right;
    }
    while (itr != head);
    System.out.println();
}
}

// Driver Code
class Main
{
    public static void main(String args[])
    {
        // Build the tree
        Tree tree = new Tree();
        tree.root = new Node(10);
        tree.root.left = new Node(12);
        tree.root.right = new Node(15);
        tree.root.left.left = new Node(25);
        tree.root.left.right = new Node(30);
        tree.root.right.left = new Node(36);

        // head refers to the head of the Link List
        Node head = tree.bTreeToCList(tree.root);

        // Display the Circular LinkedList
        tree.display(head);
    }
}
```

Output:

```
Circular Linked List is :  
25 12 30 10 36 15
```

### Source

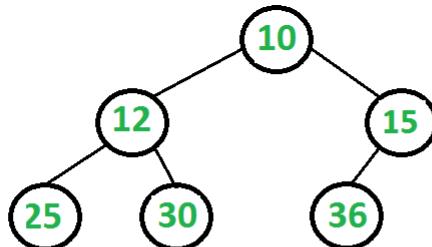
<https://www.geeksforgeeks.org/convert-a-binary-tree-to-a-circular-doubly-link-list/>

## Chapter 95

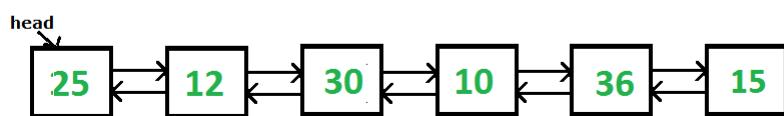
# Convert a given Binary Tree to Doubly Linked List | Set 1

Convert a given Binary Tree to Doubly Linked List | Set 1 - GeeksforGeeks

Given a Binary Tree (Bt), convert it to a Doubly Linked List(DLL). The left and right pointers in nodes are to be used as previous and next pointers respectively in converted DLL. The order of nodes in DLL must be same as Inorder of the given Binary Tree. The first node of Inorder traversal (left most node in BT) must be head node of the DLL.



**The above tree should be in-place converted to following Doubly Linked List(DLL).**



I came across this interview during one of my interviews. A similar problem is discussed in [this post](#). The problem here is simpler as we don't need to create circular DLL, but a simple DLL. The idea behind its solution is quite simple and straight.

1. If left subtree exists, process the left subtree
  - .....1.a) Recursively convert the left subtree to DLL.
  - .....1.b) Then find inorder predecessor of root in left subtree (inorder predecessor is rightmost node in left subtree).

- .....1.c) Make inorder predecessor as previous of root and root as next of inorder predecessor.  
2. If right subtree exists, process the right subtree (Below 3 steps are similar to left subtree).  
.....2.a) Recursively convert the right subtree to DLL.  
.....2.b) Then find inorder successor of root in right subtree (inorder successor is leftmost node in right subtree).  
.....2.c) Make inorder successor as next of root and root as previous of inorder successor.  
3. Find the leftmost node and return it (the leftmost node is always head of converted DLL).

Below is the source code for above algorithm.

C

```
// A C++ program for in-place conversion of Binary Tree to DLL
#include <stdio.h>

/* A binary tree node has data, and left and right pointers */
struct node
{
    int data;
    node* left;
    node* right;
};

/* This is the core function to convert Tree to list. This function follows
   steps 1 and 2 of the above algorithm */
node* bintree2listUtil(node* root)
{
    // Base case
    if (root == NULL)
        return root;

    // Convert the left subtree and link to root
    if (root->left != NULL)
    {
        // Convert the left subtree
        node* left = bintree2listUtil(root->left);

        // Find inorder predecessor. After this loop, left
        // will point to the inorder predecessor
        for (; left->right!=NULL; left=left->right);

        // Make root as next of the predecessor
        left->right = root;
    }

    // Make predecessor as previous of root
    root->left = left;
}
```

```
// Convert the right subtree and link to root
if (root->right!=NULL)
{
    // Convert the right subtree
    node* right = bintree2listUtil(root->right);

    // Find inorder successor. After this loop, right
    // will point to the inorder successor
    for (; right->left!=NULL; right = right->left);

    // Make root as previous of successor
    right->left = root;

    // Make successor as next of root
    root->right = right;
}

return root;
}

// The main function that first calls bintree2listUtil(), then follows step 3
// of the above algorithm
node* bintree2list(node *root)
{
    // Base case
    if (root == NULL)
        return root;

    // Convert to DLL using bintree2listUtil()
    root = bintree2listUtil(root);

    // bintree2listUtil() returns root node of the converted
    // DLL. We need pointer to the leftmost node which is
    // head of the constructed DLL, so move to the leftmost node
    while (root->left != NULL)
        root = root->left;

    return (root);
}

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
node* newNode(int data)
{
    node* new_node = new node;
    new_node->data = data;
    new_node->left = new_node->right = NULL;
    return (new_node);
```

```
}

/* Function to print nodes in a given doubly linked list */
void printList(node *node)
{
    while (node!=NULL)
    {
        printf("%d ", node->data);
        node = node->right;
    }
}

/* Driver program to test above functions*/
int main()
{
    // Let us create the tree shown in above diagram
    node *root      = newNode(10);
    root->left     = newNode(12);
    root->right    = newNode(15);
    root->left->left = newNode(25);
    root->left->right = newNode(30);
    root->right->left = newNode(36);

    // Convert to DLL
    node *head = bintree2list(root);

    // Print the converted list
    printList(head);

    return 0;
}
```

### Java

```
// Java program to convert binary tree to double linked list

/* A binary tree node has data, and left and right pointers */
class Node
{
    int data;
    Node left, right;

    Node(int item)
    {
        data = item;
        left = right = null;
    }
}
```

```
class BinaryTree
{
    Node root;
    /* This is the core function to convert Tree to list. This function
       follows steps 1 and 2 of the above algorithm */

    Node bintree2listUtil(Node node)
    {
        // Base case
        if (node == null)
            return node;

        // Convert the left subtree and link to root
        if (node.left != null)
        {
            // Convert the left subtree
            Node left = bintree2listUtil(node.left);

            // Find inorder predecessor. After this loop, left
            // will point to the inorder predecessor
            for (; left.right != null; left = left.right);

            // Make root as next of the predecessor
            left.right = node;

            // Make predecessor as previous of root
            node.left = left;
        }

        // Convert the right subtree and link to root
        if (node.right != null)
        {
            // Convert the right subtree
            Node right = bintree2listUtil(node.right);

            // Find inorder successor. After this loop, right
            // will point to the inorder successor
            for (; right.left != null; right = right.left);

            // Make root as previous of successor
            right.left = node;

            // Make successor as next of root
            node.right = right;
        }
    }

    return node;
}
```

```
}

// The main function that first calls bintree2listUtil(), then follows
// step 3 of the above algorithm

Node bintree2list(Node node)
{
    // Base case
    if (node == null)
        return node;

    // Convert to DLL using bintree2listUtil()
    node = bintree2listUtil(node);

    // bintree2listUtil() returns root node of the converted
    // DLL. We need pointer to the leftmost node which is
    // head of the constructed DLL, so move to the leftmost node
    while (node.left != null)
        node = node.left;

    return node;
}

/* Function to print nodes in a given doubly linked list */
void printList(Node node)
{
    while (node != null)
    {
        System.out.print(node.data + " ");
        node = node.right;
    }
}

/* Driver program to test above functions*/
public static void main(String[] args)
{
    BinaryTree tree = new BinaryTree();

    // Let us create the tree shown in above diagram
    tree.root = new Node(10);
    tree.root.left = new Node(12);
    tree.root.right = new Node(15);
    tree.root.left.left = new Node(25);
    tree.root.left.right = new Node(30);
    tree.root.right.left = new Node(36);

    // Convert to DLL
    Node head = tree.bintree2list(tree.root);
```

```
// Print the converted list
tree.printList(head);
}
}
```

### Python

```
# Python program to convert
# binary tree to doubly linked list

class Node(object):

    """Binary tree Node class has
    data, left and right child"""
    def __init__(self, item):
        self.data = item
        self.left = None
        self.right = None

def BTToDLLUtil(root):

    """This is a utility function to
    convert the binary tree to doubly
    linked list. Most of the core task
    is done by this function."""
    if root is None:
        return root

    # Convert left subtree
    # and link to root
    if root.left:

        # Convert the left subtree
        left = BTToDLLUtil(root.left)

        # Find inorder predecessor, After
        # this loop, left will point to the
        # inorder predecessor of root
        while left.right:
            left = left.right

        # Make root as next of predecessor
        left.right = root

        # Make predecessor as
        # previous of root
        root.left = left
```

```
# Convert the right subtree
# and link to root
if root.right:
    # Convert the right subtree
    right = BTToDLLUtil(root.right)

    # Find inorder successor, After
    # this loop, right will point to
    # the inorder successor of root
    while right.left:
        right = right.left

    # Make root as previous
    # of successor
    right.left = root

    # Make successor as
    # next of root
    root.right = right

return root

def BTToDLL(root):
    if root is None:
        return root

    # Convert to doubly linked
    # list using BLLToDLLUtil
    root = BTToDLLUtil(root)

    # We need pointer to left most
    # node which is head of the
    # constructed Doubly Linked list
    while root.left:
        root = root.left

    return root

def print_list(head):

    """Function to print the given
    doubly linked list"""
    if head is None:
        return
    while head:
        print(head.data, end = " ")
```

```
head = head.right

# Driver Code
if __name__ == '__main__':
    root = Node(10)
    root.left = Node(12)
    root.right = Node(15)
    root.left.left = Node(25)
    root.left.right = Node(30)
    root.right.left = Node(36)

    head = BTToDLL(root)
    print_list(head)

# This code is contributed
# by viveksyng
```

**Output:**

```
25 12 30 10 36 15
```

This article is compiled by **Ashish Mangla** and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

You may also like to see [Convert a given Binary Tree to Doubly Linked List | Set 2](#) for another simple and efficient solution.

**Improved By :** [viveksyng](#)

**Source**

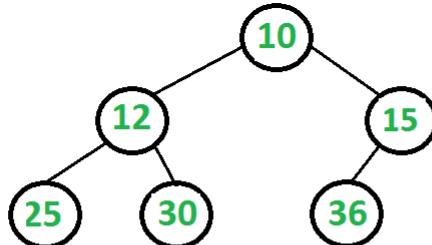
<https://www.geeksforgeeks.org/in-place-convert-a-given-binary-tree-to-doubly-linked-list/>

## Chapter 96

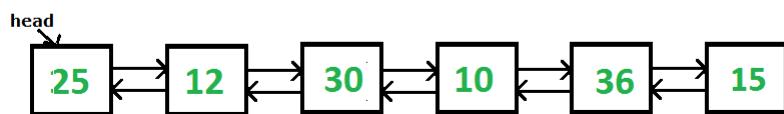
# Convert a given Binary Tree to Doubly Linked List | Set 2

Convert a given Binary Tree to Doubly Linked List | Set 2 - GeeksforGeeks

Given a Binary Tree (BT), convert it to a Doubly Linked List(DLL). The left and right pointers in nodes are to be used as previous and next pointers respectively in converted DLL. The order of nodes in DLL must be same as Inorder of the given Binary Tree. The first node of Inorder traversal (left most node in BT) must be head node of the DLL.



**The above tree should be in-place converted to following Doubly Linked List(DLL).**



A solution to this problem is discussed in [this post](#).

In this post, another simple and efficient solution is discussed. The solution discussed here has two simple steps.

- 1) **Fix Left Pointers:** In this step, we change left pointers to point to previous nodes in DLL. The idea is simple, we do inorder traversal of tree. In inorder traversal, we keep track of previous visited node and change left pointer to the previous node. See *fixPrevPtr()* in below implementation.

**2) Fix Right Pointers:** The above is intuitive and simple. How to change right pointers to point to next node in DLL? The idea is to use left pointers fixed in step 1. We start from the rightmost node in Binary Tree (BT). The rightmost node is the last node in DLL. Since left pointers are changed to point to previous node in DLL, we can linearly traverse the complete DLL using these pointers. The traversal would be from last to first node. While traversing the DLL, we keep track of the previously visited node and change the right pointer to the previous node. See *fixNextPtr()* in below implementation.

C

```
// A simple inorder traversal based program to convert a Binary Tree to DLL
#include<stdio.h>
#include<stdlib.h>

// A tree node
struct node
{
    int data;
    struct node *left, *right;
};

// A utility function to create a new tree node
struct node *newNode(int data)
{
    struct node *node = (struct node *)malloc(sizeof(struct node));
    node->data = data;
    node->left = node->right = NULL;
    return(node);
}

// Standard Inorder traversal of tree
void inorder(struct node *root)
{
    if (root != NULL)
    {
        inorder(root->left);
        printf("\t%d",root->data);
        inorder(root->right);
    }
}

// Changes left pointers to work as previous pointers in converted DLL
// The function simply does inorder traversal of Binary Tree and updates
// left pointer using previously visited node
void fixPrevPtr(struct node *root)
{
    static struct node *pre = NULL;

    if (root != NULL)
```

```
{  
    fixPrevPtr(root->left);  
    root->left = pre;  
    pre = root;  
    fixPrevPtr(root->right);  
}  
}  
  
// Changes right pointers to work as next pointers in converted DLL  
struct node *fixNextPtr(struct node *root)  
{  
    struct node *prev = NULL;  
  
    // Find the right most node in BT or last node in DLL  
    while (root && root->right != NULL)  
        root = root->right;  
  
    // Start from the rightmost node, traverse back using left pointers.  
    // While traversing, change right pointer of nodes.  
    while (root && root->left != NULL)  
    {  
        prev = root;  
        root = root->left;  
        root->right = prev;  
    }  
  
    // The leftmost node is head of linked list, return it  
    return (root);  
}  
  
// The main function that converts BST to DLL and returns head of DLL  
struct node *BTToDLL(struct node *root)  
{  
    // Set the previous pointer  
    fixPrevPtr(root);  
  
    // Set the next pointer and return head of DLL  
    return fixNextPtr(root);  
}  
  
// Traverses the DLL from left to right  
void printList(struct node *root)  
{  
    while (root != NULL)  
    {  
        printf("\t%d", root->data);  
        root = root->right;  
    }  
}
```

```
}

// Driver program to test above functions
int main(void)
{
    // Let us create the tree shown in above diagram
    struct node *root = newNode(10);
    root->left      = newNode(12);
    root->right     = newNode(15);
    root->left->left  = newNode(25);
    root->left->right = newNode(30);
    root->right->left = newNode(36);

    printf("\n\t\tInorder Tree Traversal\n\n");
    inorder(root);

    struct node *head = BTToDLL(root);

    printf("\n\n\t\tDLL Traversal\n\n");
    printList(head);
    return 0;
}
```

**Java**

```
// Java program to convert BTT to DLL using
// simple inorder traversal

public class BinaryTreeToDLL
{
    static class node
    {
        int data;
        node left, right;

        public node(int data)
        {
            this.data = data;
        }
    }

    static node prev;

    // Changes left pointers to work as previous
    // pointers in converted DLL The function
    // simply does inorder traversal of Binary
    // Tree and updates left pointer using
    // previously visited node
```

```
static void fixPrevptr(node root)
{
    if (root == null)
        return;

    fixPrevptr(root.left);
    root.left = prev;
    prev = root;
    fixPrevptr(root.right);

}

// Changes right pointers to work
// as next pointers in converted DLL
static node fixNextptr(node root)
{
    // Find the right most node in
    // BT or last node in DLL
    while (root.right != null)
        root = root.right;

    // Start from the rightmost node, traverse
    // back using left pointers. While traversing,
    // change right pointer of nodes
    while (root != null && root.left != null)
    {
        node left = root.left;
        left.right = root;
        root = root.left;
    }

    // The leftmost node is head of linked list, return it
    return root;
}

static node BTTtoDLL(node root)
{
    prev = null;

    // Set the previous pointer
    fixPrevptr(root);

    // Set the next pointer and return head of DLL
    return fixNextptr(root);
}

// Traverses the DLL from left to right
static void printlist(node root)
```

```
{  
    while (root != null)  
    {  
        System.out.print(root.data + " ");  
        root = root.right;  
    }  
}  
  
// Standard Inorder traversal of tree  
static void inorder(node root)  
{  
    if (root == null)  
        return;  
    inorder(root.left);  
    System.out.print(root.data + " ");  
    inorder(root.right);  
}  
  
public static void main(String[] args)  
{  
    // Let us create the tree shown in above diagram  
    node root = new node(10);  
    root.left = new node(12);  
    root.right = new node(15);  
    root.left.left = new node(25);  
    root.left.right = new node(30);  
    root.right.left = new node(36);  
  
    System.out.println("Inorder Tree Traversal");  
    inorder(root);  
  
    node head = BTToDLL(root);  
  
    System.out.println("\nDLL Traversal");  
    printlist(head);  
}  
}  
  
// This code is contributed by Rishabh Mahrsee
```

### Python

```
# A simple inorder traversal based program to convert a  
# Binary Tree to DLL  
  
# A Binary Tree node  
class Node:
```

```
# Constructor to create a new tree node
def __init__(self, data):
    self.data = data
    self.left = None
    self.right = None

# Standard Inorder traversal of tree
def inorder(root):

    if root is not None:
        inorder(root.left)
        print "\t%d" %(root.data),
        inorder(root.right)

# Changes left pointers to work as previous pointers
# in converted DLL
# The function simply does inorder traversal of
# Binary Tree and updates
# left pointer using previously visited node
def fixPrevPtr(root):
    if root is not None:
        fixPrevPtr(root.left)
        root.left = fixPrevPtr.pre
        fixPrevPtr.pre = root
        fixPrevPtr(root.right)

# Changes right pointers to work as next pointers in
# converted DLL
def fixNextPtr(root):

    prev = None
    # Find the right most node in BT or last node in DLL
    while(root and root.right != None):
        root = root.right

    # Start from the rightmost node, traverse back using
    # left pointers
    # While traversing, change right pointer of nodes
    while(root and root.left != None):
        prev = root
        root = root.left
        root.right = prev

    # The leftmost node is head of linked list, return it
    return root

# The main function that converts BST to DLL and returns
# head of DLL
```

```
def BTToDLL(root):

    # Set the previous pointer
    fixPrevPtr(root)

    # Set the next pointer and return head of DLL
    return fixNextPtr(root)

# Traversses the DLL from left to right
def printList(root):
    while(root != None):
        print "\t%d" %(root.data),
        root = root.right

# Driver program to test above function
root = Node(10)
root.left = Node(12)
root.right = Node(15)
root.left.left = Node(25)
root.left.right = Node(30)
root.right.left = Node(36)

print "\n\t\t Inorder Tree Traversal\n"
inorder(root)

# Static variable pre for function fixPrevPtr
fixPrevPtr.pre = None
head = BTToDLL(root)

print "\n\n\t\tDLL Traversal\n"
printList(head)

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

Output:

Inorder Tree Traversal

25      12      30      10      36      15

DLL Traversal

25      12      30      10      36      15

Time Complexity: O(n) where n is the number of nodes in given Binary Tree. The solution simply does two traversals of all Binary Tree nodes.

This article is contributed by **Bala**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

### Source

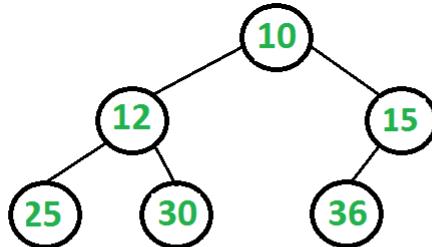
<https://www.geeksforgeeks.org/convert-a-given-binary-tree-to-doubly-linked-list-set-2/>

## Chapter 97

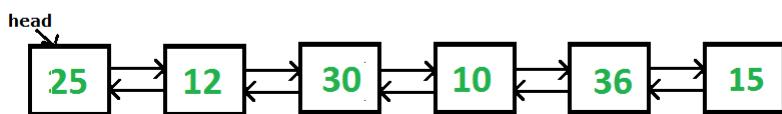
# Convert a given Binary Tree to Doubly Linked List | Set 3

Convert a given Binary Tree to Doubly Linked List | Set 3 - GeeksforGeeks

Given a Binary Tree (BT), convert it to a Doubly Linked List(DLL) In-Place. The left and right pointers in nodes are to be used as previous and next pointers respectively in converted DLL. The order of nodes in DLL must be same as Inorder of the given Binary Tree. The first node of Inorder traversal (left most node in BT) must be head node of the DLL.



**The above tree should be in-place converted to following Doubly Linked List(DLL).**



Following two different solutions have been discussed for this problem.

[Convert a given Binary Tree to Doubly Linked List | Set 1](#)

[Convert a given Binary Tree to Doubly Linked List | Set 2](#)

In this post, a third solution is discussed which seems to be the simplest of all. The idea is to do inorder traversal of the binary tree. While doing inorder traversal, keep track of the previously visited node in a variable say *prev*. For every visited node, make it next of *prev* and previous of this node as *prev*.

Thanks to rahul, wishall and all other readers for their useful comments on the above two posts.

Following is C++ implementation of this solution.

C++

```
// A C++ program for in-place conversion of Binary Tree to DLL
#include <iostream>
using namespace std;

/* A binary tree node has data, and left and right pointers */
struct node
{
    int data;
    node* left;
    node* right;
};

// A simple recursive function to convert a given Binary tree to Doubly
// Linked List
// root --> Root of Binary Tree
// head --> Pointer to head node of created doubly linked list
void BinaryTree2DoubleLinkedList(node *root, node **head)
{
    // Base case
    if (root == NULL) return;

    // Initialize previously visited node as NULL. This is
    // static so that the same value is accessible in all recursive
    // calls
    static node* prev = NULL;

    // Recursively convert left subtree
    BinaryTree2DoubleLinkedList(root->left, head);

    // Now convert this node
    if (prev == NULL)
        *head = root;
    else
    {
        root->left = prev;
        prev->right = root;
    }
    prev = root;

    // Finally convert right subtree
    BinaryTree2DoubleLinkedList(root->right, head);
}
```

```
}

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
node* newNode(int data)
{
    node* new_node = new node;
    new_node->data = data;
    new_node->left = new_node->right = NULL;
    return (new_node);
}

/* Function to print nodes in a given doubly linked list */
void printList(node *node)
{
    while (node!=NULL)
    {
        cout << node->data << " ";
        node = node->right;
    }
}

/* Driver program to test above functions*/
int main()
{
    // Let us create the tree shown in above diagram
    node *root      = newNode(10);
    root->left     = newNode(12);
    root->right    = newNode(15);
    root->left->left  = newNode(25);
    root->left->right = newNode(30);
    root->right->left = newNode(36);

    // Convert to DLL
    node *head = NULL;
    BinaryTree2DoubleLinkedList(root, &head);

    // Print the converted list
    printList(head);

    return 0;
}
```

**Java**

```
// A Java program for in-place conversion of Binary Tree to DLL

// A binary tree node has data, left pointers and right pointers
```

```
class Node
{
    int data;
    Node left, right;

    public Node(int data)
    {
        this.data = data;
        left = right = null;
    }
}

class BinaryTree
{
    Node root;

    // head --> Pointer to head node of created doubly linked list
    Node head;

    // Initialize previously visited node as NULL. This is
    // static so that the same value is accessible in all recursive
    // calls
    static Node prev = null;

    // A simple recursive function to convert a given Binary tree
    // to Doubly Linked List
    // root --> Root of Binary Tree
    void BinaryTree2DoubleLinkedList(Node root)
    {
        // Base case
        if (root == null)
            return;

        // Recursively convert left subtree
        BinaryTree2DoubleLinkedList(root.left);

        // Now convert this node
        if (prev == null)
            head = root;
        else
        {
            root.left = prev;
            prev.right = root;
        }
        prev = root;

        // Finally convert right subtree
        BinaryTree2DoubleLinkedList(root.right);
    }
}
```

```
}

/* Function to print nodes in a given doubly linked list */
void printList(Node node)
{
    while (node != null)
    {
        System.out.print(node.data + " ");
        node = node.right;
    }
}

// Driver program to test above functions
public static void main(String[] args)
{
    // Let us create the tree as shown in above diagram
    BinaryTree tree = new BinaryTree();
    tree.root = new Node(10);
    tree.root.left = new Node(12);
    tree.root.right = new Node(15);
    tree.root.left.left = new Node(25);
    tree.root.left.right = new Node(30);
    tree.root.right.left = new Node(36);

    // convert to DLL
    tree.BinaryTree2DoubleLinkedList(tree.root);

    // Print the converted List
    tree.printList(tree.head);
}

}

// This code has been contributed by Mayank Jaiswal(mayank_24)
```

Output:

```
25 12 30 10 36 15
```

Note that use of static variables like above is not a recommended practice (we have used static for simplicity). Imagine a situation where same function is called for two or more trees, the old value of *prev* would be used in next call for a different tree. To avoid such problems, we can use double pointer or reference to a pointer.

Time Complexity: The above program does a simple inorder traversal, so time complexity is  $O(n)$  where  $n$  is the number of nodes in given binary tree.

**Source**

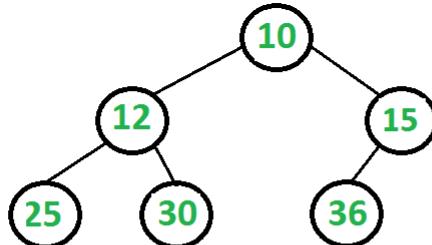
<https://www.geeksforgeeks.org/convert-given-binary-tree-doubly-linked-list-set-3/>

## Chapter 98

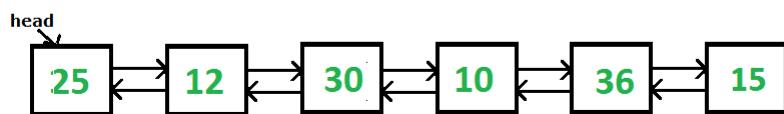
# Convert a given Binary Tree to Doubly Linked List | Set 4

Convert a given Binary Tree to Doubly Linked List | Set 4 - GeeksforGeeks

Given a Binary Tree (BT), convert it to a Doubly Linked List(DLL) In-Place. The left and right pointers in nodes are to be used as previous and next pointers respectively in converted DLL. The order of nodes in DLL must be same as Inorder of the given Binary Tree. The first node of Inorder traversal (left most node in BT) must be head node of the DLL.



**The above tree should be in-place converted to following Doubly Linked List(DLL).**



Below three different solutions have been discussed for this problem.

[Convert a given Binary Tree to Doubly Linked List | Set 1](#)

[Convert a given Binary Tree to Doubly Linked List | Set 2](#)

[Convert a given Binary Tree to Doubly Linked List | Set 3](#)

In the following implementation, we traverse the tree in inorder fashion. We add nodes at the beginning of current linked list and update head of the list using pointer to head pointer. Since we insert at the beginning, we need to process leaves in reverse order. For reverse

order, we first traverse the right subtree before the left subtree. i.e. do a reverse inorder traversal.

C++

```
// C++ program to convert a given Binary
// Tree to Doubly Linked List
#include <stdio.h>
#include <stdlib.h>

// Structure for tree and linked list
struct Node
{
    int data;
    Node *left, *right;
};

// A simple recursive function to convert a given
// Binary tree to Doubly Linked List
// root      --> Root of Binary Tree
// head_ref --> Pointer to head node of created
//                  doubly linked list
void BTodLL(Node* root, Node** head_ref)
{
    // Base cases
    if (root == NULL)
        return;

    // Recursively convert right subtree
    BTodLL(root->right, head_ref);

    // insert root into DLL
    root->right = *head_ref;

    // Change left pointer of previous head
    if (*head_ref != NULL)
        (*head_ref)->left = root;

    // Change head of Doubly linked list
    *head_ref = root;

    // Recursively convert left subtree
    BTodLL(root->left, head_ref);
}

// Utility function for allocating node for Binary
// Tree.
Node* newNode(int data)
{
```

```
Node* newNode = new Node;
node->data = data;
node->left = node->right = NULL;
return node;
}

// Utility function for printing double linked list.
void printList(Node* head)
{
    printf("Extracted Double Linked list is:\n");
    while (head)
    {
        printf("%d ", head->data);
        head = head->right;
    }
}

// Driver program to test above function
int main()
{
    /* Constructing below tree
           5
         /   \
        3     6
       / \   /
      1   4   8
     / \   / \
    0   2   7   9 */
    Node* root = newNode(5);
    root->left = newNode(3);
    root->right = newNode(6);
    root->left->left = newNode(1);
    root->left->right = newNode(4);
    root->right->right = newNode(8);
    root->left->left->left = newNode(0);
    root->left->left->right = newNode(2);
    root->right->right->left = newNode(7);
    root->right->right->right = newNode(9);

    Node* head = NULL;
    BToDLL(root, &head);

    printList(head);

    return 0;
}
```

Java

```
// Java program to convert a given Binary Tree to
// Doubly Linked List

/* Structure for tree and Linked List */
class Node
{
    int data;
    Node left, right;

    public Node(int data)
    {
        this.data = data;
        left = right = null;
    }
}

class BinaryTree
{
    // 'root' - root of binary tree
    Node root;

    // 'head' - reference to head node of created
    // double linked list
    Node head;

    // A simple recursive function to convert a given
    // Binary tree to Doubly Linked List
    void BToDLL(Node root)
    {
        // Base cases
        if (root == null)
            return;

        // Recursively convert right subtree
        BToDLL(root.right);

        // insert root into DLL
        root.right = head;

        // Change left pointer of previous head
        if (head != null)
            (head).left = root;

        // Change head of Doubly linked list
        head = root;

        // Recursively convert left subtree
        BToDLL(root.left);
    }
}
```

```
}

// Utility function for printing double linked list.
void printList(Node head)
{
    System.out.println("Extracted Double Linked List is : ");
    while (head != null)
    {
        System.out.print(head.data + " ");
        head = head.right;
    }
}

// Driver program to test the above functions
public static void main(String[] args)
{
    /* Constructing below tree
       5
      / \
     3   6
    / \   \
   1   4   8
  / \   / \
 0   2   7   9 */
    BinaryTree tree = new BinaryTree();
    tree.root = new Node(5);
    tree.root.left = new Node(3);
    tree.root.right = new Node(6);
    tree.root.left.right = new Node(4);
    tree.root.left.left = new Node(1);
    tree.root.right.right = new Node(8);
    tree.root.left.left.right = new Node(2);
    tree.root.left.left.left = new Node(0);
    tree.root.right.right.left = new Node(7);
    tree.root.right.right.right = new Node(9);

    tree.BToDLL(tree.root);
    tree.printList(tree.head);
}

// This code has been contributed by Mayank Jaiswal(mayank_24)
```

Output :

Extracted Double Linked list is:

0 1 2 3 4 5 6 7 8 9

Time Complexity: O(n), as the solution does a single traversal of given Binary Tree.

This article is contributed by **Aditya Goel**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## Source

<https://www.geeksforgeeks.org/convert-a-given-binary-tree-to-doubly-linked-list-set-4/>

## Chapter 99

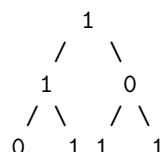
# Convert a given Binary tree to a tree that holds Logical AND property

Convert a given Binary tree to a tree that holds Logical AND property - GeeksforGeeks

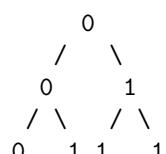
Given a Binary Tree (Every node has at most 2 children) where each node has value either 0 or 1. Convert a given Binary tree to a tree that holds Logical AND property, i.e., each node value should be the logical AND between its children.

Examples:

Input : The below tree doesn't hold the logical AND property  
convert it to a tree that holds the property.



Output :



The idea is to traverse given binary tree in postorder fashion. For each node check (recursively) if the node has one children then we don't have any need to check else if the node has both its child then simply update the node data with the logical AND of its child data.

```
// C++ code to convert a given binary tree
// to a tree that holds logical AND property.
#include<bits/stdc++.h>
using namespace std;

// Structure of binary tree
struct Node
{
    int data;
    struct Node* left;
    struct Node* right;
};

// function to create a new node
struct Node* newNode(int key)
{
    struct Node* node = new Node;
    node->data= key;
    node->left = node->right = NULL;
    return node;
}

// Convert the given tree to a tree where
// each node is logical AND of its children
// The main idea is to do Postorder traversal
void convertTree(Node *root)
{
    if (root == NULL)
        return;

    /* first recur on left child */
    convertTree(root->left);

    /* then recur on right child */
    convertTree(root->right);

    if (root->left != NULL && root->right != NULL)
        root->data = (root->left->data) &
                     (root->right->data);
}

void printInorder(Node* root)
{
    if (root == NULL)
        return;

    /* first recur on left child */
    printInorder(root->left);
```

```
/* then print the data of node */
printf("%d ", root->data);

/* now recur on right child */
printInorder(root->right);
}

// main function
int main()
{
    /* Create following Binary Tree
        1
       /   \
      1     0
     / \   / \
    0   1 1   1
    */
    Node *root=newNode(0);
    root->left=newNode(1);
    root->right=newNode(0);
    root->left->left=newNode(0);
    root->left->right=newNode(1);
    root->right->left=newNode(1);
    root->right->right=newNode(1);
    printf("\n Inorder traversal before conversion ");
    printInorder(root);

    convertTree(root);

    printf("\n Inorder traversal after conversion ");
    printInorder(root);
    return 0;
}
```

Output:

```
Inorder traversal before conversion 0 1 1 0 1 0 1
Inorder traversal after conversion 0 0 1 0 1 1 1
```

## Source

<https://www.geeksforgeeks.org/convert-given-binary-tree-tree-holds-logical-property/>

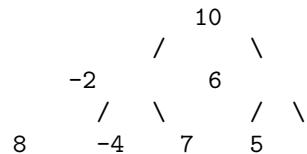
# Chapter 100

## Convert a given tree to its Sum Tree

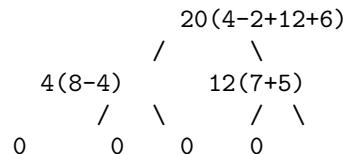
Convert a given tree to its Sum Tree - GeeksforGeeks

Given a Binary Tree where each node has positive and negative values. Convert this to a tree where each node contains the sum of the left and right sub trees in the original tree. The values of leaf nodes are changed to 0.

For example, the following tree



should be changed to



### Solution:

Do a traversal of the given tree. In the traversal, store the old value of the current node, recursively call for left and right subtrees and change the value of current node as sum of the values returned by the recursive calls. Finally return the sum of new value and value (which is sum of values in the subtree rooted with this node).

C

```
#include<stdio.h>

/* A tree node structure */
struct node
{
    int data;
    struct node *left;
    struct node *right;
};

// Convert a given tree to a tree where every node contains sum of values of
// nodes in left and right subtrees in the original tree
int toSumTree(struct node *node)
{
    // Base case
    if(node == NULL)
        return 0;

    // Store the old value
    int old_val = node->data;

    // Recursively call for left and right subtrees and store the sum as
    // new value of this node
    node->data = toSumTree(node->left) + toSumTree(node->right);

    // Return the sum of values of nodes in left and right subtrees and
    // old_value of this node
    return node->data + old_val;
}

// A utility function to print inorder traversal of a Binary Tree
void printInorder(struct node* node)
{
    if (node == NULL)
        return;
    printInorder(node->left);
    printf("%d ", node->data);
    printInorder(node->right);
}

/* Utility function to create a new Binary Tree node */
struct node* newNode(int data)
{
    struct node *temp = new struct node;
    temp->data = data;
    temp->left = NULL;
    temp->right = NULL;
}
```

```
    return temp;
}

/* Driver function to test above functions */
int main()
{
    struct node *root = NULL;
    int x;

    /* Constructing tree given in the above figure */
    root = newNode(10);
    root->left = newNode(-2);
    root->right = newNode(6);
    root->left->left = newNode(8);
    root->left->right = newNode(-4);
    root->right->left = newNode(7);
    root->right->right = newNode(5);

    toSumTree(root);

    // Print inorder traversal of the converted tree to test result of toSumTree()
    printf("Inorder Traversal of the resultant tree is: \n");
    printInorder(root);

    getchar();
    return 0;
}
```

### Java

```
// Java program to convert a tree into its sum tree

// A binary tree node
class Node
{
    int data;
    Node left, right;

    Node(int item)
    {
        data = item;
        left = right = null;
    }
}

class BinaryTree
{
    Node root;
```

```
// Convert a given tree to a tree where every node contains sum of
// values of nodes in left and right subtrees in the original tree
int toSumTree(Node node)
{
    // Base case
    if (node == null)
        return 0;

    // Store the old value
    int old_val = node.data;

    // Recursively call for left and right subtrees and store the sum
    // as new value of this node
    node.data = toSumTree(node.left) + toSumTree(node.right);

    // Return the sum of values of nodes in left and right subtrees
    // and old_value of this node
    return node.data + old_val;
}

// A utility function to print inorder traversal of a Binary Tree
void printInorder(Node node)
{
    if (node == null)
        return;
    printInorder(node.left);
    System.out.print(node.data + " ");
    printInorder(node.right);
}

/* Driver function to test above functions */
public static void main(String args[])
{
    BinaryTree tree = new BinaryTree();

    /* Constructing tree given in the above figure */
    tree.root = new Node(10);
    tree.root.left = new Node(-2);
    tree.root.right = new Node(6);
    tree.root.left.left = new Node(8);
    tree.root.left.right = new Node(-4);
    tree.root.right.left = new Node(7);
    tree.root.right.right = new Node(5);

    tree.toSumTree(tree.root);

    // Print inorder traversal of the converted tree to test result
```

```
// of toSumTree()
System.out.println("Inorder Traversal of the resultant tree is:");
tree.printInorder(tree.root);
}
}

// This code has been contributed by Mayank Jaiswal
```

Output:

```
Inorder Traversal of the resultant tree is:
0 4 0 20 0 12 0
```

Time Complexity: The solution involves a simple traversal of the given tree. So the time complexity is  $O(n)$  where  $n$  is the number of nodes in the given Binary Tree.

## Source

<https://www.geeksforgeeks.org/convert-a-given-tree-to-sum-tree/>

# Chapter 101

## Convert a normal BST to Balanced BST

Convert a normal BST to Balanced BST - GeeksforGeeks

Given a BST (**Binary Search Tree**) that may be unbalanced, convert it into a balanced BST that has minimum possible height.

Examples :

Input:

```
    30
     /
    20
     /
   10
```

Output:

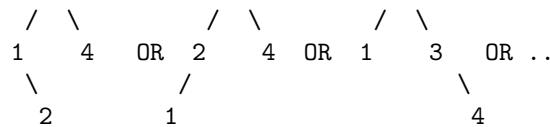
```
    20
   /   \
  10   30
```

Input:

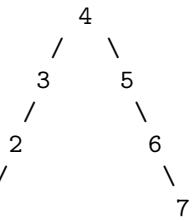
```
    4
     /
    3
     /
   2
    /
  1
```

Output:

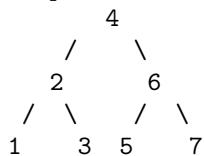
```
 3           3           2
```



Input:



Output:



A **Simple Solution** is to traverse nodes in Inorder and one by one insert into a self-balancing BST like AVL tree. Time complexity of this solution is  $O(n \log n)$  and this solution doesn't guarantee

An **Efficient Solution** can construct balanced BST in  $O(n)$  time with minimum possible height. Below are steps.

1. Traverse given BST in inorder and store result in an array. This step takes  $O(n)$  time. Note that this array would be sorted as inorder traversal of BST always produces sorted sequence.
2. Build a balanced BST from the above created sorted array using the recursive approach discussed [here](#). This step also takes  $O(n)$  time as we traverse every element exactly once and processing an element takes  $O(1)$  time.

Below is C++ implementation of above steps.

C++

```

// C++ program to convert a left unbalanced BST to
// a balanced BST
#include <bits/stdc++.h>
using namespace std;

struct Node
{
    int data;
    Node* left, *right;
}
  
```

```
};

/* This function traverse the skewed binary tree and
   stores its nodes pointers in vector nodes[] */
void storeBSTNodes(Node* root, vector<Node*> &nodes)
{
    // Base case
    if (root==NULL)
        return;

    // Store nodes in Inorder (which is sorted
    // order for BST)
    storeBSTNodes(root->left, nodes);
    nodes.push_back(root);
    storeBSTNodes(root->right, nodes);
}

/* Recursive function to construct binary tree */
Node* buildTreeUtil(vector<Node*> &nodes, int start,
                    int end)
{
    // base case
    if (start > end)
        return NULL;

    /* Get the middle element and make it root */
    int mid = (start + end)/2;
    Node *root = nodes[mid];

    /* Using index in Inorder traversal, construct
       left and right subtress */
    root->left = buildTreeUtil(nodes, start, mid-1);
    root->right = buildTreeUtil(nodes, mid+1, end);

    return root;
}

// This functions converts an unbalanced BST to
// a balanced BST
Node* buildTree(Node* root)
{
    // Store nodes of given BST in sorted order
    vector<Node *> nodes;
    storeBSTNodes(root, nodes);

    // Constructs BST from nodes[]
    int n = nodes.size();
    return buildTreeUtil(nodes, 0, n-1);
```

```
}

// Utility function to create a new node
Node* newNode(int data)
{
    Node* node = new Node;
    node->data = data;
    node->left = node->right = NULL;
    return (node);
}

/* Function to do preorder traversal of tree */
void preOrder(Node* node)
{
    if (node == NULL)
        return;
    printf("%d ", node->data);
    preOrder(node->left);
    preOrder(node->right);
}

// Driver program
int main()
{
    /* Constructed skewed binary tree is
       10
        /
       8
        /
       7
        /
       6
        /
      5   */

    Node* root = newNode(10);
    root->left = newNode(8);
    root->left->left = newNode(7);
    root->left->left->left = newNode(6);
    root->left->left->left->left = newNode(5);

    root = buildTree(root);

    printf("Preorder traversal of balanced "
          "BST is : \n");
    preOrder(root);

    return 0;
}
```

}

**Java**

```
// Java program to convert a left unbalanced BST to a balanced BST

import java.util.*;

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
class Node
{
    int data;
    Node left, right;

    public Node(int data)
    {
        this.data = data;
        left = right = null;
    }
}

class BinaryTree
{
    Node root;

    /* This function traverse the skewed binary tree and
       stores its nodes pointers in vector nodes[] */
    void storeBSTNodes(Node root, Vector<Node> nodes)
    {
        // Base case
        if (root == null)
            return;

        // Store nodes in Inorder (which is sorted
        // order for BST)
        storeBSTNodes(root.left, nodes);
        nodes.add(root);
        storeBSTNodes(root.right, nodes);
    }

    /* Recursive function to construct binary tree */
    Node buildTreeUtil(Vector<Node> nodes, int start,
                      int end)
    {
        // base case
        if (start > end)
            return null;
```

```

/* Get the middle element and make it root */
int mid = (start + end) / 2;
Node node = nodes.get(mid);

/* Using index in Inorder traversal, construct
   left and right subtress */
node.left = buildTreeUtil(nodes, start, mid - 1);
node.right = buildTreeUtil(nodes, mid + 1, end);

return node;
}

// This functions converts an unbalanced BST to
// a balanced BST
Node buildTree(Node root)
{
    // Store nodes of given BST in sorted order
    Vector<Node> nodes = new Vector<Node>();
    storeBSTNodes(root, nodes);

    // Constructs BST from nodes[]
    int n = nodes.size();
    return buildTreeUtil(nodes, 0, n - 1);
}

/* Function to do preorder traversal of tree */
void preOrder(Node node)
{
    if (node == null)
        return;
    System.out.print(node.data + " ");
    preOrder(node.left);
    preOrder(node.right);
}

// Driver program to test the above functions
public static void main(String[] args)
{
    /* Constructed skewed binary tree is
       10
       /
      8
      /
     7
      /
     6
      /
    */
}

```

```
5  */
BinaryTree tree = new BinaryTree();
tree.root = new Node(10);
tree.root.left = new Node(8);
tree.root.left.left = new Node(7);
tree.root.left.left.left = new Node(6);
tree.root.left.left.left.left = new Node(5);

tree.root = tree.buildTree(tree.root);
System.out.println("Preorder traversal of balanced BST is :");
tree.preOrder(tree.root);
}

}

// This code has been contributed by Mayank Jaiswal(mayank_24)
```

Output :

```
Preorder traversal of balanced BST is :
7 5 6 8 10
```

## Source

<https://www.geeksforgeeks.org/convert-normal-bst-balanced-bst/>

## Chapter 102

# Convert a tree to forest of even nodes

Convert a tree to forest of even nodes - GeeksforGeeks

Given a tree of  $n$  even nodes. The task is to find the maximum number of edges to be removed from the given tree to obtain forest of trees having even number of nodes. This problem is always solvable as given graph has even nodes.

Examples:

```
Input : n = 10
Edge 1: 1 3
Edge 2: 1 6
Edge 3: 1 2
Edge 4: 3 4
Edge 5: 6 8
Edge 6: 2 7
Edge 7: 2 5
Edge 8: 4 9
Edge 9: 4 10
```

```
Output : 2
```

By removing 2 edges we can obtain the forest with even node tree.

Dotted line shows removed edges. Any further removal of edge will not satisfy the even nodes condition.

Find a subtree with even number of nodes and remove it from rest of tree by removing the edge connecting it. After removal, we are left with tree with even node only because initially we have even number of nodes in the tree and removed subtree has also even node. Repeat

the same procedure until we left with the tree that cannot be further decomposed in this manner.

To do this, the idea is to use [Depth First Search](#) to traverse the tree. Implement DFS function in such a manner that it will return number of nodes in the subtree whose root is node on which DFS is performed. If the number of nodes is even then remove the edge, else ignore.

Below is C++ implementation of this approach:

```
// C++ program to find maximum number to be removed
// to convert a tree into forest containing trees of
// even number of nodes
#include<bits/stdc++.h>
#define N 12
using namespace std;

// Return the number of nodes of subtree having
// node as a root.
int dfs(vector<int> tree[N], int visit[N],
        int *ans, int node)
{
    int num = 0, temp = 0;

    // Mark node as visited.
    visit[node] = 1;

    // Traverse the adjacency list to find non-
    // visited node.
    for (int i = 0; i < tree[node].size(); i++)
    {
        if (visit[tree[node][i]] == 0)
        {
            // Finding number of nodes of the subtree
            // of a subtree.
            temp = dfs(tree, visit, ans, tree[node][i]);

            // If nodes are even, increment number of
            // edges to remove.
            // Else leave the node as child of subtree.
            (*temp%2)?(num += temp):((*ans)++);
        }
    }
}

return num+1;
}

// Return the maximum number of edge to remove
// to make forest.
```

```
int minEdge(vector<int> tree[N], int n)
{
    int visit[n+2];
    int ans = 0;
    memset(visit, 0, sizeof visit);

    dfs(tree, visit, &ans, 1);

    return ans;
}

// Driven Program
int main()
{
    int n = 10;

    vector<int> tree[n+2];
    tree[1].push_back(3);
    tree[3].push_back(1);

    tree[1].push_back(6);
    tree[6].push_back(1);

    tree[1].push_back(2);
    tree[2].push_back(1);

    tree[3].push_back(4);
    tree[4].push_back(3);

    tree[6].push_back(8);
    tree[8].push_back(6);

    tree[2].push_back(7);
    tree[7].push_back(2);

    tree[2].push_back(5);
    tree[5].push_back(2);

    tree[4].push_back(9);
    tree[9].push_back(4);

    tree[4].push_back(10);
    tree[10].push_back(4);

    cout << minEdge(tree, n) << endl;
    return 0;
}
```

Output:

2

**Time Complexity:**  $O(n)$ .

### Source

<https://www.geeksforgeeks.org/convert-tree-forest-even-nodes/>

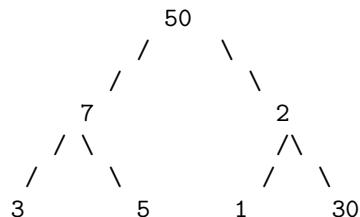
## Chapter 103

# Convert an arbitrary Binary Tree to a tree that holds Children Sum Property

Convert an arbitrary Binary Tree to a tree that holds Children Sum Property - Geeks-forGeeks

**Question:** Given an arbitrary binary tree, convert it to a binary tree that holds [Children Sum Property](#). You can only increment data values in any node (You cannot change the structure of the tree and cannot decrement the value of any node).

For example, the below tree doesn't hold the children sum property, convert it to a tree that holds the property.



### Algorithm:

Traverse the given tree in post order to convert it, i.e., first change left and right children to hold the children sum property then change the parent node.

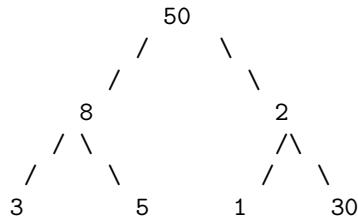
Let difference between node's data and children sum be diff.

```
diff = node's children sum - node's data
```

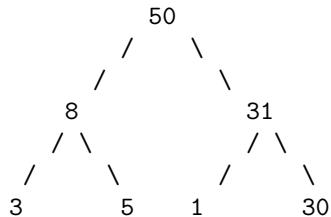
If diff is 0 then nothing needs to be done.

If diff > 0 ( node's data is smaller than node's children sum) increment the node's data by diff.

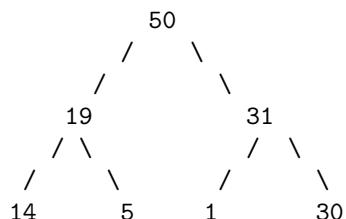
If diff < 0 (node's data is greater than the node's children sum) then increment one child's data. We can choose to increment either left or right child if they both are not NULL. Let us always first increment the left child. Incrementing a child changes the subtree's children sum property so we need to change left subtree also. So we recursively increment the left child. If left child is empty then we recursively call increment() for right child. Let us run the algorithm for the given example. First convert the left subtree (increment 7 to 8).



Then convert the right subtree (increment 2 to 31)



Now convert the root, we have to increment left subtree for converting the root.



Please note the last step – we have incremented 8 to 19, and to fix the subtree we have incremented 3 to 14.

#### **Implementation:**

C

```
/* Program to convert an arbitrary binary tree to
   a tree that holds children sum property */

#include <stdio.h>
#include <stdlib.h>

struct node
{
    int data;
    struct node* left;
    struct node* right;
};

/* This function is used to increment left subtree */
void increment(struct node* node, int diff);

/* Helper function that allocates a new node
   with the given data and NULL left and right
   pointers. */
struct node* newNode(int data);

/* This function changes a tree to hold children sum
   property */
void convertTree(struct node* node)
{
    int left_data = 0, right_data = 0, diff;

    /* If tree is empty or it's a leaf node then
       return true */
    if (node == NULL ||
        (node->left == NULL && node->right == NULL))
        return;
    else
    {
        /* convert left and right subtrees */
        convertTree(node->left);
        convertTree(node->right);

        /* If left child is not present then 0 is used
           as data of left child */
        if (node->left != NULL)
            left_data = node->left->data;

        /* If right child is not present then 0 is used
           as data of right child */
        if (node->right != NULL)
            right_data = node->right->data;

        /* Increment left subtree by right subtree */
        increment(node, right_data);
    }
}
```

```
/* get the diff of node's data and children sum */
diff = left_data + right_data - node->data;

/* If node's children sum is greater than the node's data */
if (diff > 0)
    node->data = node->data + diff;

/* THIS IS TRICKY --> If node's data is greater than children sum,
   then increment subtree by diff */
if (diff < 0)
    increment(node, -diff); // -diff is used to make diff positive
}

}

/* This function is used to increment subtree by diff */
void increment(struct node* node, int diff)
{
    /* IF left child is not NULL then increment it */
    if(node->left != NULL)
    {
        node->left->data = node->left->data + diff;

        // Recursively call to fix the descendants of node->left
        increment(node->left, diff);
    }
    else if (node->right != NULL) // Else increment right child
    {
        node->right->data = node->right->data + diff;

        // Recursively call to fix the descendants of node->right
        increment(node->right, diff);
    }
}

/* Given a binary tree, printInorder() prints out its
   inorder traversal*/
void printInorder(struct node* node)
{
    if (node == NULL)
        return;

    /* first recur on left child */
    printInorder(node->left);

    /* then print the data of node */
    printf("%d ", node->data);

    /* now recur on right child */
}
```

```
    printInorder(node->right);
}

/* Helper function that allocates a new node
   with the given data and NULL left and right
   pointers. */
struct node* newNode(int data)
{
    struct node* node =
        (struct node*)malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;
    return(node);
}

/* Driver program to test above functions */
int main()
{
    struct node *root = newNode(50);
    root->left      = newNode(7);
    root->right     = newNode(2);
    root->left->left  = newNode(3);
    root->left->right = newNode(5);
    root->right->left = newNode(1);
    root->right->right = newNode(30);

    printf("\n Inorder traversal before conversion ");
    printInorder(root);

    convertTree(root);

    printf("\n Inorder traversal after conversion ");
    printInorder(root);

    getchar();
    return 0;
}
```

**Java**

```
// Java program to convert an arbitrary binary tree to a tree that holds
// children sum property

// A binary tree node
class Node
{
    int data;
```

```
Node left, right;

Node(int item)
{
    data = item;
    left = right = null;
}
}

class BinaryTree
{
    Node root;
    /* This function changes a tree to hold children sum
       property */

    void convertTree(Node node)
    {
        int left_data = 0, right_data = 0, diff;

        /* If tree is empty or it's a leaf node then
           return true */
        if (node == null
            || (node.left == null && node.right == null))
            return;
        else
        {
            /* convert left and right subtrees */
            convertTree(node.left);
            convertTree(node.right);

            /* If left child is not present then 0 is used
               as data of left child */
            if (node.left != null)
                left_data = node.left.data;

            /* If right child is not present then 0 is used
               as data of right child */
            if (node.right != null)
                right_data = node.right.data;

            /* get the diff of node's data and children sum */
            diff = left_data + right_data - node.data;

            /* If node's children sum is greater than the node's data */
            if (diff > 0)
                node.data = node.data + diff;

            /* THIS IS TRICKY --> If node's data is greater than children
               data, then we have to swap them. This is done by calling
               swap() function */
            if (node.data > left_data && node.data > right_data)
                swap(left, right);
        }
    }
}
```

```
        sum, then increment subtree by diff */
if (diff < 0)

    // -diff is used to make diff positive
    increment(node, -diff);
}

/*
 * This function is used to increment subtree by diff */
void increment(Node node, int diff)
{
    /* IF left child is not NULL then increment it */
    if (node.left != null)
    {
        node.left.data = node.left.data + diff;

        // Recursively call to fix the descendants of node->left
        increment(node.left, diff);
    }
    else if (node.right != null) // Else increment right child
    {
        node.right.data = node.right.data + diff;

        // Recursively call to fix the descendants of node->right
        increment(node.right, diff);
    }
}

/*
 * Given a binary tree, printInorder() prints out its
 * inorder traversal*/
void printInorder(Node node)
{
    if (node == null)
        return;

    /* first recur on left child */
    printInorder(node.left);

    /* then print the data of node */
    System.out.print(node.data + " ");

    /* now recur on right child */
    printInorder(node.right);
}

// Driver program to test above functions
public static void main(String args[])
{
```

```
BinaryTree tree = new BinaryTree();
tree.root = new Node(50);
tree.root.left = new Node(7);
tree.root.right = new Node(2);
tree.root.left.left = new Node(3);
tree.root.left.right = new Node(5);
tree.root.right.left = new Node(1);
tree.root.right.right = new Node(30);

System.out.println("Inorder traversal before conversion is :");
tree.printInorder(tree.root);

tree.convertTree(tree.root);
System.out.println("");

System.out.println("Inorder traversal after conversion is :");
tree.printInorder(tree.root);

}

}

// This code has been contributed by Mayank Jaiswal(mayank_24)

Output :
Inorder traversal before conversion is :
3 7 5 50 1 2 30
Inorder traversal after conversion is :
14 19 5 50 1 31 30
```

**Time Complexity:**  $O(n^2)$ , Worst case complexity is for a skewed tree such that nodes are in decreasing order from root to leaf.

## Source

<https://www.geeksforgeeks.org/convert-an-arbitrary-binary-tree-to-a-tree-that-holds-children-sum-property/>

## Chapter 104

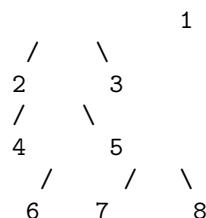
# Convert left-right representation of a binary tree to down-right

Convert left-right representation of a binary tree to down-right - GeeksforGeeks

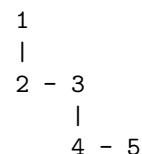
**Left-Right representation** of a binary tree is standard representation where every node has a pointer to left child and another pointer to right child.

**Down-Right representation** is an alternate representation where every node has a pointer to left (or first) child and another pointer to next sibling. So siblings at every level are connected from left to right.

Given a binary tree in left-right representation as below



Convert the structure of the tree to down-right representation like the below tree.



```
    |   |
  6   7 - 8
```

The conversion should happen in-place, i.e., left child pointer should be used as down pointer and right child pointer should be used as right sibling pointer.

**We strongly recommend to minimize your browser and try this yourself.**

The idea is to first convert left and right children, then convert the root. Following is C++ implementation of the idea.

```
/* C++ program to convert left-right to down-right
representation of binary tree */
#include <iostream>
#include <queue>
using namespace std;

// A Binary Tree Node
struct node
{
    int key;
    struct node *left, *right;
};

// An Iterative level order traversal based function to
// convert left-right to down-right representation.
void convert(node *root)
{
    // Base Case
    if (root == NULL)  return;

    // Recursively convert left an right subtrees
    convert(root->left);
    convert(root->right);

    // If left child is NULL, make right child as left
    // as it is the first child.
    if (root->left == NULL)
        root->left = root->right;

    // If left child is NOT NULL, then make right child
    // as right of left child
    else
        root->left->right = root->right;

    // Set root's right as NULL
    root->right = NULL;
}
```

```
// A utility function to traverse a tree stored in
// down-right form.
void downRightTraversal(node *root)
{
    if (root != NULL)
    {
        cout << root->key << " ";
        downRightTraversal(root->right);
        downRightTraversal(root->left);
    }
}

// Utility function to create a new tree node
node* newNode(int key)
{
    node *temp = new node;
    temp->key = key;
    temp->left = temp->right = NULL;
    return temp;
}

// Driver program to test above functions
int main()
{
    // Let us create binary tree shown in above diagram
    /*
        1
       / \
      2   3
         / \
        4   5
       /   / \
      6   7   8
    */
    node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->right->left = newNode(4);
    root->right->right = newNode(5);
    root->right->left->left = newNode(6);
    root->right->right->left = newNode(7);
    root->right->right->right = newNode(8);

    convert(root);

    cout << "Traversal of the tree converted to down-right form\n";
    downRightTraversal(root);
```

```
    return 0;  
}
```

Output:

```
Traversal of the tree converted to down-right form  
1 2 3 4 5 7 8 6
```

Time complexity of the above program is  $O(n)$ .

This article is contributed by **Abhishek**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## Source

<https://www.geeksforgeeks.org/convert-left-right-representation-binary-tree-right/>

## Chapter 105

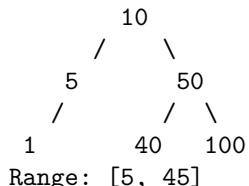
# Count BST nodes that lie in a given range

Count BST nodes that lie in a given range - GeeksforGeeks

Given a Binary Search Tree (BST) and a range, count number of nodes that lie in the given range.

Examples:

Input:



Range: [5, 45]

Output: 3

There are three nodes in range, 5, 10 and 40

The idea is to traverse the given binary search tree starting from root. For every node being visited, check if this node lies in range, if yes, then add 1 to result and recur for both of its children. If current node is smaller than low value of range, then recur for right child, else recur for left child.

Below is the implementation of above idea.

C++

```
// C++ program to count BST nodes within a given range
#include<bits/stdc++.h>
```

```
using namespace std;

// A BST node
struct node
{
    int data;
    struct node* left, *right;
};

// Utility function to create new node
node *newNode(int data)
{
    node *temp = new node;
    temp->data = data;
    temp->left = temp->right = NULL;
    return (temp);
}

// Returns count of nodes in BST in range [low, high]
int getCount(node *root, int low, int high)
{
    // Base case
    if (!root) return 0;

    // Special Optional case for improving efficiency
    if (root->data == high && root->data == low)
        return 1;

    // If current node is in range, then include it in count and
    // recur for left and right children of it
    if (root->data <= high && root->data >= low)
        return 1 + getCount(root->left, low, high) +
               getCount(root->right, low, high);

    // If current node is smaller than low, then recur for right
    // child
    else if (root->data < low)
        return getCount(root->right, low, high);

    // Else recur for left child
    else return getCount(root->left, low, high);
}

// Driver program
int main()
{
    // Let us construct the BST shown in the above figure
    node *root = newNode(10);
```

```
root->left      = newNode(5);
root->right     = newNode(50);
root->left->left  = newNode(1);
root->right->left = newNode(40);
root->right->right = newNode(100);
/* Let us constructed BST shown in above example
      10
     /   \
    5     50
   /   /   \
  1   40   100  */
int l = 5;
int h = 45;
cout << "Count of nodes between [" << l << ", " << h
     << "] is " << getCount(root, l, h);
return 0;
}
```

### Java

```
// Java code to count BST nodes that
// lie in a given range
class BinarySearchTree {

    /* Class containing left and right child
    of current node and key value*/
    static class Node {
        int data;
        Node left, right;

        public Node(int item) {
            data = item;
            left = right = null;
        }
    }

    // Root of BST
    Node root;

    // Constructor
    BinarySearchTree() {
        root = null;
    }

    // Returns count of nodes in BST in
    // range [low, high]
    int getCount(Node node, int low, int high)
    {
```

```

// Base Case
if(node == null)
    return 0;

// If current node is in range, then
// include it in count and recur for
// left and right children of it
if(node.data >= low && node.data <= high)
    return 1 + this.getCount(node.left, low, high) +
        this.getCount(node.right, low, high);

// If current node is smaller than low,
// then recur for right child
else if(node.data < low)
    return this.getCount(node.right, low, high);

// Else recur for left child
else
    return this.getCount(node.left, low, high);
}

// Driver function
public static void main(String[] args) {
    BinarySearchTree tree = new BinarySearchTree();

    tree.root = new Node(10);
    tree.root.left      = new Node(5);
    tree.root.right     = new Node(50);
    tree.root.left.left = new Node(1);
    tree.root.right.left = new Node(40);
    tree.root.right.right = new Node(100);
    /* Let us constructed BST shown in above example
       10
      /   \
     5     50
    /   /   \
   1   40   100 */
    int l=5;
    int h=45;
    System.out.println("Count of nodes between [" + l + ", "
                       + h + "] is " + tree.getCount(tree.root,
                                         l, h));
}
}

// This code is contributed by Kamal Rawal

```

Output:

Count of nodes between [5, 45] is 3

Time complexity of the above program is  $O(h + k)$  where h is height of BST and k is number of nodes in given range.

This article is contributed by [Gaurav Ahirwar](#). Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## Source

<https://www.geeksforgeeks.org/count-bst-nodes-that-are-in-a-given-range/>

## Chapter 106

# Count Balanced Binary Trees of Height h

Count Balanced Binary Trees of Height h - GeeksforGeeks

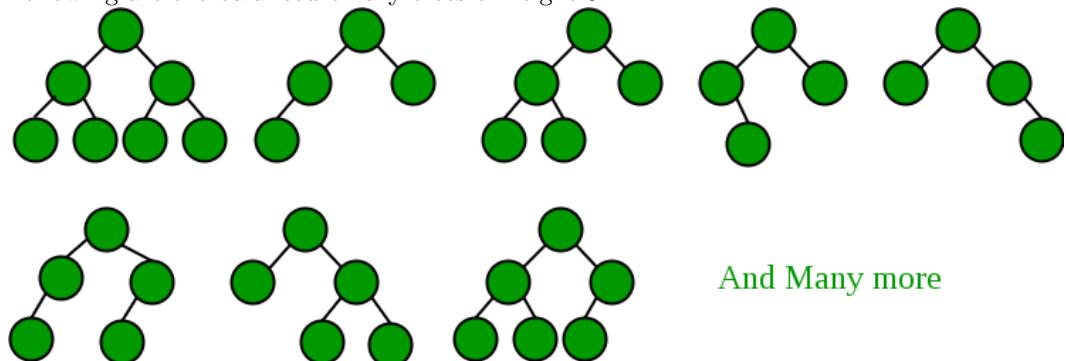
Given a height h, count and return the maximum number of balanced binary trees possible with height h. A balanced binary tree is one in which for every node, the difference between heights of left and right subtree is not more than 1.

**Examples :**

Input : h = 3  
Output : 15

Input : h = 4  
Output : 315

Following are the balanced binary trees of height 3.



Height of tree,  $h = 1 + \max(\text{left height}, \text{right height})$

Since the difference between the heights of left and right subtree is not more than one, possible heights of left and right part can be one of the following:

1. (h-1), (h-2)
2. (h-2), (h-1)
3. (h-1), (h-1)

```

count(h) = count(h-1) * count(h-2) +
          count(h-2) * count(h-1) +
          count(h-1) * count(h-1)
      = 2 * count(h-1) * count(h-2) +
          count(h-1) * count(h-1)
      = count(h-1) * (2*count(h - 2) +
          count(h - 1))

```

Hence we can see that the problem has optimal substructure property.

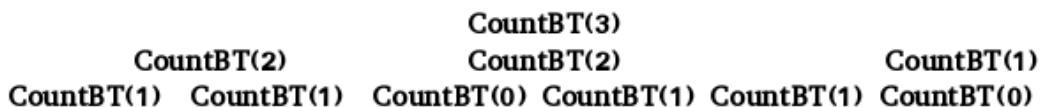
A **recursive function** to count no of balanced binary trees of height h is:

```

int countBT(int h)
{
    // One tree is possible with height 0 or 1
    if (h == 0 || h == 1)
        return 1;
    return countBT(h-1) * (2 * countBT(h-2) +
                           countBT(h-1));
}

```

The time complexity of this recursive approach will be exponential. The recursion tree for the problem with  $h = 3$  looks like :



As we can see, sub-problems are solved repeatedly. Therefore we store the results as we compute them.

An efficient dynamic programming approach will be as follows :

**CPP**

```

// C++ program to count number of balanced
// binary trees of height h.
#include <bits/stdc++.h>
#define mod 1000000007
using namespace std;

```

```

long long int countBT(int h) {

    long long int dp[h + 1];
    //base cases
    dp[0] = dp[1] = 1;
    for(int i = 2; i <= h; i++) {
        dp[i] = (dp[i - 1] * ((2 * dp[i - 2])%mod + dp[i - 1])%mod) % mod;
    }
    return dp[h];
}

// Driver program
int main()
{
    int h = 3;
    cout << "No. of balanced binary trees"
        " of height h is: "
        << countBT(h) << endl;
}

```

### PHP

```

<?php
// PHP program to count
// number of balanced

$mod =1000000007;

function countBT($h)
{
    global $mod;

    // base cases
    $dp[0] = $dp[1] = 1;
    for($i = 2; $i <= $h; $i++)
    {
        $dp[$i] = ($dp[$i - 1] *
                    ((2 * $dp[$i - 2]) %
                     $mod + $dp[$i - 1]) %
                     $mod) % $mod;
    }
    return $dp[$h];
}

// Driver Code
$h = 3;

```

```
echo "No. of balanced binary trees",
      " of height h is: ",
      countBT($h) ,"\n";

// This code is contributed by aj_36
?>
```

**Output :**

No of balanced binary trees of height h is: 15

**Improved By :** [Anuj\\_Sharma, jit\\_t](#)

**Source**

<https://www.geeksforgeeks.org/count-balanced-binary-trees-height-h/>

## Chapter 107

# Count Non-Leaf nodes in a Binary Tree

Count Non-Leaf nodes in a Binary Tree - GeeksforGeeks

Given a Binary tree, count total number of non-leaf nodes in the tree

Examples:

Input :

Output :2

Explanation

In the above tree only two nodes 1 and 2 are non-leaf nodes

We recursively traverse the given tree. While traversing, we count non-leaf nodes in left and right subtrees and add 1 to the result.

```
// CPP program to count total number of
// non-leaf nodes in a binary tree
#include <bits/stdc++.h>
using namespace std;

/* A binary tree node has data, pointer to
   left child and a pointer to right child */
struct Node {
    int data;
    struct Node* left;
    struct Node* right;
};


```

```
/* Helper function that allocates a new node with the
given data and NULL left and right pointers. */
struct Node* newNode(int data)
{
    struct Node* node = new Node;
    node->data = data;
    node->left = node->right = NULL;
    return (node);
}

/* Computes the number of non-leaf nodes in a tree. */
int countNonleaf(struct Node* root)
{
    // Base cases.
    if (root == NULL || (root->left == NULL &&
                          root->right == NULL))
        return 0;

    // If root is Not NULL and its one of its
    // child is also not NULL
    return 1 + countNonleaf(root->left) +
           countNonleaf(root->right);
}

/* Driver program to test size function*/
int main()
{
    struct Node* root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    cout << countNonleaf(root);
    return 0;
}
```

Output:

2

## Source

<https://www.geeksforgeeks.org/count-non-leaf-nodes-binary-tree/>

## Chapter 108

# Count elements which divide all numbers in range L-R

Count elements which divide all numbers in range L-R - GeeksforGeeks

Given N numbers and Q queries, each query consists of L and R. Task is to write a program which prints the count of numbers which divides all numbers in the given range L-R.

Examples :

```
Input : a = {3, 4, 2, 2, 4, 6}
        Q = 2
        L = 1 R = 4
        L = 2 R = 6

Output : 0
        2
```

Explanation : The range 1-4 has {3, 4, 2, 2} which does not have any number that divides all the numbers in this range.

The range 2-6 has {4, 2, 2, 4, 6} which has 2 numbers {2, 2} which divides all numbers in the given range.

```
Input: a = {1, 2, 3, 5}
      Q = 2
      L = 1 R = 4
      L = 2 R = 4
Output: 1
      0
```

**Naive approach :** Iterate from range L-R for every query and check if the given element

at index-i divide all the numbers in the range. We keep a count for of all the elements which divides all the numbers. The complexity of every query at worst case will be  $\mathbf{O(n^2)}$ .

**Below is the implementation of Naive Approach :**

```
// CPP program to Count elements which
// divides all numbers in range L-R
#include <bits/stdc++.h>
using namespace std;

// function to count element
// Time complexity O(n^2) worst case
int answerQuery(int a[], int n,
                 int l, int r)
{
    // answer for query
    int count = 0;

    // 0 based index
    l = l - 1;

    // iterate for all elements
    for (int i = l; i < r; i++)
    {
        int element = a[i];
        int divisors = 0;

        // check if the element divides
        // all numbers in range
        for (int j = l; j < r; j++)
        {
            // no of elements
            if (a[j] % a[i] == 0)
                divisors++;
            else
                break;
        }

        // if all elements are divisible by a[i]
        if (divisors == (r - l))
            count++;
    }

    // answer for every query
    return count;
}

// Driver Code
int main()
```

```
{  
    int a[] = { 1, 2, 3, 5 };  
    int n = sizeof(a) / sizeof(a[0]);  
  
    int l = 1, r = 4;  
    cout << answerQuery(a, n, l, r) << endl;  
  
    l = 2, r = 4;  
    cout << answerQuery(a, n, l, r) << endl;  
    return 0;  
}
```

**Output:**

```
1  
0
```

**Efficient approach :** Use [Segment Trees](#) to solve this problem. If an element divides all the numbers in a given range, then the element is the minimum number in that range and it is the gcd of all elements in the given range L-R. So the count of the number of minimums in range L-R, given that minimum is equal to the gcd of that range will be our answer to every query. The problem boils down to finding the [GCD](#), [MINIMUM](#) and [countMINIMUM](#) for every range using Segment trees. On every node of the tree, three values are stored.

On querying for a given range, if the gcd and minimum of the given range are equal, [count-MINIMUM](#) is returned as the answer. If they are unequal, 0 is returned as the answer.

**Below is the implementation of efficient approach :**

```
// CPP program to Count elements  
// which divides all numbers in  
// range L-R efficient approach  
#include <bits/stdc++.h>  
using namespace std;  
  
#define N 100005  
  
// predefines the tree with nodes  
// storing gcd, min and count  
struct node  
{  
    int gcd;  
    int min;  
    int cnt;  
} tree[5 * N];
```

```
// function to construct the tree
void buildtree(int low, int high,
               int pos, int a[])
{
    // base condition
    if (low == high)
    {
        // initially always gcd and min
        // are same at leaf node
        tree[pos].min = tree[pos].gcd = a[low];
        tree[pos].cnt = 1;

        return;
    }

    int mid = (low + high) >> 1;

    // left-subtree
    buildtree(low, mid, 2 * pos + 1, a);

    // right-subtree
    buildtree(mid + 1, high, 2 * pos + 2, a);

    // finds gcd of left and right subtree
    tree[pos].gcd = __gcd(tree[2 * pos + 1].gcd,
                          tree[2 * pos + 2].gcd);

    // left subtree has the minimum element
    if (tree[2 * pos + 1].min < tree[2 * pos + 2].min)
    {
        tree[pos].min = tree[2 * pos + 1].min;
        tree[pos].cnt = tree[2 * pos + 1].cnt;
    }

    // right subtree has the minimum element
    else
    if (tree[2 * pos + 1].min > tree[2 * pos + 2].min)
    {
        tree[pos].min = tree[2 * pos + 2].min;
        tree[pos].cnt = tree[2 * pos + 2].cnt;
    }

    // both subtree has the same minimum element
    else
    {
        tree[pos].min = tree[2 * pos + 1].min;
        tree[pos].cnt = tree[2 * pos + 1].cnt +
```

```
        tree[2 * pos + 2].cnt;
    }
}

// function that answers every query
node query(int s, int e, int low, int high, int pos)
{
    node dummy;

    // out of range
    if (e < low or s > high)
    {
        dummy.gcd = dummy.min = dummy.cnt = 0;
        return dummy;
    }

    // in range
    if (s >= low and e <= high)
    {
        node dummy;
        dummy.gcd = tree[pos].gcd;
        dummy.min = tree[pos].min;
        if (dummy.gcd != dummy.min)
            dummy.cnt = 0;
        else
            dummy.cnt = tree[pos].cnt;

        return dummy;
    }

    int mid = (s + e) >> 1;

    // left-subtree
    node ans1 = query(s, mid, low,
                      high, 2 * pos + 1);

    // right-subtree
    node ans2 = query(mid + 1, e, low,
                      high, 2 * pos + 2);

    node ans;

    // when both left subtree and
    // right subtree is in range
    if (ans1.gcd and ans2.gcd)
    {
        // merge two trees
        ans.gcd = __gcd(ans1.gcd, ans2.gcd);
        ans.min = min(ans1.min, ans2.min);
        ans.cnt = ans1.cnt + ans2.cnt;
    }
}
```

```
ans.min = min(ans1.min, ans2.min);

// when gcd is not equal to min
if (ans.gcd != ans.min)
    ans.cnt = 0;
else
{
    // add count when min is
    // same of both subtree
    if (ans1.min == ans2.min)
        ans.cnt = ans2.cnt + ans1.cnt;

    // store the minimal's count
    else
        if (ans1.min < ans2.min)
            ans.cnt = ans1.cnt;
        else
            ans.cnt = ans2.cnt;
}

return ans;
}

// only left subtree is in range
else if (ans1.gcd)
    return ans1;

// only right subtree is in range
else if (ans2.gcd)
    return ans2;
}

// function to answer query in range l-r
int answerQuery(int a[], int n, int l, int r)
{
    // calls the function which returns
    // a node this function returns the
    // count which will be the answer
    return query(0, n - 1, l - 1, r - 1, 0).cnt;
}

// Driver Code
int main()
{
    int a[] = { 3, 4, 2, 2, 4, 6 };

    int n = sizeof(a) / sizeof(a[0]);
    buildtree(0, n - 1, 0, a);
```

```
int l = 1, r = 4;

// answers 1-st query
cout << answerQuery(a, n, l, r) << endl;

l = 2, r = 6;
// answers 2nd query
cout << answerQuery(a, n, l, r) << endl;
return 0;
}
```

**Output:**

```
0
2
```

**Time Complexity:** Time Complexity for tree construction is **O(n logn)** since tree construction takes O(n) and finding out gcd takes O(log n). The time taken for every query in worst case will be **O(log n \* log n)** since the inbuilt function `__gcd` takes **O(log n)**

**Source**

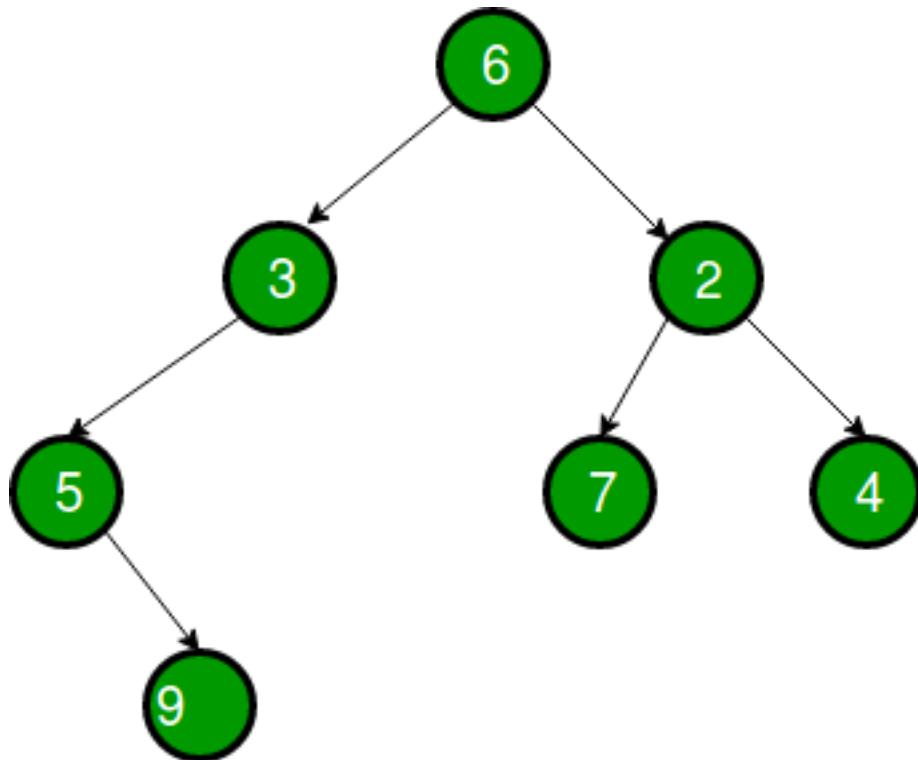
<https://www.geeksforgeeks.org/count-elements-which-divide-all-numbers-in-range-l-r/>

## Chapter 109

# Count full nodes in a Binary tree (Iterative and Recursive)

Count full nodes in a Binary tree (Iterative and Recursive) - GeeksforGeeks

Given A binary Tree, how do you count all the full nodes (Nodes which have both children as not NULL) without using recursion and with recursion? Note leaves should not be touched as they have both children as NULL.



Nodes 2 and 6 are full nodes has both child's. So count of full nodes in the above tree is 2

**Method: Iterative**

The idea is to use level-order traversal to solve this problem efficiently.

- 1) Create an empty Queue Node and push root node to Queue.
- 2) Do following while nodeQueue is not empty.
  - a) Pop an item from Queue and process it.
    - a.1) If it is full node then increment count++.
  - b) Push left child of popped item to Queue, if available.
  - c) Push right child of popped item to Queue, if available.

Below is the implementation of this idea.

C++

```
// C++ program to count
// full nodes in a Binary Tree
#include <bits/stdc++.h>
using namespace std;

// A binary tree Node has data, pointer to left
// child and a pointer to right child
struct Node
{
    int data;
    struct Node* left, *right;
};

// Function to get the count of full Nodes in
// a binary tree
unsigned int getfullCount(struct Node* node)
{
    // If tree is empty
    if (!node)
        return 0;
    queue<Node *> q;

    // Do level order traversal starting from root
    int count = 0; // Initialize count of full nodes
    q.push(node);
    while (!q.empty())
    {
        struct Node *temp = q.front();
        q.pop();

        if (temp->left && temp->right)
```

```
        count++;

        if (temp->left != NULL)
            q.push(temp->left);
        if (temp->right != NULL)
            q.push(temp->right);
    }
    return count;
}

/* Helper function that allocates a new Node with the
given data and NULL left and right pointers. */
struct Node* newNode(int data)
{
    struct Node* node = new Node;
    node->data = data;
    node->left = node->right = NULL;
    return (node);
}

// Driver program
int main(void)
{
    /*      2
         / \
        7     5
       / \   \
      6   9
     / \ /
    1 11 4
    Let us create Binary Tree as shown
    */

    struct Node *root = newNode(2);
    root->left      = newNode(7);
    root->right     = newNode(5);
    root->left->right = newNode(6);
    root->left->right->left = newNode(1);
    root->left->right->right = newNode(11);
    root->right->right = newNode(9);
    root->right->right->left = newNode(4);

    cout << getfullCount(root);

    return 0;
}
```

Java

```
// Java program to count
// full nodes in a Binary Tree
// using Iterative approach
import java.util.Queue;
import java.util.LinkedList;

// Class to represent Tree node
class Node
{
    int data;
    Node left, right;

    public Node(int item)
    {
        data = item;
        left = null;
        right = null;
    }
}

// Class to count full nodes of Tree
class BinaryTree
{

    Node root;

    /* Function to get the count of full Nodes in
     a binary tree*/
    int getfullCount()
    {
        // If tree is empty
        if (root==null)
            return 0;

        // Initialize empty queue.
        Queue<Node> queue = new LinkedList<Node>();

        // Do level order traversal starting from root
        queue.add(root);

        int count=0; // Initialize count of full nodes
        while (!queue.isEmpty())
        {

            Node temp = queue.poll();
            if (temp.left!=null && temp.right!=null)
                count++;
        }
    }
}
```

```
// Enqueue left child
if (temp.left != null)
{
    queue.add(temp.left);
}

// Enqueue right child
if (temp.right != null)
{
    queue.add(temp.right);
}
}
return count;
}

public static void main(String args[])
{
    /*      2
           / \
          7     5
         /   \
        6     9
       / \ /
      1 11 4
Let us create Binary Tree as shown
*/
    BinaryTree tree_level = new BinaryTree();
    tree_level.root = new Node(2);
    tree_level.root.left = new Node(7);
    tree_level.root.right = new Node(5);
    tree_level.root.left.right = new Node(6);
    tree_level.root.left.right.left = new Node(1);
    tree_level.root.left.right.right = new Node(11);
    tree_level.root.right.right = new Node(9);
    tree_level.root.right.right.left = new Node(4);

    System.out.println(tree_level.getfullCount());
}

}
```

### Python

```
# Python program to count
# full nodes in a Binary Tree
# using iterative approach
```

```
# A node structure
class Node:
    # A utility function to create a new node
    def __init__(self ,key):
        self.data = key
        self.left = None
        self.right = None

# Iterative Method to count full nodes of binary tree
def getfullCount(root):
    # Base Case
    if root is None:
        return 0

    # Create an empty queue for level order traversal
    queue = []

    # Enqueue Root and initialize count
    queue.append(root)

    count = 0 #initialize count for full nodes
    while(len(queue) > 0):
        node = queue.pop(0)

        # if it is full node then increment count
        if node.left is not None and node.right is not None:
            count = count+1

        # Enqueue left child
        if node.left is not None:
            queue.append(node.left)

        # Enqueue right child
        if node.right is not None:
            queue.append(node.right)

    return count

# Driver Program to test above function
root = Node(2)
root.left = Node(7)
root.right = Node(5)
root.left.right = Node(6)
root.left.right.left = Node(1)
root.left.right.right = Node(11)
root.right.right = Node(9)
root.right.right.left = Node(4)
```

```
print "%d" %(getfullCount(root))
```

Output:

2

**Time Complexity:** O(n)

**Auxiliary Space :** O(n) where, n is number of nodes in given binary tree

**Method: Recursive**

The idea is to traverse the tree in postorder. If the current node is full, we increment result by 1 and add returned values of left and right subtrees.

```
// C++ program to count full nodes in a Binary Tree
#include <bits/stdc++.h>
using namespace std;

// A binary tree Node has data, pointer to left
// child and a pointer to right child
struct Node
{
    int data;
    struct Node* left, *right;
};

// Function to get the count of full Nodes in
// a binary tree
unsigned int getfullCount(struct Node* root)
{
    if (root == NULL)
        return 0;

    int res = 0;
    if (root->left && root->right)
        res++;

    res += (getfullCount(root->left) +
            getfullCount(root->right));
    return res;
}

/* Helper function that allocates a new
   Node with the given data and NULL left
   and right pointers. */
struct Node* newNode(int data)
```

```
{  
    struct Node* node = new Node;  
    node->data = data;  
    node->left = node->right = NULL;  
    return (node);  
}  
  
// Driver program  
int main(void)  
{  
    /* 2  
     / \  
    7   5  
   \   \  
   6   9  
  / \ /  
 1 11 4  
Let us create Binary Tree as shown  
*/  
  
    struct Node *root = newNode(2);  
    root->left = newNode(7);  
    root->right = newNode(5);  
    root->left->right = newNode(6);  
    root->left->right->left = newNode(1);  
    root->left->right->right = newNode(11);  
    root->right->right = newNode(9);  
    root->right->right->left = newNode(4);  
  
    cout << getfullCount(root);  
  
    return 0;  
}  
  
Output:
```

2

**Time Complexity:** O(n)

**Auxiliary Space :** O(n)

where, n is number of nodes in given binary tree

**Similar Articles:**

- [Count half nodes in a Binary tree \(Iterative and Recursive\)](#)
- [Program to get count of leaf nodes in Binary Tree](#)
- [Given a binary tree, how do you remove all the half nodes?](#)
- [Print all full nodes in a Binary Tree](#)

**Source**

<https://www.geeksforgeeks.org/count-full-nodes-binary-tree-iterative-recursive/>

## Chapter 110

# Count half nodes in a Binary tree (Iterative and Recursive)

Count half nodes in a Binary tree (Iterative and Recursive) - GeeksforGeeks

Given A binary Tree, how do you count all the half nodes (which has only one child) without using recursion? Note leaves should not be touched as they have both children as NULL.

Input : Root of below tree

Output : 3  
Nodes 7, 5 and 9 are half nodes as one of their child is Null. So count of half nodes in the above tree is 3

### Iterative

The idea is to use level-order traversal to solve this problem efficiently.

- 1) Create an empty Queue Node and push root node to Queue.
- 2) Do following while nodeQueue is not empty.
  - a) Pop an item from Queue and process it.
    - a.1) If it is half node then increment count++.
    - b) Push left child of popped item to Queue, if available.
    - c) Push right child of popped item to Queue, if available.

Below is the implementation of this idea.

C++

```
// C++ program to count half nodes in a Binary Tree
#include <bits/stdc++.h>
using namespace std;

// A binary tree Node has data, pointer to left
// child and a pointer to right child
struct Node
{
    int data;
    struct Node* left, *right;
};

// Function to get the count of half Nodes in
// a binary tree
unsigned int gethalfCount(struct Node* node)
{
    // If tree is empty
    if (!node)
        return 0;

    int count = 0; // Initialize count of half nodes

    // Do level order traversal starting from root
    queue<Node *> q;
    q.push(node);
    while (!q.empty())
    {
        struct Node *temp = q.front();
        q.pop();

        if (!temp->left && temp->right ||
            temp->left && !temp->right)
            count++;

        if (temp->left != NULL)
            q.push(temp->left);
        if (temp->right != NULL)
            q.push(temp->right);
    }
    return count;
}

/* Helper function that allocates a new
   Node with the given data and NULL left
   and right pointers. */
struct Node* newNode(int data)
{
    struct Node* node = new Node;
```

```
node->data = data;
node->left = node->right = NULL;
return (node);
}

// Driver program
int main(void)
{
    /* 2
     / \
    7   5
    \   \
    6   9
   / \ /
  1 11 4
Let us create Binary Tree shown in
above example */

struct Node *root = newNode(2);
root->left      = newNode(7);
root->right     = newNode(5);
root->left->right = newNode(6);
root->left->right->left = newNode(1);
root->left->right->right = newNode(11);
root->right->right = newNode(9);
root->right->right->left = newNode(4);

cout << gethalfCount(root);

return 0;
}
```

### Java

```
// Java program to count half nodes in a Binary Tree
// using Iterative approach
import java.util.Queue;
import java.util.LinkedList;

// Class to represent Tree node
class Node
{
    int data;
    Node left, right;

    public Node(int item)
    {
        data = item;
```

```
        left = null;
        right = null;
    }
}

// Class to count half nodes of Tree
class BinaryTree
{

    Node root;

    /* Function to get the count of half Nodes in
     a binary tree*/
    int gethalfCount()
    {
        // If tree is empty
        if (root==null)
            return 0;

        // Do level order traversal starting from root
        Queue<Node> queue = new LinkedList<Node>();
        queue.add(root);

        int count=0; // Initialize count of half nodes
        while (!queue.isEmpty())
        {

            Node temp = queue.poll();
            if (temp.left!=null && temp.right==null ||
                temp.left==null && temp.right!=null)
                count++;

            // Enqueue left child
            if (temp.left != null)
                queue.add(temp.left);

            // Enqueue right child
            if (temp.right != null)
                queue.add(temp.right);
        }
        return count;
    }

    public static void main(String args[])
    {
        /* 2
           / \
          7   5
        */
    }
}
```

7        5

```
\      \
6      9
/ \ /
1 11 4
Let us create Binary Tree shown in
above example */
BinaryTree tree_level = new BinaryTree();
tree_level.root = new Node(2);
tree_level.root.left = new Node(7);
tree_level.root.right = new Node(5);
tree_level.root.left.right = new Node(6);
tree_level.root.left.right.left = new Node(1);
tree_level.root.left.right.right = new Node(11);
tree_level.root.right.right = new Node(9);
tree_level.root.right.right.left = new Node(4);

System.out.println(tree_level.gethalfCount());

}
}
```

### Python

```
# Python program to count
# half nodes in a Binary Tree
# using iterative approach

# A node structure
class Node:

    # A utility function to create a new node
    def __init__(self ,key):
        self.data = key
        self.left = None
        self.right = None

# Iterative Method to count half nodes of binary tree
def gethalfCount(root):

    # Base Case
    if root is None:
        return 0

    # Create an empty queue for level order traversal
    queue = []

    # Enqueue Root and initialize count
```

```
queue.append(root)

count = 0 #initialize count for half nodes
while(len(queue) > 0):

    node = queue.pop(0)

    # if it is half node then increment count
    if node.left is not None and node.right is None or node.left is None and node.right is not None:
        count = count+1

    #Enqueue left child
    if node.left is not None:
        queue.append(node.left)

    # Enqueue right child
    if node.right is not None:
        queue.append(node.right)

return count

#Driver Program to test above function

root = Node(2)
root.left = Node(7)
root.right = Node(5)
root.left.right = Node(6)
root.left.right.left = Node(1)
root.left.right.right = Node(11)
root.right.right = Node(9)
root.right.right.left = Node(4)

print "%d" %(gethalfCount(root))
```

Output:

3

**Time Complexity:** O(n)

**Auxiliary Space:** O(n)

where, n is number of nodes in given binary tree

### Recursive

The idea is to traverse the tree in postorder. If current node is half, we increment result by 1 and add returned values of left and right subtrees.

```
// C++ program to count half nodes in a Binary Tree
#include <bits/stdc++.h>
using namespace std;

// A binary tree Node has data, pointer to left
// child and a pointer to right child
struct Node
{
    int data;
    struct Node* left, *right;
};

// Function to get the count of half Nodes in
// a binary tree
unsigned int gethalfCount(struct Node* root)
{
    if (root == NULL)
        return 0;

    int res = 0;
    if ((root->left == NULL && root->right != NULL) ||
        (root->left != NULL && root->right == NULL))
        res++;

    res += (gethalfCount(root->left) + gethalfCount(root->right));
    return res;
}

/* Helper function that allocates a new
   Node with the given data and NULL left
   and right pointers. */
struct Node* newNode(int data)
{
    struct Node* node = new Node;
    node->data = data;
    node->left = node->right = NULL;
    return (node);
}

// Driver program
int main(void)
{
    /* 2
       / \
      7     5
     \     \
      6     9
     / \ /
    */
}
```

1 11 4

Let us create Binary Tree shown in  
above example \*/

```
struct Node *root = newNode(2);
root->left      = newNode(7);
root->right     = newNode(5);
root->left->right = newNode(6);
root->left->right->left = newNode(1);
root->left->right->right = newNode(11);
root->right->right = newNode(9);
root->right->right->left = newNode(4);

cout << gethalfCount(root);

return 0;
}
```

Output :

3

**Time Complexity:** O(n)

**Auxiliary Space:** O(n)

where, n is number of nodes in given binary tree

#### Similar Articles:

- Program to get count of leaf nodes in Binary Tree
- Given a binary tree, how do you remove all the half nodes?

#### Source

<https://www.geeksforgeeks.org/count-half-nodes-in-a-binary-tree-iterative-and-recursive/>

## Chapter 111

# Count pairs in a binary tree whose sum is equal to a given value x

Count pairs in a binary tree whose sum is equal to a given value x - GeeksforGeeks

Given a binary tree containing **n** distinct numbers and a value **x**. The problem is to count pairs in the given binary tree whose sum is equal to the given value **x**.

Examples:

Input :

```
      5
     / \
    3   7
   / \ / \
  2  4 6  8
```

**x = 10**

Output : 3

The pairs are (3, 7), (2, 8) and (4, 6).

**Naive Approach:** One by one get each node of the binary tree through any of the tree traversals method. Pass the node say **temp**, the **root** of the tree and value **x** to another function say **findPair()**. In the function with the help of the **root** pointer traverse the tree again. One by one sum up these nodes with **temp** and check whether **sum == x**. If so, increment **count**. Calculate **count = count / 2** as a single pair has been counted twice by the aforementioned method.

```
// C++ implementation to count pairs in a binary tree
```

```
// whose sum is equal to given value x
#include <bits/stdc++.h>
using namespace std;

// structure of a node of a binary tree
struct Node {
    int data;
    Node *left, *right;
};

// function to create and return a node
// of a binary tree
Node* getNode(int data)
{
    // allocate space for the node
    Node* new_node = (Node*)malloc(sizeof(Node));

    // put in the data
    new_node->data = data;
    new_node->left = new_node->right = NULL;
}

// returns true if a pair exists with given sum 'x'
bool findPair(Node* root, Node* temp, int x)
{
    // base case
    if (!root)
        return false;

    // pair exists
    if (root != temp && ((root->data + temp->data) == x))
        return true;

    // find pair in left and right subtress
    if (findPair(root->left, temp, x) || findPair(root->right, temp, x))
        return true;

    // pair does not exists with given sum 'x'
    return false;
}

// function to count pairs in a binary tree
// whose sum is equal to given value x
void countPairs(Node* root, Node* curr, int x, int& count)
{
    // if tree is empty
    if (!curr)
        return;
```

```
// check whether pair exists for current node 'curr'  
// in the binary tree that sum up to 'x'  
if (findPair(root, curr, x))  
    count++;  
  
// recursively count pairs in left subtree  
countPairs(root, curr->left, x, count);  
  
// recursively count pairs in right subtree  
countPairs(root, curr->right, x, count);  
}  
  
// Driver program to test above  
int main()  
{  
    // formation of binary tree  
    Node* root = getNode(5); /*      5      */  
    root->left = getNode(3); /*      / \      */  
    root->right = getNode(7); /*      3  7      */  
    root->left->left = getNode(2); /*      / \ / \      */  
    root->left->right = getNode(4); /*      2 4 6 8      */  
    root->right->left = getNode(6);  
    root->right->right = getNode(8);  
  
    int x = 10;  
    int count = 0;  
  
    countPairs(root, root, x, count);  
    count = count / 2;  
  
    cout << "Count = " << count;  
  
    return 0;  
}
```

Output:

Count = 3

Time Complexity:  $O(n^2)$ .

**Efficient Approach:** Following are the steps:

1. Convert given binary tree to doubly linked list. Refer [this](#) post.
2. Sort the doubly linked list obtained in Step 1. Refer [this](#) post.
3. Count Pairs in sorted doubly linked with sum equal to 'x'. Refer [this](#) post.

4. Display the count obtained in Step 4.

```
// C++ implementation to count pairs in a binary tree
// whose sum is equal to given value x
#include <bits/stdc++.h>
using namespace std;

// structure of a node of a binary tree
struct Node {
    int data;
    Node *left, *right;
};

// function to create and return a node
// of a binary tree
Node* getNode(int data)
{
    // allocate space for the node
    Node* new_node = (Node*)malloc(sizeof(Node));

    // put in the data
    new_node->data = data;
    new_node->left = new_node->right = NULL;
}

// A simple recursive function to convert a given
// Binary tree to Doubly Linked List
// root      --> Root of Binary Tree
// head_ref --> Pointer to head node of created
// doubly linked list
void BToDLL(Node* root, Node** head_ref)
{
    // Base cases
    if (root == NULL)
        return;

    // Recursively convert right subtree
    BToDLL(root->right, head_ref);

    // insert root into DLL
    root->right = *head_ref;

    // Change left pointer of previous head
    if (*head_ref != NULL)
        (*head_ref)->left = root;

    // Change head of Doubly linked list
    *head_ref = root;
}
```

```
// Recursively convert left subtree
BToDLL(root->left, head_ref);
}

// Split a doubly linked list (DLL) into 2 DLLs of
// half sizes
Node* split(Node* head)
{
    Node *fast = head, *slow = head;
    while (fast->right && fast->right->right) {
        fast = fast->right->right;
        slow = slow->right;
    }
    Node* temp = slow->right;
    slow->right = NULL;
    return temp;
}

// Function to merge two sorted doubly linked lists
Node* merge(Node* first, Node* second)
{
    // If first linked list is empty
    if (!first)
        return second;

    // If second linked list is empty
    if (!second)
        return first;

    // Pick the smaller value
    if (first->data < second->data) {
        first->right = merge(first->right, second);
        first->right->left = first;
        first->left = NULL;
        return first;
    }
    else {
        second->right = merge(first, second->right);
        second->right->left = second;
        second->left = NULL;
        return second;
    }
}

// Function to do merge sort
Node* mergeSort(Node* head)
{
```

```
if (!head || !head->right)
    return head;
Node* second = split(head);

// Recur for left and right halves
head = mergeSort(head);
second = mergeSort(second);

// Merge the two sorted halves
return merge(head, second);
}

// Function to count pairs in a sorted doubly linked list
// whose sum equal to given value x
int pairSum(Node* head, int x)
{
    // Set two pointers, first to the beginning of DLL
    // and second to the end of DLL.
    Node* first = head;
    Node* second = head;
    while (second->right != NULL)
        second = second->right;

    int count = 0;

    // The loop terminates when either of two pointers
    // become NULL, or they cross each other (second->right
    // == first), or they become same (first == second)
    while (first != NULL && second != NULL && first != second && second->right != first) {
        // pair found
        if ((first->data + second->data) == x) {
            count++;

            // move first in forward direction
            first = first->right;

            // move second in backward direction
            second = second->left;
        }
        else {
            if ((first->data + second->data) < x)
                first = first->right;
            else
                second = second->left;
        }
    }

    return count;
}
```

```
}

// function to count pairs in a binary tree
// whose sum is equal to given value x
int countPairs(Node* root, int x)
{
    Node* head = NULL;
    int count = 0;

    // Convert binary tree to
    // doubly linked list
    BToDLL(root, &head);

    // sort DLL
    head = mergeSort(head);

    // count pairs
    return pairSum(head, x);
}

// Driver program to test above
int main()
{
    // formation of binary tree
    Node* root = getNode(5); /*      5      */
    root->left = getNode(3); /*      / \      */
    root->right = getNode(7); /*      3  7      */
    root->left->left = getNode(2); /*      / \ / \      */
    root->left->right = getNode(4); /*      2 4 6 8      */
    root->right->left = getNode(6);
    root->right->right = getNode(8);

    int x = 10;

    cout << "Count = "
        << countPairs(root, x);

    return 0;
}
```

Output:

```
Count = 3
```

Time Complexity: O(nLog n).

**Source**

<https://www.geeksforgeeks.org/count-pairs-in-a-binary-tree-whose-sum-is-equal-to-a-given-value-x/>

## Chapter 112

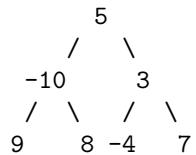
# Count subtrees that sum up to a given value x

Count subtrees that sum up to a given value x - GeeksforGeeks

Given a binary tree containing **n** nodes. The problem is to count subtress having total node's data sum equal to a given value **x**.

**Examples:**

**Input :**



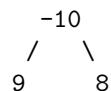
**x = 7**

**Output :** 2

There are 2 subtrees with sum 7.

1st one is leaf node:  
7.

2nd one is:



Source: Microsoft Interview Experience | Set 157.

Algorithm:

```
countSubtreesWithSumX(root, count, x)
if !root then
    return 0

ls = countSubtreesWithSumX(root->left, count, x)
rs = countSubtreesWithSumX(root->right, count, x)
sum = ls + rs + root->data

if sum == x then
    count++
    return sum

countSubtreesWithSumXUtil(root, x)
if !root then
    return 0

Initialize count = 0
ls = countSubtreesWithSumX(root->left, count, x)
rs = countSubtreesWithSumX(root->right, count, x)

if (ls + rs + root->data) == x
    count++
return count
```

C++

```
// C++ implementation to count subtress that
// sum up to a given value x
#include <bits/stdc++.h>

using namespace std;

// structure of a node of binary tree
struct Node {
    int data;
    Node *left, *right;
};

// function to get a new node
Node* getNode(int data)
{
    // allocate space
    Node* newNode = (Node*)malloc(sizeof(Node));
```

```
// put in the data
newNode->data = data;
newNode->left = newNode->right = NULL;
return newNode;
}

// function to count subtress that
// sum up to a given value x
int countSubtreesWithSumX(Node* root,
                           int& count, int x)
{
    // if tree is empty
    if (!root)
        return 0;

    // sum of nodes in the left subtree
    int ls = countSubtreesWithSumX(root->left, count, x);

    // sum of nodes in the right subtree
    int rs = countSubtreesWithSumX(root->right, count, x);

    // sum of nodes in the subtree rooted
    // with 'root->data'
    int sum = ls + rs + root->data;

    // if true
    if (sum == x)
        count++;

    // return subtree's nodes sum
    return sum;
}

// utility function to count subtress that
// sum up to a given value x
int countSubtreesWithSumXUtil(Node* root, int x)
{
    // if tree is empty
    if (!root)
        return 0;

    int count = 0;

    // sum of nodes in the left subtree
    int ls = countSubtreesWithSumX(root->left, count, x);

    // sum of nodes in the right subtree
```

```
int rs = countSubtreesWithSumX(root->right, count, x);

// if tree's nodes sum == x
if ((ls + rs + root->data) == x)
    count++;

// required count of subtrees
return count;
}

// Driver program to test above
int main()
{
    /* binary tree creation
        5
       /   \
      -10   3
     / \   / \
    9  8  -4  7
    */
    Node* root = getNode(5);
    root->left = getNode(-10);
    root->right = getNode(3);
    root->left->left = getNode(9);
    root->left->right = getNode(8);
    root->right->left = getNode(-4);
    root->right->right = getNode(7);

    int x = 7;

    cout << "Count = "
        << countSubtreesWithSumXUtil(root, x);

    return 0;
}
```

### Java

```
// Java program to find if
// there is a subtree with
// given sum
import java.util.*;
class GFG
{
    // structure of a node
    // of binary tree
    static class Node
    {
```

```
int data;
Node left, right;
}

static class INT
{
int v;
INT(int a)
{
v = a;
}
}

// function to get a new node
static Node getNode(int data)
{
// allocate space
Node newNode = new Node();

// put in the data
newNode.data = data;
newNode.left = newNode.right = null;
return newNode;
}

// function to count subtress that
// sum up to a given value x
static int countSubtreesWithSumX(Node root,
INT count, int x)
{
// if tree is empty
if (root == null)
return 0;

// sum of nodes in the left subtree
int ls = countSubtreesWithSumX(root.left,
count, x);

// sum of nodes in the right subtree
int rs = countSubtreesWithSumX(root.right,
count, x);

// sum of nodes in the subtree
// rooted with 'root.data'
int sum = ls + rs + root.data;

// if true
if (sum == x)
count.v++;

// return subtree's nodes sum
return sum;
}
```

```
// utility function to
// count subtress that
// sum up to a given value x
static int countSubtreesWithSumXUtil(Node root,
int x)
{
// if tree is empty
if (root == null)
return 0;

INT count = new INT(0);

// sum of nodes in the left subtree
int ls = countSubtreesWithSumX(root.left,
count, x);

// sum of nodes in the right subtree
int rs = countSubtreesWithSumX(root.right,
count, x);

// if tree's nodes sum == x
if ((ls + rs + root.data) == x)
count.v++;

// required count of subtrees
return count.v;
}

// Driver Code
public static void main(String args[])
{
/* binary tree creation
5
/
-10 3
/ \ / \
9 8 -4 7
*/
Node root = getNode(5);
root.left = getNode(-10);
root.right = getNode(3);
root.left.left = getNode(9);
root.left.right = getNode(8);
root.right.left = getNode(-4);
root.right.right = getNode(7);

int x = 7;

System.out.println("Count = " +
countSubtreesWithSumXUtil(root, x));
}
```

```
// This code is contributed  
// by Arnab Kundu
```

**Output:**

```
Count = 2
```

**Time Complexity:** O(n).

**Improved By :** [andrew1234](#)

**Source**

<https://www.geeksforgeeks.org/count-subtrees-sum-given-value-x/>

## Chapter 113

# Count the number of nodes at given level in a tree using BFS.

Count the number of nodes at given level in a tree using BFS. - GeeksforGeeks

Given a tree represented as undirected graph. Count the number of nodes at given level l. It may be assumed that vertex 0 is root of the tree.

Examples:

```
Input : 7
        0 1
        0 2
        1 3
        1 4
        1 5
        2 6
        2
Output : 4
```

```
Input : 6
        0 1
        0 2
        1 3
        2 4
        2 5
        2
Output : 3
```

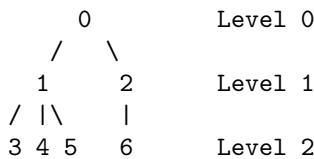
**BFS** is a traversing algorithm which start traversing from a selected node (source or starting node) and traverse the graph layer wise thus exploring the neighbour nodes (nodes which

are directly connected to source node). Then, move towards the next-level neighbour nodes. As the name BFS suggests, traverse the graph breadth wise as follows:

1. First move horizontally and visit all the nodes of the current layer.
2. Move to the next layer.

In this code, while visiting each node, the level of that node is set with an increment in the level of its parent node i.e.,  $\text{level}[\text{child}] = \text{level}[\text{parent}] + 1$ . This is how the level of each node is determined. The root node lies at level zero in the tree.

**Explanation :**



Given a tree with 7 nodes and 6 edges in which node 0 lies at 0 level. Level of 1 can be updated as :  $\text{level}[1] = \text{level}[0] + 1$  as 0 is the parent node of 1. Similarly, the level of other nodes can be updated by adding 1 to the level of their parent.

$\text{level}[2] = \text{level}[0] + 1$ , i.e  $\text{level}[2] = 0 + 1 = 1$ .

$\text{level}[3] = \text{level}[1] + 1$ , i.e  $\text{level}[3] = 1 + 1 = 2$ .

$\text{level}[4] = \text{level}[1] + 1$ , i.e  $\text{level}[4] = 1 + 1 = 2$ .

$\text{level}[5] = \text{level}[1] + 1$ , i.e  $\text{level}[5] = 1 + 1 = 2$ .

$\text{level}[6] = \text{level}[2] + 1$ , i.e  $\text{level}[6] = 1 + 1 = 2$ .

Then, count of number of nodes which are at level l(i.e, l=2) is 4 (node:- 3, 4, 5, 6)

C++

```
// C++ Program to print
// count of nodes
// at given level.
#include <iostream>
#include <list>

using namespace std;

// This class represents
// a directed graph
// using adjacency
// list representation
class Graph {
    // No. of vertices
    int V;

    // Pointer to an
    // array containing
```

```
// adjacency lists
list<int>* adj;

public:
    // Constructor
    Graph(int V);

    // function to add
    // an edge to graph
    void addEdge(int v, int w);

    // Returns count of nodes at
    // level l from given source.
    int BFS(int s, int l);
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
    // Add w to v's list.
    adj[v].push_back(w);

    // Add v to w's list.
    adj[w].push_back(v);
}

int Graph::BFS(int s, int l)
{
    // Mark all the vertices
    // as not visited
    bool* visited = new bool[V];
    int level[V];

    for (int i = 0; i < V; i++) {
        visited[i] = false;
        level[i] = 0;
    }

    // Create a queue for BFS
    list<int> queue;

    // Mark the current node as
    // visited and enqueue it
```

```
visited[s] = true;
queue.push_back(s);
level[s] = 0;

while (!queue.empty()) {

    // Dequeue a vertex from
    // queue and print it
    s = queue.front();
    queue.pop_front();

    // Get all adjacent vertices
    // of the dequeued vertex s.
    // If a adjacent has not been
    // visited, then mark it
    // visited and enqueue it
    for (auto i = adj[s].begin();
         i != adj[s].end(); ++i) {
        if (!visited[*i]) {

            // Setting the level
            // of each node with
            // an increment in the
            // level of parent node
            level[*i] = level[s] + 1;
            visited[*i] = true;
            queue.push_back(*i);
        }
    }
}

int count = 0;
for (int i = 0; i < V; i++)
    if (level[i] == 1)
        count++;
return count;
}

// Driver program to test
// methods of graph class
int main()
{
    // Create a graph given
    // in the above diagram
    Graph g(6);
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 3);
```

```
g.addEdge(2, 4);
g.addEdge(2, 5);

int level = 2;

cout << g.BFS(0, level);

return 0;
}
```

Output:

3

## Source

<https://www.geeksforgeeks.org/count-number-nodes-given-level-using-bfs/>

## Chapter 114

# Counting the number of words in a Trie

Counting the number of words in a Trie - GeeksforGeeks

A [Trie](#) is used to store dictionary words so that they can be searched efficiently and prefix search can be done. The task is to write a function to count the number of words.

Example :

```
Input :      root
            /   \   \
          t     a     b
          |     |     |
          h     n     y
          |     |   \
          e     s     y
          |     |     |
          i     r     e
          |     |     |
          r     e     e
                      |
                      r

Output : 8
Explanation : Words formed in the Trie :
"the", "a", "there", "answer", "any", "by",
"bye", "their".
```

In Trie structure, we have a field to store end of word marker, we call it `isLeaf` in below implementation. To count words, we need to simply traverse the Trie and count all nodes where `isLeaf` is set.

C++

```
// C++ implementation to count words in a trie
#include <bits/stdc++.h>
using namespace std;

#define ARRAY_SIZE(a) sizeof(a)/sizeof(a[0])

// Alphabet size (# of symbols)
#define ALPHABET_SIZE (26)

// Converts key current character into index
// use only 'a' through 'z' and lower case
#define CHAR_TO_INDEX(c) ((int)c - (int)'a')

// Trie node
struct TrieNode
{
    struct TrieNode *children[ALPHABET_SIZE];

    // isLeaf is true if the node represents
    // end of a word
    bool isLeaf;
};

// Returns new trie node (initialized to NULLs)
struct TrieNode *getNode(void)
{
    struct TrieNode *pNode = new TrieNode;
    pNode->isLeaf = false;

    for (int i = 0; i < ALPHABET_SIZE; i++)
        pNode->children[i] = NULL;

    return pNode;
}

// If not present, inserts key into trie
// If the key is prefix of trie node, just
// marks leaf node
void insert(struct TrieNode *root, const char *key)
{
    int length = strlen(key);

    struct TrieNode *pCrawl = root;

    for (int level = 0; level < length; level++)
    {
```

```
int index = CHAR_TO_INDEX(key[level]);
if (!pCrawl->children[index])
    pCrawl->children[index] = getNode();

pCrawl = pCrawl->children[index];
}

// mark last node as leaf
pCrawl->isLeaf = true;
}

// Function to count number of words
int wordCount(struct TrieNode *root)
{
    int result = 0;

    // Leaf denotes end of a word
    if (root -> isLeaf)
        result++;

    for (int i = 0; i < ALPHABET_SIZE; i++)
        if (root -> children[i])
            result += wordCount(root -> children[i]);

    return result;
}

// Driver
int main()
{
    // Input keys (use only 'a' through 'z'
    // and lower case)
    char keys[] [8] = {"the", "a", "there", "answer",
                       "any", "by", "bye", "their"};

    struct TrieNode *root = getNode();

    // Construct Trie
    for (int i = 0; i < ARRAY_SIZE(keys); i++)
        insert(root, keys[i]);

    cout << wordCount(root);

    return 0;
}
```

Java

```
// Java implementation to count words in a trie
public class Words_in_trie {

    // Alphabet size (# of symbols)
    static final int ALPHABET_SIZE = 26;

    // Trie node
    static class TrieNode
    {
        TrieNode[] children = new TrieNode[ALPHABET_SIZE];
        // isLeaf is true if the node represents
        // end of a word
        boolean isLeaf;

        TrieNode(){
            isLeaf = false;
            for (int i = 0; i < ALPHABET_SIZE; i++)
                children[i] = null;
        }
    };
    static TrieNode root;

    // If not present, inserts key into trie
    // If the key is prefix of trie node, just
    // marks leaf node
    static void insert(String key)
    {
        int length = key.length();

        TrieNode pCrawl = root;

        for (int level = 0; level < length; level++)
        {
            int index = key.charAt(level) - 'a';
            if (pCrawl.children[index] == null)
                pCrawl.children[index] = new TrieNode();

            pCrawl = pCrawl.children[index];
        }

        // mark last node as leaf
        pCrawl.isLeaf = true;
    }

    // Function to count number of words
    static int wordCount(TrieNode root)
    {
        int result = 0;
```

```
// Leaf denotes end of a word
if (root.isLeaf)
    result++;

for (int i = 0; i < ALPHABET_SIZE; i++)
    if (root.children[i] != null )
        result += wordCount(root.children[i]);

return result;
}

// Driver Program
public static void main(String args[])
{
    // Input keys (use only 'a' through 'z'
    // and lower case)
    String keys[] = {"the", "a", "there", "answer",
                     "any", "by", "bye", "their"};

    root = new TrieNode();

    // Construct Trie
    for (int i = 0; i < keys.length; i++)
        insert(keys[i]);

    System.out.println(wordCount(root));
}
}

// This code is contributed by Sumit Ghosh
```

Output:

8

## Source

<https://www.geeksforgeeks.org/counting-number-words-trie/>

## Chapter 115

# Create a Doubly Linked List from a Ternary Tree

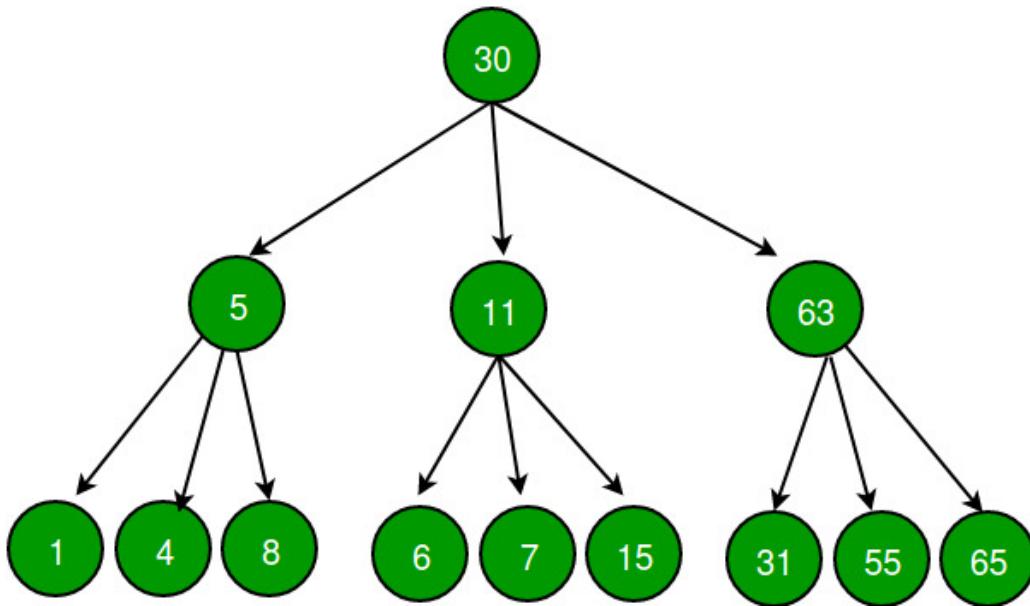
Create a Doubly Linked List from a Ternary Tree - GeeksforGeeks

Given a ternary tree, create a doubly linked list out of it. A ternary tree is just like binary tree but instead of having two nodes, it has three nodes i.e. left, middle, right.

The doubly linked list should holds following properties –

1. Left pointer of ternary tree should act as prev pointer of doubly linked list.
2. Middle pointer of ternary tree should not point to anything.
3. Right pointer of ternary tree should act as next pointer of doubly linked list.
4. Each node of ternary tree is inserted into doubly linked list before its subtrees and for any node, its left child will be inserted first, followed by mid and right child (if any).

For the above example, the linked list formed for below tree should be NULL 5 1 4 8 11 6  
7 15 63 31 55 65 -> NULL



We strongly recommend you to minimize your browser and try this yourself first.

The idea is to traverse the tree in preorder fashion similar to binary tree preorder traversal. Here, when we visit a node, we will insert it into doubly linked list in the end using a tail pointer. That we use to maintain the required insertion order. We then recursively call for left child, middle child and right child in that order.

Below is the implementation of this idea.

C++

```

// C++ program to create a doubly linked list out
// of given a ternary tree.
#include <bits/stdc++.h>
using namespace std;

/* A ternary tree */
struct Node
{
    int data;
    struct Node *left, *middle, *right;
};

/* Helper function that allocates a new node with the
   given data and assign NULL to left, middle and right
   pointers.*/
Node* newNode(int data)
{
    Node* node = new Node;
    node->data = data;
    node->left = node->middle = node->right = NULL;
    return node;
}

/* Function to print doubly linked list */
void printList(Node* head)
{
    Node* temp = head;
    while (temp != NULL)
    {
        cout << temp->data << " ";
        temp = temp->right;
    }
}
  
```

```
node->data = data;
node->left = node->middle = node->right = NULL;
return node;
}

/* Utility function that constructs doubly linked list
by inserting current node at the end of the doubly
linked list by using a tail pointer */
void push(Node** tail_ref, Node* node)
{
    // initialize tail pointer
    if (*tail_ref == NULL)
    {
        *tail_ref = node;

        // set left, middle and right child to point
        // to NULL
        node->left = node->middle = node->right = NULL;

        return;
    }

    // insert node in the end using tail pointer
    (*tail_ref)->right = node;

    // set prev of node
    node->left = (*tail_ref);

    // set middle and right child to point to NULL
    node->right = node->middle = NULL;

    // now tail pointer will point to inserted node
    (*tail_ref) = node;
}

/* Create a doubly linked list out of given a ternary tree.
by traversing the tree in preorder fashion. */
Node* TernaryTreeToList(Node* root, Node** head_ref)
{
    // Base case
    if (root == NULL)
        return NULL;

    //create a static tail pointer
    static Node* tail = NULL;

    // store left, middle and right nodes
    // for future calls.
```

```
Node* left = root->left;
Node* middle = root->middle;
Node* right = root->right;

// set head of the doubly linked list
// head will be root of the ternary tree
if (*head_ref == NULL)
    *head_ref = root;

// push current node in the end of DLL
push(&tail, root);

//recurse for left, middle and right child
TernaryTreeToList(left, head_ref);
TernaryTreeToList(middle, head_ref);
TernaryTreeToList(right, head_ref);
}

// Utility function for printing double linked list.
void printList(Node* head)
{
    printf("Created Double Linked list is:\n");
    while (head)
    {
        printf("%d ", head->data);
        head = head->right;
    }
}

// Driver program to test above functions
int main()
{
    // Construting ternary tree as shown in above figure
    Node* root = newNode(30);

    root->left = newNode(5);
    root->middle = newNode(11);
    root->right = newNode(63);

    root->left->left = newNode(1);
    root->left->middle = newNode(4);
    root->left->right = newNode(8);

    root->middle->left = newNode(6);
    root->middle->middle = newNode(7);
    root->middle->right = newNode(15);

    root->right->left = newNode(31);
```

```
root->right->middle = newNode(55);
root->right->right = newNode(65);

Node* head = NULL;

TernaryTreeToList(root, &head);

printList(head);

return 0;
}
```

**Java**

```
//Java program to create a doubly linked list
// from a given ternary tree.

//Custom node class.
class newNode
{
    int data;
    newNode left,middle,right;
    public newNode(int data)
    {
        this.data = data;
        left = middle = right = null;
    }
}

class GFG {

    //tail of the linked list.
    static newNode tail;

    //function to push the node to the tail.
    public static void push(newNode node)
    {
        //to put the node at the end of
        // the already existing tail.
        tail.right = node;

        //to point to the previous node.
        node.left = tail;

        // middle pointer should point to
        // nothing so null. initiate right
        // pointer to null.
        node.middle = node.right = null;
    }
}
```

```
//update the tail position.  
tail = node;  
}  
  
/* Create a doubly linked list out of given a ternary tree.  
by traversing the tree in preoder fashion. */  
public static void ternaryTree(newNode node,newNode head)  
{  
    if(node == null)  
        return;  
    newNode left = node.left;  
    newNode middle = node.middle;  
    newNode right = node.right;  
    if(tail != node)  
  
        // already root is in the tail so dont push  
        // the node when it was root.In the first  
        // case both node and tail have root in them.  
        push(node);  
  
        // First the left child is to be taken.  
        // Then middle and then right child.  
        ternaryTree(left,head);  
        ternaryTree(middle,head);  
        ternaryTree(right,head);  
    }  
  
    //function to initiate the list process.  
    public static newNode startTree(newNode root)  
{  
        //Initiate the head and tail with root.  
        newNode head = root;  
        tail = root;  
        ternaryTree(root,head);  
  
        //since the head,root are passed  
        // with reference the changes in  
        // root will be reflected in head.  
        return head;  
    }  
  
    // Utility function for printing double linked list.  
    public static void printList(newNode head)  
{  
        System.out.print("Created Double Linked list is:\n");  
        while(head != null)  
        {
```

```
        System.out.print(head.data + " ");
        head = head.right;
    }
}

// Driver program to test above functions
public static void main(String args[])
{
    // Construting ternary tree as shown
    // in above figure
    newNode root = new newNode(30);
    root.left = new newNode(5);
    root.middle = new newNode(11);
    root.right = new newNode(63);
    root.left.left = new newNode(1);
    root.left.middle = new newNode(4);
    root.left.right = new newNode(8);
    root.middle.left = new newNode(6);
    root.middle.middle = new newNode(7);
    root.middle.right = new newNode(15);
    root.right.left = new newNode(31);
    root.right.middle = new newNode(55);
    root.right.right = new newNode(65);

    // The function which initiates the list
    // process returns the head.
    newNode head = startTree(root);
    printList(head);
}
}

// This code is contributed by M.V.S.Surya Teja.
```

Output:

```
Created Double Linked list is:
30 5 1 4 8 11 6 7 15 63 31 55 65
```

Improved By : [MvssTeja](#)

## Source

<https://www.geeksforgeeks.org/create-doubly-linked-list-ternary-tree/>

## Chapter 116

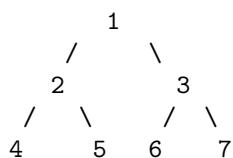
# Create loops of even and odd values in a binary tree

Create loops of even and odd values in a binary tree - GeeksforGeeks

Given a binary tree with the node structure containing a data part, left and right pointers and an arbitrary pointer(abtr). The node's value can be any positive integer. The problem is to create odd and even loops in a binary tree. An odd loop is a loop which connects all the nodes having odd numbers and similarly even loop is for nodes having even numbers. To create such loops, the abtr pointer of each node is used. An abtr pointer of an odd node(node having odd number) points to some other odd node in the tree. A loop must be created in such way that from any node we could traverse all the nodes in the loop to which the node belongs.

Examples:

Consider the binary tree given below



Now with the help of abtr pointers of node,  
we connect odd and even nodes as:

Odd loop  
1 -> 5 -> 3 -> 7 -> 1(again pointing to first node  
in the loop)

Even loop

```
2 -> 4 -> 6 -> 2(again pointing to first node  
in the loop)
```

Nodes in the respective loop can be arranged in any order. But from any node in the loop we should be able to traverse all the nodes in the loop.

**Approach:** The following steps are:

1. Add pointers of the nodes having even and odd numbers to **even\_ptrs** and **odd\_ptrs** arrays respectively. Through any tree traversal we could get the respective node pointers.
2. For both the **even\_ptrs** and **odd\_ptrs** array, perform:
  - As the array contains node pointers, consider an element at ith index, let it be **node**, and the assign **node->abtr** = element at (i+1)th index.
  - For last element of the array, **node->abtr** = element at index 0.

```
// C++ implementation to create odd and even loops  
// in a binary tree  
#include <bits/stdc++.h>  
  
using namespace std;  
  
// structure of a node  
struct Node  
{  
    int data;  
    Node *left, *right, *abtr;  
};  
  
// Utility function to create a new node  
struct Node* newNode(int data)  
{  
    struct Node* node = new Node;  
    node->data = data;  
    node->left = node->right = node->abtr = NULL;  
    return node;  
}  
  
// preorder traversal to place the node pointer  
// in the respective even_ptrs or odd_ptrs list  
void preorderTraversal(Node *root, vector<Node*> *even_ptrs,  
                      vector<Node*> *odd_ptrs)  
{  
    if (!root)  
        return;
```

```
// place node ptr in even_ptrs list if
// node contains even number
if (root->data % 2 == 0)
    (*even_ptrs).push_back(root);

// else place node ptr in odd_ptrs list
else
    (*odd_ptrs).push_back(root);

preorderTraversal(root->left, even_ptrs, odd_ptrs);
preorderTraversal(root->right, even_ptrs, odd_ptrs);
}

// function to create the even and odd loops
void createLoops(Node *root)
{
    vector<Node*> even_ptrs, odd_ptrs;
    preorderTraversal(root, &even_ptrs, &odd_ptrs);

    int i;

    // forming even loop
    for (i=1; i<even_ptrs.size(); i++)
        even_ptrs[i-1]->abtr = even_ptrs[i];

    // for the last element
    even_ptrs[i-1]->abtr = even_ptrs[0];

    // Similarly forming odd loop
    for (i=1; i<odd_ptrs.size(); i++)
        odd_ptrs[i-1]->abtr = odd_ptrs[i];
    odd_ptrs[i-1]->abtr = odd_ptrs[0];
}

// traversing the loop from any random
// node in the loop
void traverseLoop(Node *start)
{
    Node *curr = start;
    do
    {
        cout << curr->data << " ";
        curr = curr->abtr;
    } while (curr != start);
}

// Driver program to test above
```

```
int main()
{
    // Binary tree formation
    struct Node* root = NULL;
    root = newNode(1);                                /*      1      */
    root->left = newNode(2);                          /*   / \      */
    root->right = newNode(3);                         /* 2   3      */
    root->left->left = newNode(4);                  /* / \ / \    */
    root->left->right = newNode(5);                 /* 4   5   6   */
    root->right->left = newNode(6);                 /*       \   7 */
    root->right->right = newNode(7);

    createLoops(root);

    // traversing odd loop from any
    // random odd node
    cout << "Odd nodes: ";
    traverseLoop(root->right);

    cout << endl << "Even nodes: ";
    // traversing even loop from any
    // random even node
    traverseLoop(root->left);

    return 0;
}
```

Output:

```
Odd nodes: 3 7 1 5
Even nodes: 2 4 6
```

Time Complexity: Equal to the time complexity of any recursive tree traversal which is  $O(n)$

## Source

<https://www.geeksforgeeks.org/create-loops-of-even-and-odd-values-in-a-binary-tree/>

## Chapter 117

# Creating a tree with Left-Child Right-Sibling Representation

Creating a tree with Left-Child Right-Sibling Representation - GeeksforGeeks

[Left-Child Right-Sibling Representation](#) is a different representation of an n-ary tree where instead of holding a reference to each and every child node, a node holds just two references, first a reference to its first child, and the other to its immediate next sibling. This new transformation not only removes the need of advance knowledge of the number of children a node has, but also limits the number of references to a maximum of two, thereby making it so much easier to code.

At each node, link children of same parent from left to right.  
Parent should be linked with only first child.

Examples:

Left Child Right Sibling tree representation

```
10
 |
2 -> 3 -> 4 -> 5
 |
6   7 -> 8 -> 9
```

Prerequisite : [Left-Child Right-Sibling Representation of Tree](#)  
Below is the implementation.

C++

```
// CPP program to create a tree with left child
```

```
// right sibling representation.  
#include<bits/stdc++.h>  
using namespace std;  
  
struct Node  
{  
    int data;  
    struct Node *next;  
    struct Node *child;  
};  
  
// Creating new Node  
Node* newNode(int data)  
{  
    Node *newNode = new Node;  
    newNode->next = newNode->child = NULL;  
    newNode->data = data;  
    return newNode;  
}  
  
// Adds a sibling to a list with starting with n  
Node *addSibling(Node *n, int data)  
{  
    if (n == NULL)  
        return NULL;  
  
    while (n->next)  
        n = n->next;  
  
    return (n->next = newNode(data));  
}  
  
// Add child Node to a Node  
Node *addChild(Node * n, int data)  
{  
    if (n == NULL)  
        return NULL;  
  
    // Check if child list is not empty.  
    if (n->child)  
        return addSibling(n->child, data);  
    else  
        return (n->child = newNode(data));  
}  
  
// Traverses tree in level order  
void traverseTree(Node * root)  
{
```

```
if (root == NULL)
    return;

while (root)
{
    cout << " " << root->data;
    if (root->child)
        traverseTree(root->child);
    root = root->next;
}
}

//Driver code

int main()
{
    /* Let us create below tree
     *          10
     *         /   \
     *        2   3   4   5
     *           |   /   |   \
     *           6   7   8   9 */
}

// Left child right sibling
/* 10
 *   |
 *   2 -> 3 -> 4 -> 5
 *       |
 *       6   7 -> 8 -> 9 */

Node *root = newNode(10);
Node *n1 = addChild(root, 2);
Node *n2 = addChild(root, 3);
Node *n3 = addChild(root, 4);
Node *n4 = addChild(n3, 6);
Node *n5 = addChild(root, 5);
Node *n6 = addChild(n5, 7);
Node *n7 = addChild(n5, 8);
Node *n8 = addChild(n5, 9);
traverseTree(root);
return 0;
}
```

**Java**

```
// CPP program to create a tree with left child
// right sibling representation.

class GFG {
```

```
static class NodeTemp
{
    int data;
    NodeTemp next, child;
    public NodeTemp(int data)
    {
        this.data = data;
        next = child = null;
    }
}

// Adds a sibling to a list with starting with n
static public NodeTemp addSibling(NodeTemp node, int data)
{
    if(node == null)
        return null;
    while(node.next != null)
        node = node.next;
    return(node.next = new NodeTemp(data));
}

// Add child Node to a Node
static public NodeTemp addChild(NodeTemp node,int data)
{
    if(node == null)
        return null;

    // Check if child is not empty.
    if(node.child != null)
        return(addSibling(node.child,data));
    else
        return(node.child = new NodeTemp(data));
}

// Traverses tree in level order
static public void traverseTree(NodeTemp root)
{
    if(root == null)
        return;
    while(root != null)
    {
        System.out.print(root.data + " ");
        if(root.child != null)
            traverseTree(root.child);
        root = root.next;
    }
}
```

```
// Driver code
public static void main(String args[])
{
    /* Let us create below tree
     *          10
     *         /   \
     *        2   3     4   5
     *           |   /   |
     *           6   7   8   9 */
    // Left child right sibling
    /* 10
     *  |
     * 2 -> 3 -> 4 -> 5
     *      |
     *      6   7 -> 8 -> 9 */

    NodeTemp root = new NodeTemp(10);
    NodeTemp n1 = addChild(root,2);
    NodeTemp n2 = addChild(root,3);
    NodeTemp n3 = addChild(root,4);
    NodeTemp n4 = addChild(n3,6);
    NodeTemp n5 = addChild(root,5);
    NodeTemp n6 = addChild(n5,7);
    NodeTemp n7 = addChild(n5,8);
    NodeTemp n8 = addChild(n5,9);

    traverseTree(root);
}
}

// This code is contributed by M.V.S.Surya Teja.
```

Output:

```
10 2 3 4 6 5 7 8 9
```

Improved By : [MvssTeja](#)

## Source

<https://www.geeksforgeeks.org/creating-tree-left-child-right-sibling-representation/>

## Chapter 118

# Custom Tree Problem

Custom Tree Problem - GeeksforGeeks

You are given a set of links, e.g.

```
a ----> b  
b ----> c  
b ----> d  
a ----> e
```

Print the tree that would form when each pair of these links that has the same character as start and end point is joined together. You have to maintain fidelity w.r.t. the height of nodes, i.e. nodes at height n from root should be printed at same row or column. For set of links given above, tree printed should be –

```
-->a  
|-->b  
|   |-->c  
|   |-->d  
|-->e
```

Note that these links need not form a single tree; they could form, ahem, a forest. Consider the following links

```
a ----> b  
a ----> g  
b ----> c  
c ----> d  
d ----> e  
c ----> f  
z ----> y  
y ----> x  
x ----> w
```

The output would be following forest.

```
-->a
|--->b
|   |--->c
|   |   |--->d
|   |   |   |--->e
|   |   |--->f
|--->g

-->z
|--->y
|   |--->x
|   |--->w
```

You can assume that given links can form a tree or forest of trees only, and there are no duplicates among links.

**Solution:** The idea is to maintain two arrays, one array for tree nodes and other for trees themselves (we call this array forest). An element of the node array contains the `TreeNode` object that corresponds to respective character. An element of the forest array contains `Tree` object that corresponds to respective root of tree.

It should be obvious that the crucial part is creating the forest here, once it is created, printing it out in required format is straightforward. To create the forest, following procedure is used –

Do following for each input link,

1. If start of link is not present in node array  
Create `TreeNode` objects for start character  
Add entries of start in both arrays.
2. If end of link is not present in node array  
Create `TreeNode` objects for start character  
Add entry of end in node array.
3. If end of link is present in node array.  
If end of link is present in forest array, then remove it from there.
4. Add an edge (in tree) between start and end nodes of link.

It should be clear that this procedure runs in linear time in number of nodes as well as of links – it makes only one pass over the links. It also requires linear space in terms of alphabet size.

Following is Java implementation of above algorithm. In the following implementation characters are assumed to be only lower case characters from ‘a’ to ‘z’.

```
// Java program to create a custom tree from a given set of links.
```

```

// The main class that represents tree and has main method
public class Tree {

    private TreeNode root;

    /* Returns an array of trees from links input. Links are assumed to
       be Strings of the form "<s> <e>" where <s> and <e> are starting
       and ending points for the link. The returned array is of size 26
       and has non-null values at indexes corresponding to roots of trees
       in output */
    public Tree[] buildFromLinks(String [] links) {

        // Create two arrays for nodes and forest
        TreeNode[] nodes = new TreeNode[26];
        Tree[] forest = new Tree[26];

        // Process each link
        for (String link : links) {

            // Find the two ends of current link
            String[] ends = link.split(" ");
            int start = (int) (ends[0].charAt(0) - 'a'); // Start node
            int end   = (int) (ends[1].charAt(0) - 'a'); // End node

            // If start of link not seen before, add it to both arrays
            if (nodes[start] == null)
            {
                nodes[start] = new TreeNode((char) (start + 'a'));

                // Note that it may be removed later when this character is
                // last character of a link. For example, let we first see
                // a--->b, then c--->a. We first add 'a' to array of trees
                // and when we see link c--->a, we remove it from trees array.
                forest[start] = new Tree(nodes[start]);
            }

            // If end of link is not seen before, add it to the nodes array
            if (nodes[end] == null)
                nodes[end] = new TreeNode((char) (end + 'a'));

            // If end of link is seen before, remove it from forest if
            // it exists there.
            else forest[end] = null;

            // Establish Parent-Child Relationship between Start and End
            nodes[start].addChild(nodes[end], end);
        }
    }
}

```

```

        return forest;
    }

    // Constructor
    public Tree(TreeNode root) { this.root = root; }

    public static void printForest(String[] links)
    {
        Tree t = new Tree(new TreeNode('\0'));
        for (Tree t1 : t.buildFromLinks(links)) {
            if (t1 != null)
            {
                t1.root.printTreeIndented("");
                System.out.println("");
            }
        }
    }

    // Driver method to test
    public static void main(String[] args) {
        String [] links1 = {"a b", "b c", "b d", "a e"};
        System.out.println("----- Forest 1 -----");
        printForest(links1);

        String [] links2 = {"a b", "a g", "b c", "c d", "d e", "c f",
                           "z y", "y x", "x w"};
        System.out.println("----- Forest 2 -----");
        printForest(links2);
    }
}

// Class to represent a tree node
class TreeNode {
    TreeNode [] children;
    char c;

    // Adds a child 'n' to this node
    public void addChild(TreeNode n, int index) { this.children[index] = n; }

    // Constructor
    public TreeNode(char c) { this.c = c; this.children = new TreeNode[26]; }

    // Recursive method to print indented tree rooted with this node.
    public void printTreeIndented(String indent) {
        System.out.println(indent + "-->" + c);
        for (TreeNode child : children) {
            if (child != null)
                child.printTreeIndented(indent + "    |");
        }
    }
}

```

```
        }
    }
}
```

Output:

```
----- Forest 1 -----
-->a
|-->b
|   |-->c
|   |-->d
|-->e

----- Forest 2 -----
-->a
|-->b
|   |-->c
|   |   |-->d
|   |   |   |-->e
|   |   |-->f
|-->g

-->z
|-->y
|   |-->x
|   |   |-->w
```

**Exercise:**

In the above implementation, endpoints of input links are assumed to be from set of only 26 characters. Extend the implementation where endpoints are strings of any length.

This article is contributed by **Ciphe**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## Source

<https://www.geeksforgeeks.org/custom-tree-problem/>

## Chapter 119

# DFS for a n-ary tree (acyclic graph) represented as adjacency list

DFS for a n-ary tree (acyclic graph) represented as adjacency list - GeeksforGeeks

A tree consisting of n nodes is given, we need to print its [DFS](#).

Examples :

Input : Edges of graph

```
1 2  
1 3  
2 4  
3 5
```

Output : 1 2 4 3 5

A simple solution is to do implement standard [DFS](#).

We can modify our approach to avoid extra space for visited nodes. Instead of using the visited array, we can keep track of parent. We traverse all adjacent nodes but the parent.

Below is the implementation :

C++

```
/* CPP code to perform DFS of given tree : */  
#include <bits/stdc++.h>  
using namespace std;  
  
// DFS on tree
```

```
void dfs(vector<int> list[], int node, int arrival)
{
    // Printing traversed node
    cout << node << '\n';

    // Traversing adjacent edges
    for (int i = 0; i < list[node].size(); i++) {

        // Not traversing the parent node
        if (list[node][i] != arrival)
            dfs(list, list[node][i], node);
    }
}

int main()
{
    // Number of nodes
    int nodes = 5;

    // Adjacency list
    vector<int> list[10000];

    // Designing the tree
    list[1].push_back(2);
    list[2].push_back(1);

    list[1].push_back(3);
    list[3].push_back(1);

    list[2].push_back(4);
    list[4].push_back(2);

    list[3].push_back(5);
    list[5].push_back(3);

    // Function call
    dfs(list, 1, 0);

    return 0;
}
```

### Java

```
//JAVA Code For DFS for a n-ary tree (acyclic graph)
// represented as adjacency list
import java.util.*;

class GFG {
```

```
// DFS on tree
public static void dfs(LinkedList<Integer> list[],
                      int node, int arrival)
{
    // Printing traversed node
    System.out.println(node);

    // Traversing adjacent edges
    for (int i = 0; i < list[node].size(); i++) {

        // Not traversing the parent node
        if (list[node].get(i) != arrival)
            dfs(list, list[node].get(i), node);
    }
}

/* Driver program to test above function */
public static void main(String[] args)
{

    // Number of nodes
    int nodes = 5;

    // Adjacency list
    LinkedList<Integer> list[] = new LinkedList[nodes+1];

    for (int i = 0; i < list.length; i ++){
        list[i] = new LinkedList<Integer>();
    }

    // Designing the tree
    list[1].add(2);
    list[2].add(1);

    list[1].add(3);
    list[3].add(1);

    list[2].add(4);
    list[4].add(2);

    list[3].add(5);
    list[5].add(3);

    // Function call
    dfs(list, 1, 0);
}
```

```
    }
}

// This code is contributed by Arnav Kr. Mandal.
```

Output:

```
1
2
4
3
5
```

## Source

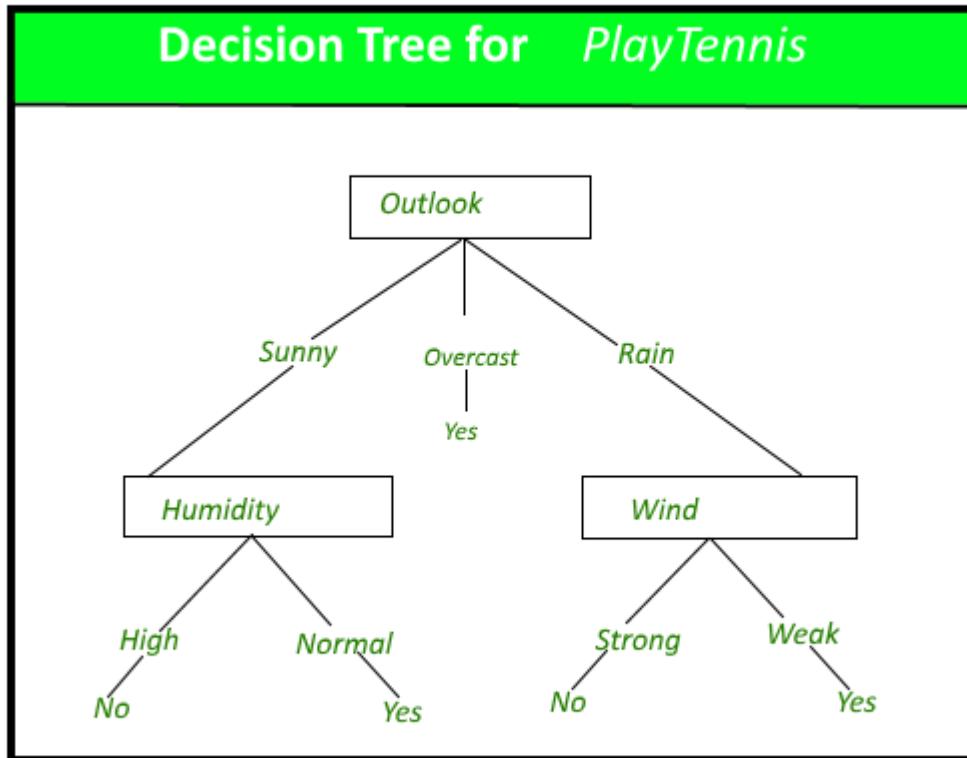
<https://www.geeksforgeeks.org/dfs-n-ary-tree-acyclic-graph-represented-adjacency-list/>

## Chapter 120

# Decision Tree

Decision Tree - GeeksforGeeks

**Decision Tree :** Decision tree is the most powerful and popular tool for classification and prediction. A Decision tree is a flowchart like tree structure, where each internal node denotes a test on an attribute, each branch represents an outcome of the test, and each leaf node (terminal node) holds a class label.



A decision tree for the concept *PlayTennis*.

#### **Construction of Decision Tree :**

A tree can be “learned” by splitting the source set into subsets based on an attribute value test. This process is repeated on each derived subset in a recursive manner called *recursive partitioning*. The recursion is completed when the subset at a node all has the same value of the target variable, or when splitting no longer adds value to the predictions. The construction of decision tree classifier does not require any domain knowledge or parameter setting, and therefore is appropriate for exploratory knowledge discovery. Decision trees can handle high dimensional data. In general decision tree classifier has good accuracy. Decision tree induction is a typical inductive approach to learn knowledge on classification.

#### **Decision Tree Representation :**

Decision trees classify instances by sorting them down the tree from the root to some leaf node, which provides the classification of the instance. An instance is classified by starting at the root node of the tree, testing the attribute specified by this node, then moving down the tree branch corresponding to the value of the attribute as shown in the above figure. This

process is then repeated for the subtree rooted at the new node.

The decision tree in above figure classifies a particular morning according to whether it is suitable for playing tennis and returning the classification associated with the particular leaf.(in this case Yes or No).

For example, the instance

(Outlook = Rain, Temperature = Hot, Humidity = High, Wind = Strong )

would be sorted down the leftmost branch of this decision tree and would therefore be classified as a negative instance.

In other words we can say that decision tree represent a disjunction of conjunctions of constraints on the attribute values of instances.

(Outlook = Sunny  $\wedge$  Humidity = Normal)  $\vee$  (Outlook = Overcast)  $\vee$  (Outlook = Rain  $\wedge$  Wind = Weak)

### Strengths and Weakness of Decision Tree approach

The strengths of decision tree methods are:

- Decision trees are able to generate understandable rules.
- Decision trees perform classification without requiring much computation.
- Decision trees are able to handle both continuous and categorical variables.
- Decision trees provide a clear indication of which fields are most important for prediction or classification.

The weaknesses of decision tree methods :

- Decision trees are less appropriate for estimation tasks where the goal is to predict the value of a continuous attribute.
- Decision trees are prone to errors in classification problems with many class and relatively small number of training examples.
- Decision tree can be computationally expensive to train. The process of growing a decision tree is computationally expensive. At each node, each candidate splitting field must be sorted before its best split can be found. In some algorithms, combinations of fields are used and a search must be made for optimal combining weights. Pruning algorithms can also be expensive since many candidate sub-trees must be formed and compared.

### References :

*Machine Learning, Tom Mitchell, McGraw Hill, 1997.*

In the next post we will be discussing about ID3 algorithm for the construction of Decision tree given by J. R. Quinlan.

### Source

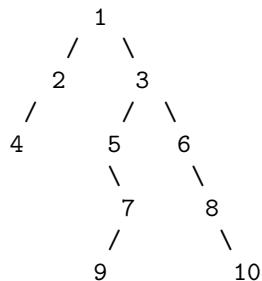
<https://www.geeksforgeeks.org/decision-tree/>

## Chapter 121

# Deepest left leaf node in a binary tree

Deepest left leaf node in a binary tree - GeeksforGeeks

Given a Binary Tree, find the deepest leaf node that is left child of its parent. For example, consider the following tree. The deepest left leaf node is the node with value 9.



The idea is to recursively traverse the given binary tree and while traversing, maintain “level” which will store the current node’s level in the tree. If current node is left leaf, then check if its level is more than the level of deepest left leaf seen so far. If level is more then update the result. If current node is not leaf, then recursively find maximum depth in left and right subtrees, and return maximum of the two depths. Thanks to [Coder011](#) for suggesting this approach.

C/C++

```
// A C++ program to find the deepest left leaf in a given binary tree
#include <stdio.h>
#include <iostream>
using namespace std;
```

```
struct Node
{
    int val;
    struct Node *left, *right;
};

Node *newNode(int data)
{
    Node *temp = new Node;
    temp->val = data;
    temp->left = temp->right = NULL;
    return temp;
}

// A utility function to find deepest leaf node.
// lvl: level of current node.
// maxlvl: pointer to the deepest left leaf node found so far
// isLeft: A bool indicate that this node is left child of its parent
// resPtr: Pointer to the result
void deepestLeftLeafUtil(Node *root, int lvl, int *maxlvl,
                         bool isLeft, Node **resPtr)
{
    // Base case
    if (root == NULL)
        return;

    // Update result if this node is left leaf and its level is more
    // than the maxl level of the current result
    if (isLeft && !root->left && !root->right && lvl > *maxlvl)
    {
        *resPtr = root;
        *maxlvl = lvl;
        return;
    }

    // Recur for left and right subtrees
    deepestLeftLeafUtil(root->left, lvl+1, maxlvl, true, resPtr);
    deepestLeftLeafUtil(root->right, lvl+1, maxlvl, false, resPtr);
}

// A wrapper over deepestLeftLeafUtil().
Node* deepestLeftLeaf(Node *root)
{
    int maxlevel = 0;
    Node *result = NULL;
    deepestLeftLeafUtil(root, 0, &maxlevel, false, &result);
    return result;
}
```

```
}

// Driver program to test above function
int main()
{
    Node* root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->right->left = newNode(5);
    root->right->right = newNode(6);
    root->right->left->right = newNode(7);
    root->right->right->right = newNode(8);
    root->right->left->right->left = newNode(9);
    root->right->right->right->right = newNode(10);

    Node *result = deepestLeftLeaf(root);
    if (result)
        cout << "The deepest left child is " << result->val;
    else
        cout << "There is no left leaf in the given tree";

    return 0;
}
```

**Java**

```
// A java program to find
// the deepest left leaf
// in a binary tree

// A Binary Tree node
class Node
{
    int data;
    Node left, right;

    // Constructor
    public Node(int data)
    {
        this.data = data;
        left = right = null;
    }
}

// Class to evaluate pass
// by reference
class Level
```

```
{  
    // maxlevel: gives the  
    // value of level of  
    // maximum left leaf  
    int maxlevel = 0;  
}  
  
class BinaryTree  
{  
    Node root;  
  
    // Node to store resultant  
    // node after left traversal  
    Node result;  
  
    // A utility function to  
    // find deepest leaf node.  
    // lvl: level of current node.  
    // isLeft: A bool indicate  
    // that this node is left child  
    void deepestLeftLeafUtil(Node node,  
                            int lvl,  
                            Level level,  
                            boolean isLeft)  
{  
    // Base case  
    if (node == null)  
        return;  
  
    // Update result if this node  
    // is left leaf and its level  
    // is more than the maxl level  
    // of the current result  
    if (isLeft != false &&  
        node.left == null &&  
        node.right == null &&  
        lvl > level.maxlevel)  
    {  
        result = node;  
        level.maxlevel = lvl;  
    }  
  
    // Recur for left and right subtrees  
    deepestLeftLeafUtil(node.left, lvl + 1,  
                        level, true);  
    deepestLeftLeafUtil(node.right, lvl + 1,  
                        level, false);  
}
```

```
// A wrapper over deepestLeftLeafUtil().
void deepestLeftLeaf(Node node)
{
    Level level = new Level();
    deepestLeftLeafUtil(node, 0, level, false);
}

// Driver program to test above functions
public static void main(String[] args)
{
    BinaryTree tree = new BinaryTree();
    tree.root = new Node(1);
    tree.root.left = new Node(2);
    tree.root.right = new Node(3);
    tree.root.left.left = new Node(4);
    tree.root.right.left = new Node(5);
    tree.root.right.right = new Node(6);
    tree.root.right.left.right = new Node(7);
    tree.root.right.right.right = new Node(8);
    tree.root.right.left.right.left = new Node(9);
    tree.root.right.right.right.right = new Node(10);

    tree.deepestLeftLeaf(tree.root);
    if (tree.result != null)
        System.out.println("The deepest left child"+
                           " is " + tree.result.data);
    else
        System.out.println("There is no left leaf in"+
                           " the given tree");
}
}

// This code has been contributed by Mayank Jaiswal(mayank_24)
```

### Python

```
# Python program to find the deepest left leaf in a given
# Binary tree

# A binary tree node
class Node:

    # Constructor to create a new node
    def __init__(self, val):
        self.val = val
        self.left = None
        self.right = None
```

```
# A utility function to find deepest leaf node.
# lvl: level of current node.
# maxlvl: pointer to the deepest left leaf node found so far
# isLeft: A bool indicate that this node is left child
# of its parent
# resPtr: Pointer to the result
def deepestLeftLeafUtil(root, lvl, maxlvl, isLeft):

    # Base Case
    if root is None:
        return

    # Update result if this node is left leaf and its
    # level is more than the max level of the current result
    if(isLeft is True):
        if (root.left == None and root.right == None):
            if lvl > maxlvl[0] :
                deepestLeftLeafUtil.resPtr = root
                maxlvl[0] = lvl
            return

    # Recur for left and right subtrees
    deepestLeftLeafUtil(root.left, lvl+1, maxlvl, True)
    deepestLeftLeafUtil(root.right, lvl+1, maxlvl, False)

# A wrapper for left and right subtree
def deepestLeftLeaf(root):
    maxlvl = [0]
    deepestLeftLeafUtil.resPtr = None
    deepestLeftLeafUtil(root, 0, maxlvl, False)
    return deepestLeftLeafUtil.resPtr

# Driver program to test above function
root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.right.left = Node(5)
root.right.right = Node(6)
root.right.left.right = Node(7)
root.right.right.right = Node(8)
root.right.left.right.left = Node(9)
root.right.right.right= Node(10)

result = deepestLeftLeaf(root)
```

```
if result is None:  
    print "There is not left leaf in the given tree"  
else:  
    print "The deepest left child is", result.val  
  
# This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

Output:

The deepest left child is 9

Time Complexity: The function does a simple traversal of the tree, so the complexity is  $O(n)$ .

This article is contributed by **Abhay Rathi**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## Source

<https://www.geeksforgeeks.org/deepest-left-leaf-node-in-a-binary-tree/>

## Chapter 122

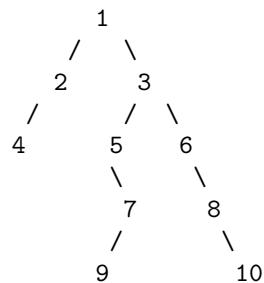
# Deepest left leaf node in a binary tree | iterative approach

Deepest left leaf node in a binary tree | iterative approach - GeeksforGeeks

Given a Binary Tree, find the deepest leaf node that is left child of its parent. For example, consider the following tree. The deepest left leaf node is the node with value 9.

Examples:

Input :



Output : 9

Recursive approach to this problem is discussed [here](#)

For iterative approach, idea is similar to [Method 2 of level order traversal](#)

The idea is to traverse the tree iteratively and whenever a left tree node is pushed to queue, check if it is leaf node, if it's leaf node, then update the result. Since we go level by level, the last stored leaf node is deepest one,

```
// CPP program to find deepest left leaf
```

```
// node of binary tree
#include <bits/stdc++.h>
using namespace std;

// tree node
struct Node {
    int data;
    Node *left, *right;
};

// returns a new tree Node
Node* newNode(int data)
{
    Node* temp = new Node();
    temp->data = data;
    temp->left = temp->right = NULL;
    return temp;
}

// return the deepest left leaf node
// of binary tree
Node* getDeepestLeftLeafNode(Node* root)
{
    if (!root)
        return NULL;

    // create a queue for level order traversal
    queue<Node*> q;
    q.push(root);

    Node* result = NULL;

    // traverse until the queue is empty
    while (!q.empty()) {
        Node* temp = q.front();
        q.pop();

        // Since we go level by level, the last
        // stored left leaf node is deepest one,
        if (temp->left) {
            q.push(temp->left);
            if (!temp->left->left && !temp->left->right)
                result = temp->left;
        }

        if (temp->right)
            q.push(temp->right);
    }
}
```

```
        }
        return result;
    }

// driver program
int main()
{
    // construct a tree
    Node* root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->right->left = newNode(5);
    root->right->right = newNode(6);
    root->right->left->right = newNode(7);
    root->right->right->right = newNode(8);
    root->right->left->right->left = newNode(9);
    root->right->right->right->right = newNode(10);

    Node* result = getDeepestLeftLeafNode(root);
    if (result)
        cout << "Deepest Left Leaf Node :: "
            << result->data << endl;
    else
        cout << "No result, left leaf not found\n";
    return 0;
}
```

Output:

```
Deepest Left Leaf Node :: 9
```

– **Mandeep Singh**

## Source

<https://www.geeksforgeeks.org/deepest-left-leaf-node-binary-tree-iterative-approach/>

## Chapter 123

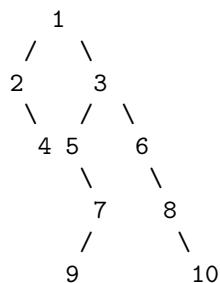
# Deepest right leaf node in a binary tree | Iterative approach

Deepest right leaf node in a binary tree | Iterative approach - GeeksforGeeks

Given a Binary Tree, find the deepest leaf node that is right child of its parent. For example, consider the following tree. The deepest right leaf node is the node with value 10.

Examples:

Input :



Output : 10

The idea is similar to [Method 2 of level order traversal](#)

Traverse the tree level by level and while pushing right child to queue, check if it is leaf node, if it's leaf node, then update the result and since we are traversing level by level, the last stored right leaf will be the deepest right leaf node.

```
// CPP program to find deepest right leaf
// node of binary tree
#include <bits/stdc++.h>
```

```
using namespace std;

// tree node
struct Node {
    int data;
    Node *left, *right;
};

// returns a new tree Node
Node* newNode(int data)
{
    Node* temp = new Node();
    temp->data = data;
    temp->left = temp->right = NULL;
    return temp;
}

// return the deepest right leaf node
// of binary tree
Node* getDeepestRightLeafNode(Node* root)
{
    if (!root)
        return NULL;

    // create a queue for level order traversal
    queue<Node*> q;
    q.push(root);

    Node* result = NULL;

    // traverse until the queue is empty
    while (!q.empty()) {
        Node* temp = q.front();
        q.pop();

        if (temp->left) {
            q.push(temp->left);
        }

        // Since we go level by level, the last
        // stored right leaf node is deepest one
        if (temp->right){
            q.push(temp->right);
            if (!temp->right->left && !temp->right->right)
                result = temp->right;
        }
    }
}
```

```
    return result;
}

// driver program
int main()
{
    // construct a tree
    Node* root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->right = newNode(4);
    root->right->left = newNode(5);
    root->right->right = newNode(6);
    root->right->left->right = newNode(7);
    root->right->right->right = newNode(8);
    root->right->left->right->left = newNode(9);
    root->right->right->right->right = newNode(10);

    Node* result = getDeepestRightLeafNode(root);
    if (result)
        cout << "Deepest Right Leaf Node :: "
            << result->data << endl;
    else
        cout << "No result, right leaf not found\n";
    return 0;
}
```

Output:

```
Deepest Right Leaf Node :: 10
```

Time Complexity : O(n)

– Mandeep Singh

## Source

<https://www.geeksforgeeks.org/deepest-right-leaf-node-binary-tree-iterative-approach/>

## Chapter 124

### Delete leaf nodes with value as x

Delete leaf nodes with value as x - GeeksforGeeks

Given a binary tree and a target integer x, delete all the leaf nodes having value as x. Also, delete the newly formed leaves with the target value as x.

```
Input : x = 5
        6
       /   \
      5     4
     / \   \
    1   2   5
Output :
        6
       /   \
      5     4
     / \   \
    1   2
Inorder Traversal is 1 5 2 6 4
```

Source: [Microsoft Interview](#)

We traverse the tree in postorder fashion and recursively delete the nodes. The approach is very similar to [this](#) and [this](#) problem.

```
// CPP code to delete all leaves with given
// value.
#include <bits/stdc++.h>
using namespace std;

// A binary tree node
struct Node {
```

```
int data;
struct Node *left, *right;
};

// A utility function to allocate a new node
struct Node* newNode(int data)
{
    struct Node* newNode = new Node;
    newNode->data = data;
    newNode->left = newNode->right = NULL;
    return (newNode);
}

Node* deleteLeaves(Node* root, int x)
{
    if (root == NULL)
        return NULL;
    root->left = deleteLeaves(root->left, x);
    root->right = deleteLeaves(root->right, x);

    if (root->data == x && root->left == NULL &&
        root->right == NULL) {
        delete(root);
        return NULL;
    }
    return root;
}

void inorder(Node* root)
{
    if (root == NULL)
        return;
    inorder(root->left);
    cout << root->data << " ";
    inorder(root->right);
}

// Driver program
int main(void)
{
    struct Node* root = newNode(10);
    root->left = newNode(3);
    root->right = newNode(10);
    root->left->left = newNode(3);
    root->left->right = newNode(1);
    root->right->right = newNode(3);
    root->right->right->left = newNode(3);
    root->right->right->right = newNode(3);
}
```

```
deleteLeaves(root, 3);
cout << "Inorder traversal after deletion : ";
inorder(root);
return 0;
}
```

Output:

Inorder traversal after deletion : 3 1 10 10

### Source

<https://www.geeksforgeeks.org/delete-leaf-nodes-value-x/>

## Chapter 125

# Delete leaf nodes with value k

Delete leaf nodes with value k - GeeksforGeeks

Given a binary tree and value k. delete all the leaf nodes with value equal to k. If a node becomes leaf after deletion, then it should be deleted if it has value k.

Examples:

Input :        4  
          /        \  
        5         5  
      / \        /  
    3    1     5

Output :        4  
          /  
        5  
      / \        /  
    3    1

1. Use PostOrder traversal.
2. When we encounter leaf nodes, then we check whether it is leaf node or not.
3. If it is leaf node and value equal to k, then delete it.
4. Else, Recurse for other nodes.

```
// Java program to delete leaf nodes with  
// value equal to k.
```

```
class Node {  
    int data;  
    Node left;  
    Node right;
```

```
public Node(int data)
{
    this.data = data;
    this.left = null;
    this.right = null;
}
}

public class LeafNodesWithValueK {

    // Function to delete leaf Node with value
    // equal to k
    static Node delLeafValueK(Node root, int k)
    {
        if (root == null)
            return null;

        root.left = delLeafValueK(root.left, k);
        root.right = delLeafValueK(root.right, k);

        // If the node is leaf, and its
        // value is equal to k
        if ((root.left == null &&
            root.right == null) &&
            root.data == k)
            return null;

        return root;
    }

    static void postOrder(Node root)
    {
        if (root == null)
            return;
        System.out.print(root.data + " ");
        postOrder(root.left);
        postOrder(root.right);
    }
}

// Driver Code
public static void main(String[] args)
{
    Node root = new Node(4);
    root.left = new Node(5);
    root.right = new Node(5);
    root.left.left = new Node(3);
    root.left.right = new Node(1);
```

```
root.right.left = new Node(5);

System.out.println("Nodes in postorder before deletion");
postOrder(root);
System.out.println();
System.out.println("Nodes in post order after required deletion");
int k = 5;
delLeafValueK(root, k);
postOrder(root);
System.out.println();
}
}
```

**Output:**

```
Nodes in postorder before deletion
4 5 3 1 5 5
Nodes in post order after required deletion
4 5 3 1
```

**Source**

<https://www.geeksforgeeks.org/delete-leaf-nodes-with-value-k/>

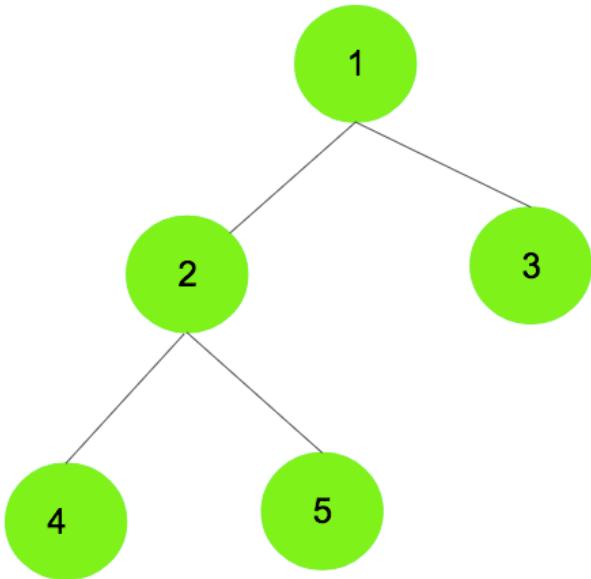
## Chapter 126

# Deleting a binary tree using the delete keyword

Deleting a binary tree using the delete keyword - GeeksforGeeks

A [recursive](#) and a [non-recursive](#) program to delete an entire binary tree has already been discussed in the previous posts. In this post, deleting the entire binary tree using the **delete** keyword in C++ is discussed.

Declare a **destructor** function in the ‘BinaryTreeNode’ class which has been defined to create a tree node. Using ‘delete’ keyword on an object of a class deletes the entire binary tree., it’s destructor is called within the destructor. Use the ‘**delete**’ keyword for the children; so the destructors for the children will be called one by one, and the process will go on recursively until the entire binary tree is deleted. Consider the tree shown below, as soon as the destructor is called for the root i.e., ‘1’, it will call the destructors for ‘2’ and ‘3’, and 2 will then call the same for its left and right child with data ‘4’ and ‘5’ respectively. Eventually, the tree will be deleted in the order: **4->5->2->3->1** (Post-order)



Below is the C++ implementation of the above approach:

```
// C++ program to delete the entire binary
// tree using the delete keyword
#include <iostream>
using namespace std;

class BinaryTreeNode {

    // Making data members public to
    // avoid the usage of getter and setter functions
public:
    int data;
    BinaryTreeNode* left;
    BinaryTreeNode* right;

    // Constructor function to
    // assign data to the node
    BinaryTreeNode(int data)
    {
        this->data = data;
        this->left = NULL;
        this->right = NULL;
    }

    // Destructor function to delete the tree
    ~BinaryTreeNode()
    {
        // using keyword to delete the tree
    }
}
```

```
    delete left;
    delete right;

    // printing the node which has been deleted
    cout << "Deleting " << this->data << endl;
}
};

// Driver Code
int main()
{
    // Creating the nodes dynamically
    BinaryTreeNode* root = new BinaryTreeNode(1);
    BinaryTreeNode* node1 = new BinaryTreeNode(2);
    BinaryTreeNode* node2 = new BinaryTreeNode(3);
    BinaryTreeNode* node3 = new BinaryTreeNode(4);
    BinaryTreeNode* node4 = new BinaryTreeNode(5);

    // Creating the binary tree
    root->left = node1;
    root->right = node2;
    node1->left = node3;
    node1->right = node4;

    // Calls the destructor function which actually deletes the tree entirely
    delete root;

    return 0;
}
```

**Output:**

```
Deleting 4
Deleting 5
Deleting 2
Deleting 3
Deleting 1
```

**Source**

<https://www.geeksforgeeks.org/deleting-a-binary-tree-using-the-delete-keyword/>

## Chapter 127

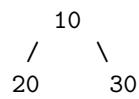
# Deletion in a Binary Tree

Deletion in a Binary Tree - GeeksforGeeks

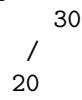
Given a binary tree, delete a node from it by making sure that tree shrinks from the bottom (i.e. the deleted node is replaced by bottom most and rightmost node). This different from [BST deletion](#). Here we do not have any order among elements, so we replace with last element.

Examples :

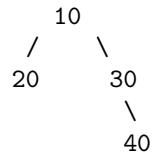
Delete 10 in below tree



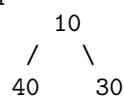
Output :



Delete 20 in below tree



Output :

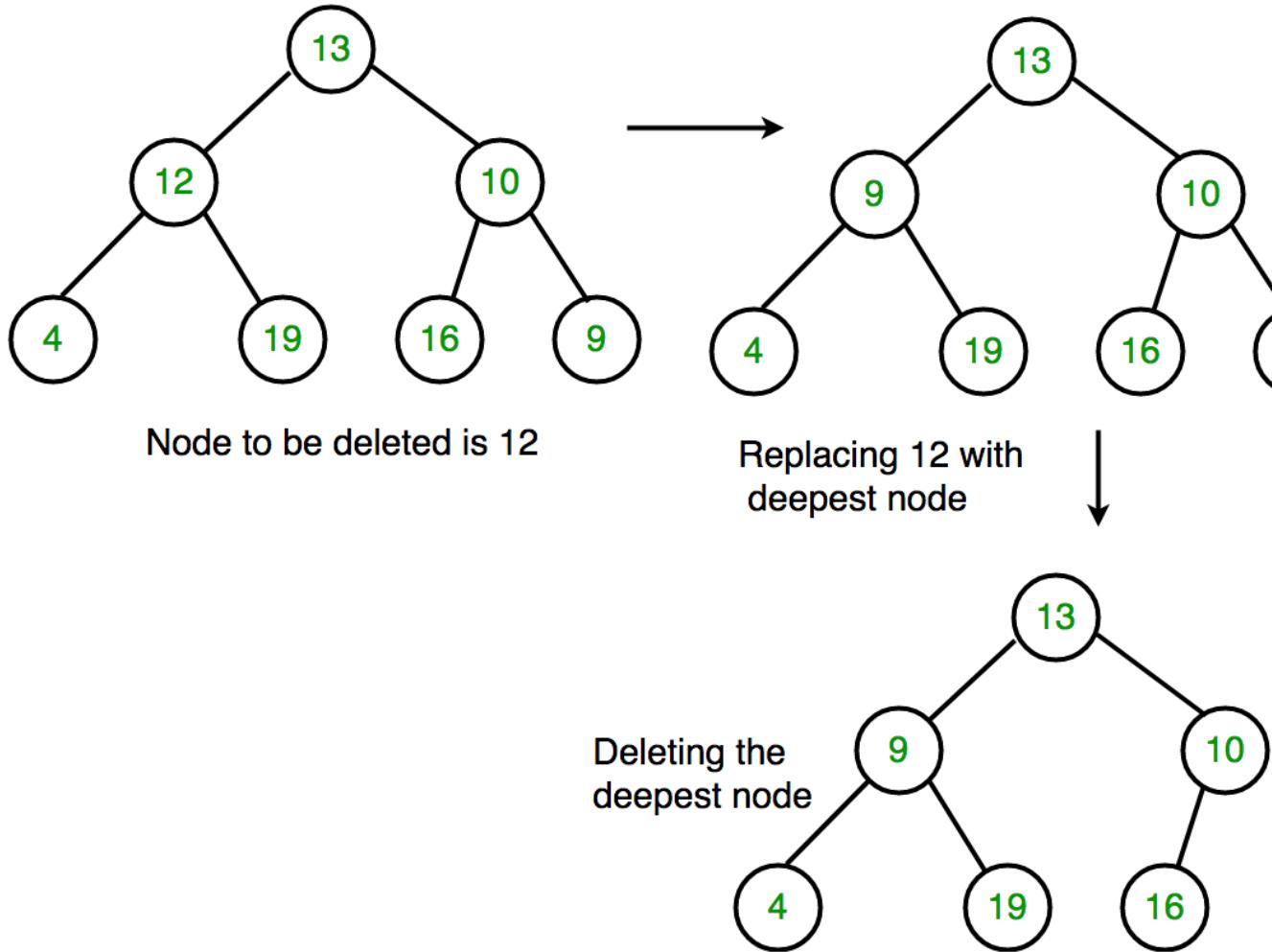


### Algorithm

- Starting at root, find the deepest and rightmost node in binary tree and node which we

want to delete.

2. Replace the deepest rightmost node's data with node to be deleted.
3. Then delete the deepest rightmost node.



```
// C++ program to delete element in binary tree
#include <bits/stdc++.h>
using namespace std;

/* A binary tree node has key, pointer to left
   child and a pointer to right child */
struct Node
{
    int key;
    struct Node* left, *right;
};
```

```
/* function to create a new node of tree and
   return pointer */
struct Node* newNode(int key)
{
    struct Node* temp = new Node;
    temp->key = key;
    temp->left = temp->right = NULL;
    return temp;
}

/* Inorder traversal of a binary tree*/
void inorder(struct Node* temp)
{
    if (!temp)
        return;
    inorder(temp->left);
    cout << temp->key << " ";
    inorder(temp->right);
}

/* function to delete the given deepest node
   (d_node) in binary tree */
void deletDeepest(struct Node *root,
                  struct Node *d_node)
{
    queue<struct Node*> q;
    q.push(root);

    // Do level order traversal until last node
    struct Node* temp;
    while(!q.empty())
    {
        temp = q.front();
        q.pop();

        if (temp->right)
        {
            if (temp->right == d_node)
            {
                temp->right = NULL;
                delete(d_node);
                return;
            }
            else
                q.push(temp->right);
        }
    }
}
```

```
if (temp->left)
{
    if (temp->left == d_node)
    {
        temp->left=NULL;
        delete(d_node);
        return;
    }
    else
        q.push(temp->left);
}
}

/* function to delete element in binary tree */
void deletion(struct Node* root, int key)
{
    queue<struct Node*> q;
    q.push(root);

    struct Node *temp;
    struct Node *key_node = NULL;

    // Do level order traversal to find deepest
    // node(temp) and node to be deleted (key_node)
    while (!q.empty())
    {
        temp = q.front();
        q.pop();

        if (temp->key == key)
            key_node = temp;

        if (temp->left)
            q.push(temp->left);

        if (temp->right)
            q.push(temp->right);
    }

    int x = temp->key;
    deletDeepest(root, temp);
    key_node->key = x;
}

// Driver code
int main()
{
```

```
struct Node* root = newNode(10);
root->left = newNode(11);
root->left->left = newNode(7);
root->left->right = newNode(12);
root->right = newNode(9);
root->right->left = newNode(15);
root->right->right = newNode(8);

cout << "Inorder traversal before deletion : ";
inorder(root);

int key = 11;
deletion(root, key);

cout << endl;
cout << "Inorder traversal after deletion : ";
inorder(root);

return 0;
}
```

Output:

```
Inorder traversal before deletion : 7 11 12 10 15 9 8
Inorder traversal after deletion : 7 8 12 10 15 9
```

**Note:** We can also replace node's data that is to be deleted with any node whose left and right child points to NULL but we only use deepest node in order to maintain the Balance of a binary tree.

**Improved By :** [programmer2k17](#)

## Source

<https://www.geeksforgeeks.org/deletion-binary-tree/>

## Chapter 128

# Density of Binary Tree in One Traversal

Density of Binary Tree in One Traversal - GeeksforGeeks

Given a Binary Tree, find density of it by doing one traversal of it.

Density of Binary Tree = Size / Height

Examples:

Input: Root of following tree  
      10  
      /    \  
    20    30

Output: 1.5  
Height of given tree = 2  
Size of given tree = 3

Input: Root of following tree  
      10  
      /  
    20  
  /  
30  
Output: 1  
Height of given tree = 3  
Size of given tree = 3

Density of a Binary Tree indicates, how balanced Binary Tree is. For example density of a skewed tree is minimum and that of a perfect tree is maximum.

**We strongly recommend you to minimize your browser and try this yourself first.**

Two traversal based approach is very simple. First find the height using one traversal, then find the size using another traversal. Finally return the ratio of two values.

To do it in one traversal, we compute size of Binary Tree while finding its height. Below is C++ implementation.

### C++

```
//C++ program to find density of a binary tree
#include<stdio.h>
#include<stdlib.h>

// A binary tree node
struct Node
{
    int data;
    Node *left, *right;
};

// Helper function to allocates a new node
Node* newNode(int data)
{
    Node* node = new Node;
    node->data = data;
    node->left = node->right = NULL;
    return node;
}

// Function to compute height and
// size of a binary tree
int heighAndSize(Node* node, int &size)
{
    if (node==NULL)
        return 0;

    // compute height of each subtree
    int l = heighAndSize(node->left, size);
    int r = heighAndSize(node->right, size);

    //increase size by 1
    size++;

    //return larger of the two
}
```

```
        return (l > r) ? l + 1 : r + 1;
    }

//function to calculate density of a binary tree
float density(Node* root)
{
    if (root == NULL)
        return 0;

    int size = 0; // To store size

    // Finds height and size
    int _height = heighAndSize(root, size);

    return (float)size/_height;
}

// Driver code to test above methods
int main()
{
    Node* root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);

    printf("Density of given binary tree is %f",
           density(root));

    return 0;
}
```

**Java**

```
// Java program to find density of Binary Tree

// A binary tree node
class Node
{
    int data;
    Node left, right;

    public Node(int data)
    {
        this.data = data;
        left = right = null;
    }
}

// Class to implement pass by reference of size
```

```
class Size
{
    // variable to calculate size of tree
    int size = 0;
}

class BinaryTree
{
    Node root;

    // Function to compute height and
    // size of a binary tree
    int heighAndSize(Node node, Size size)
    {
        if (node == null)
            return 0;

        // compute height of each subtree
        int l = heighAndSize(node.left, size);
        int r = heighAndSize(node.right, size);

        //increase size by 1
        size.size++;

        //return larger of the two
        return (l > r) ? l + 1 : r + 1;
    }

    //function to calculate density of a binary tree
    float density(Node root)
    {
        Size size = new Size();
        if (root == null)
            return 0;

        // Finds height and size
        int _height = heighAndSize(root, size);

        return (float) size.size / _height;
    }

    // Driver code to test above methods
    public static void main(String[] args)
    {
        BinaryTree tree = new BinaryTree();
        tree.root = new Node(1);
        tree.root.left = new Node(2);
        tree.root.right = new Node(3);
```

```
System.out.println("Density of given Binary Tree is : "
+ tree.density(tree.root));
}

// This code has been contributed by Mayank Jaiswal(mayank_24)
```

Output:

```
Density of given binary tree is 1.5
```

**Reference:**

<http://www.eem.anadolu.edu.tr/egermen/EEM%20480/icerik/EEM%20480%20Algorithms%20and%20Complexity%20Week%208.pdf>

This article is contributed by **Aditya Goel**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

**Source**

<https://www.geeksforgeeks.org/density-of-binary-tree-in-one-traversal/>

## Chapter 129

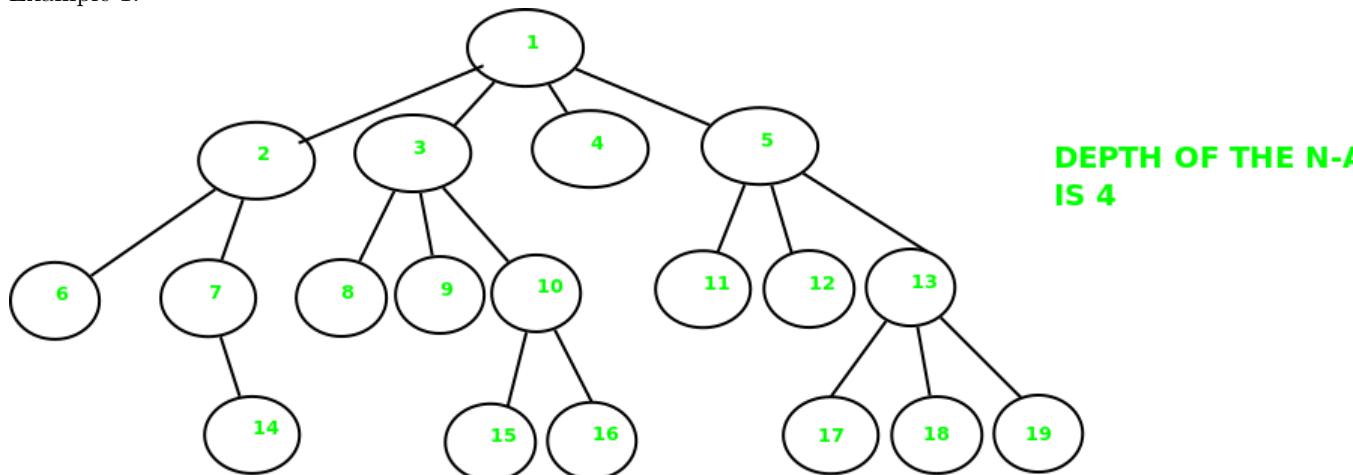
# Depth of an N-Ary tree

Depth of an N-Ary tree - GeeksforGeeks

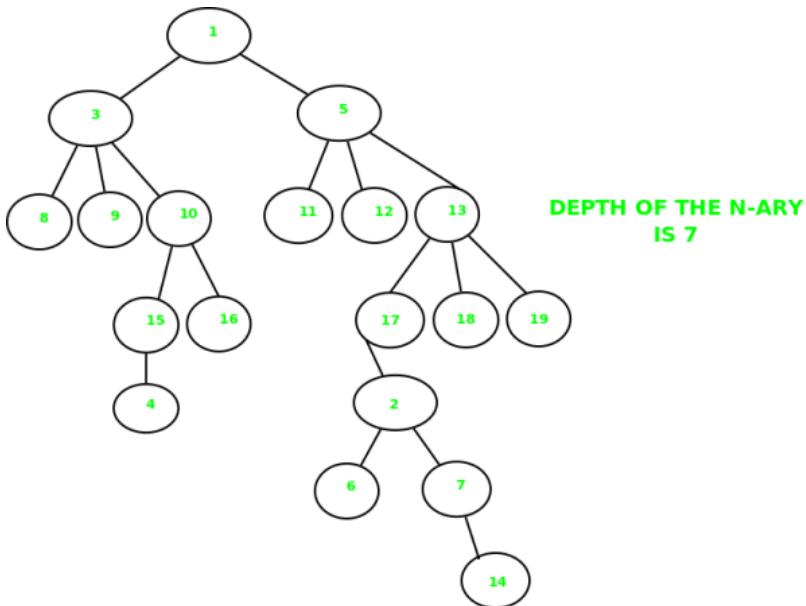
Given an [N-Ary tree](#), find depth of the tree. An N-Ary tree is a tree in which nodes can have **at most N** children.

Examples:

Example 1:



Example 2:



N-Ary tree can be traversed just like a normal tree. We just have to consider all childs of a given node and recursively call that function on every node.

```
// C++ program to find the height of
// an N-ary tree
#include <bits/stdc++.h>
using namespace std;

// Structure of a node of an n-ary tree
struct Node
{
    char key;
    vector<Node *> child;
};

// Utility function to create a new tree node
Node *newNode(int key)
{
    Node *temp = new Node;
    temp->key = key;
    return temp;
}

// Function that will return the depth
// of the tree
int depthOfTree(struct Node *ptr)
{
    // Base case
    if (!ptr)
```

```

        return 0;

    // Check for all children and find
    // the maximum depth
    int maxdepth = 0;
    for (vector<Node*>::iterator it = ptr->child.begin();
         it != ptr->child.end(); it++)
        maxdepth = max(maxdepth, depthOfTree(*it));

    return maxdepth + 1 ;
}

// Driver program
int main()
{
    /* Let us create below tree
     *          A
     *        /   \   \
     *      B     F   D   E
     *      / \   |   / \ \
     *    K   J   G   C   H   I
     *    / \           \
     *  N   M           L
    */

    Node *root = newNode('A');
    (root->child).push_back(newNode('B'));
    (root->child).push_back(newNode('F'));
    (root->child).push_back(newNode('D'));
    (root->child).push_back(newNode('E'));
    (root->child[0]->child).push_back(newNode('K'));
    (root->child[0]->child).push_back(newNode('J'));
    (root->child[2]->child).push_back(newNode('G'));
    (root->child[3]->child).push_back(newNode('C'));
    (root->child[3]->child).push_back(newNode('H'));
    (root->child[3]->child).push_back(newNode('I'));
    (root->child[0]->child[0]->child).push_back(newNode('N'));
    (root->child[0]->child[0]->child).push_back(newNode('M'));
    (root->child[3]->child[2]->child).push_back(newNode('L'));

    cout << depthOfTree(root) << endl;

    return 0;
}

```

Output:

## Source

<https://www.geeksforgeeks.org/depth-n-ary-tree/>

## Chapter 130

# Depth of the deepest odd level node in Binary Tree

Depth of the deepest odd level node in Binary Tree - GeeksforGeeks

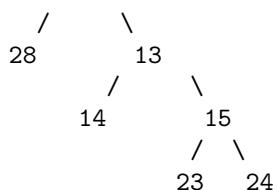
Given a Binary tree, find out depth of the deepest odd level leaf node. Take root level as depth 1.

Examples:

Input :

Output : 5

Input : 10



Output : 3

We can traverse the tree starting from the root level and keep *curr\_level* of the node. Increment the *curr\_level* each time we go to left or a right subtree.

Return the max depth of an odd level,if it exists.

Algorithm:

```
1) return 0 if curr_node == NULL
```

```
2) if curr_node is leaf and curr_level is odd,
   return curr_level
3) else maximum(depthOdd(left subtree),
                depthOdd(right subtree))
```

Below is C++ implementation.

```
// C++ program to find depth of the deepest
// odd level node
#include<bits/stdc++.h>
using namespace std;

// A Tree node
struct Node
{
    int key;
    struct Node *left, *right;
};

// Utility function to create a new node
Node* newNode(int key)
{
    Node* temp = new Node;
    temp->key = key;
    temp->left = temp->right = NULL;
    return (temp);
}

// Utility function which
// returns whether the current node
// is a leaf or not
bool isleaf(Node *curr_node)
{
    return (curr_node->left == NULL &&
            curr_node->right == NULL);
}

// function to return the longest
// odd level depth if it exists
// otherwise 0
int deepestOddLevelDepthUtil(Node *curr_node, int curr_level)
{
    // Base case
    // return from here
    if ( curr_node == NULL)
        return 0;

    // increment current level
```

```
curr_level += 1;

// if curr_level is odd
// and its a leaf node
if ( curr_level % 2 != 0 && isleaf(curr_node))
    return curr_level;

return max(deepestOddLevelDepthUtil(curr_node->left,curr_level),
           deepestOddLevelDepthUtil(curr_node->right,curr_level));
}

// A wrapper over deepestOddLevelDepth()
int deepestOddLevelDepth(Node *curr_node)
{
    return deepestOddLevelDepthUtil(curr_node, 0);
}

int main()
{
    /*      10
         /     \
        28      13
       /       \
      14      15
     /   \
    23   24

    Let us create Binary Tree shown in above example */
    Node *root = newNode(10);
    root->left = newNode(28);
    root->right = newNode(13);

    root->right->left = newNode(14);
    root->right->right = newNode(15);

    root->right->right->left = newNode(23);
    root->right->right->right = newNode(24);

    cout << deepestOddLevelDepth(root) << endl;
}

return 0;
}
```

Output:

**Source**

<https://www.geeksforgeeks.org/depth-deepest-odd-level-node-binary-tree/>

## Chapter 131

# Diagonal Sum of a Binary Tree

Diagonal Sum of a Binary Tree - GeeksforGeeks

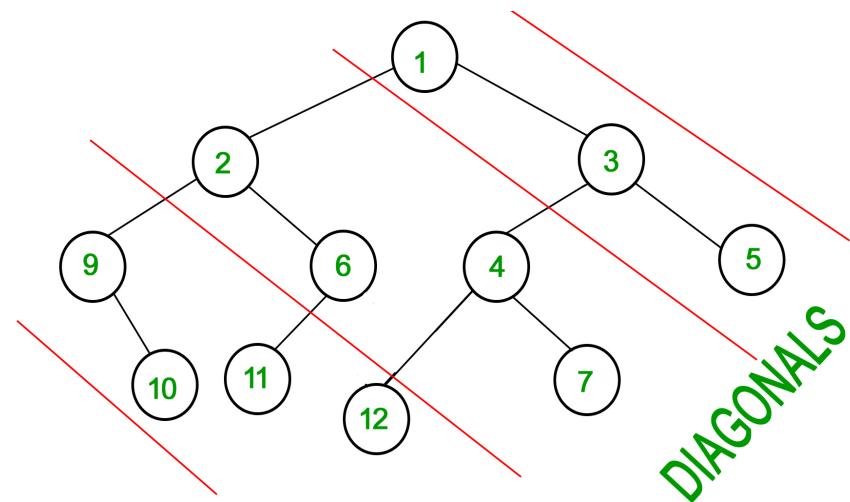
Consider lines of slope -1 passing between nodes (dotted lines in below diagram). Diagonal sum in a binary tree is sum of all node's data lying between these lines. Given a Binary Tree, print all diagonal sums.

For the following input tree, output should be 9, 19, 42.

9 is sum of 1, 3 and 5.

19 is sum of 2, 6, 4 and 7.

42 is sum of 9, 10, 11 and 12.



### Algorithm:

The idea is to keep track of vertical distance from top diagonal passing through root. We increment the vertical distance we go down to next diagonal.

1. Add root with vertical distance as 0 to the queue.
2. Process the sum of all right child and right of right child and so on.
3. Add left child current node into the queue for later processing. The vertical distance of

left child is vertical distance of current node plus 1.

4. Keep doing 2nd, 3rd and 4th step till the queue is empty.

Following is the implementation of above idea.

C++

```
// C++ Program to find diagonal
// sum in a Binary Tree
#include <iostream>
#include <stdlib.h>
#include <map>
using namespace std;

struct Node
{
    int data;
    struct Node* left;
    struct Node* right;
};

struct Node* newNode(int data)
{
    struct Node* Node =
        (struct Node*)malloc(sizeof(struct Node));

    Node->data = data;
    Node->left = Node->right = NULL;

    return Node;
}

// root - root of the binary tree
// vd - vertical distance diagonally
// diagonalSum - map to store Diagonal
// Sum(Passed by Reference)
void diagonalSumUtil(struct Node* root,
                     int vd, map<int, int> &diagonalSum)
{
    if(!root)
        return;

    diagonalSum[vd] += root->data;

    // increase the vertical distance if left child
    diagonalSumUtil(root->left, vd + 1, diagonalSum);

    // vertical distance remains same for right child
    diagonalSumUtil(root->right, vd, diagonalSum);
}
```

```
}

// Function to calculate diagonal
// sum of given binary tree
void diagonalSum(struct Node* root)
{

    // create a map to store Diagonal Sum
    map<int, int> diagonalSum;

    diagonalSumUtil(root, 0, diagonalSum);

    map<int, int>::iterator it;
    cout << "Diagonal sum in a binary tree is - ";

    for(it = diagonalSum.begin();
        it != diagonalSum.end(); ++it)
    {
        cout << it->second << " ";
    }
}

// Driver code
int main()
{
    struct Node* root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(9);
    root->left->right = newNode(6);
    root->right->left = newNode(4);
    root->right->right = newNode(5);
    root->right->left->right = newNode(7);
    root->right->left->left = newNode(12);
    root->left->right->left = newNode(11);
    root->left->left->right = newNode(10);

    diagonalSum(root);

    return 0;
}

// This code is contributed by Aditya Goel
```

### Java

```
// Java Program to find diagonal sum in a Binary Tree
import java.util.*;
```

```
import java.util.Map.Entry;

//Tree node
class TreeNode
{
    int data; //node data
    int vd; //vertical distance diagonally
    TreeNode left, right; //left and right child's reference

    // Tree node constructor
    public TreeNode(int data)
    {
        this.data = data;
        vd = Integer.MAX_VALUE;
        left = right = null;
    }
}

// Tree class
class Tree
{
    TreeNode root;//Tree root

    // Tree constructor
    public Tree(TreeNode root) { this.root = root; }

    // Diagonal sum method
    public void diagonalSum()
    {
        // Queue which stores tree nodes
        Queue<TreeNode> queue = new LinkedList<TreeNode>();

        // Map to store sum of node's data lying diagonally
        Map<Integer, Integer> map = new TreeMap<>();

        // Assign the root's vertical distance as 0.
        root.vd = 0;

        // Add root node to the queue
        queue.add(root);

        // Loop while the queue is not empty
        while (!queue.isEmpty())
        {
            // Remove the front tree node from queue.
            TreeNode curr = queue.remove();

            // Get the vertical distance of the dequeued node.
```

```
int vd = curr.vd;

// Sum over this node's right-child, right-of-right-child
// and so on
while (curr != null)
{
    int prevSum = (map.get(vd) == null)? 0: map.get(vd);
    map.put(vd, prevSum + curr.data);

    // If for any node the left child is not null add
    // it to the queue for future processing.
    if (curr.left != null)
    {
        curr.left.vd = vd+1;
        queue.add(curr.left);
    }

    // Move to the current node's right child.
    curr = curr.right;
}

// Make an entry set from map.
Set<Entry<Integer, Integer>> set = map.entrySet();

// Make an iterator
Iterator<Entry<Integer, Integer>> iterator = set.iterator();

// Traverse the map elements using the iterator.
System.out.print("Diagonal sum in a binary tree is - ");
while (iterator.hasNext())
{
    Map.Entry<Integer, Integer> me = iterator.next();

    System.out.print(me.getValue()+" ");
}
}

//Driver class
public class DiagonalSum
{
    public static void main(String[] args)
    {
        TreeNode root = new TreeNode(1);
        root.left = new TreeNode(2);
        root.right = new TreeNode(3);
        root.left.left = new TreeNode(9);
```

```
root.left.right = new TreeNode(6);
root.right.left = new TreeNode(4);
root.right.right = new TreeNode(5);
root.right.left.left = new TreeNode(12);
root.right.left.right = new TreeNode(7);
root.left.right.left = new TreeNode(11);
root.left.left.right = new TreeNode(10);
Tree tree = new Tree(root);
tree.diagonalSum();
}
}
```

Output:

```
Diagonal sum in a binary tree is - 9 19 42
```

**Exercise:**

This problem was for diagonals from top to bottom and slope -1. Try the same problem for slope +1.

This article is contributed by **Kumar Gautam**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

**Source**

<https://www.geeksforgeeks.org/diagonal-sum-binary-tree/>

## Chapter 132

# Diagonal Traversal of Binary Tree

Diagonal Traversal of Binary Tree - GeeksforGeeks

Consider lines of slope -1 passing between nodes. Given a Binary Tree, print all diagonal elements in a binary tree belonging to same line.

Input : Root of below tree

Output :

Diagonal Traversal of binary tree :  
8 10 14  
3 6 7 13  
1 4

The idea is to use map. We use different slope distances and use them as key in map. Value in map is vector (or dynamic array) of nodes. We traverse the tree to store values in map. Once map is built, we print contents of it.

Below is implementation of above idea.

C++

```
// C++ program for diagonal traversal of Binary Tree
#include <bits/stdc++.h>
using namespace std;

// Tree node
struct Node
{
```

```

        int data;
        Node *left, *right;
    };

/* root - root of the binary tree
   d - distance of current line from rightmost
       -topmost slope.
   diagonalPrint - multimap to store Diagonal
                   elements (Passed by Reference) */
void diagonalPrintUtil(Node* root, int d,
                      map<int, vector<int>> &diagonalPrint)
{
    // Base case
    if (!root)
        return;

    // Store all nodes of same line together as a vector
    diagonalPrint[d].push_back(root->data);

    // Increase the vertical distance if left child
    diagonalPrintUtil(root->left, d + 1, diagonalPrint);

    // Vertical distance remains same for right child
    diagonalPrintUtil(root->right, d, diagonalPrint);
}

// Print diagonal traversal of given binary tree
void diagonalPrint(Node* root)
{
    // create a map of vectors to store Diagonal elements
    map<int, vector<int> > diagonalPrint;
    diagonalPrintUtil(root, 0, diagonalPrint);

    cout << "Diagonal Traversal of binary tree : n";
    for (auto it = diagonalPrint.begin();
         it != diagonalPrint.end(); ++it)
    {
        for (auto itr = it->second.begin();
             itr != it->second.end(); ++itr)
            cout << *itr << ' ';

        cout << 'n';
    }
}

// Utility method to create a new node
Node* newNode(int data)
{

```

```
Node* node = new Node;
node->data = data;
node->left = node->right = NULL;
return node;
}

// Driver program
int main()
{
    Node* root = newNode(8);
    root->left = newNode(3);
    root->right = newNode(10);
    root->left->left = newNode(1);
    root->left->right = newNode(6);
    root->right->right = newNode(14);
    root->right->right->left = newNode(13);
    root->left->right->left = newNode(4);
    root->left->right->right = newNode(7);

/*  Node* root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(9);
    root->left->right = newNode(6);
    root->right->left = newNode(4);
    root->right->right = newNode(5);
    root->right->left->right = newNode(7);
    root->right->left->left = newNode(12);
    root->left->right->left = newNode(11);
    root->left->left->right = newNode(10); */

diagonalPrint(root);

    return 0;
}
```

### Java

```
// Java program for diagonal traversal of Binary Tree
import java.util.HashMap;
import java.util.Map.Entry;
import java.util.Vector;

public class DiagonalTraversalBTree
{
    // Tree node
    static class Node{
        int data;
```

```
Node left;
Node right;

//constructor
Node(int data)
{
    this.data=data;
    left = null;
    right =null;
}

/*
 * root - root of the binary tree
 * d - distance of current line from rightmost
 * -topmost slope.
 * diagonalPrint - HashMap to store Diagonal
 * elements (Passed by Reference) */
static void diagonalPrintUtil(Node root,int d,
    HashMap<Integer,Vector<Integer>> diagonalPrint){

    // Base case
    if (root == null)
        return;

    // get the list at the particular d value
    Vector<Integer> k = diagonalPrint.get(d);

    // k is null then create a vector and store the data
    if (k == null)
    {
        k = new Vector<>();
        k.add(root.data);
    }

    // k is not null then update the list
    else
    {
        k.add(root.data);
    }

    // Store all nodes of same line together as a vector
    diagonalPrint.put(d,k);

    // Increase the vertical distance if left child
    diagonalPrintUtil(root.left, d + 1, diagonalPrint);

    // Vertical distance remains same for right child
    diagonalPrintUtil(root.right, d, diagonalPrint);
}
```

```
}

// Print diagonal traversal of given binary tree
static void diagonalPrint(Node root)
{
    // create a map of vectors to store Diagonal elements
    HashMap<Integer,Vector<Integer>> diagonalPrint = new HashMap<>();
    diagonalPrintUtil(root, 0, diagonalPrint);

    System.out.println("Diagonal Traversal of Binnary Tree");
    for (Entry<Integer, Vector<Integer>> entry : diagonalPrint.entrySet())
    {
        System.out.println(entry.getValue());
    }
}

// Driver program
public static void main(String[] args) {

    Node root = new Node(8);
    root.left = new Node(3);
    root.right = new Node(10);
    root.left.left = new Node(1);
    root.left.right = new Node(6);
    root.right.right = new Node(14);
    root.right.right.left = new Node(13);
    root.left.right.left = new Node(4);
    root.left.right.right = new Node(7);

    diagonalPrint(root);
}
}

// This code is contributed by Sumit Ghosh
```

### Python

```
# Python program for diagonal traversal of Binary Tree

# A binary tree node
class Node:

    # Constructor to create a new binary tree node
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None
```

```
""" root - root of the binary tree
d - distance of current line from rightmost
    -topmost slope.
diagonalPrint - multimap to store Diagonal
                elements (Passed by Reference) """
def diagonalPrintUtil(root, d, diagonalPrintMap):

    # Base Case
    if root is None:
        return

    # Store all nodes of same line together as a vector
    try :
        diagonalPrintMap[d].append(root.data)
    except KeyError:
        diagonalPrintMap[d] = [root.data]

    # Increase the vertical distance if left child
    diagonalPrintUtil(root.left, d+1, diagonalPrintMap)

    # Vertical distance remains same for right child
    diagonalPrintUtil(root.right, d, diagonalPrintMap)

# Print diagonal traversal of given binary tree
def diagonalPrint(root):

    # Create a dict to store diagonal elements
    diagonalPrintMap = dict()

    # Find the diagonal traversal
    diagonalPrintUtil(root, 0, diagonalPrintMap)

    print "Diagonal Traversal of binary tree : "
    for i in diagonalPrintMap:
        for j in diagonalPrintMap[i]:
            print j,
        print ""

# Driver Program
root = Node(8)
root.left = Node(3)
root.right = Node(10)
root.left.left = Node(1)
root.left.right = Node(6)
root.right.right = Node(14)
```

```
root.right.right.left = Node(13)
root.left.right.left = Node(4)
root.left.right.right = Node(7)

diagonalPrint(root)

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

Output :

```
Diagonal Traversal of binary tree :
8 10 14
3 6 7 13
1 4
```

This article is contributed by **Aditya Goel**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

**Improved By :** [Sachin Verma 7](#)

## Source

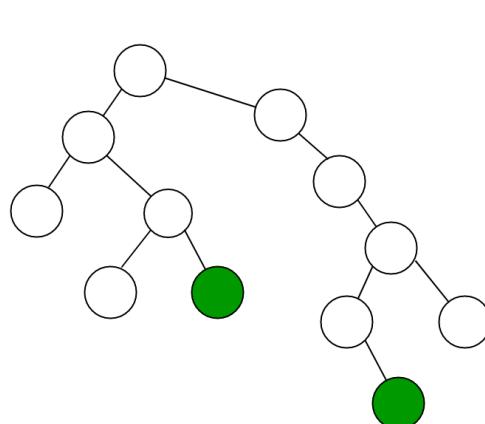
<https://www.geeksforgeeks.org/diagonal-traversal-of-binary-tree/>

## Chapter 133

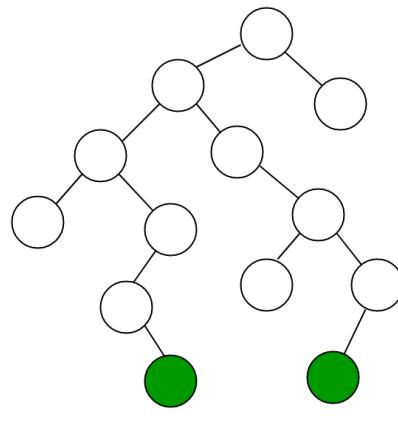
# Diameter of a Binary Tree

Diameter of a Binary Tree - GeeksforGeeks

The diameter of a tree (sometimes called the width) is the number of nodes on the longest path between two end nodes. The diagram below shows two trees each with diameter nine, the leaves that form the ends of a longest path are shaded (note that there is more than one path in each tree of length nine, but no path longer than nine nodes).



Diameter 9 nodes through root



Diameter 9 nodes not through root

The diameter of a tree T is the largest of the following quantities:

- \* the diameter of T's left subtree
- \* the diameter of T's right subtree
- \* the longest path between leaves that goes through the root of T (this can be computed from the heights of the subtrees of T)

Implementation:

C

```
#include <stdio.h>
#include <stdlib.h>

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
    int data;
    struct node* left, *right;
};

/* function to create a new node of tree and returns pointer */
struct node* newNode(int data);

/* returns max of two integers */
int max(int a, int b);

/* function to Compute height of a tree. */
int height(struct node* node);

/* Function to get diameter of a binary tree */
int diameter(struct node * tree)
{
    /* base case where tree is empty */
    if (tree == NULL)
        return 0;

    /* get the height of left and right sub-trees */
    int lheight = height(tree->left);
    int rheight = height(tree->right);

    /* get the diameter of left and right sub-trees */
    int ldiameter = diameter(tree->left);
    int rdiameter = diameter(tree->right);

    /* Return max of following three
     1) Diameter of left subtree
     2) Diameter of right subtree
     3) Height of left subtree + height of right subtree + 1 */
    return max(lheight + rheight + 1, max(ldiameter, rdiameter));
}

/* UTILITY FUNCTIONS TO TEST diameter() FUNCTION */

/* The function Compute the "height" of a tree. Height is the
   number f nodes along the longest path from the root node
   down to the farthest leaf node.*/
int height(struct node* node)
```

```

{
    /* base case tree is empty */
    if(node == NULL)
        return 0;

    /* If tree is not empty then height = 1 + max of left
       height and right heights */
    return 1 + max(height(node->left), height(node->right));
}

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* node = (struct node*)
                           malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;

    return(node);
}

/* returns maximum of two integers */
int max(int a, int b)
{
    return (a >= b)? a: b;
}

/* Driver program to test above functions*/
int main()
{

    /* Constructed binary tree is
           1
         /   \
        2     3
       / \   /
      4   5
    */
    struct node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);

    printf("Diameter of the given binary tree is %d\n", diameter(root));
}

```

```
    getchar();
    return 0;
}
```

**Java**

```
// Recursive optimized Java program to find the diameter of a
// Binary Tree

/* Class containing left and right child of current
node and key value*/
class Node
{
    int data;
    Node left, right;

    public Node(int item)
    {
        data = item;
        left = right = null;
    }
}

/* Class to print the Diameter */
class BinaryTree
{
    Node root;

    /* Method to calculate the diameter and return it to main */
    int diameter(Node root)
    {
        /* base case if tree is empty */
        if (root == null)
            return 0;

        /* get the height of left and right sub trees */
        int lheight = height(root.left);
        int rheight = height(root.right);

        /* get the diameter of left and right subtrees */
        int ldiameter = diameter(root.left);
        int rdiameter = diameter(root.right);

        /* Return max of following three
           1) Diameter of left subtree
           2) Diameter of right subtree
           3) Height of left subtree + height of right subtree + 1 */
        return Math.max(lheight + rheight + 1,
```

```

        Math.max(ldiameter, rdiameter));

    }

/* A wrapper over diameter(Node root) */
int diameter()
{
    return diameter(root);
}

/*The function Compute the "height" of a tree. Height is the
 number f nodes along the longest path from the root node
 down to the farthest leaf node.*/
static int height(Node node)
{
    /* base case tree is empty */
    if (node == null)
        return 0;

    /* If tree is not empty then height = 1 + max of left
       height and right heights */
    return (1 + Math.max(height(node.left), height(node.right)));
}

public static void main(String args[])
{
    /* creating a binary tree and entering the nodes */
    BinaryTree tree = new BinaryTree();
    tree.root = new Node(1);
    tree.root.left = new Node(2);
    tree.root.right = new Node(3);
    tree.root.left.left = new Node(4);
    tree.root.left.right = new Node(5);

    System.out.println("The diameter of given binary tree is : "
                       + tree.diameter());
}
}

```

### Python

```

# Python program to find the diameter of binary tree

# A binary tree node
class Node:

    # Constructor to create a new node
    def __init__(self, data):

```

```

        self.data = data
        self.left = None
        self.right = None

"""
The function Compute the "height" of a tree. Height is the
number f nodes along the longest path from the root node
down to the farthest leaf node.
"""
def height(node):

    # Base Case : Tree is empty
    if node is None:
        return 0 ;

    # If tree is not empty then height = 1 + max of left
    # height and right heights
    return 1 + max(height(node.left) ,height(node.right))

# Function to get the diamtere of a binary tree
def diameter(root):

    # Base Case when tree is empty
    if root is None:
        return 0;

    # Get the height of left and right sub-trees
    lheight = height(root.left)
    rheight = height(root.right)

    # Get the diameter of left and irgh sub-trees
    ldiameter = diameter(root.left)
    rdiameter = diameter(root.right)

    # Return max of the following tree:
    # 1) Diameter of left subtree
    # 2) Diameter of right subtree
    # 3) Height of left subtree + height of right subtree +1
    return max(lheight + rheight + 1, max(ldiameter, rdiameter))

# Driver program to test above functions

"""
Constructed binary tree is
      1
     / \

```

```

      2      3
     / \ 
    4   5
"""

root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)
print "Diameter of given binary tree is %d" %(diameter(root))

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)

```

Time Complexity:  $O(n^2)$

Output:

```
Diameter of the given binary tree is 4
```

**Optimized implementation:** The above implementation can be optimized by calculating the height in the same recursion rather than calling a `height()` separately. Thanks to Amar for suggesting this optimized version. This optimization reduces time complexity to  $O(n)$ .

C

```

/*The second parameter is to store the height of tree.
Initially, we need to pass a pointer to a location with value
as 0. So, function should be used as follows:

int height = 0;
struct node *root = SomeFunctionToMakeTree();
int diameter = diameterOpt(root, &height); */
int diameterOpt(struct node *root, int* height)
{
    /* lh --> Height of left subtree
       rh --> Height of right subtree */
    int lh = 0, rh = 0;

    /* ldiameter --> diameter of left subtree
       rdiameter --> Diameter of right subtree */
    int ldiameter = 0, rdiameter = 0;

    if(root == NULL)
    {
        *height = 0;
        return 0; /* diameter is also 0 */
    }
}
```

```
/* Get the heights of left and right subtrees in lh and rh
   And store the returned values in ldiameter and rdiameter */
ldiameter = diameterOpt(root->left, &lh);
rdiameter = diameterOpt(root->right, &rh);

/* Height of current node is max of heights of left and
   right subtrees plus 1*/
*height = max(lh, rh) + 1;

return max(lh + rh + 1, max(ldiameter, rdiameter));
}
```

Java

```
// Recursive Java program to find the diameter of a
// Binary Tree

/* Class containing left and right child of current
   node and key value*/
class Node
{
    int data;
    Node left, right;

    public Node(int item)
    {
        data = item;
        left = right = null;
    }
}

// A utility class to pass height object
class Height
{
    int h;
}

/* Class to print the Diameter */
class BinaryTree
{
    Node root;

    /* Define height = 0 globally and call diameterOpt(root, height)
       from main */
    int diameterOpt(Node root, Height height)
    {
        /* lh --> Height of left subtree
```

```

    rh --> Height of right subtree */
Height lh = new Height(), rh = new Height();

if (root == null)
{
    height.h = 0;
    return 0; /* diameter is also 0 */
}

/* ldiameter --> diameter of left subtree
   rdiameter --> Diameter of right subtree */
/* Get the heights of left and right subtrees in lh and rh
   And store the returned values in ldiameter and ldiameter */
int ldiameter = diameterOpt(root.left, lh);
int rdiameter = diameterOpt(root.right, rh);

/* Height of current node is max of heights of left and
   right subtrees plus 1*/
height.h = Math.max(lh.h, rh.h) + 1;

return Math.max(lh.h + rh.h + 1, Math.max(ldiameter, rdiameter));
}

/* A wrapper over diameter(Node root) */
int diameter()
{
    Height height = new Height();
    return diameterOpt(root, height);
}

/*The function Compute the "height" of a tree. Height is the
   number f nodes along the longest path from the root node
   down to the farthest leaf node.*/
static int height(Node node)
{
    /* base case tree is empty */
    if (node == null)
        return 0;

    /* If tree is not empty then height = 1 + max of left
       height and right heights */
    return (1 + Math.max(height(node.left), height(node.right)));
}

public static void main(String args[])
{
    /* creating a binary tree and entering the nodes */
    BinaryTree tree = new BinaryTree();
}

```

```
tree.root = new Node(1);
tree.root.left = new Node(2);
tree.root.right = new Node(3);
tree.root.left.left = new Node(4);
tree.root.left.right = new Node(5);

System.out.println("The diameter of given binary tree is : "
+ tree.diameter());
}
}
```

Time Complexity: O(n)

Output:

4

1. [Diameter of a Binary Tree in O\(n\) \[A new method\]](#)
2. [Diameter of an N-ary tree](#)

References:

<http://www.cs.duke.edu/courses/spring00/cps100/assign/trees/diameter.html>

## Source

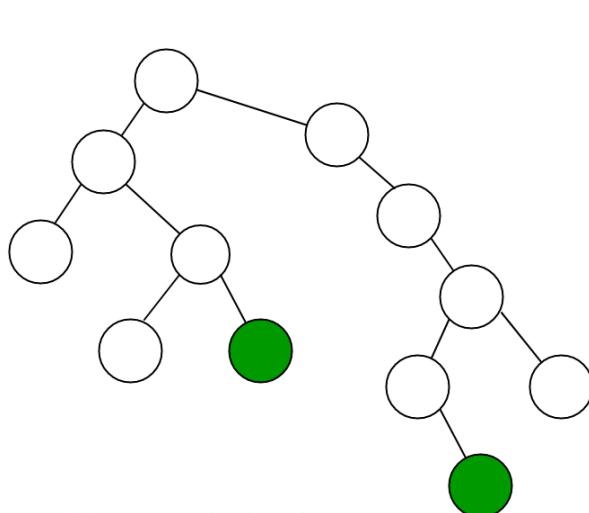
<https://www.geeksforgeeks.org/diameter-of-a-binary-tree/>

## Chapter 134

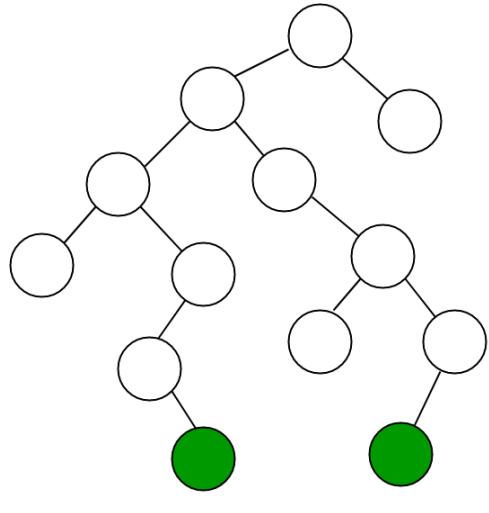
# Diameter of a Binary Tree in O(n) [A new method]

Diameter of a Binary Tree in O(n) [A new method] - GeeksforGeeks

The diameter of a tree is the number of nodes on the longest path between two leaves in the tree. The diagram below shows two trees each with diameter nine, the leaves that form the ends of a longest path are colored (note that there may be more than one path in tree of same diameter).



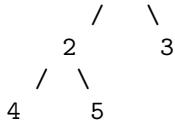
Diameter 9 nodes through root



Diameter 9 nodes not through root

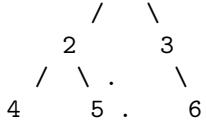
Examples:

Input : 1



Output : 4

Input : 1



Output : 5

We have discussed a solution in below post.

### Diameter of a Binary Tree

In this post a new simple O(n) method is discussed. Diameter of a tree can be calculated by only using the height function, because the diameter of a tree is nothing but maximum value of ( $\text{left\_height} + \text{right\_height} + 1$ ) for each node. So we need to calculate this value ( $\text{left\_height} + \text{right\_height} + 1$ ) for each node and update the result. Time complexity – O(n)

```

// Simple C++ program to find diameter
// of a binary tree.
#include <bits/stdc++.h>
using namespace std;

/* Tree node structure used in the program */
struct Node {
    int data;
    Node* left, *right;
};

/* Function to find height of a tree */
int height(Node* root, int& ans)
{
    if (root == NULL)
        return 0;

    int left_height = height(root->left, ans);

    int right_height = height(root->right, ans);

    // update the answer, because diameter of a
    // tree is nothing but maximum value of
    // (left_height + right_height + 1) for each node
  
```

```
ans = max(ans, 1 + left_height + right_height);

    return 1 + max(left_height, right_height);
}

/* Computes the diameter of binary tree with given root. */
int diameter(Node* root)
{
    if (root == NULL)
        return 0;
    int ans = INT_MIN; // This will store the final answer
    int height_of_tree = height(root, ans);
    return ans;
}

struct Node* newNode(int data)
{
    struct Node* node = new Node;
    node->data = data;
    node->left = node->right = NULL;

    return (node);
}

// Driver code
int main()
{
    struct Node* root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);

    printf("Diameter is %d\n", diameter(root));

    return 0;
}
```

Output:

Diameter is 4

## Source

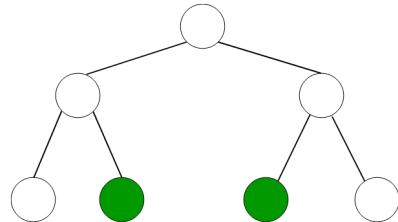
<https://www.geeksforgeeks.org/diameter-of-a-binary-tree-in-on-a-new-method/>

## Chapter 135

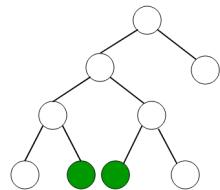
# Diameter of a tree using DFS

Diameter of a tree using DFS - GeeksforGeeks

The diameter of a tree (sometimes called the width) is the number of nodes on the longest path between two leaves in the tree. The diagram below shows two trees each with diameter five, the leaves that form the ends of the longest path are shaded (note that there is more than one path in each tree of length five, but no path longer than five nodes)



Diameter, 5 nodes through root



Diameter, 5 nodes not through root

We have discussed a solution in below post

### Diameter of a binary tree

In this post a different DFS based solution is discussed. After observing above tree we can see that the longest path will always occur between two leaf nodes. We start DFS from a random node and then see which node is farthest from it. Let the node farthest be X. It is clear that X will always be a leaf node and a corner of DFS. Now if we start DFS from X

and check the farthest node from it, we will get the diameter of the tree.  
The C++ implementation uses adjacency list representation of graphs. STL's list container is used to store lists of adjacent nodes.

### C++

```
// C++ program to find diameter of a binary tree
// using DFS.
#include <iostream>
#include <limits.h>
#include <list>
using namespace std;

// Used to track farthest node.
int x;

// Sets maxCount as maximum distance from node.
void dfsUtil(int node, int count, bool visited[],
             int& maxCount, list<int>* adj)
{
    visited[node] = true;
    count++;
    for (auto i = adj[node].begin(); i != adj[node].end(); ++i) {
        if (!visited[*i]) {
            if (count >= maxCount) {
                maxCount = count;
                x = *i;
            }
            dfsUtil(*i, count, visited, maxCount, adj);
        }
    }
}

// The function to do DFS traversal. It uses recursive
// dfsUtil()
void dfs(int node, int n, list<int>* adj, int& maxCount)
{
    bool visited[n + 1];
    int count = 0;

    // Mark all the vertices as not visited
    for (int i = 1; i <= n; ++i)
        visited[i] = false;

    // Increment count by 1 for visited node
    dfsUtil(node, count + 1, visited, maxCount, adj);
}
```

```
// Returns diameter of binary tree represented
// as adjacency list.
int diameter(list<int>* adj, int n)
{
    int maxCount = INT_MIN;

    /* DFS from a random node and then see
    farthest node X from it*/
    dfs(1, n, adj, maxCount);

    /* DFS from X and check the farthest node
    from it */
    dfs(x, n, adj, maxCount);

    return maxCount;
}

/* Driver program to test above functions*/
int main()
{
    int n = 5;

    /* Constructed tree is
        1
       / \
      2   3
     / \
    4   5 */
    list<int>* adj = new list<int>[n + 1];

    /*create undirected edges */
    adj[1].push_back(2);
    adj[2].push_back(1);
    adj[1].push_back(3);
    adj[3].push_back(1);
    adj[2].push_back(4);
    adj[4].push_back(2);
    adj[2].push_back(5);
    adj[5].push_back(2);

    /* maxCount will have diameter of tree */
    cout << "Diameter of the given tree is "
        << diameter(adj, n) << endl;
    return 0;
}
```

Java

```
// Java program to find diameter of a
// binary tree using DFS.
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
public class Diametre_tree {

    // Used to track farthest node.
    static int x;
    static int maxCount;
    static List<Integer> adj[];

    // Sets maxCount as maximum distance
    // from node
    static void dfsUtil(int node, int count,
                        boolean visited[],
                        List<Integer> adj[])
    {
        visited[node] = true;
        count++;

        List<Integer> l = adj[node];
        for(Integer i: l)
        {
            if(!visited[i]){
                if (count >= maxCount) {
                    maxCount = count;
                    x = i;
                }
                dfsUtil(i, count, visited, adj);
            }
        }
    }

    // The function to do DFS traversal. It uses
    // recursive dfsUtil()
    static void dfs(int node, int n, List<Integer>
                    adj[])
    {
        boolean[] visited = new boolean[n + 1];
        int count = 0;

        // Mark all the vertices as not visited
        Arrays.fill(visited, false);

        // Increment count by 1 for visited node
        dfsUtil(node, count + 1, visited, adj);
    }
}
```

```
}

// Returns diameter of binary tree represented
// as adjacency list.
static int diameter(List<Integer> adj[], int n)
{
    maxCount = Integer.MIN_VALUE;

    /* DFS from a random node and then see
    farthest node X from it*/
    dfs(1, n, adj);

    /* DFS from X and check the farthest node
    from it */
    dfs(x, n, adj);

    return maxCount;
}

/* Driver program to test above functions*/
public static void main(String args[])
{
    int n = 5;

    /* Constructed tree is
       1
      / \
     2   3
      / \
     4   5 */
    adj = new List[n + 1];
    for(int i = 0; i < n+1 ; i++)
        adj[i] = new ArrayList<Integer>();

    /*create undirected edges */
    adj[1].add(2);
    adj[2].add(1);
    adj[1].add(3);
    adj[3].add(1);
    adj[2].add(4);
    adj[4].add(2);
    adj[2].add(5);
    adj[5].add(2);

    /* maxCount will have diameter of tree */
    System.out.println("Diameter of the given " +
                       "tree is " + diameter(adj, n));
}
```

```
}
```

```
// This code is contributed by Sumit Ghosh
```

Output:

```
Diameter of the given tree is 4
```

### Source

<https://www.geeksforgeeks.org/diameter-tree-using-dfs/>

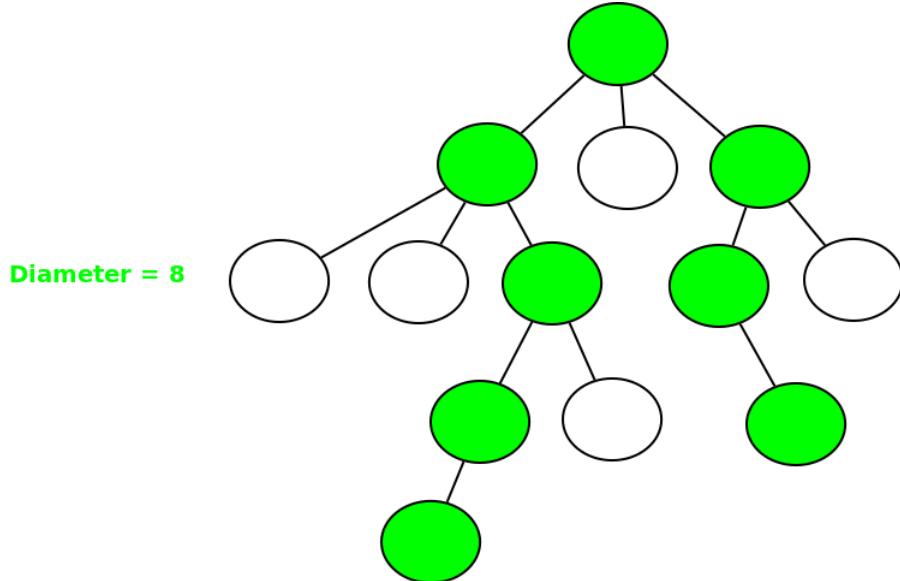
## Chapter 136

### Diameter of an N-ary tree

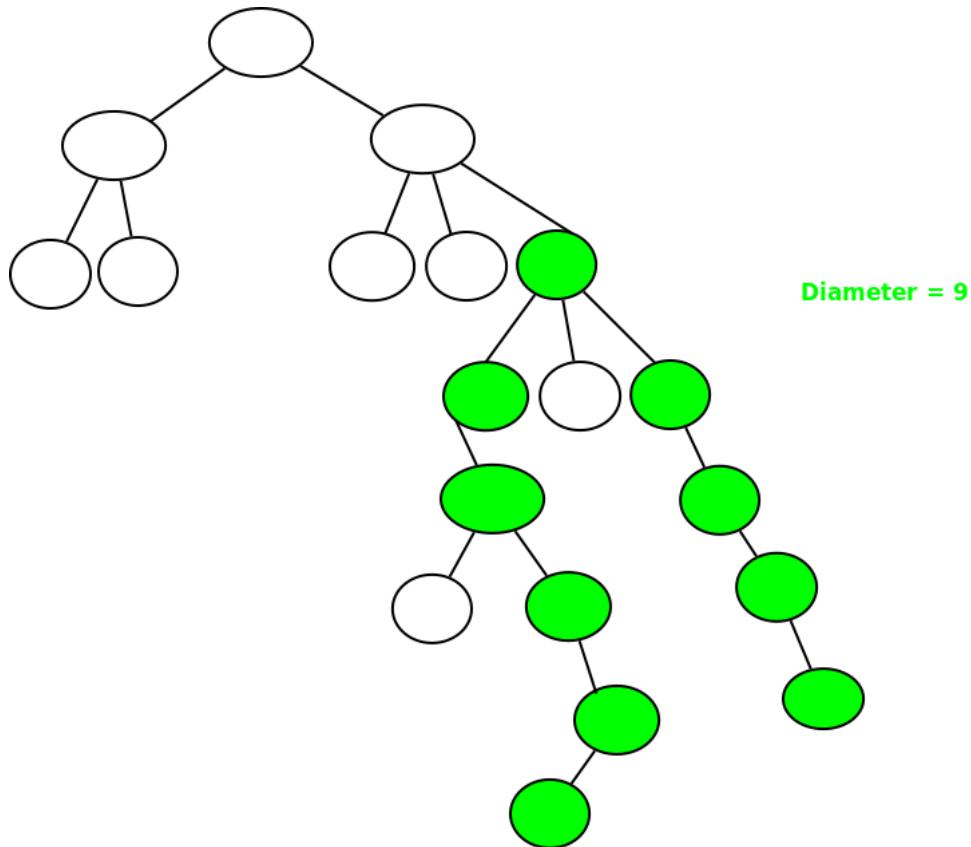
Diameter of an N-ary tree - GeeksforGeeks

The diameter of an N-ary tree is the longest path present between any two nodes of the tree. These two nodes must be two leaf nodes. The following examples have the longest path[diameter] shaded.

Examples:



Example 2:



Prerequisite : [Diameter of a binary tree](#).

The path can either start from one of the node and goes up to one of the LCAs of these nodes and again come down to the deepest node of some other subtree **or** can exist as a diameter of one of the child of the current node.

The solution will exist in any one of these:

- I] Diameter of one of the children of the current node
- II] Sum of Height of highest two subtree + 1

```
// C++ program to find the height of an N-ary
// tree
#include <bits/stdc++.h>
using namespace std;

// Structure of a node of an n-ary tree
struct Node
{
    char key;
    vector<Node *> child;
};
```

```

// Utility function to create a new tree node
Node *newNode(int key)
{
    Node *temp = new Node;
    temp->key = key;
    return temp;
}

// Utility function that will return the depth
// of the tree
int depthOfTree(struct Node *ptr)
{
    // Base case
    if (!ptr)
        return 0;

    int maxdepth = 0;

    // Check for all children and find
    // the maximum depth
    for (vector<Node*>::iterator it = ptr->child.begin();
         it != ptr->child.end(); it++)
        maxdepth = max(maxdepth , depthOfTree(*it));

    return maxdepth + 1;
}

// Function to calculate the diameter
// of the tree
int diameter(struct Node *ptr)
{
    // Base case
    if (!ptr)
        return 0;

    // Find top two highest children
    int max1 = 0, max2 = 0;
    for (vector<Node*>::iterator it = ptr->child.begin();
         it != ptr->child.end(); it++)
    {
        int h = depthOfTree(*it);
        if (h > max1)
            max2 = max1, max1 = h;
        else if (h > max2)
            max2 = h;
    }
}

```

```

// Iterate over each child for diameter
int maxChildDia = 0;
for (vector<Node*>::iterator it = ptr->child.begin();
     it != ptr->child.end(); it++)
    maxChildDia = max(maxChildDia, diameter(*it));

return max(maxChildDia, max1 + max2 + 1);
}

// Driver program
int main()
{
    /* Let us create below tree
     *      A
     *      / \   \
     *      B   F   D   E
     *      / \           |   / \
     *      K   J           G   C   H   I
     *      / \                   \
     *      N   M                   L
    */
    Node *root = newNode('A');
    (root->child).push_back(newNode('B'));
    (root->child).push_back(newNode('F'));
    (root->child).push_back(newNode('D'));
    (root->child).push_back(newNode('E'));
    (root->child[0]->child).push_back(newNode('K'));
    (root->child[0]->child).push_back(newNode('J'));
    (root->child[2]->child).push_back(newNode('G'));
    (root->child[3]->child).push_back(newNode('C'));
    (root->child[3]->child).push_back(newNode('H'));
    (root->child[3]->child).push_back(newNode('I'));
    (root->child[0]->child[0]->child).push_back(newNode('N'));
    (root->child[0]->child[0]->child).push_back(newNode('M'));
    (root->child[3]->child[2]->child).push_back(newNode('L'));

    cout << diameter(root) << endl;

    return 0;
}

```

Output:

We can make a hash table to store heights of all nodes. If we precompute these heights, we don't need to call depthOfTree() for every node.

**A different optimized solution :**

[Longest path in an undirected tree](#)

## Source

<https://www.geeksforgeeks.org/diameter-n-ary-tree/>

## Chapter 137

# Diameter of n-ary tree using BFS

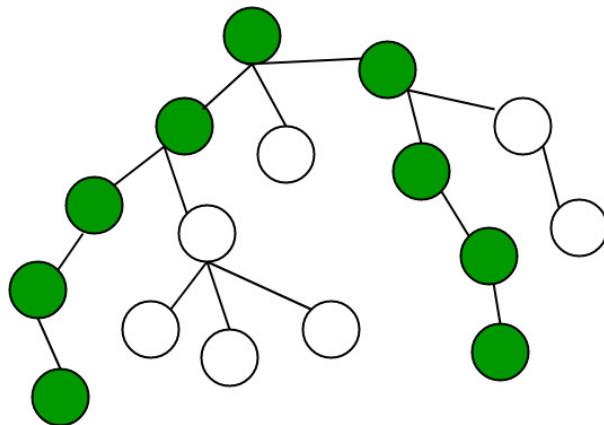
Diameter of n-ary tree using BFS - GeeksforGeeks

**N-ary** tree refers to the rooted tree in which each node having atmost k child nodes. The diameter of n-ary tree is the longest path between two leaf nodes.

Various approaches have already been discussed to compute diameter of tree.

1. [Diameter of an N-ary tree](#)
2. [Diameter of a Binary Tree in O\(n\)](#)
3. [Diameter of a Binary Tree](#)
4. [Diameter of a tree using DFS](#)

This article discuss another approach for computing diameter tree of n-ary tree using bfs.



**Step 1:** Run bfs to find the farthest node from rooted tree let say A

**Step 2:** Then run bfs from A to find farthest node from A let B

**Step 3:** Distance between node A and B is the diameter of given tree

```
// C++ Program to find Diameter of n-ary tree
```

```
#include <bits/stdc++.h>
using namespace std;

// Here 10000 is maximum number of nodes in
// given tree.
int diameter[10001];

// The Function to do bfs traversal.
// It uses iterative approach to do bfs
// bfsUtil()
int bfs(int init, vector<int> arr[], int n)
{
    // Initializing queue
    queue<int> q;
    q.push(init);

    int visited[n + 1];
    for (int i = 0; i <= n; i++) {
        visited[i] = 0;
        diameter[i] = 0;
    }

    // Pushing each node in queue
    q.push(init);

    // Mark the traversed node visited
    visited[init] = 1;
    while (!q.empty()) {
        int u = q.front();
        q.pop();
        for (int i = 0; i < arr[u].size(); i++) {
            if (visited[arr[u][i]] == 0) {
                visited[arr[u][i]] = 1;

                // Considering weight of edges equal to 1
                diameter[arr[u][i]] += diameter[u] + 1;
                q.push(arr[u][i]);
            }
        }
    }

    // return index of max value in diameter
    return int(max_element(diameter + 1,
                           diameter + n + 1)
              - diameter);
}

int findDiameter(vector<int> arr[], int n)
```

```
{  
    int init = bfs(1, arr, n);  
    int val = bfs(init, arr, n);  
    return diameter[val];  
}  
  
// Driver Code  
int main()  
{  
    // Input number of nodes  
    int n = 6;  
  
    vector<int> arr[n + 1];  
  
    // Input nodes in adjacency list  
    arr[1].push_back(2);  
    arr[1].push_back(3);  
    arr[1].push_back(6);  
    arr[2].push_back(4);  
    arr[2].push_back(1);  
    arr[2].push_back(5);  
    arr[3].push_back(1);  
    arr[4].push_back(2);  
    arr[5].push_back(2);  
    arr[6].push_back(1);  
  
    printf("Diameter of n-ary tree is %d\n",  
          findDiameter(arr, n));  
  
    return 0;  
}
```

Output:

Diameter of n-ary tree is 3

## Source

<https://www.geeksforgeeks.org/diameter-n-ary-tree-using-bfs/>

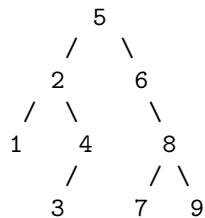
## Chapter 138

# Difference between sums of odd level and even level nodes of a Binary Tree

Difference between sums of odd level and even level nodes of a Binary Tree - GeeksforGeeks

Given a a Binary Tree, find the difference between the sum of nodes at odd level and the sum of nodes at even level. Consider root as level 1, left and right children of root as level 2 and so on.

For example, in the following tree, sum of nodes at odd level is  $(5 + 1 + 4 + 8)$  which is 18. And sum of nodes at even level is  $(2 + 6 + 3 + 7 + 9)$  which is 27. The output for following tree should be  $18 - 27$  which is -9.



A straightforward method is to use **level order traversal**. In the traversal, check level of current node, if it is odd, increment odd sum by data of current node, otherwise increment even sum. Finally return difference between odd sum and even sum. See following for implementation of this approach.

[C implementation of level order traversal based approach to find the difference.](#)

This approach is provided by [Mandeep Singh](#). For **Iterative approach**, simply traverse the tree level by level (level order traversal), store sum of node values in even no. level in evenSum and rest in variable oddSum and finally return the difference.

Below is the simple implementation of the approach.

C++

```
// CPP program to find
// difference between
// sums of odd level
// and even level nodes
// of binary tree
#include <bits/stdc++.h>
using namespace std;

// tree node
struct Node
{
    int data;
    Node *left, *right;
};

// returns a new
// tree Node
Node* newNode(int data)
{
    Node* temp = new Node();
    temp->data = data;
    temp->left = temp->right = NULL;
    return temp;
}

// return difference of
// sums of odd level
// and even level
int evenOddLevelDifference(Node* root)
{
    if (!root)
        return 0;

    // create a queue for
    // level order traversal
    queue<Node*> q;
    q.push(root);

    int level = 0;
    int evenSum = 0, oddSum = 0;

    // traverse until the
    // queue is empty
    while (!q.empty())
```

```
{  
    int size = q.size();  
    level += 1;  
  
    // traverse for  
    // complete level  
    while(size > 0)  
    {  
        Node* temp = q.front();  
        q.pop();  
  
        // check if level no.  
        // is even or odd and  
        // accordingly update  
        // the evenSum or oddSum  
        if(level % 2 == 0)  
            evenSum += temp->data;  
        else  
            oddSum += temp->data;  
  
        // check for left child  
        if (temp->left)  
        {  
            q.push(temp->left);  
        }  
  
        // check for right child  
        if (temp->right)  
        {  
            q.push(temp->right);  
        }  
        size -= 1;  
    }  
}  
  
return (oddSum - evenSum);  
}  
  
// driver program  
int main()  
{  
    // construct a tree  
    Node* root = newNode(5);  
    root->left = newNode(2);  
    root->right = newNode(6);  
    root->left->left = newNode(1);  
    root->left->right = newNode(4);  
    root->left->right->left = newNode(3);
```

```
root->right->right = newNode(8);
root->right->right->right = newNode(9);
root->right->right->left = newNode(7);

int result = evenOddLevelDifference(root);
cout << "difference between sums is :: ";
cout << result << endl;
return 0;
}

// This article is contributed by Mandeep Singh.
```

The problem can also be solved **using simple recursive traversal**. We can recursively calculate the required difference as, value of root's data subtracted by the difference for subtree under left child and the difference for subtree under right child.

Below is the implementation of this approach.

## C

```
// A recursive program to find difference between sum of nodes at
// odd level and sum at even level
#include <stdio.h>
#include <stdlib.h>

// Binary Tree node
struct node
{
    int data;
    struct node* left, *right;
};

// A utility function to allocate a new tree node with given data
struct node* newNode(int data)
{
    struct node* node = (struct node*)malloc(sizeof(struct node));
    node->data = data;
    node->left = node->right = NULL;
    return (node);
}

// The main function that return difference between odd and even level
// nodes
int getLevelDiff(struct node *root)
{
    // Base case
    if (root == NULL)
        return 0;
```

```
// Difference for root is root's data - difference for
// left subtree - difference for right subtree
return root->data - getLevelDiff(root->left) -
           getLevelDiff(root->right);
}

// Driver program to test above functions
int main()
{
    struct node *root = newNode(5);
    root->left = newNode(2);
    root->right = newNode(6);
    root->left->left = newNode(1);
    root->left->right = newNode(4);
    root->left->right->left = newNode(3);
    root->right->right = newNode(8);
    root->right->right->right = newNode(9);
    root->right->right->left = newNode(7);
    printf("%d is the required difference\n", getLevelDiff(root));
    getchar();
    return 0;
}
```

### Java

```
// A recursive java program to find difference between sum of nodes at
// odd level and sum at even level

// A binary tree node
class Node
{
    int data;
    Node left, right;

    Node(int item)
    {
        data = item;
        left = right;
    }
}

class BinaryTree
{
    // The main function that return difference between odd and even level
    // nodes
    Node root;

    int getLevelDiff(Node node)
```

```
{  
    // Base case  
    if (node == null)  
        return 0;  
  
    // Difference for root is root's data - difference for  
    // left subtree - difference for right subtree  
    return node.data - getLevelDiff(node.left) -  
           getLevelDiff(node.right);  
}  
  
// Driver program to test above functions  
public static void main(String args[])  
{  
    BinaryTree tree = new BinaryTree();  
    tree.root = new Node(5);  
    tree.root.left = new Node(2);  
    tree.root.right = new Node(6);  
    tree.root.left.left = new Node(1);  
    tree.root.left.right = new Node(4);  
    tree.root.left.right.left = new Node(3);  
    tree.root.right.right = new Node(8);  
    tree.root.right.right.right = new Node(9);  
    tree.root.right.right.left = new Node(7);  
    System.out.println(tree.getLevelDiff(tree.root) +  
                       " is the required difference");  
  
}  
}  
  
// This code has been contributed by Mayank Jaiswal
```

### Python

```
# A recursive program to find difference between sum of nodes  
# at odd level and sum at even level  
  
# A Binary Tree node  
class Node:  
  
    # Constructor to create a new node  
    def __init__(self, data):  
        self.data = data  
        self.left = None  
        self.right = None  
  
    # The main function that returns difference between odd and  
    # even level nodes
```

```
def getLevelDiff(root):

    # Base Case
    if root is None:
        return 0

    # Difference for root is root's data - difference for
    # left subtree - difference for right subtree
    return (root.data - getLevelDiff(root.left)-
            getLevelDiff(root.right))

# Driver program to test above function
root = Node(5)
root.left = Node(2)
root.right = Node(6)
root.left.left = Node(1)
root.left.right = Node(4)
root.left.right.left = Node(3)
root.right.right = Node(8)
root.right.right.right = Node(9)
root.right.right.left = Node(7)
print "%d is the required difference" %(getLevelDiff(root))

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

Output:

```
-9 is the required difference
```

Time complexity of both methods is  $O(n)$ , but the second method is simple and easy to implement.

This article is contributed by **Chandra Prakash**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## Source

<https://www.geeksforgeeks.org/difference-between-sums-of-odd-and-even-levels/>

## Chapter 139

# Disjoint Set Union on trees | Set 1

Disjoint Set Union on trees | Set 1 - GeeksforGeeks

Given a tree and weights of nodes. Weights are non-negative integers. Task is to find maximum size of a subtree of a given tree such that all nodes are even in weights.

**Prerequisite :** [Disjoint Set Union](#)

Examples :

```
Input : Number of nodes = 7
        Weights of nodes = 1 2 6 4 2 0 3
        Edges = (1, 2), (1, 3), (2, 4),
                 (2, 5), (4, 6), (6, 7)
Output : Maximum size of the subtree
with even weighted nodes = 4
Explanation :
Subtree of nodes {2, 4, 5, 6} gives the maximum size.
```

```
Input : Number of nodes = 6
        Weights of nodes = 2 4 0 2 2 6
        Edges = (1, 2), (2, 3), (3, 4),
                 (4, 5), (1, 6)
Output : Maximum size of the subtree
with even weighted nodes = 6
Explanation :
The given tree gives the maximum size.
```

**Approach :** We can find solution by simply running [DFS](#) on tree. DFS solution gives us answer in  $O(n)$ . But, how can we use DSU for this problem? We first iterate through all

edges. If both nodes are even in weights, we make union of them. Set of nodes with maximum size is the answer. If we use union-find with path compression then time complexity is  $O(n)$ .

**Below is the implementation of above approach :**

```

// CPP code to find maximum subtree such
// that all nodes are even in weight
#include<bits/stdc++.h>

using namespace std;

#define N 100010

// Structure for Edge
struct Edge
{
    int u, v;
};

/*
'id': stores parent of a node.
'sz': stores size of a DSU tree.
*/
int id[N], sz[N];

// Function to assign root
int Root(int idx)
{
    int i = idx;

    while(i != id[i])
        id[i] = id[id[i]], i = id[i];

    return i;
}

// Function to find Union
void Union(int a, int b)
{
    int i = Root(a), j = Root(b);

    if (i != j)
    {
        if(sz[i] >= sz[j])
        {
            id[j] = i, sz[i] += sz[j];
            sz[j] = 0;
        }
        else
    }
}

```

```

    {
        id[i] = j, sz[j] += sz[i];
        sz[i] = 0;
    }
}

// Utility function for Union
void UnionUtil(struct Edge e[], int W[], int q)
{

    for(int i = 0; i < q; i++)
    {
        // Edge between 'u' and 'v'
        int u, v;
        u = e[i].u, v = e[i].v;

        // 0-indexed nodes
        u--, v--;

        // If weights of both 'u' and 'v'
        // are even then we make union of them.
        if(W[u] % 2 == 0 && W[v] % 2 == 0)
            Union(u,v);
    }
}

// Function to find maximum
// size of DSU tree
int findMax(int n, int W[])
{
    int maxi = 0;
    for(int i = 1; i <= n; i++)
        if(W[i] % 2 == 0)
            maxi = max(maxi, sz[i]);

    return maxi;
}

// Driver code
int main()
{
    /*
        Nodes are 0-indexed in this code
        So we have to make necessary changes
        while taking inputs
    */
}

```

```
// Weights of nodes
int W[] = {1, 2, 6, 4, 2, 0, 3};

// Number of nodes in a tree
int n = sizeof(W) / sizeof(W[0]);

// Initializing every node as
// a tree with single node.
for(int i = 0; i < n; i++)
    id[i] = i, sz[i] = 1;

Edge e[] = {{1, 2}, {1, 3}, {2, 4},
            {2, 5}, {4, 6}, {6, 7}};

int q = sizeof(e) / sizeof(e[0]);

UnionUtil(e, W, q);

// Find maximum size of DSU tree.
int maxi = findMax(n, W);

printf("Maximum size of the subtree with ");
printf("even weighted nodes = %d\n", maxi);

return 0;
}
```

**Output:**

```
Maximum size of the subtree with even weighted nodes = 4
```

**Source**

<https://www.geeksforgeeks.org/disjoint-set-union-trees-set-1/>

## Chapter 140

# Disjoint Set Union on trees | Set 2

Disjoint Set Union on trees | Set 2 - GeeksforGeeks

Given a tree, and the cost of a subtree is defined as  $|S| * \text{AND}(S)$  where  $|S|$  is the size of the subtree and  $\text{AND}(S)$  is bitwise AND of all indices of nodes from the subtree, task is to find maximum cost of possible subtree.

**Prerequisite :** [Disjoint Set Union](#)

Examples:

```
Input : Number of nodes = 4
        Edges = (1, 2), (3, 4), (1, 3)
Output : Maximum cost = 4
Explanation :
Subtree with single node {4} gives the maximum cost.
```

```
Input : Number of nodes = 6
        Edges = (1, 2), (2, 3), (3, 4), (3, 5), (5, 6)
Output : Maximum cost = 8
Explanation :
Subtree with nodes {5, 6} gives the maximum cost.
```

**Approach :** The strategy is to fix the AND, and find the maximum size of a subtree such that AND of all indices equals to the given AND. Suppose we fix AND as 'A'. In binary representation of A, if the ith bit is '1', then all indices(nodes) of the required subtree should have '1' in ith position in binary representation. If ith bit is '0' then indices either have '0' or '1' in ith position. That means all elements of subtree are super masks of A. All super masks of A can be generated in  $O(2^k)$  time where 'k' is the number of bits which are '0' in A.

Now, the maximum size of subtree with a given AND ‘A’ can be found using DSU on the tree. Let, ‘u’ be the super mask of A and ‘p[u]’ be the parent of u. If p[u] is also a super mask of A, then, we have to update the DSU by merging the components of u and p[u]. Simultaneously, we also have to keep track of the maximum size of subtree. DSU helps us to do it. It will be more clear if we look at following code.

```

// CPP code to find maximum possible cost
#include <bits/stdc++.h>
using namespace std;

#define N 100010

// Edge structure
struct Edge {
    int u, v;
};

/* v : Adjacency list representation of Graph
   p : stores parents of nodes */
vector<int> v[N];
int p[N];

// Weighted union-find with path compression
struct wunionfind {
    int id[N], sz[N];
    void initial(int n)
    {
        for (int i = 1; i <= n; i++)
            id[i] = i, sz[i] = 1;
    }

    int Root(int idx)
    {
        int i = idx;
        while (i != id[i])
            id[i] = id[id[i]], i = id[i];

        return i;
    }

    void Union(int a, int b)
    {
        int i = Root(a), j = Root(b);
        if (i != j) {
            if (sz[i] >= sz[j]) {
                id[j] = i, sz[i] += sz[j];
                sz[j] = 0;
            }
        }
    }
};

```

```

        else {
            id[i] = j, sz[j] += sz[i];
            sz[i] = 0;
        }
    }
};

wunionfind W;

// DFS is called to generate parent of
// a node from adjacency list representation
void dfs(int u, int parent)
{
    for (int i = 0; i < v[u].size(); i++) {
        int j = v[u][i];
        if (j != parent) {
            p[j] = u;
            dfs(j, u);
        }
    }
}

// Utility function for Union
int UnionUtil(int n)
{
    int ans = 0;

    // Fixed 'i' as AND
    for (int i = 1; i <= n; i++) {
        int maxi = 1;

        // Generating supermasks of 'i'
        for (int x = i; x <= n; x = (i | (x + 1))) {
            int y = p[x];

            // Checking whether p[x] is
            // also a supermask of i.
            if ((y & i) == i) {
                W.Union(x, y);

                // Keep track of maximum
                // size of subtree
                maxi = max(maxi, W.sz[W.Root(x)]);
            }
        }
    }

    // Storing maximum cost of
}

```

```

// subtree with a given AND
ans = max(ans, maxi * i);

// Separating components which are merged
// during Union operation for next AND value.
for (int x = i; x <= n; x = (i | (x + 1))) {
    W.sz[x] = 1;
    W.id[x] = x;
}
}

return ans;
}

// Driver code
int main()
{
    int n, i;

    // Number of nodes
    n = 6;

    W.initial(n);

    Edge e[] = { { 1, 2 }, { 2, 3 }, { 3, 4 },
                 { 3, 5 }, { 5, 6 } };

    int q = sizeof(e) / sizeof(e[0]);

    // Taking edges as input and put
    // them in adjacency list representation
    for (i = 0; i < q; i++) {
        int x, y;
        x = e[i].u, y = e[i].v;
        v[x].push_back(y);
        v[y].push_back(x);
    }

    // Initializing parent vertex of '1' as '1'
    p[1] = 1;

    // Call DFS to generate 'p' array
    dfs(1, -1);

    int ans = UnionUtil(n);

    printf("Maximum Cost = %d\n", ans);
}

```

```
        return 0;  
    }
```

**Output:**

```
Maximum Cost = 8
```

**Time Complexity :** Union in DSU takes O(1) time. Generating all supermasks takes  $O(3^k)$  time where k is the maximum number of bits which are '0'. DFS takes O(n). Overall time complexity is  $O(3^k+n)$ .

**Source**

<https://www.geeksforgeeks.org/disjoint-set-union-trees-set-2/>

## Chapter 141

### Double Tree

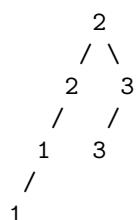
Double Tree - GeeksforGeeks

Write a program that converts a given tree to its Double tree. To create Double tree of the given tree, create a new duplicate for each node, and insert the duplicate as the left child of the original node.

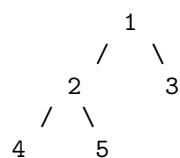
So the tree...



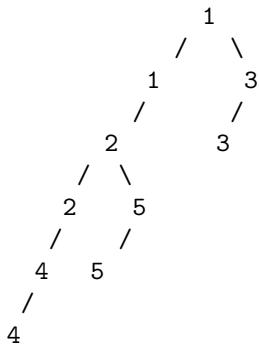
is changed to...



And the tree



is changed to



Algorithm:

Recursively convert the tree to double tree in postorder fashion. For each node, first convert the left subtree of the node, then right subtree, finally create a duplicate node of the node and fix the left child of the node and left child of left child.

Implementation:

C

```

#include <stdio.h>
#include <stdlib.h>

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

/* function to create a new node of tree and returns pointer */
struct node* newNode(int data);

/* Function to convert a tree to double tree */
void doubleTree(struct node* node)
{
    struct node* oldLeft;

    if (node==NULL) return;

    /* do the subtrees */
    doubleTree(node->left);
    doubleTree(node->right);
  
```

```

/* duplicate this node to its left */
oldLeft = node->left;
node->left = newNode(node->data);
node->left->left = oldLeft;
}

/* UTILITY FUNCTIONS TO TEST doubleTree() FUNCTION */
/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* node = (struct node*)
                           malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;

    return(node);
}

/* Given a binary tree, print its nodes in inorder*/
void printInorder(struct node* node)
{
    if (node == NULL)
        return;
    printInorder(node->left);
    printf("%d ", node->data);
    printInorder(node->right);
}

/* Driver program to test above functions*/
int main()
{

    /* Constructed binary tree is
           1
         /   \
        2     3
       /   \
      4     5
    */
    struct node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
}

```

```
root->left->right = newNode(5);

printf("Inorder traversal of the original tree is \n");
printInorder(root);

doubleTree(root);

printf("\n Inorder traversal of the double tree is \n");
printInorder(root);

getchar();
return 0;
}
```

**Java**

```
// Java program to convert binary tree to double tree

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
class Node
{
    int data;
    Node left, right;

    Node(int item)
    {
        data = item;
        left = right = null;
    }
}

class BinaryTree
{
    Node root;

    /* Function to convert a tree to double tree */
    void doubleTree(Node node)
    {
        Node oldleft;

        if (node == null)
            return;

        /* do the subtrees */
        doubleTree(node.left);
        doubleTree(node.right);
```

```
/* duplicate this node to its left */
oldleft = node.left;
node.left = new Node(node.data);
node.left.left = oldleft;
}

/* Given a binary tree, print its nodes in inorder*/
void printInorder(Node node)
{
    if (node == null)
        return;
    printInorder(node.left);
    System.out.print(node.data + " ");
    printInorder(node.right);
}

/* Driver program to test the above functions */
public static void main(String args[])
{
    /* Constructed binary tree is
       1
      / \
     2   3
    / \
   4   5
*/
    BinaryTree tree = new BinaryTree();
    tree.root = new Node(1);
    tree.root.left = new Node(2);
    tree.root.right = new Node(3);
    tree.root.left.left = new Node(4);
    tree.root.left.right = new Node(5);

    System.out.println("Original tree is : ");
    tree.printInorder(tree.root);
    tree.doubleTree(tree.root);
    System.out.println("");
    System.out.println("Inorder traversal of double tree is : ");
    tree.printInorder(tree.root);
}
}

// This code has been contributed by Mayank Jaiswal(mayank_24)
```

Time Complexity: O(n) where n is the number of nodes in the tree.

References:

<http://cslibrary.stanford.edu/110/BinaryTrees.html>

**Source**

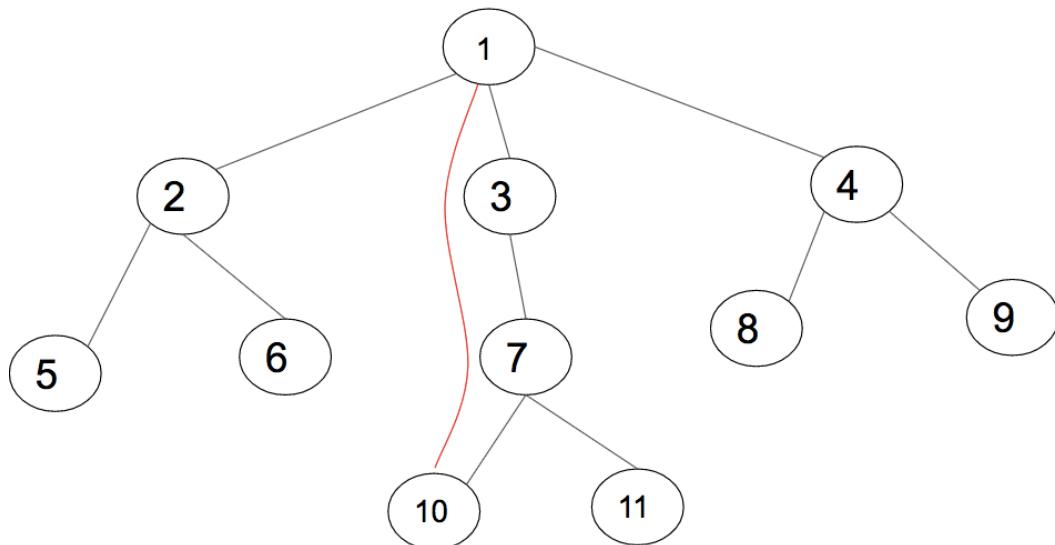
<https://www.geeksforgeeks.org/double-tree/>

## Chapter 142

# Dynamic Programming on Trees | Set 2

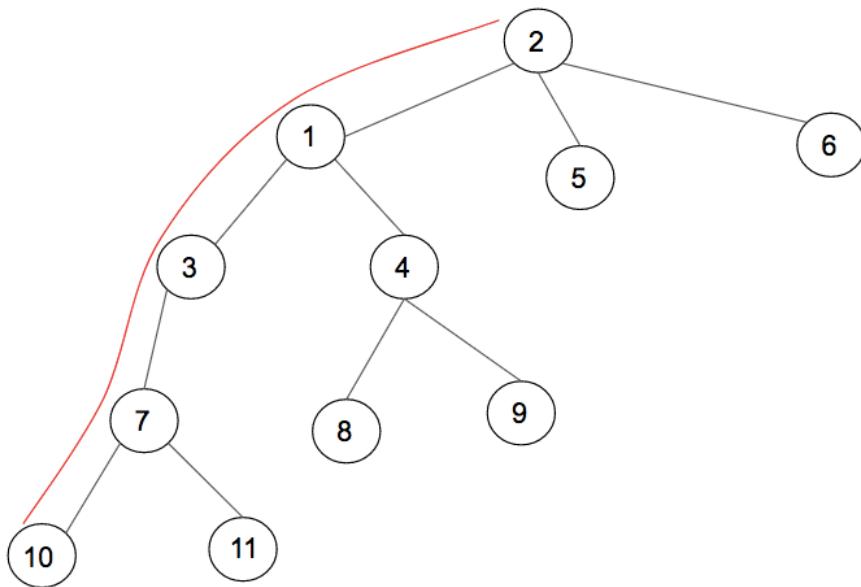
Dynamic Programming on Trees | Set 2 - GeeksforGeeks

Given a tree with N nodes and N-1 edges, find out the **maximum height of tree** when any node in the tree is considered as the root of the tree.



**Red line denotes the maximum height of tree when node-1 is the root**

The above diagram represents a tree with 11 nodes and 10 edges, and the path which gives us the maximum height when node 1 is considered as root. The maximum height is 3.



**Red line denotes the maximum height of the tree when node-2 is taken as root.**

In the above diagram, when 2 is considered as root, then the longest path found is in RED color. A **naive approach** will be to traverse the tree using [DFS traversal](#) for every node and calculate the maximum height when the node is treated as the root of the tree. The time complexity for DFS traversal of a tree is  $O(N)$ . **The overall time complexity of DFS for all N nodes will be  $O(N)*N$  i.e.,  $O(N^2)$ .**

The above problem can be solved by using **Dynamic Programming on Trees**. To solve this problem, pre-calculate two things for every node. One will be the maximum height while traveling downwards via its branches to the leaves. While the other will be the maximum height when traveling upwards via its parent to any of the leaves.

#### Optimal Substructure :

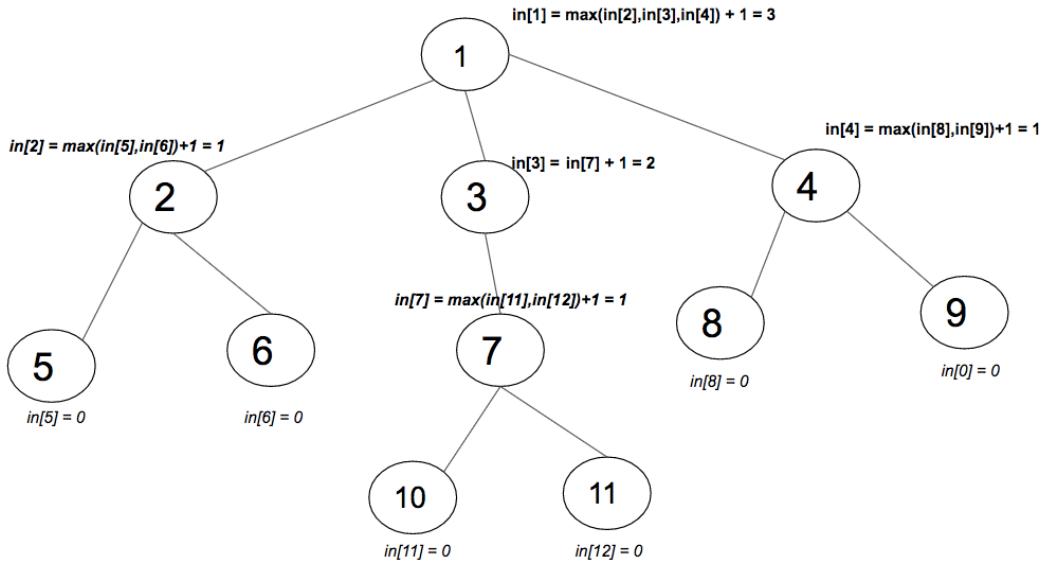
When node  $i$  is considered as root,

**in[i]** be the maximum height of tree when we travel downwards via its sub-trees and leaves.

Also, **out[i]** be the maximum height of the tree while traveling upwards via its parent.

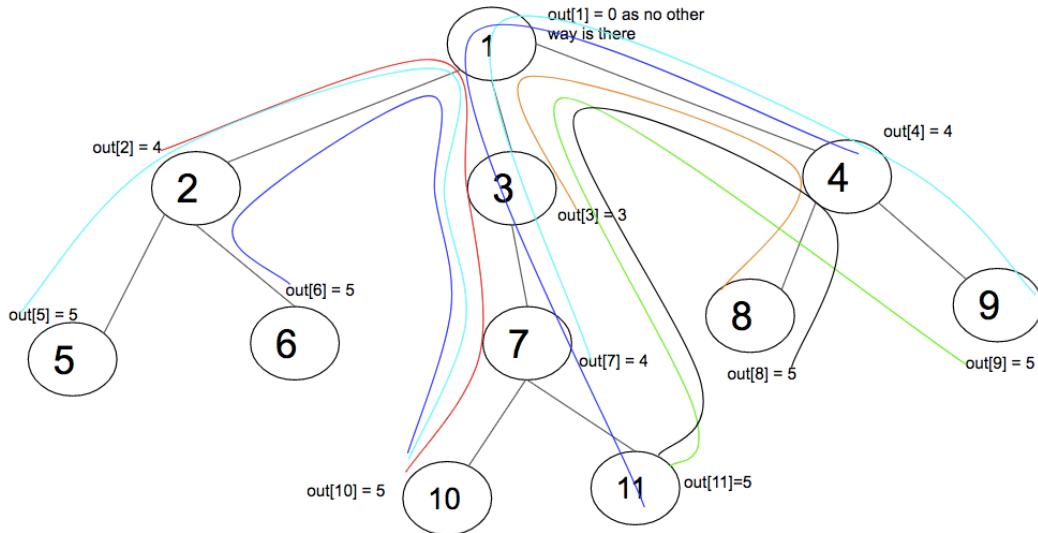
The maximum height of tree when node  $i$  is considered as root will be **max(in[i], out[i])**.

#### Calculation of in[i] :



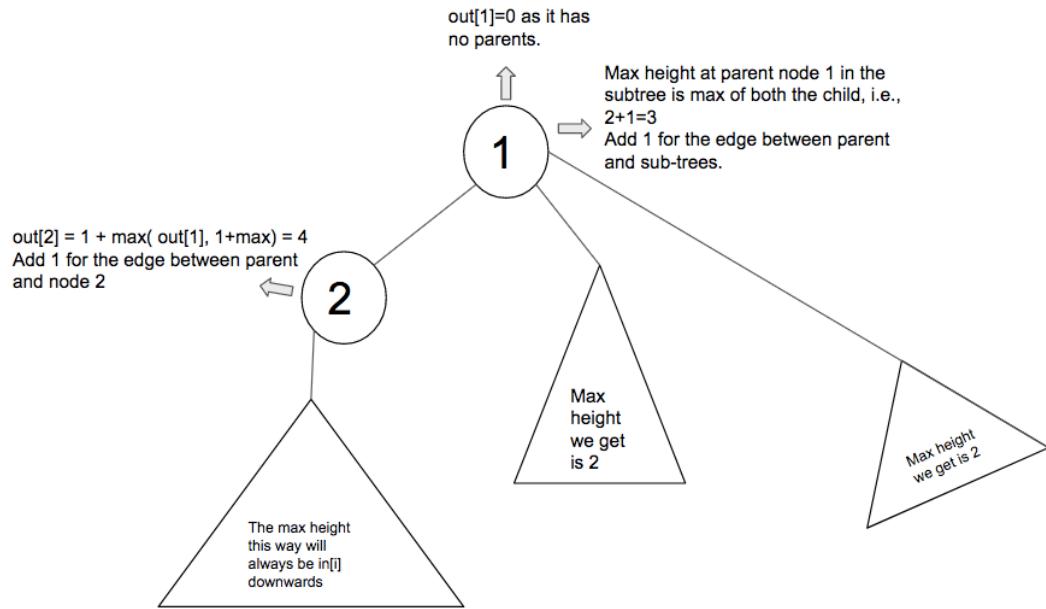
In the image above, values of  $\text{in}[i]$  have been calculated for every node  $i$ . The maximum of every subtree is taken and added with 1 to the parent of that subtree. Add 1 for the edge between parent and subtree. Traverse the tree using DFS and calculate  $\text{in}[i]$  as  $\max(\text{in}[i], 1 + \text{in}[\text{child}])$  for every node.

**Calculation of out[i] :**



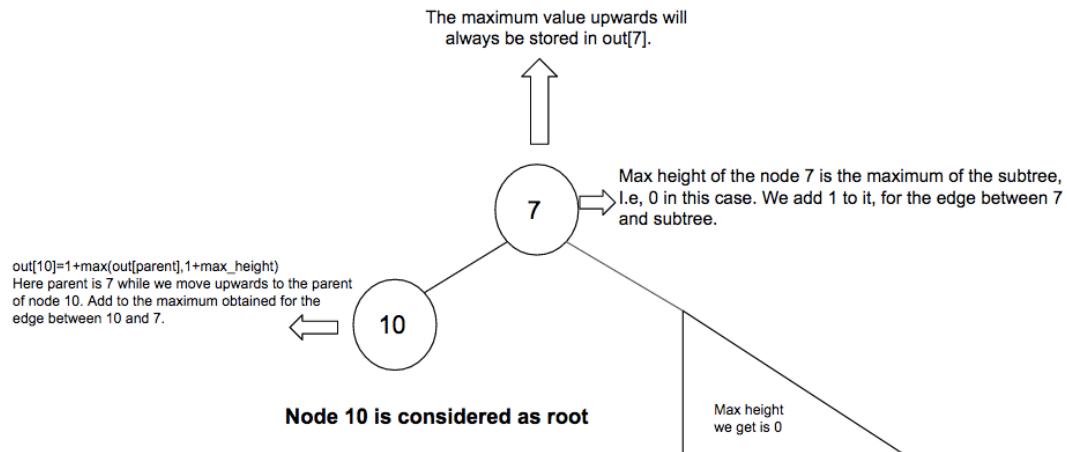
The above diagram shows all the  $\text{out}[i]$  values and the path. For calculation of  $\text{out}[i]$ , move upwards to the parent of node  $i$ . From the parent of node  $i$ , there are two ways to move in, one will be in all the branches of the parent. The other direction is to move to the parent(call it parent2 to avoid confusion) of the parent(call it parent1) of node  $i$ . The maximum height upwards via parent2 is  $\text{out}[\text{parent1}]$  itself. Generally,  $\text{out}[\text{node } i]$  as  $1 + \max(\text{out}[i], 1 + \max(\text{out}[\text{parent1}], \dots))$ .

of all branches). Add 1 for the edges between node and parent.



#### Node 2 is considered as root

The above diagram explains the calculation of  $out[i]$  when 2 is considered as the root of the tree. The branches of node 2 is not taken into count since the maximum height via that path has already been calculated and stored in  $i[2]$ . Moving up, in this case, the parent of 2 i.e., 1 has no parent. So, the branches except for the one which has the node are considered while calculating the maximum.



The above diagram explains the calculation of  $out[10]$ . The parent of node 10, i.e., 7 has a parent and a branch(precisely a child in this case). So the maximum height of both has been taken to count in such cases when parent and branches exist.

In case of multiple branches of a parent, take the longest of them to count(excluding the branch in which the node lies)

**Calculating the maximum height of all the branches connected to parent :**

in[i] stores the maximum height while moving downwards. No need to store all the lengths of branches. Only the first and second maximum length among all the branches will give answer. Since, algorithm used is based on DFS, all the branches connected to parent will be considered, including the branch which has the node. If the first maximum path thus obtained is same as in[i], then maximum1 is the length of the branch in which node i lies. In this case, our longest path will be maximum2.

**Recurrence relation of in[i] and out[i] :**

$$\begin{aligned} \text{in}[i] &= \max(\text{in}[i], 1 + \text{in}[\text{child}]) \\ \text{out}[i] &= 1 + \max(\text{out}[\text{parent of } i], 1 + \text{longest path of all branches of parent of } i) \end{aligned}$$

Below is the implementation of the above idea :

C++

```
// CPP code to find the maximum path length
// considering any node as root
#include <bits/stdc++.h>
using namespace std;
const int MAX_NODES = 100;

int in[MAX_NODES];
int out[MAX_NODES];

// function to pre-calculate the array in[]
// which stores the maximum height when travelled
// via branches
void dfs1(vector<int> v[], int u, int parent)
{
    // initially every node has 0 height
    in[u] = 0;

    // traverse in the subtree of u
    for (int child : v[u]) {

        // if child is same as parent
        if (child == parent)
            continue;

        // dfs called
        dfs1(v, child, u);
    }
}
```

```

        // recursively calculate the max height
        in[u] = max(in[u], 1 + in[child]);
    }
}

// function to pre-calculate the array ouut[]
// which stores the maximum height when traveled
// via parent
void dfs2(vector<int> v[], int u, int parent)
{
    // stores the longest and second
    // longest branches
    int mx1 = -1, mx2 = -1;

    // traverse in the subtress of u
    for (int child : v[u]) {
        if (child == parent)
            continue;

        // compare and store the longest
        // and second longest
        if (in[child] >= mx1) {
            mx2 = mx1;
            mx1 = in[child];
        }

        else if (in[child] > mx2)
            mx2 = in[child];
    }

    // traverse in the subtree of u
    for (int child : v[u]) {
        if (child == parent)
            continue;

        int longest = mx1;

        // if longest branch has the node, then
        // consider the second longest branch
        if (mx1 == in[child])
            longest = mx2;

        // recursively calculate out[i]
        out[child] = 1 + max(out[u], 1 + longest);

        // dfs fucntion call
        dfs2(v, child, u);
    }
}

```

```

}

// function to print all the maximum heights
// from every node
void printHeights(vector<int> v[], int n)
{
    // traversal to calculate in[] array
    dfs1(v, 1, 0);

    // traversal to calculate out[] array
    dfs2(v, 1, 0);

    // print all maximum heights
    for (int i = 1; i <= n; i++)
        cout << "The maximum height when node "
            << i << " is considered as root"
            << " is " << max(in[i], out[i])
            << "\n";
}

// Driver Code
int main()
{
    int n = 11;
    vector<int> v[n + 1];

    // initialize the tree given in the diagram
    v[1].push_back(2), v[2].push_back(1);
    v[1].push_back(3), v[3].push_back(1);
    v[1].push_back(4), v[4].push_back(1);
    v[2].push_back(5), v[5].push_back(2);
    v[2].push_back(6), v[6].push_back(2);
    v[3].push_back(7), v[7].push_back(3);
    v[7].push_back(10), v[10].push_back(7);
    v[7].push_back(11), v[11].push_back(7);
    v[4].push_back(8), v[8].push_back(4);
    v[4].push_back(9), v[9].push_back(4);

    // function to print the maximum height from every node
    printHeights(v, n);

    return 0;
}

```

**Output :**

The maximum height when node 1 is considered as root is 3

The maximum height when node 2 is considered as root is 4  
The maximum height when node 3 is considered as root is 3  
The maximum height when node 4 is considered as root is 4  
The maximum height when node 5 is considered as root is 5  
The maximum height when node 6 is considered as root is 5  
The maximum height when node 7 is considered as root is 4  
The maximum height when node 8 is considered as root is 5  
The maximum height when node 9 is considered as root is 5  
The maximum height when node 10 is considered as root is 5  
The maximum height when node 11 is considered as root is 5

**Time Complexity :**  $O(N)$   
**Auxiliary Space :**  $O(N)$

## Source

<https://www.geeksforgeeks.org/dynamic-programming-trees-set-2/>

## Chapter 143

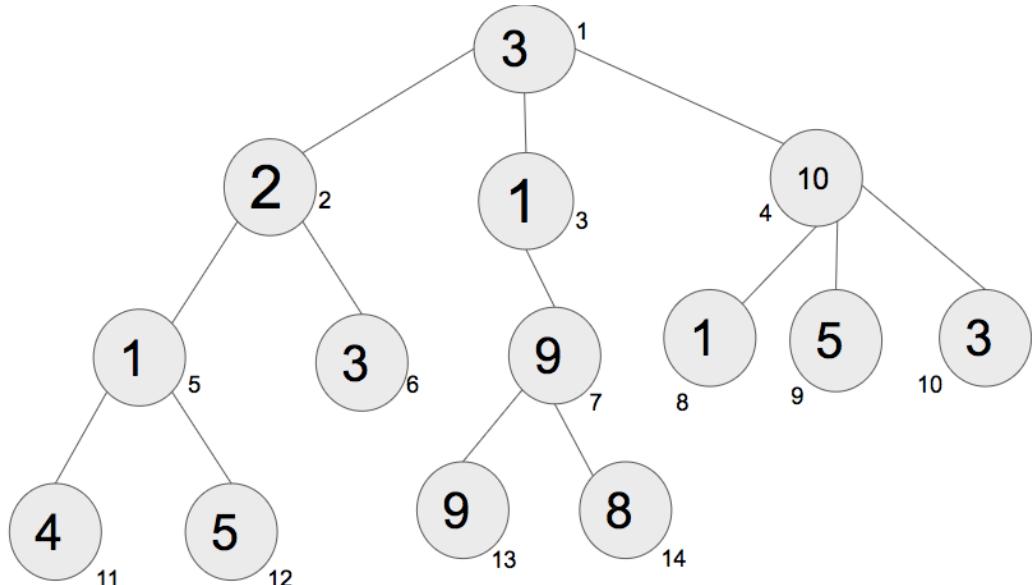
# Dynamic Programming on Trees | Set-1

Dynamic Programming on Trees | Set-1 - GeeksforGeeks

Dynamic Programming(DP) is a technique to solve problems by breaking them down into overlapping sub-problems which follows the optimal substructure. There are various problems using DP like subset sum, knapsack, coin change etc. DP can also be applied on trees to solve some specific problems.

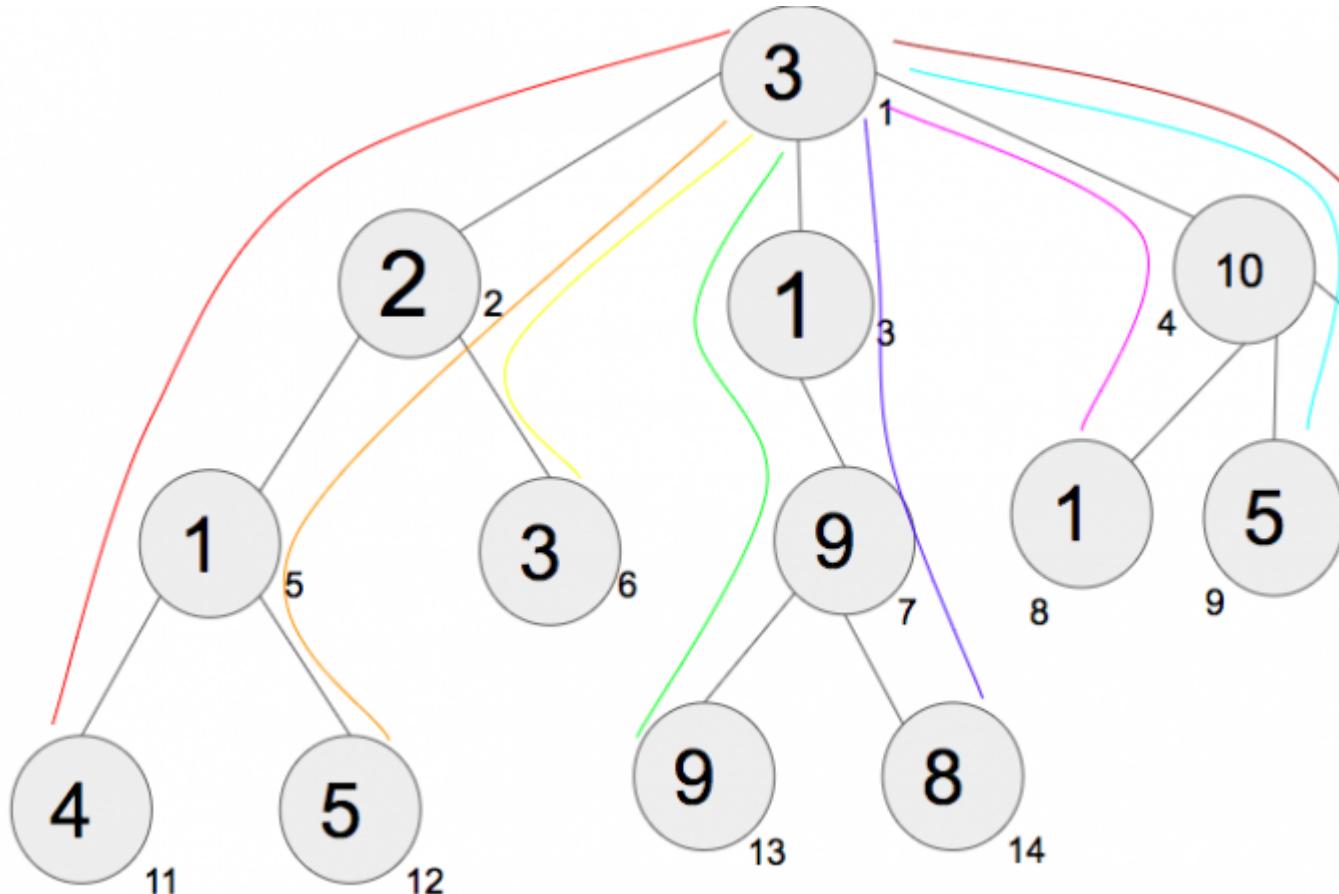
**Pre-requisite:** [DFS](#)

Given a tree with N nodes and N-1 edges, calculate the maximum sum of the node values from root to any of the leaves without re-visiting any node.



Given above is a diagram of a tree with  $N=14$  nodes and  $N-1=13$  edges. The values at node being **3, 2, 1, 10, 1, 3, 9, 1, 5, 3, 4, 5, 9** and **8** respectively for nodes 1, 2, 3, 4....14.

The diagram below shows all the paths from root to leaves :



All the paths are marked by different colors :

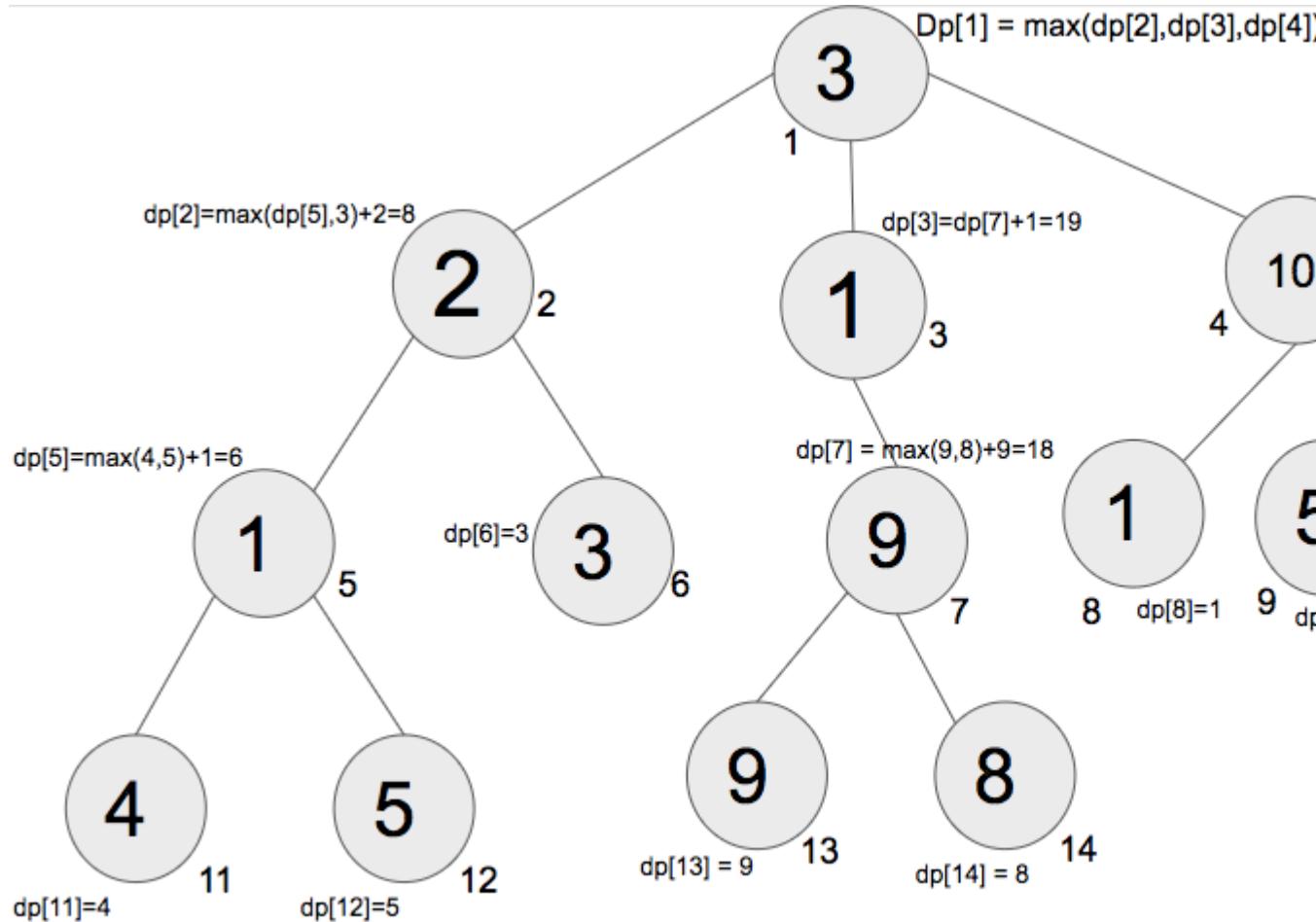
- Path 1(red, 3-2-1-4) : sum of all node values = 10
- Path 2(orange, 3-2-1-5) : sum of all node values = 11
- Path 3(yellow, 3-2-3) : sum of all node values = 8
- Path 4(green, 3-1-9-9) : sum of all node values = 22
- Path 5(violet, 3-1-9-8) : sum of all node values = 21
- Path 6(pink, 3-10-1) : sum of all node values = 14
- Path 7(blue, 3-10-5) : sum of all node values = 18
- Path 8(brown, 3-10-3) : sum of all node values = 16

The answer is 22, as Path 4 has the maximum sum of values of nodes in its path from a root to leaves.

**The greedy approach fails in this case.** Starting from the root and take 3 from the

first level, 10 from the next level and 5 from the third level greedily. Result is path-7 if after following greedy approach, hence do not apply greedy approach over here.

The problem can be solved using **Dynamic Programming on trees**. Start memoizing from the leaves and add the maximum of leaves to the root of every sub-tree. At the last step, there will be root and the sub-tree under it, adding the value at node and maximum of sub-tree will give us the maximum sum of the node values from root to any of the leaves.



The diagram above shows how to **start from the leaves and add the maximum of leaves of a sub-tree to its root**. Move upward and repeat the same procedure of storing the maximum of every sub-tree leaves and adding it to its root. In this example, the maximum of node 11 and 12 is taken to count and then added to node 5(**In this sub-tree, 5 is the root and 11, 12 are its leaves**). Similarly, the maximum of node 13 and 15 is taken to count and then added to node 7. Repeat the steps for every sub-tree till we reach the node.

Let  $DP_i$  be the maximum summation of node values in the path between  $i$  and any of its leaves moving downwards. **Traverse the tree using DFS traversal**. Store the maximum

of all the leaves of the sub-tree, and add it to the root of the sub-tree. At the end,  $DP_1$  will have the maximum sum of the node values from root to any of the leaves without re-visiting any node.

Below is the implementation of the above idea :

### CPP

```
// CPP code to find the maximum path sum
#include <bits/stdc++.h>
using namespace std;

int dp[100];

// function for dfs traversal and to store the
// maximum value in dp[] for every node till the leaves
void dfs(int a[], vector<int> v[], int u, int parent)
{
    // initially dp[u] is always a[u]
    dp[u] = a[u - 1];

    // stores the maximum value from nodes
    int maximum = 0;

    // traverse the tree
    for (int child : v[u]) {

        // if child is parent, then we continue
        // without recursing further
        if (child == parent)
            continue;

        // call dfs for further traversal
        dfs(a, v, child, u);

        // store the maximum of previous visited node
        // and present visited node
        maximum = max(maximum, dp[child]);
    }

    // add the maximum value returned to the parent node
    dp[u] += maximum;
}

// function that returns the maximum value
int maximumValue(int a[], vector<int> v[])
{
    dfs(a, v, 1, 0);
```

```
    return dp[1];
}

// Driver Code
int main()
{
    // number of nodes
    int n = 14;

    // adjacency list
    vector<int> v[n + 1];

    // create undirected edges
    // initialize the tree given in the diagram
    v[1].push_back(2), v[2].push_back(1);
    v[1].push_back(3), v[3].push_back(1);
    v[1].push_back(4), v[4].push_back(1);
    v[2].push_back(5), v[5].push_back(2);
    v[2].push_back(6), v[6].push_back(2);
    v[3].push_back(7), v[7].push_back(3);
    v[4].push_back(8), v[8].push_back(4);
    v[4].push_back(9), v[9].push_back(4);
    v[4].push_back(10), v[10].push_back(4);
    v[5].push_back(11), v[11].push_back(5);
    v[5].push_back(12), v[12].push_back(5);
    v[7].push_back(13), v[13].push_back(7);
    v[7].push_back(14), v[14].push_back(7);

    // values of node 1, 2, 3....14
    int a[] = { 3, 2, 1, 10, 1, 3, 9, 1, 5, 3, 4, 5, 9, 8 };

    // function call
    cout << maximumValue(a, v);

    return 0;
}
```

**Output:**

22

**Time Complexity :**  $O(N)$ , where  $N$  is the number of nodes.

**Source**

<https://www.geeksforgeeks.org/dynamic-programming-trees-set-1/>

## Chapter 144

# Enumeration of Binary Trees

Enumeration of Binary Trees - GeeksforGeeks

A Binary Tree is labeled if every node is assigned a label and a Binary Tree is unlabeled if nodes are not assigned any label.

Below two are considered same unlabeled trees



Below two are considered different labeled trees



How many different Unlabeled Binary Trees can be there with n nodes?

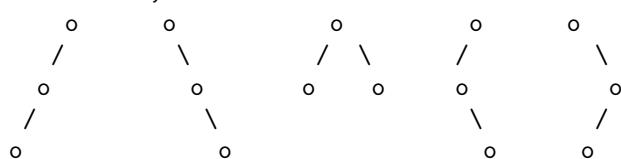
For n = 1, there is only one tree



For n = 2, there are two trees



For n = 3, there are five trees



The idea is to consider all possible pair of counts for nodes in left and right subtrees and multiply the counts for a particular pair. Finally add results of all pairs.

For example, let  $T(n)$  be count for  $n$  nodes.

$T(0) = 1$  [There is only 1 empty tree]

$T(1) = 1$

$T(2) = 2$

$$T(3) = T(0)*T(2) + T(1)*T(1) + T(2)*T(0) = 1*2 + 1*1 + 2*1 = 5$$

$$\begin{aligned} T(4) &= T(0)*T(3) + T(1)*T(2) + T(2)*T(1) + T(3)*T(0) \\ &= 1*5 + 1*2 + 2*1 + 5*1 \\ &= 14 \end{aligned}$$

The above pattern basically represents [n'th Catalan Numbers](#). First few catalan numbers are 1 1 2 5 14 42 132 429 1430 4862,...

$$T(n) = \sum_{i=1}^n T(i-1)T(n-i) = \sum_{i=0}^{n-1} T(i)T(n-i-1) =$$

Here,

$T(i-1)$  represents number of nodes on the left-sub-tree

$T(n-i-1)$  represents number of nodes on the right-sub-tree

n'th Catalan Number can also be evaluated using direct formula.

$$T(n) = (2n)! / (n+1)!n!$$

Number of Binary Search Trees (BST) with  $n$  nodes is also same as number of unlabeled trees. The reason for this is simple, in BST also we can make any key as root, If root is  $i$ 'th key in sorted order, then  $i-1$  keys can go on one side and  $(n-i)$  keys can go on other side.

#### How many labeled Binary Trees can be there with $n$ nodes?

To count labeled trees, we can use above count for unlabeled trees. The idea is simple, every unlabeled tree with  $n$  nodes can create  $n!$  different labeled trees by assigning different permutations of labels to all nodes.

Therefore,

$$\begin{aligned} \text{Number of Labeled Trees} &= (\text{Number of unlabeled trees}) * n! \\ &= [(2n)! / (n+1)!n!] * n! \end{aligned}$$

For example for  $n = 3$ , there are  $5 * 3! = 5*6 = 30$  different labeled trees

#### Source

<https://www.geeksforgeeks.org/enumeration-of-binary-trees/>

## Chapter 145

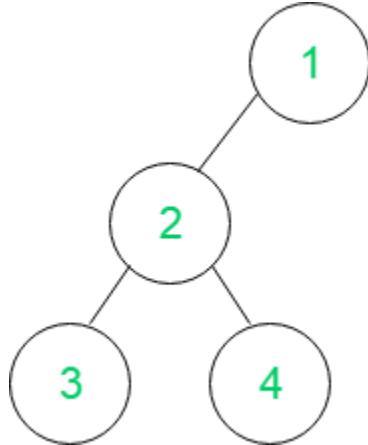
# Euler Tour of Tree

Euler Tour of Tree - GeeksforGeeks

A Tree is a generalization of connected graph where it has N nodes that will have exactly N-1 edges, i.e one edge between every pair of vertices. Find the Euler tour of tree represented by adjacency list.

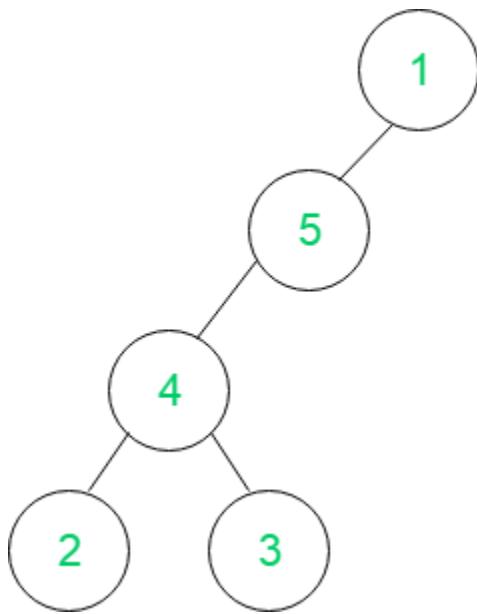
Examples:

Input :



Output : 1 2 3 2 4 2 1

Input :

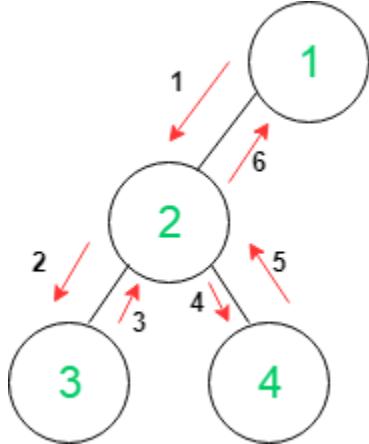


Output : 1 5 4 2 4 3 4 5 1

**Euler tour** is defined as a way of traversing tree such that each vertex is added to the tour when we visit it (either moving down from parent vertex or returning from child vertex). We start from root and reach back to root after visiting all vertices.

**It requires exactly  $2*N-1$  vertices to store Euler tour.**

**Approach:** We will run DFS(Depth first search) algorithm on Tree as:



(1) **Visit root node, i.e 1**

vis[1]=1, Euler[0]=1

run dfs() for all unvisited adjacent nodes(2)

(2) **Visit node 2**

vis[2]=1, Euler[1]=2

run dfs() for all unvisited adjacent nodes(3, 4)

**(3) Visit node 3**

vis[3]=1, Euler[2]=3

All adjacent nodes are already visited, return to parent node  
and add parent to Euler tour Euler[3]=2

**(4) Visit node 4**

vis[4]=1, Euler[4]=4

All adjacent nodes are already visited, return to parent node  
and add parent to Euler tour, Euler[5]=2

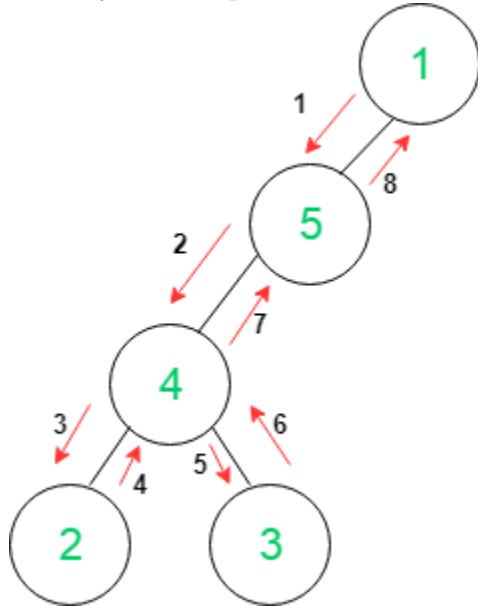
**(5) Visit node 2**

All adjacent nodes are already visited, return to parent node  
and add parent to Euler tour, Euler[6]=1

**(6) Visit node 1**

All adjacent nodes are already visited, and node 1 is root node  
so, we stop our recursion here.

Similarly, for example 2:



```

// CPP program to print Euler tour of a
// tree.
#include <bits/stdc++.h>
using namespace std;

#define MAX 1001

// Adjacency list representation of tree
vector<int> adj[MAX];

// Visited array to keep track visited
// nodes on tour

```

```
int vis[MAX];

// Array to store Euler Tour
int Euler[2 * MAX];

// Function to add edges to tree
void add_edge(int u, int v)
{
    adj[u].push_back(v);
    adj[v].push_back(u);
}

// Function to store Euler Tour of tree
void eulerTree(int u, int &indx)
{
    vis[u] = 1;
    Euler[indx++] = u;
    for (auto it : adj[u]) {
        if (!vis[it]) {
            eulerTree(it, indx);
            Euler[indx++] = u;
        }
    }
}

// Function to print Euler Tour of tree
void printEulerTour(int root, int N)
{
    int index = 0;
    eulerTree(root, index);
    for (int i = 0; i < (2*N-1); i++)
        cout << Euler[i] << " ";
}

// Driver code
int main()
{
    int N = 4;

    add_edge(1, 2);
    add_edge(2, 3);
    add_edge(2, 4);

    // Consider 1 as root and print
    // Euler tour
    printEulerTour(1, N);

    return 0;
}
```

}

**Output:**

1 2 3 2 4 2 1

Auxiliary Space : $O(N)$   
Time Complexity:  $O(N)$

**Source**

<https://www.geeksforgeeks.org/euler-tour-tree/>

## Chapter 146

# Euler Tour | Subtree Sum using Segment Tree

Euler Tour | Subtree Sum using Segment Tree - GeeksforGeeks

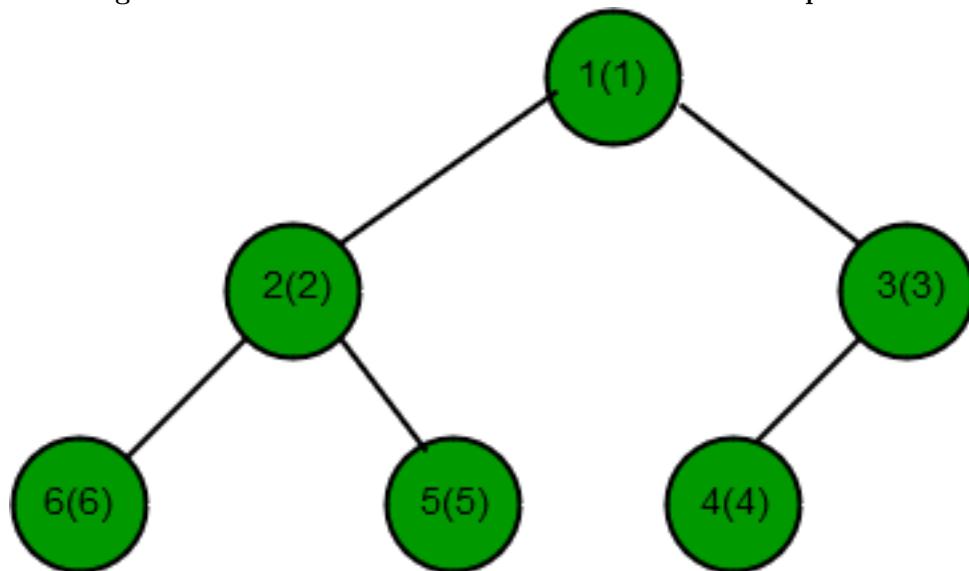
**Euler tour tree (ETT)** is a method for representing a rooted tree as a number sequence. When traversing the tree using [Depth for search\(DFS\)](#), insert each node in a vector twice, once while entered it and next after visiting all its children. This method is very useful for solving subtree problems and one such problem is **Subtree Sum**.

**Prerequisite :** [Segment Tree\(Sum of given range\)](#)

**Naive Approach :**

Consider a rooted tree with 6 vertices connected as given in the below diagram. Apply DFS for different queries.

The weight associated with each node is written inside the parenthesis.

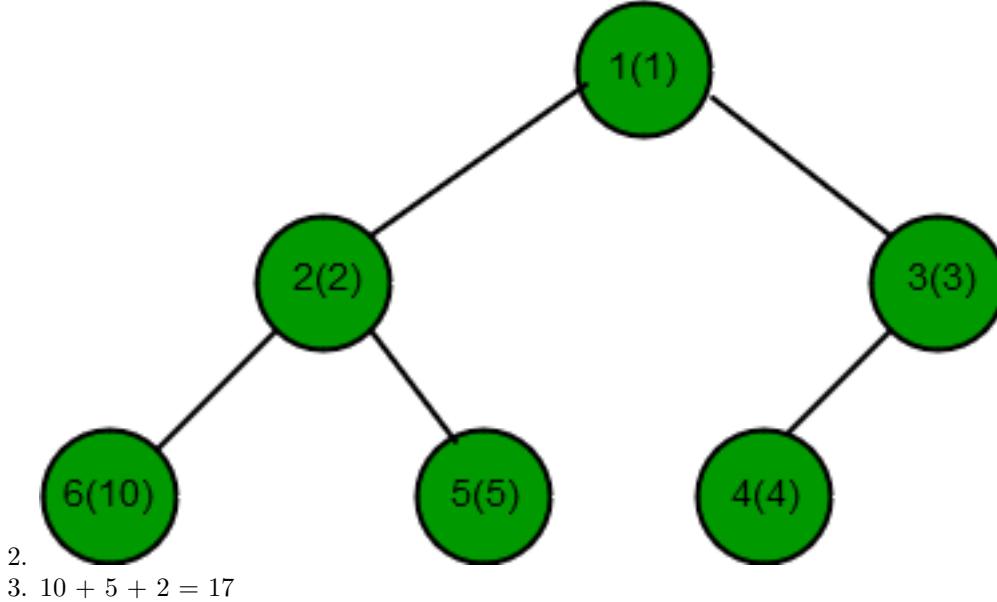


**Queries :**

1. Sum of all the subtrees of node 1.
2. Update the value of node 6 to 10.
3. Sum of all the subtrees of node 2.

**Answers :**

1.  $6 + 5 + 4 + 3 + 2 + 1 = 21$



2.

3.  $10 + 5 + 2 = 17$

**Time Complexity Analysis :**

Such queries can be performed using depth first search(dfs) in **O(n)** time complexity.

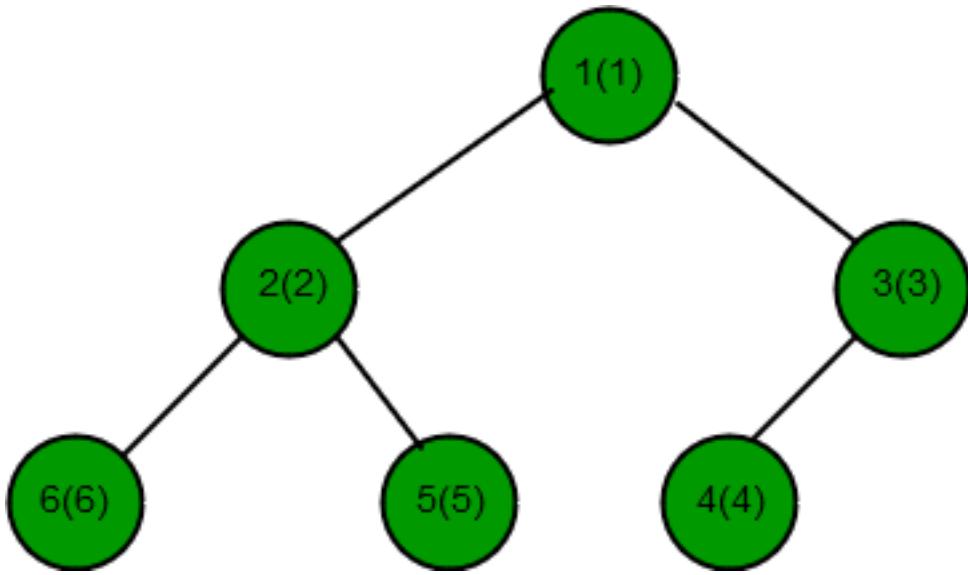
**Efficient Approach :**

The time complexity for such queries can be reduced to **O(log(n))** time by converting the rooted tree into segment tree using Euler tour technique. So, When the number of queries are q, the total complexity becomes **O(q\*5log(n))**.

**Euler Tour :**

In Euler tour Technique, each vertex is added to the vector twice, while descending into it and while leaving it.

Let us understand with the help of previous example :



On performing depth first search(DFS) using euler tour technique on the given rooted tree, the vector so formed is :

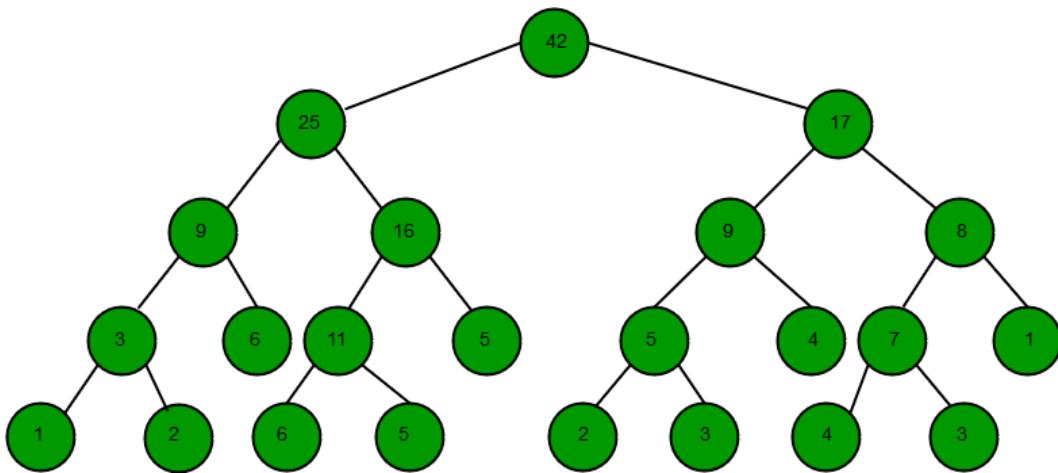
```
s[]={1, 2, 6, 6, 5, 5, 2, 3, 4, 4, 3, 1}
```

```
// DFS function to traverse the tree
int dfs(int root)
{
    s.push_back(root);
    if (v[root].size() == 0)
        return root;

    for (int i = 0; i < v[root].size(); i++) {
        int temp = dfs(v[root][i]);
        s.push_back(temp);
    }
    return root;
}
```

Now, use vector s[ ] to [Create Segment Tree](#).

Below is the representation of segment tree of vector s[ ].



For the output and update query, store the entry time and exit time(which serve as index range) for each node of the rooted tree.

```
s[]={1, 2, 6, 6, 5, 5, 2, 3, 4, 4, 3, 1}
```

Node	Entry time	Exit time
1	1	12
2	2	7
3	8	11
4	9	10
5	5	6
6	3	4

#### Query of type 1 :

Find the range sum on segment tree for output query where range is exit time and entry time of the rooted tree node. Deduce that the answer is always twice the expected answer because each node is added twice in segment tree. So reduce the answer by half.

#### Query of type 2 :

For update query, update the leaf node of segment tree at the entry time and exit time of the rooted tree node.

**Below is the implementation of above approach :**

```
// C++ program for implementation of
// Euler Tour | Subtree Sum.
#include <bits/stdc++.h>
using namespace std;

vector<int> v[1001];
vector<int> s;
int seg[1001] = { 0 };
```

```

// Value/Weight of each node of tree,
// value of 0th(no such node) node is 0.
int ar[] = { 0, 1, 2, 3, 4, 5, 6 };

int vertices = 6;
int edges = 5;

// A recursive function that constructs
// Segment Tree for array ar[] = { }.
// 'pos' is index of current node
// in segment tree seg[].
int segment(int low, int high, int pos)
{
    if (high == low) {
        seg[pos] = ar[s[low]];
    }
    else {
        int mid = (low + high) / 2;
        segment(low, mid, 2 * pos);
        segment(mid + 1, high, 2 * pos + 1);
        seg[pos] = seg[2 * pos] + seg[2 * pos + 1];
    }
}

/* Return sum of elements in range
   from index l to r . It uses the
   seg[] array created using segment()
   function. 'pos' is index of current
   node in segment tree seg[].

*/
int query(int node, int start,
          int end, int l, int r)
{
    if (r < start || end < l) {
        return 0;
    }

    if (l <= start && end <= r) {
        return seg[node];
    }

    int mid = (start + end) / 2;
    int p1 = query(2 * node, start,
                  mid, l, r);
    int p2 = query(2 * node + 1, mid + 1,
                  end, l, r);

    return (p1 + p2);
}

```

```

}

/* A recursive function to update the
   nodes which have the given index in
   their range. The following are
   parameters pos --> index of current
   node in segment tree seg[]. idx -->
   index of the element to be updated.
   This index is in input array.
   val --> Value to be change at node idx
*/
int update(int pos, int low, int high,
           int idx, int val)
{
    if (low == high) {
        seg[pos] = val;
    }
    else {
        int mid = (low + high) / 2;

        if (low <= idx && idx <= mid) {
            update(2 * pos, low, mid,
                   idx, val);
        }
        else {
            update(2 * pos + 1, mid + 1,
                   high, idx, val);
        }

        seg[pos] = seg[2 * pos] + seg[2 * pos + 1];
    }
}

/* A recursive function to form array
   ar[] from a directed tree .
*/
int dfs(int root)
{
    // pushing each node in vector s
    s.push_back(root);
    if (v[root].size() == 0)
        return root;

    for (int i = 0; i < v[root].size(); i++) {
        int temp = dfs(v[root][i]);
        s.push_back(temp);
    }
    return root;
}

```

```

}

// Driver program to test above functions
int main()
{
    // Edges between the nodes
    v[1].push_back(2);
    v[1].push_back(3);
    v[2].push_back(6);
    v[2].push_back(5);
    v[3].push_back(4);

    // Calling dfs function.
    int temp = dfs(1);
    s.push_back(temp);

    // Storing entry time and exit
    // time of each node
    vector<pair<int, int> > p;

    for (int i = 0; i <= vertices; i++)
        p.push_back(make_pair(0, 0));

    for (int i = 0; i < s.size(); i++) {
        if (p[s[i]].first == 0)
            p[s[i]].first = i + 1;
        else
            p[s[i]].second = i + 1;
    }

    // Build segment tree from array ar[].
    segment(0, s.size() - 1, 1);

    // query of type 1 return the
    // sum of subtree at node 1.
    int node = 1;
    int e = p[node].first;
    int f = p[node].second;

    int ans = query(1, 1, s.size(), e, f);

    // print the sum of subtree
    cout << "Subtree sum of node " << node << " is : " << (ans / 2) << endl;

    // query of type 2 return update
    // the subtree at node 6.
    int val = 10;
    node = 6;
}

```

```
e = p[node].first;
f = p[node].second;
update(1, 1, s.size(), e, val);
update(1, 1, s.size(), f, val);

// query of type 1 return the
// sum of subtree at node 2.
node = 2;

e = p[node].first;
f = p[node].second;

ans = query(1, 1, s.size(), e, f);

// print the sum of subtree
cout << "Subtree sum of node " << node << " is : " << (ans / 2) << endl;

return 0;
}
```

**Output:**

```
Subtree sum of node 1 is : 21
Subtree sum of node 2 is : 17
```

**Time Complexity :**  $O(q * \log(n))$

**Source**

<https://www.geeksforgeeks.org/euler-tour-subtree-sum-using-segment-tree/>

## Chapter 147

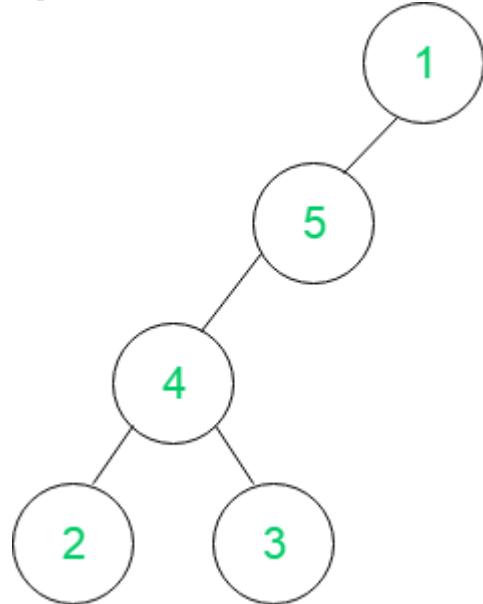
# Euler tour of Binary Tree

Euler tour of Binary Tree - GeeksforGeeks

Given a binary tree where each node can have at most two child nodes, the task is to find the Euler tour of the binary tree. Euler tour is represented by a pointer to the topmost node in the tree. If the tree is empty, then value of root is NULL.

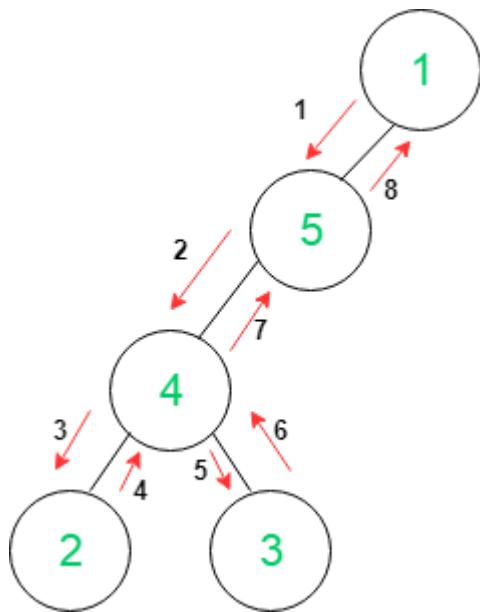
Examples:

Input :



Output: 1 5 4 2 4 3 4 5 1

Approach:



- (1) First, start with root node 1, **Euler[0]=1**
- (2) Go to left node i.e, node 5, **Euler[1]=5**
- (3) Go to left node i.e, node 4, **Euler[2]=4**
- (4) Go to left node i.e, node 2, **Euler[3]=2**
- (5) Go to left node i.e, NULL, go to parent node 4 **Euler[4]=4**
- (6) Go to right node i.e, node 3 **Euler[5]=3**
- (7) No child, go to parent, node 4 **Euler[6]=4**
- (8) All child discovered, go to parent node 5 **Euler[7]=5**
- (9) All child discovered, go to parent node 1 **Euler[8]=1**

[Euler tour of tree](#) has been already discussed where it can be applied to N-ary tree which is represented by adjacency list. If a Binary tree is represented by the classical structured way by links and nodes, then there need to first convert the tree into adjacency list representation and then we can find the Euler tour if we want to apply method discussed in the original post. But this increases the space complexity of the program. Here, In this post, a generalized space-optimized version is discussed which can be directly applied to binary trees represented by structure nodes.

This method :

- (1) Works without the use of Visited arrays.
- (2) Requires exactly  $2*N-1$  vertices to store Euler tour.

```

// C++ program to find euler tour of binary tree
#include <bits/stdc++.h>
using namespace std;

/* A tree node structure */
struct Node {
    int data;
    struct Node* left;
    
```

```
    struct Node* right;
};

/* Utility function to create a new Binary Tree node */
struct Node* newNode(int data)
{
    struct Node* temp = new struct Node;
    temp->data = data;
    temp->left = temp->right = NULL;
    return temp;
}

// Find Euler Tour
void eulerTree(struct Node* root, vector<int> &Euler)
{
    // store current node's data
    Euler.push_back(root->data);

    // If left node exists
    if (root->left)
    {
        // traverse left subtree
        eulerTree(root->left, Euler);

        // store parent node's data
        Euler.push_back(root->data);
    }

    // If right node exists
    if (root->right)
    {
        // traverse right subtree
        eulerTree(root->right, Euler);

        // store parent node's data
        Euler.push_back(root->data);
    }
}

// Function to print Euler Tour of tree
void printEulerTour(Node *root)
{
    // Stores Euler Tour
    vector<int> Euler;

    eulerTree(root, Euler);

    for (int i = 0; i < Euler.size(); i++)
```

```
        cout << Euler[i] << " ";
}

/* Driver function to test above functions */
int main()
{
    // Constructing tree given in the above figure
    Node* root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->left = newNode(6);
    root->right->right = newNode(7);
    root->right->left->right = newNode(8);

    // print Euler Tour
    printEulerTour(root);

    return 0;
}
```

**Output:**

1 2 4 2 5 2 1 3 6 8 6 3 7 3 1

Time Complexity:  $O(2*N-1)$  where N is number of nodes in the tree.  
Auxiliary Space :  $O(2*N-1)$  where N is number of nodes in the tree.

**Source**

<https://www.geeksforgeeks.org/euler-tour-binary-tree/>

## Chapter 148

# Evaluation of Expression Tree

Evaluation of Expression Tree - GeeksforGeeks

Given a simple [expression tree](#), consisting of basic binary operators i.e., + , - ; \* and / and some integers, evaluate the expression tree.

Examples:

Input :

Root node of the below tree

Output :

100

Input :

Root node of the below tree

Output :

110

As all the operators in the tree are binary hence each node will have either 0 or 2 children.  
As it can be inferred from the examples above , the integer values would appear at the leaf nodes , while the interior nodes represent the operators.

To evaluate the syntax tree , a recursive approach can be followed .

Algorithm :

```
Let t be the syntax tree  
If t is not null then
```

```
If t.info is operand then
    Return t.info
Else
    A = solve(t.left)
    B = solve(t.right)
    return A operator B
where operator is the info contained in t
```

The time complexity would be  $O(n)$ , as each node is visited once. Below is a C++ program for the same:

### C/C++

```
// C++ program to evaluate an expression tree
#include <bits/stdc++.h>
using namespace std;

// Class to represent the nodes of syntax tree
class node
{
public:
    string info;
    node *left = NULL, *right = NULL;
    node(string x)
    {
        info = x;
    }
};

// Utility function to return the integer value
// of a given string
int toInt(string s)
{
    int num = 0;
    for (int i=0; i<s.length(); i++)
        num = num*10 + (int(s[i])-48);
    return num;
}

// This function receives a node of the syntax tree
// and recursively evaluates it
int eval(node* root)
{
    // empty tree
    if (!root)
        return 0;
```

```
// leaf node i.e, an integer
if (!root->left && !root->right)
    return toInt(root->info);

// Evaluate left subtree
int l_val = eval(root->left);

// Evaluate right subtree
int r_val = eval(root->right);

// Check which operator to apply
if (root->info=="+" )
    return l_val+r_val;

if (root->info=="-")
    return l_val-r_val;

if (root->info=="*")
    return l_val*r_val;

return l_val/r_val;
}

//driver function to check the above program
int main()
{
    // create a syntax tree
    node *root = new node("+");
    root->left = new node("*");
    root->left->left = new node("5");
    root->left->right = new node("4");
    root->right = new node("-");
    root->right->left = new node("100");
    root->right->right = new node("20");
    cout << eval(root) << endl;

    delete(root);

    root = new node("+");
    root->left = new node("*");
    root->left->left = new node("5");
    root->left->right = new node("4");
    root->right = new node("-");
    root->right->left = new node("100");
    root->right->right = new node("/");
    root->right->right->left = new node("20");
    root->right->right->right = new node("2");
}
```

```
    cout << eval(root);
    return 0;
}
```

### Python

```
# Python program to evaluate expression tree

# Class to represent the nodes of syntax tree
class node:
    def __init__(self, value):
        self.left = None
        self.data = value
        self.right = None

# This function receives a node of the syntax tree
# and recursively evaluate it
def evaluateExpressionTree(root):

    # empty tree
    if root is None:
        return 0

    # leaf node
    if root.left is None and root.right is None:
        return int(root.data)

    # evaluate left tree
    left_sum = evaluateExpressionTree(root.left)

    # evaluate right tree
    right_sum = evaluateExpressionTree(root.right)

    # check which operation to apply
    if root.data == '+':
        return left_sum + right_sum

    elif root.data == '-':
        return left_sum - right_sum

    elif root.data == '*':
        return left_sum * right_sum

    else:
        return left_sum / right_sum

# Driver function to test above problem
```

```
if __name__=='__main__':  
  
    # creating a sample tree  
    root = node('+')  
    root.left = node('*')  
    root.left.left = node('5')  
    root.left.right = node('4')  
    root.right = node('-')  
    root.right.left = node('100')  
    root.right.right = node('20')  
    print evaluateExpressionTree(root)  
  
root = None  
  
#creating a sample tree  
root = node('+')  
root.left = node('*')  
root.left.left = node('5')  
root.left.right = node('4')  
root.right = node('-')  
root.right.left = node('100')  
root.right.right = node('/')  
root.right.right.left = node('20')  
root.right.right.right = node('2')  
print evaluateExpressionTree(root)  
  
# This code is contributed by Harshit Sidhwa
```

Output:

```
100  
110
```

## Source

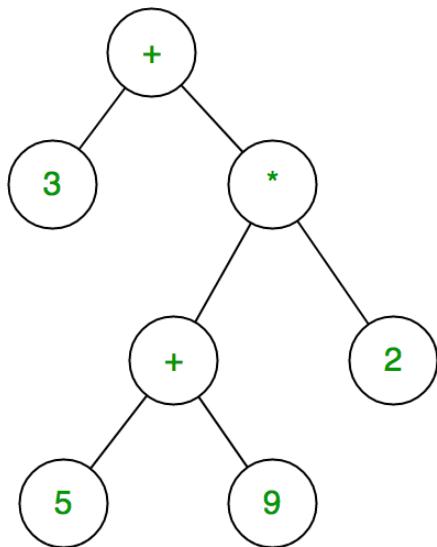
<https://www.geeksforgeeks.org/evaluation-of-expression-tree/>

## Chapter 149

# Expression Tree

Expression Tree - GeeksforGeeks

Expression tree is a binary tree in which each internal node corresponds to operator and each leaf node corresponds to operand so for example expression tree for  $3 + ((5+9)*2)$  would be:



Inorder traversal of expression tree produces infix version of given postfix expression (same with preorder traversal it gives prefix expression)

**Evaluating the expression represented by expression tree:**

```
Let t be the expression tree  
If t is not null then
```

```
If t.value is operand then
    Return t.value
A = solve(t.left)
B = solve(t.right)

// calculate applies operator 't.value'
// on A and B, and returns value
Return calculate(A, B, t.value)
```

**Construction of Expression Tree:**

Now For constructing expression tree we use a stack. We loop through input expression and do following for every character.

- 1) If character is operand push that into stack
- 2) If character is operator pop two values from stack make them its child and push current node again.

At the end only element of stack will be root of expression tree.

Below is the implementation :

**C/C++**

```
// C++ program for expression tree
#include<bits/stdc++.h>
using namespace std;

// An expression tree node
struct et
{
    char value;
    et* left, *right;
};

// A utility function to check if 'c'
// is an operator
bool isOperator(char c)
{
    if (c == '+' || c == '-' ||
        c == '*' || c == '/' ||
        c == '^')
        return true;
    return false;
}

// Utility function to do inorder traversal
void inorder(et *t)
{
    if(t)
    {
```

```

        inorder(t->left);
        printf("%c ", t->value);
        inorder(t->right);
    }
}

// A utility function to create a new node
et* newNode(int v)
{
    et *temp = new et;
    temp->left = temp->right = NULL;
    temp->value = v;
    return temp;
};

// Returns root of constructed tree for given
// postfix expression
et* constructTree(char postfix[])
{
    stack<et *> st;
    et *t, *t1, *t2;

    // Traverse through every character of
    // input expression
    for (int i=0; i<strlen(postfix); i++)
    {
        // If operand, simply push into stack
        if (!isOperator(postfix[i]))
        {
            t = newNode(postfix[i]);
            st.push(t);
        }
        else // operator
        {
            t = newNode(postfix[i]);

            // Pop two top nodes
            t1 = st.top(); // Store top
            st.pop(); // Remove top
            t2 = st.top();
            st.pop();

            // make them children
            t->right = t1;
            t->left = t2;

            // Add this subexpression to stack
            st.push(t);
        }
    }
}

```

```
        }
    }

    // only element will be root of expression
    // tree
    t = st.top();
    st.pop();

    return t;
}

// Driver program to test above
int main()
{
    char postfix[] = "ab+ef*g*-";
    et* r = constructTree(postfix);
    printf("infix expression is \n");
    inorder(r);
    return 0;
}
```

**Java**

```
// Java program to construct an expression tree

import java.util.Stack;

// Java program for expression tree
class Node {

    char value;
    Node left, right;

    Node(char item) {
        value = item;
        left = right = null;
    }
}

class ExpressionTree {

    // A utility function to check if 'c'
    // is an operator

    boolean isOperator(char c) {
        if (c == '+' || c == '-'
            || c == '*' || c == '/'
            || c == '^') {
```

```
        return true;
    }
    return false;
}

// Utility function to do inorder traversal
void inorder(Node t) {
    if (t != null) {
        inorder(t.left);
        System.out.print(t.value + " ");
        inorder(t.right);
    }
}

// Returns root of constructed tree for given
// postfix expression
Node constructTree(char postfix[]) {
    Stack<Node> st = new Stack();
    Node t, t1, t2;

    // Traverse through every character of
    // input expression
    for (int i = 0; i < postfix.length; i++) {

        // If operand, simply push into stack
        if (!isOperator(postfix[i])) {
            t = new Node(postfix[i]);
            st.push(t);
        } else // operator
        {
            t = new Node(postfix[i]);

            // Pop two top nodes
            // Store top
            t1 = st.pop();           // Remove top
            t2 = st.pop();

            // make them children
            t.right = t1;
            t.left = t2;

            // System.out.println(t1 + "" + t2);
            // Add this subexpression to stack
            st.push(t);
        }
    }

    // only element will be root of expression
}
```

```
// tree
t = st.peek();
st.pop();

return t;
}

public static void main(String args[]) {

    ExpressionTree et = new ExpressionTree();
    String postfix = "ab+ef*g*-";
    char[] charArray = postfix.toCharArray();
    Node root = et.constructTree(charArray);
    System.out.println("infix expression is");
    et.inorder(root);

}
}

// This code has been contributed by Mayank Jaiswal
```

### Python

```
# Python program for expression tree

# An expression tree node
class Et:

    # Constructor to create a node
    def __init__(self , value):
        self.value = value
        self.left = None
        self.right = None

    # A utility function to check if 'c'
    # is an operator
    def isOperator(c):
        if (c == '+' or c == '-' or c == '*'
            or c == '/' or c == '^'):
            return True
        else:
            return False

    # A utility function to do inorder traversal
    def inorder(t):
        if t is not None:
            inorder(t.left)
            print t.value,
```

```
inorder(t.right)

# Returns root of constructed tree for
# given postfix expression
def constructTree(postfix):
    stack = []

    # Traverse through every character of input expression
    for char in postfix :

        # if operand, simply push into stack
        if not isOperator(char):
            t = Et(char)
            stack.append(t)

        # Operator
        else:

            # Pop two top nodes
            t = Et(char)
            t1 = stack.pop()
            t2 = stack.pop()

            # make them children
            t.right = t1
            t.left = t2

            # Add this subexpression to stack
            stack.append(t)

    # Only element will be the root of expression tree
    t = stack.pop()

    return t

# Driver program to test above
postfix = "ab+ef*g*-"
r = constructTree(postfix)
print "Infix expression is"
inorder(r)
```

Output:

```
infix expression is
a + b - e * f * g
```

This article is contributed by [Utkarsh Trivedi](#). Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

**Source**

<https://www.geeksforgeeks.org/expression-tree/>

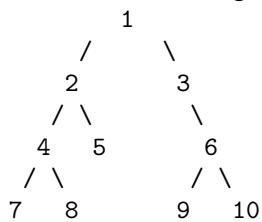
## Chapter 150

# Extract Leaves of a Binary Tree in a Doubly Linked List

Extract Leaves of a Binary Tree in a Doubly Linked List - GeeksforGeeks

Given a Binary Tree, extract all leaves of it in a **Doubly Linked List (DLL)**. Note that the DLL need to be created in-place. Assume that the node structure of DLL and Binary Tree is same, only the meaning of left and right pointers are different. In DLL, left means previous pointer and right means next pointer.

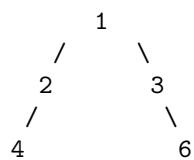
Let the following be input binary tree



Output:

Doubly Linked List  
785910

Modified Tree:



We need to traverse all leaves and connect them by changing their left and right pointers.

We also need to remove them from Binary Tree by changing left or right pointers in parent nodes. There can be many ways to solve this. In the following implementation, we add leaves at the beginning of current linked list and update head of the list using pointer to head pointer. Since we insert at the beginning, we need to process leaves in reverse order. For reverse order, we first traverse the right subtree then the left subtree. We use return values to update left or right pointers in parent nodes.

C

```
// C program to extract leaves of a Binary Tree in a Doubly Linked List
#include <stdio.h>
#include <stdlib.h>

// Structure for tree and linked list
struct Node
{
    int data;
    struct Node *left, *right;
};

// Main function which extracts all leaves from given Binary Tree.
// The function returns new root of Binary Tree (Note that root may change
// if Binary Tree has only one node). The function also sets *head_ref as
// head of doubly linked list. left pointer of tree is used as prev in DLL
// and right pointer is used as next
struct Node* extractLeafList(struct Node *root, struct Node **head_ref)
{
    // Base cases
    if (root == NULL)  return NULL;

    if (root->left == NULL && root->right == NULL)
    {
        // This node is going to be added to doubly linked list
        // of leaves, set right pointer of this node as previous
        // head of DLL. We don't need to set left pointer as left
        // is already NULL
        root->right = *head_ref;

        // Change left pointer of previous head
        if (*head_ref != NULL) (*head_ref)->left = root;

        // Change head of linked list
        *head_ref = root;

        return NULL; // Return new root
    }

    // Recur for right and left subtrees
    root->right = extractLeafList(root->right, head_ref);
}
```

```
root->left = extractLeafList(root->left, head_ref);

return root;
}

// Utility function for allocating node for Binary Tree.
struct Node* newNode(int data)
{
    struct Node* node = (struct Node*)malloc(sizeof(struct Node));
    node->data = data;
    node->left = node->right = NULL;
    return node;
}

// Utility function for printing tree in In-Order.
void print(struct Node *root)
{
    if (root != NULL)
    {
        print(root->left);
        printf("%d ",root->data);
        print(root->right);
    }
}

// Utility function for printing double linked list.
void printList(struct Node *head)
{
    while (head)
    {
        printf("%d ", head->data);
        head = head->right;
    }
}

// Driver program to test above function
int main()
{
    struct Node *head = NULL;
    struct Node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->right = newNode(6);
    root->left->left->left = newNode(7);
    root->left->left->right = newNode(8);
    root->right->right->left = newNode(9);
```

```
root->right->right->right = newNode(10);

printf("Inorder Traversal of given Tree is:\n");
print(root);

root = extractLeafList(root, &head);

printf("\nExtracted Double Linked list is:\n");
printList(head);

printf("\nInorder traversal of modified tree is:\n");
print(root);
return 0;
}
```

**Java**

```
// Java program to extract leaf nodes from binary tree
// using double linked list

// A binary tree node
class Node
{
    int data;
    Node left, right;

    Node(int item)
    {
        data = item;
        right = left = null;
    }
}

public class BinaryTree
{
    Node root;
    Node head; // will point to head of DLL
    Node prev; // temporary pointer

    // The main function that links the list list to be traversed
    public Node extractLeafList(Node root)
    {
        if (root == null)
            return null;
        if (root.left == null && root.right == null)
        {
            if (head == null)
            {
```

```
        head = root;
        prev = root;
    }
    else
    {
        prev.right = root;
        root.left = prev;
        prev = root;
    }
    return null;
}
root.left = extractLeafList(root.left);
root.right = extractLeafList(root.right);
return root;
}

//Prints the DLL in both forward and reverse directions.
public void printDLL(Node head)
{
    Node last = null;
    while (head != null)
    {
        System.out.print(head.data + " ");
        last = head;
        head = head.right;
    }
}

void inorder(Node node)
{
    if (node == null)
        return;
    inorder(node.left);
    System.out.print(node.data + " ");
    inorder(node.right);
}

// Driver program to test above functions
public static void main(String args[])
{
    BinaryTree tree = new BinaryTree();
    tree.root = new Node(1);
    tree.root.left = new Node(2);
    tree.root.right = new Node(3);

    tree.root.left.left = new Node(4);
    tree.root.left.right = new Node(5);
    tree.root.right.right = new Node(6);
```

```
tree.root.left.left.left = new Node(7);
tree.root.left.left.right = new Node(8);
tree.root.right.right.left = new Node(9);
tree.root.right.right.right = new Node(10);

System.out.println("Inorder traversal of given tree is : ");
tree.inorder(tree.root);
tree.extractLeafList(tree.root);
System.out.println("");
System.out.println("Extracted double link list is : ");
tree.printDLL(tree.head);
System.out.println("");
System.out.println("Inorder traversal of modified tree is : ");
tree.inorder(tree.root);
}

}

// This code has been contributed by Mayank Jaiswal(mayank_24)
```

### Python

```
# Python program to extract leaf nodes from binary tree
# using double linked list

# A binary tree node
class Node:

    # Constructor to create a new node
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

    # Main function which extracts all leaves from given Binary Tree.
    # The function returns new root of Binary Tree (Note that
    # root may change if Binary Tree has only one node).
    # The function also sets *head_ref as head of doubly linked list.
    # left pointer of tree is used as prev in DLL
    # and right pointer is used as next
    def extractLeafList(root):

        # Base Case
        if root is None:
            return None

        if root.left is None and root.right is None:
            # This node is going to be added to doubly linked
            # list of leaves, set pointer of this node as
```

```
# previous head of DLL. We don't need to set left
# pointer as left is already None
root.right = extractLeafList.head

# Change the left pointer of previous head
if extractLeafList.head is not None:
    extractLeafList.head.left = root

# Change head of linked list
extractLeafList.head = root

return None # Return new root

# Recur for right and left subtrees
root.right = extractLeafList(root.right)
root.left = extractLeafList(root.left)

return root

# Utility function for printing tree in InOrder
def printInorder(root):
    if root is not None:
        printInorder(root.left)
        print root.data,
        printInorder(root.right)

def printList(head):
    while(head):
        if head.data is not None:
            print head.data,
        head = head.right

# Driver program to test above function
extractLeafList.head = Node(None)
root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)
root.right.right = Node(6)
root.left.left.left = Node(7)
root.left.left.right = Node(8)
root.right.right.left = Node(9)
root.right.right.right = Node(10)

print "Inorder traversal of given tree is:"
printInorder(root)
```

```
root = extractLeafList(root)

print "\nExtract Double Linked List is:"
printList(extractLeafList.head)

print "\nInorder traversal of modified tree is:"
printInorder(root)
```

Output:

```
Inorder Traversal of given Tree is:
7 4 8 2 5 1 3 9 6 10
Extracted Double Linked list is:
7 8 5 9 10
Inorder traversal of modified tree is:
4 2 1 3 6
```

Time Complexity: O(n), the solution does a single traversal of given Binary Tree.

This article is contributed by [Chandra Prakash](#). Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## Source

<https://www.geeksforgeeks.org/connect-leaves-doubly-linked-list/>

## Chapter 151

# Factor Tree of a given Number

Factor Tree of a given Number - GeeksforGeeks

Factor Tree is an intuitive method to understand factors of a number. It shows how all the factors are been derived from the number. It is a special diagram where you find the factors of a number, then the factors of those numbers, etc until you can't factor anymore. The ends are all the prime factors of the original number.

Example:

```
Input : v = 48
Output : Root of below tree
        48
        / \
       2   24
          / \
         2   12
            / \
           2   6
              / \
             2   3
```

The factor tree is created recursively. A binary tree is used.

1. We start with a number and find the minimum divisor possible.
2. Then, we divide the parent number by the minimum divisor.
3. We store both the divisor and quotient as two children of the parent number.
4. Both the children are sent into function recursively.
5. If a divisor less than half the number is not found, two children are stored as NULL.

```
// C++ program to construct Factor Tree for
```

```
// a given number
#include<bits/stdc++.h>
using namespace std;

// Tree node
struct Node
{
    struct Node *left, *right;
    int key;
};

// Utility function to create a new tree Node
Node* newNode(int key)
{
    Node* temp = new Node;
    temp->key = key;
    temp->left = temp->right = NULL;
    return temp;
}

// Constructs factor tree for given value and stores
// root of tree at given reference.
void createFactorTree(struct Node **node_ref, int v)
{
    (*node_ref) = newNode(v);

    // the number is factorized
    for (int i = 2 ; i < v/2 ; i++)
    {
        if (v % i != 0)
            continue;

        // If we found a factor, we construct left
        // and right subtrees and return. Since we
        // traverse factors starting from smaller
        // to greater, left child will always have
        // smaller factor
        createFactorTree(&(*node_ref)->left), i);
        createFactorTree(&(*node_ref)->right), v/i);
        return;
    }
}

// Iterative method to find height of Binary Tree
void printLevelOrder(Node *root)
{
    // Base Case
    if (root == NULL)  return;
```

```
queue<Node *> q;
q.push(root);

while (q.empty() == false)
{
    // Print front of queue and remove
    // it from queue
    Node *node = q.front();
    cout << node->key << " ";
    q.pop();
    if (node->left != NULL)
        q.push(node->left);
    if (node->right != NULL)
        q.push(node->right);
}
}

// driver program
int main()
{
    int val = 48;// sample value
    struct Node *root = NULL;
    createFactorTree(&root, val);
    cout << "Level order traversal of "
         "constructed factor tree";
    printLevelOrder(root);
    return 0;
}
```

Output:

```
Level order traversal of constructed factor tree
48 2 24 2 12 2 6 2 3
```

## Source

<https://www.geeksforgeeks.org/factor-tree-of-a-given-number/>

## Chapter 152

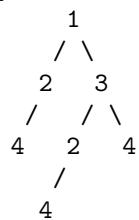
# Find All Duplicate Subtrees

Find All Duplicate Subtrees - GeeksforGeeks

Given a binary tree, find all duplicate subtrees. For each duplicate subtrees, we only need to return the root node of any one of them. Two trees are duplicate if they have the same structure with same node values.

Examples:

Input :



Output :

```
2
/   and   4
4
```

Explanation: Above Trees are two duplicate subtrees.

Therefore, you need to return above trees root in the form of a list.

The idea is to use [hashing](#). We store [inorder traversals](#) of subtrees in a hash. Since simple inorder traversal cannot uniquely identify a tree, we use symbols like '(' and ')' to represent NULL nodes.

We pass a [Unordered Map in C++](#) as an argument to the helper function which recursively calculates inorder string and increases its count in map. If any string gets repeated, then it will imply duplication of the subtree rooted at that node so push that node in Final result and return the vector of these nodes.

C++

```
// C++ program to find averages of all levels
// in a binary tree.
#include <bits/stdc++.h>
using namespace std;

/* A binary tree node has data, pointer to
left child and a pointer to right child */
struct Node {
    int data;
    struct Node* left, *right;
};

string inorder(Node* node, unordered_map<string, int>& m)
{
    if (!node)
        return "";

    string str = "(";
    str += inorder(node->left, m);
    str += to_string(node->data);
    str += inorder(node->right, m);
    str += ")";

    // Subtree already present (Note that we use
    // unordered_map instead of unordered_set
    // because we want to print multiple duplicates
    // only once, consider example of 4 in above
    // subtree, it should be printed only once.
    if (m[str] == 1)
        cout << node->data << " ";

    m[str]++;
}

return str;
}

// Wrapper over inorder()
void printAllDups(Node* root)
{
    unordered_map<string, int> m;
    inorder(root, m);
}

/* Helper function that allocates a
new node with the given data and
NULL left and right pointers. */
```

```
Node* newNode(int data)
{
    Node* temp = new Node;
    temp->data = data;
    temp->left = temp->right = NULL;
    return temp;
}

// Driver code
int main()
{
    Node* root = NULL;
    root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->right->left = newNode(2);
    root->right->left->left = newNode(4);
    root->right->right = newNode(4);
    printAllDups(root);
    return 0;
}
```

**Java**

```
// A java program to find all duplicate subtrees
// in a binary tree.
import java.util.HashMap;
public class Duplicate_subtress {

    /* A binary tree node has data, pointer to
     left child and a pointer to right child */
    static HashMap<String, Integer> m;
    static class Node {
        int data;
        Node left;
        Node right;
        Node(int data){
            this.data = data;
            left = null;
            right = null;
        }
    }
    static String inorder(Node node)
    {
        if (node == null)
            return "";
        String str = "" + node.data;
        str += inorder(node.left);
        str += inorder(node.right);
        return str;
    }
    static void printDups(String str)
    {
        if (m.get(str) > 1)
            System.out.println(str);
        else
            m.put(str, 1);
    }
    static void printAllDups(Node root)
    {
        if (root == null)
            return;
        printDups(inorder(root));
        printAllDups(root.left);
        printAllDups(root.right);
    }
}
```

```
String str = "(";
str += inorder(node.left);
str += Integer.toString(node.data);
str += inorder(node.right);
str += ")";

// Subtree already present (Note that we use
// HashMap instead of HashSet
// because we want to print multiple duplicates
// only once, consider example of 4 in above
// subtree, it should be printed only once.
if (m.get(str) != null && m.get(str)==1 )
    System.out.print( node.data + " ");

if (m.containsKey(str))
    m.put(str, m.get(str) + 1);
else
    m.put(str, 1);

return str;
}

// Wrapper over inorder()
static void printAllDups(Node root)
{
    m = new HashMap<>();
    inorder(root);
}
// Driver code
public static void main(String args[])
{
    Node root = null;
    root = new Node(1);
    root.left = new Node(2);
    root.right = new Node(3);
    root.left.left = new Node(4);
    root.right.left = new Node(2);
    root.right.left.left = new Node(4);
    root.right.right = new Node(4);
    printAllDups(root);
}
}

// This code is contributed by Sumit Ghosh
```

Output:

4 2

**Source**

<https://www.geeksforgeeks.org/find-duplicate-subtrees/>

## Chapter 153

# Find Count of Single Valued Subtrees

Find Count of Single Valued Subtrees - GeeksforGeeks

Given a binary tree, write a program to count the number of Single Valued Subtrees. A Single Valued Subtree is one in which all the nodes have same value. Expected time complexity is  $O(n)$ .

Example:

```
Input: root of below tree
      5
     / \
    1   5
   / \   \
  5   5   5
Output: 4
There are 4 subtrees with single values.
```

```
Input: root of below tree
      5
     / \
    4   5
   / \   \
  4   4   5
Output: 5
There are five subtrees with single values.
```

We strongly recommend you to minimize your browser and try this yourself first.

A **Simple Solution** is to traverse the tree. For every traversed node, check if all values under this node are same or not. If same, then increment count. Time complexity of this solution is  $O(n^2)$ .

An **Efficient Solution** is to traverse the tree in bottom up manner. For every subtree visited, return true if subtree rooted under it is single valued and increment count. So the idea is to use count as a reference parameter in recursive calls and use returned values to find out if left and right subtrees are single valued or not.

Below is the implementation of above idea.

### C++

```
// C++ program to find count of single valued subtrees
#include<bits/stdc++.h>
using namespace std;

// A Tree node
struct Node
{
    int data;
    struct Node* left, *right;
};

// Utility function to create a new node
Node* newNode(int data)
{
    Node* temp = new Node;
    temp->data = data;
    temp->left = temp->right = NULL;
    return (temp);
}

// This function increments count by number of single
// valued subtrees under root. It returns true if subtree
// under root is Singly, else false.
bool countSingleRec(Node* root, int &count)
{
    // Return false to indicate NULL
    if (root == NULL)
        return true;

    // Recursively count in left and right subtrees also
    bool left = countSingleRec(root->left, count);
    bool right = countSingleRec(root->right, count);

    // If any of the subtrees is not singly, then this
    // cannot be singly.
```

```
if (left == false || right == false)
    return false;

// If left subtree is singly and non-empty, but data
// doesn't match
if (root->left && root->data != root->left->data)
    return false;

// Same for right subtree
if (root->right && root->data != root->right->data)
    return false;

// If none of the above conditions is true, then
// tree rooted under root is single valued, increment
// count and return true.
count++;
return true;
}

// This function mainly calls countSingleRec()
// after initializing count as 0
int countSingle(Node* root)
{
    // Initialize result
    int count = 0;

    // Recursive function to count
    countSingleRec(root, count);

    return count;
}

// Driver program to test
int main()
{
    /* Let us construct the below tree
        5
       / \
      4   5
     / \   \
    4   4   5 */
    Node* root      = newNode(5);
    root->left     = newNode(4);
    root->right    = newNode(5);
    root->left->left = newNode(4);
    root->left->right = newNode(4);
    root->right->right = newNode(5);
```

```
    cout << "Count of Single Valued Subtrees is "
          << countSingle(root);
    return 0;
}
```

**Java**

```
// Java program to find count of single valued subtrees

/* Class containing left and right child of current
node and key value*/
class Node
{
    int data;
    Node left, right;

    public Node(int item)
    {
        data = item;
        left = right = null;
    }
}

class Count
{
    int count = 0;
}

class BinaryTree
{
    Node root;
    Count ct = new Count();

    // This function increments count by number of single
    // valued subtrees under root. It returns true if subtree
    // under root is Singly, else false.
    boolean countSingleRec(Node node, Count c)
    {
        // Return false to indicate NULL
        if (node == null)
            return true;

        // Recursively count in left and right subtrees also
        boolean left = countSingleRec(node.left, c);
        boolean right = countSingleRec(node.right, c);

        // If any of the subtrees is not singly, then this
        // cannot be singly.
```

```
if (left == false || right == false)
    return false;

// If left subtree is singly and non-empty, but data
// doesn't match
if (node.left != null && node.data != node.left.data)
    return false;

// Same for right subtree
if (node.right != null && node.data != node.right.data)
    return false;

// If none of the above conditions is true, then
// tree rooted under root is single valued, increment
// count and return true.
c.count++;
return true;
}

// This function mainly calls countSingleRec()
// after initializing count as 0
int countSingle()
{
    return countSingle(root);
}

int countSingle(Node node)
{
    // Recursive function to count
    countSingleRec(node, ct);
    return ct.count;
}

// Driver program to test above functions
public static void main(String args[])
{
    /* Let us construct the below tree
        5
       / \
      4   5
     / \   \
    4   4   5 */
    BinaryTree tree = new BinaryTree();
    tree.root = new Node(5);
    tree.root.left = new Node(4);
    tree.root.right = new Node(5);
    tree.root.left.left = new Node(4);
    tree.root.left.right = new Node(4);
```

```
        tree.root.right.right = new Node(5);

        System.out.println("The count of single valued sub trees is : "
                           + tree.countSingle());
    }

}

// This code has been contributed by Mayank Jaiswal
```

### Python

```
# Python program to find the count of single valued subtrees

# Node Structure
class Node:
    # Utility function to create a new node
    def __init__(self ,data):
        self.data = data
        self.left = None
        self.right = None

    # This function increments count by number of single
    # valued subtrees under root. It returns true if subtree
    # under root is Singly, else false.
    def countSingleRec(root , count):
        # Return False to indicate None
        if root is None :
            return True

        # Recursively count in left and right subtress also
        left = countSingleRec(root.left , count)
        right = countSingleRec(root.right , count)

        # If any of the subtress is not singly, then this
        # cannot be singly
        if left == False or right == False :
            return False

        # If left subtree is singly and non-empty , but data
        # doesn't match
        if root.left and root.data != root.left.data:
            return False

        # same for right subtree
        if root.right and root.data != root.right.data:
            return False
```

```
# If none of the above conditions is True, then
# tree rooted under root is single valued,increment
# count and return true
count[0] += 1
return True

# This function mainly calss countSingleRec()
# after initializing count as 0
def countSingle(root):
    # initialize result
    count = [0]

    # Recursive function to count
    countSingleRec(root , count)

    return count[0]

# Driver program to test

"""Let us construct the below tree
      5
     /   \
    4     5
   / \   \
  4   4   5
"""
root = Node(5)
root.left = Node(4)
root.right = Node(5)
root.left.left = Node(4)
root.left.right = Node(4)
root.right.right = Node(5)
countSingle(root)
print "Count of Single Valued Subtrees is" , countSingle(root)

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

Output:

Count of Single Valued Subtrees is 5

Time complexity of this solution is O(n) where n is number of nodes in given binary tree.  
Thanks to Gaurav Ahirwar for suggesting above solution.

**Source**

<https://www.geeksforgeeks.org/find-count-of-singly-subtrees/>

## Chapter 154

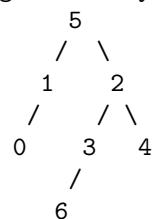
# Find Height of Binary Tree represented by Parent array

Find Height of Binary Tree represented by Parent array - GeeksforGeeks

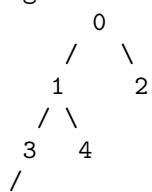
A given array represents a tree in such a way that the array value gives the parent node of that particular index. The value of the root node index would always be -1. Find the height of the tree.

Height of a Binary Tree is number of nodes on the path from root to the deepest leaf node, the number includes both root and leaf.

Input: parent[] = {1 5 5 2 2 -1 3}  
Output: 4  
The given array represents following Binary Tree



Input: parent[] = {-1, 0, 0, 1, 1, 3, 5};  
Output: 5  
The given array represents following Binary Tree



```
5
/
6
```

Source: [Amazon Interview experience | Set 128 \(For SDET\)](#)

We strongly recommend to minimize your browser and try this yourself first.

A simple solution is to first construct the tree and then find height of the constructed binary tree. The tree can be constructed recursively by first searching the current root, then recurring for the found indexes and making them left and right subtrees of root. This solution takes  $O(n^2)$  as we have to linearly search for every node.

An efficient solution can solve the above problem in  $O(n)$  time. The idea is to first calculate depth of every node and store in an array  $\text{depth}[]$ . Once we have depths of all nodes, we return maximum of all depths.

- 1) Find depth of all nodes and fill in an auxiliary array  $\text{depth}[]$ .
- 2) Return maximum value in  $\text{depth}[]$ .

Following are steps to find depth of a node at index  $i$ .

- 1) If it is root,  $\text{depth}[i]$  is 1.
- 2) If depth of  $\text{parent}[i]$  is evaluated,  $\text{depth}[i]$  is  $\text{depth}[\text{parent}[i]] + 1$ .
- 3) If depth of  $\text{parent}[i]$  is not evaluated, recur for parent and assign  $\text{depth}[i]$  as  $\text{depth}[\text{parent}[i]] + 1$  (same as above).

Following is the implementation of above idea.

C++

```
// C++ program to find height using parent array
#include <iostream>
using namespace std;

// This function fills depth of i'th element in parent[]. The depth is
// filled in depth[i].
void fillDepth(int parent[], int i, int depth[])
{
    // If depth[i] is already filled
    if (depth[i])
        return;

    // If node at index i is root
    if (parent[i] == -1)
    {
        depth[i] = 1;
        return;
    }

    // If depth of parent is not evaluated before, then evaluate
    // depth of parent first
    if (depth[parent[i]] == 0)
```

```
        fillDepth(parent, parent[i], depth);

    // Depth of this node is depth of parent plus 1
    depth[i] = depth[parent[i]] + 1;
}

// This function returns height of binary tree represented by
// parent array
int findHeight(int parent[], int n)
{
    // Create an array to store depth of all nodes/ and
    // initialize depth of every node as 0 (an invalid
    // value). Depth of root is 1
    int depth[n];
    for (int i = 0; i < n; i++)
        depth[i] = 0;

    // fill depth of all nodes
    for (int i = 0; i < n; i++)
        fillDepth(parent, i, depth);

    // The height of binary tree is maximum of all depths.
    // Find the maximum value in depth[] and assign it to ht.
    int ht = depth[0];
    for (int i=1; i<n; i++)
        if (ht < depth[i])
            ht = depth[i];
    return ht;
}

// Driver program to test above functions
int main()
{
    // int parent[] = {1, 5, 5, 2, 2, -1, 3};
    int parent[] = {-1, 0, 0, 1, 1, 3, 5};

    int n = sizeof(parent)/sizeof(parent[0]);
    cout << "Height is " << findHeight(parent, n);
    return 0;
}
```

### Java

```
// Java program to find height using parent array
class BinaryTree {

    // This function fills depth of i'th element in parent[]. The depth is
    // filled in depth[i].
```

```
void fillDepth(int parent[], int i, int depth[]) {  
  
    // If depth[i] is already filled  
    if (depth[i] != 0) {  
        return;  
    }  
  
    // If node at index i is root  
    if (parent[i] == -1) {  
        depth[i] = 1;  
        return;  
    }  
  
    // If depth of parent is not evaluated before, then evaluate  
    // depth of parent first  
    if (depth[parent[i]] == 0) {  
        fillDepth(parent, parent[i], depth);  
    }  
  
    // Depth of this node is depth of parent plus 1  
    depth[i] = depth[parent[i]] + 1;  
}  
  
// This function returns height of binary tree represented by  
// parent array  
int findHeight(int parent[], int n) {  
  
    // Create an array to store depth of all nodes/ and  
    // initialize depth of every node as 0 (an invalid  
    // value). Depth of root is 1  
    int depth[] = new int[n];  
    for (int i = 0; i < n; i++) {  
        depth[i] = 0;  
    }  
  
    // fill depth of all nodes  
    for (int i = 0; i < n; i++) {  
        fillDepth(parent, i, depth);  
    }  
  
    // The height of binary tree is maximum of all depths.  
    // Find the maximum value in depth[] and assign it to ht.  
    int ht = depth[0];  
    for (int i = 1; i < n; i++) {  
        if (ht < depth[i]) {  
            ht = depth[i];  
        }  
    }  
}
```

```
        return ht;
    }

// Driver program to test above functions
public static void main(String args[]) {

    BinaryTree tree = new BinaryTree();

    // int parent[] = {1, 5, 5, 2, 2, -1, 3};
    int parent[] = new int[]{-1, 0, 0, 1, 1, 3, 5};

    int n = parent.length;
    System.out.println("Height is " + tree.findHeight(parent, n));
}
}
```

### Python

```
# Python program to find height using parent array

# This function fills depth of i'th element in parent[]
# The depth is filled in depth[i]

def fillDepth(parent, i, depth):

    # If depth[i] is already filled
    if depth[i] != 0:
        return

    # If node at index i is root
    if parent[i] == -1:
        depth[i] = 1
        return

    # If depth of parent is not evaluated before,
    # then evaluate depth of parent first
    if depth[parent[i]] == 0:
        fillDepth(parent, parent[i], depth)

    # Depth of this node is depth of parent plus 1
    depth[i] = depth[parent[i]] + 1

# This function returns height of binary tree represented
# by parent array
def findHeight(parent):
    n = len(parent)
    # Create an array to store depth of all nodes and
    # initialize depth of every node as 0
```

```
# Depth of root is 1
depth = [0 for i in range(n)]

# fill depth of all nodes
for i in range(n):
    fillDepth(parent, i, depth)

# The height of binary tree is maximum of all
# depths. Find the maximum in depth[] and assign
# it to ht
ht = depth[0]
for i in range(1,n):
    ht = max(ht, depth[i])

return ht

# Driver program to test above function
parent = [-1 , 0 , 0 , 1 , 1 , 3 , 5]
print "Height is %d" %(findHeight(parent))

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

Output:

```
Height is 5
```

Note that the time complexity of this programs seems more than  $O(n)$ . If we take a closer look, we can observe that depth of every node is evaluated only once.

This article is contributed by **Siddharth**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## Source

<https://www.geeksforgeeks.org/find-height-binary-tree-represented-parent-array/>

## Chapter 155

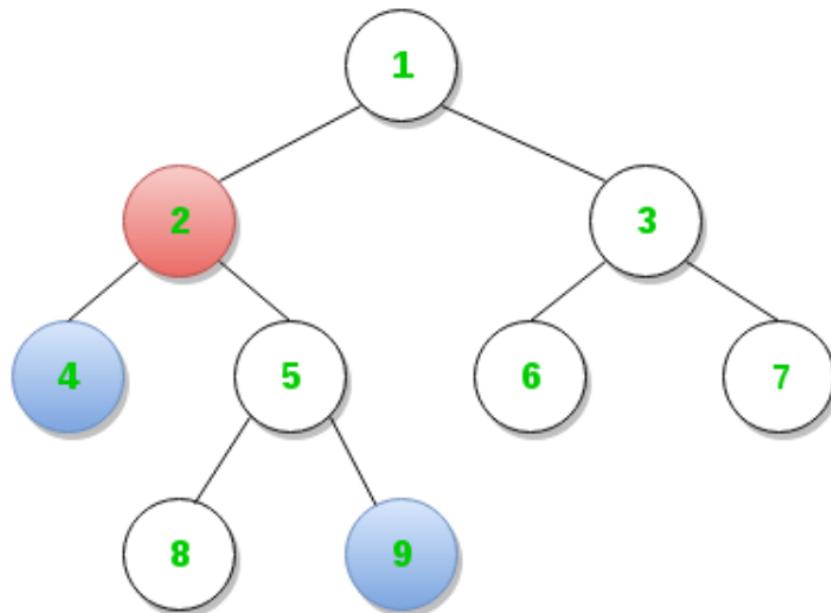
# Find LCA in Binary Tree using RMQ

Find LCA in Binary Tree using RMQ - GeeksforGeeks

The article describes an approach to solving the problem of finding the LCA of two nodes in a tree by reducing it to a RMQ problem.

**Lowest Common Ancestor (LCA)** of two nodes u and v in a rooted tree T is defined as the node located farthest from the root that has both u and v as descendants.

For example, in below diagram, LCA of node 4 and node 9 is node 2.

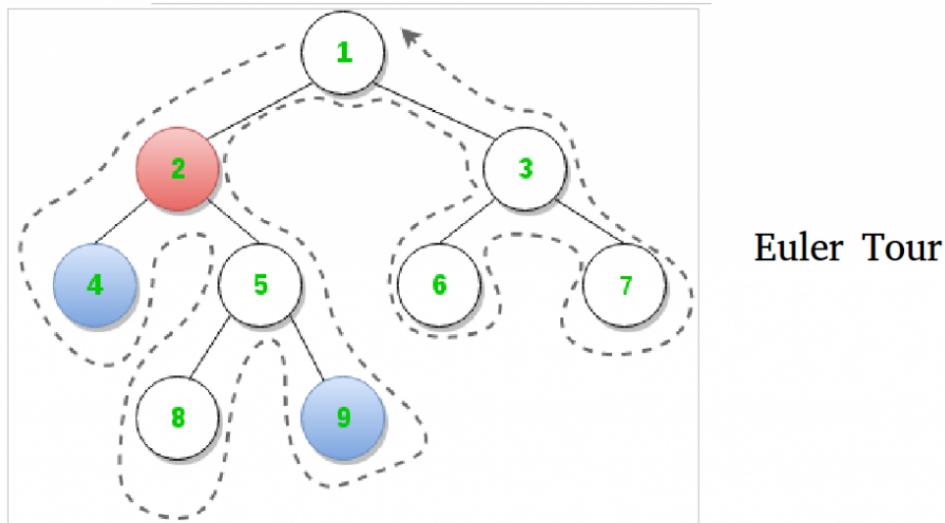


There can be many approaches to solve the LCA problem. The approaches differ in their time and space complexities. [Here](#) is a link to a couple of them (these do not involve reduction to RMQ).

**Range Minimum Query (RMQ)** is used on arrays to find the position of an element with the minimum value between two specified indices. Different approaches for solving RMQ have been discussed [here](#) and [here](#). In this article, Segment Tree based approach is discussed. With segment tree, preprocessing time is  $O(n)$  and time to for range minimum query is  $O(\log n)$ . The extra space required is  $O(n)$  to store the segment tree.

#### Reduction of LCA to RMQ:

The idea is to traverse the tree starting from root by an Euler tour (traversal without lifting pencil), which is a DFS-type traversal with preorder traversal characteristics.



An euler tour of the tree starting from node 1 will yield:

1	2	4	2	5	8	5	9	5	2	1	3	6	3	7	3	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

The corresponding levels for every node in Euler tour:

0	1	2	1	2	3	2	3	2	1	0	1	2	1	2	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

**Observation:** The LCA of nodes 4 and 9 is node 2, which happens to be the node closest to the root amongst all those encountered between the visits of 4 and 9 during a DFS of T. This observation is the key to the reduction. Let's rephrase: Our node is the node at the smallest level and the only node at that level amongst all the nodes that occur between consecutive occurrences (any) of u and v in the Euler tour of T.

We require three arrays for implementation:

- Nodes visited in order of Euler tour of T
- Level of each node visited in Euler tour of T

3. Index of the **first** occurrence of a node in Euler tour of T (since any occurrence would be good, let's track the first one)

**The first occurrences corresponding to every node in Euler tour of T:**

Node	1	2	3	4	5	6	7	8	9
First Occurrence	0	1	11	2	4	12	14	5	7

**Algorithm:**

1. Do a Euler tour on the tree, and fill the euler, level and first occurrence arrays.
2. Using the first occurrence array, get the indices corresponding to the two nodes which will be the corners of the range in the level array that is fed to the RMQ algorithm for the minimum value.
3. Once the algorithm return the index of the minimum level in the range, we use it to determine the LCA using Euler tour array.

Below is the implementation of above algorithm.

C++

```
/* C++ Program to find LCA of u and v by reducing the problem to RMQ */
#include<bits/stdc++.h>
#define V 9           // number of nodes in input tree

int euler[2*V - 1];      // For Euler tour sequence
int level[2*V - 1];      // Level of nodes in tour sequence
int firstOccurrence[V+1]; // First occurrences of nodes in tour
int ind;                  // Variable to fill-in euler and level arrays

// A Binary Tree node
struct Node
{
    int key;
    struct Node *left, *right;
};

// Utility function creates a new binary tree node with given key
Node * newNode(int k)
{
    Node *temp = new Node;
    temp->key = k;
    temp->left = temp->right = NULL;
    return temp;
}
```

```

// log base 2 of x
int Log2(int x)
{
    int ans = 0 ;
    while (x>>=1) ans++;
    return ans ;
}

/* A recursive function to get the minimum value in a given range
   of array indexes. The following are parameters for this function.

   st    --> Pointer to segment tree
   index --> Index of current node in the segment tree. Initially
              0 is passed as root is always at index 0
   ss & se  --> Starting and ending indexes of the segment represented
                  by current node, i.e., st[index]
   qs & qe  --> Starting and ending indexes of query range */
int RMQUtil(int index, int ss, int se, int qs, int qe, int *st)
{
    // If segment of this node is a part of given range, then return
    // the min of the segment
    if (qs <= ss && qe >= se)
        return st[index];

    // If segment of this node is outside the given range
    else if (se < qs || ss > qe)
        return -1;

    // If a part of this segment overlaps with the given range
    int mid = (ss + se)/2;

    int q1 = RMQUtil(2*index+1, ss, mid, qs, qe, st);
    int q2 = RMQUtil(2*index+2, mid+1, se, qs, qe, st);

    if (q1===-1) return q2;
    else if (q2===-1) return q1;

    return (level[q1] < level[q2]) ? q1 : q2;
}

// Return minimum of elements in range from index qs (quey start) to
// qe (query end). It mainly uses RMQUtil()
int RMQ(int *st, int n, int qs, int qe)
{
    // Check for erroneous input values
    if (qs < 0 || qe > n-1 || qs > qe)
    {

```

```

        printf("Invalid Input");
        return -1;
    }

    return RMQUtil(0, 0, n-1, qs, qe, st);
}

// A recursive function that constructs Segment Tree for array[ss..se].
// si is index of current node in segment tree st
void constructSTUtil(int si, int ss, int se, int arr[], int *st)
{
    // If there is one element in array, store it in current node of
    // segment tree and return
    if (ss == se) st[si] = ss;

    else
    {
        // If there are more than one elements, then recur for left and
        // right subtrees and store the minimum of two values in this node
        int mid = (ss + se)/2;
        constructSTUtil(si*2+1, ss, mid, arr, st);
        constructSTUtil(si*2+2, mid+1, se, arr, st);

        if (arr[st[2*si+1]] < arr[st[2*si+2]])
            st[si] = st[2*si+1];
        else
            st[si] = st[2*si+2];
    }
}

/* Function to construct segment tree from given array. This function
   allocates memory for segment tree and calls constructSTUtil() to
   fill the allocated memory */
int *constructST(int arr[], int n)
{
    // Allocate memory for segment tree

    // Height of segment tree
    int x = Log2(n)+1;

    // Maximum size of segment tree
    int max_size = 2*(1<<x) - 1; // 2*pow(2,x) -1

    int *st = new int[max_size];

    // Fill the allocated memory st
    constructSTUtil(0, 0, n-1, arr, st);
}

```

```

// Return the constructed segment tree
return st;
}

// Recursive version of the Euler tour of T
void eulerTour(Node *root, int l)
{
    /* if the passed node exists */
    if (root)
    {
        euler[ind] = root->key; // insert in euler array
        level[ind] = l;           // insert l in level array
        ind++;                   // increment index

        /* if unvisited, mark first occurrence */
        if (firstOccurrence[root->key] == -1)
            firstOccurrence[root->key] = ind-1;

        /* tour left subtree if exists, and remark euler
           and level arrays for parent on return */
        if (root->left)
        {
            eulerTour(root->left, l+1);
            euler[ind]=root->key;
            level[ind] = l;
            ind++;
        }

        /* tour right subtree if exists, and remark euler
           and level arrays for parent on return */
        if (root->right)
        {
            eulerTour(root->right, l+1);
            euler[ind]=root->key;
            level[ind] = l;
            ind++;
        }
    }
}

// Returns LCA of nodes n1, n2 (assuming they are
// present in the tree)
int findLCA(Node *root, int u, int v)
{
    /* Mark all nodes unvisited. Note that the size of
       firstOccurrence is 1 as node values which vary from
       1 to 9 are used as indexes */
    memset(firstOccurrence, -1, sizeof(int)*(V+1));
}

```

```
/* To start filling euler and level arrays from index 0 */
ind = 0;

/* Start Euler tour with root node on level 0 */
eulerTour(root, 0);

/* construct segment tree on level array */
int *st = constructST(level, 2*V-1);

/* If v before u in Euler tour. For RMQ to work, first
parameter 'u' must be smaller than second 'v' */
if (firstOccurrence[u]>firstOccurrence[v])
    std::swap(u, v);

// Starting and ending indexes of query range
int qs = firstOccurrence[u];
int qe = firstOccurrence[v];

// query for index of LCA in tour
int index = RMQ(st, 2*V-1, qs, qe);

/* return LCA node */
return euler[index];
}

// Driver program to test above functions
int main()
{
    // Let us create the Binary Tree as shown in the diagram.
    Node * root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->left = newNode(6);
    root->right->right = newNode(7);
    root->left->right->left = newNode(8);
    root->left->right->right = newNode(9);

    int u = 4, v = 9;
    printf("The LCA of node %d and node %d is node %d.\n",
           u, v, findLCA(root, u, v));
    return 0;
}
```

Java

```
// Java program to find LCA of u and v by reducing problem to RMQ

import java.util.*;

// A binary tree node
class Node
{
    Node left, right;
    int data;

    Node(int item)
    {
        data = item;
        left = right = null;
    }
}

class St_class
{
    int st;
    int stt[] = new int[10000];
}

class BinaryTree
{
    Node root;
    int v = 9; // v is the highest value of node in our tree
    int euler[] = new int[2 * v - 1]; // for euler tour sequence
    int level[] = new int[2 * v - 1]; // level of nodes in tour sequence
    int f_occur[] = new int[2 * v - 1]; // to store 1st occurrence of nodes
    int fill; // variable to fill euler and level arrays
    St_class sc = new St_class();

    // log base 2 of x
    int Log2(int x)
    {
        int ans = 0;
        int y = x >>= 1;
        while (y-- != 0)
            ans++;
        return ans;
    }

    int swap(int a, int b)
    {
        return a;
    }
}
```

```

/* A recursive function to get the minimum value in a given range
of array indexes. The following are parameters for this function.

st    --> Pointer to segment tree
index --> Index of current node in the segment tree. Initially
0 is passed as root is always at index 0
ss & se --> Starting and ending indexes of the segment represented
by current node, i.e., st[index]
qs & qe --> Starting and ending indexes of query range */
int RMQUtil(int index, int ss, int se, int qs, int qe, St_class st)
{
    // If segment of this node is a part of given range, then return
    // the min of the segment
    if (qs <= ss && qe >= se)
        return st.sst[index];

    // If segment of this node is outside the given range
    else if (se < qs || ss > qe)
        return -1;

    // If a part of this segment overlaps with the given range
    int mid = (ss + se) / 2;

    int q1 = RMQUtil(2 * index + 1, ss, mid, qs, qe, st);
    int q2 = RMQUtil(2 * index + 2, mid + 1, se, qs, qe, st);

    if (q1 == -1)
        return q2;
    else if (q2 == -1)
        return q1;

    return (level[q1] < level[q2]) ? q1 : q2;
}

// Return minimum of elements in range from index qs (quey start) to
// qe (query end). It mainly uses RMQUtil()
int RMQ(St_class st, int n, int qs, int qe)
{
    // Check for erroneous input values
    if (qs < 0 || qe > n - 1 || qs > qe)
    {
        System.out.println("Invalid input");
        return -1;
    }

    return RMQUtil(0, 0, n - 1, qs, qe, st);
}

```

```

// A recursive function that constructs Segment Tree for array[ss..se].
// si is index of current node in segment tree st
void constructSTUtil(int si, int ss, int se, int arr[], St_class st)
{
    // If there is one element in array, store it in current node of
    // segment tree and return
    if (ss == se)
        st.stt[si] = ss;
    else
    {
        // If there are more than one elements, then recur for left and
        // right subtrees and store the minimum of two values in this node
        int mid = (ss + se) / 2;
        constructSTUtil(si * 2 + 1, ss, mid, arr, st);
        constructSTUtil(si * 2 + 2, mid + 1, se, arr, st);

        if (arr[st.stt[2 * si + 1]] < arr[st.stt[2 * si + 2]])
            st.stt[si] = st.stt[2 * si + 1];
        else
            st.stt[si] = st.stt[2 * si + 2];
    }
}

/* Function to construct segment tree from given array. This function
allocates memory for segment tree and calls constructSTUtil() to
fill the allocated memory */
int constructST(int arr[], int n)
{
    // Allocate memory for segment tree
    // Height of segment tree
    int x = Log2(n) + 1;

    // Maximum size of segment tree
    int max_size = 2 * (1 << x) - 1; // 2*pow(2,x) -1

    sc.stt = new int[max_size];

    // Fill the allocated memory st
    constructSTUtil(0, 0, n - 1, arr, sc);

    // Return the constructed segment tree
    return sc.st;
}

// Recursive version of the Euler tour of T
void eulerTour(Node node, int l)
{
    /* if the passed node exists */

```

```

        if (node != null)
        {
            euler[fill] = node.data; // insert in euler array
            level[fill] = 1;           // insert 1 in level array
            fill++;                  // increment index

            /* if unvisited, mark first occurrence */
            if (f_occur[node.data] == -1)
                f_occur[node.data] = fill - 1;

            /* tour left subtree if exists, and remark euler
               and level arrays for parent on return */
            if (node.left != null)
            {
                eulerTour(node.left, l + 1);
                euler[fill] = node.data;
                level[fill] = l;
                fill++;
            }

            /* tour right subtree if exists, and remark euler
               and level arrays for parent on return */
            if (node.right != null)
            {
                eulerTour(node.right, l + 1);
                euler[fill] = node.data;
                level[fill] = l;
                fill++;
            }
        }

        // returns LCA of node n1 and n2 assuming they are present in tree
        int findLCA(Node node, int u, int v)
        {
            /* Mark all nodes unvisited. Note that the size of
               firstOccurrence is 1 as node values which vary from
               1 to 9 are used as indexes */
            Arrays.fill(f_occur, -1);

            /* To start filling euler and level arrays from index 0 */
            fill = 0;

            /* Start Euler tour with root node on level 0 */
            eulerTour(root, 0);

            /* construct segment tree on level array */
            sc.st = constructST(level, 2 * v - 1);
        }
    }
}

```

```
/* If v before u in Euler tour. For RMQ to work, first
parameter 'u' must be smaller than second 'v' */
if (f_occur[u] > f_occur[v])
    u = swap(u, u = v);

// Starting and ending indexes of query range
int qs = f_occur[u];
intqe = f_occur[v];

// query for index of LCA in tour
int index = RMQ(sc, 2 * v - 1, qs, qe);

/* return LCA node */
return euler[index];

}

// Driver program to test above functions
public static void main(String args[])
{
    BinaryTree tree = new BinaryTree();

    // Let us create the Binary Tree as shown in the diagram.
    tree.root = new Node(1);
    tree.root.left = new Node(2);
    tree.root.right = new Node(3);
    tree.root.left.left = new Node(4);
    tree.root.left.right = new Node(5);
    tree.root.right.left = new Node(6);
    tree.root.right.right = new Node(7);
    tree.root.left.right.left = new Node(8);
    tree.root.left.right.right = new Node(9);

    int u = 4, v = 9;
    System.out.println("The LCA of node " + u + " and " + v + " is "
        + tree.findLCA(tree.root, u, v));
}

// This code has been contributed by Mayank Jaiswal
```

Output:

The LCA of node 4 and node 9 is node 2.

Note:

1. We assume that the nodes queried are present in the tree.
2. We also assumed that if there are V nodes in tree, then keys (or data) of these nodes are in range from 1 to V.

#### Time complexity:

1. Euler tour: Number of nodes is V. For a tree,  $E = V-1$ . Euler tour (DFS) will take  $O(V+E)$  which is  $O(2*V)$  which can be written as  $O(V)$ .
2. Segment Tree construction :  $O(n)$  where  $n = V + E = 2*V - 1$ .
3. Range Minimum query:  $O(\log(n))$

Overall this method takes  $O(n)$  time for preprocessing, but takes  $O(\log n)$  time for query. Therefore, it can be useful when we have a single tree on which we want to perform large number of LCA queries (Note that LCA is useful for finding shortest path between two nodes of Binary Tree)

#### Auxiliary Space:

1. Euler tour array:  $O(n)$  where  $n = 2*V - 1$
2. Node Levels array:  $O(n)$
3. First Occurrences array:  $O(V)$
4. Segment Tree:  $O(n)$

Overall:  $O(n)$

Another observation is that the adjacent elements in level array differ by 1. This can be used to convert a RMQ problem to a LCA problem.

This article is contributed by **Yash Varyani**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

#### Source

<https://www.geeksforgeeks.org/find-lca-in-binary-tree-using-rmq/>

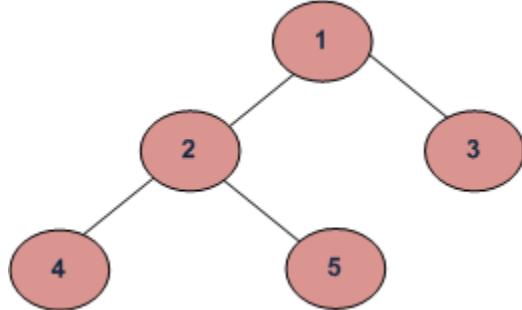
## Chapter 156

# Find Minimum Depth of a Binary Tree

Find Minimum Depth of a Binary Tree - GeeksforGeeks

Given a binary tree, find its minimum depth. The minimum depth is the number of nodes along the shortest path from root node down to the nearest leaf node.

For example, minimum height of below Binary Tree is 2.



Note that the path must end on a leaf node. For example, minimum height of below Binary Tree is also 2.

```
    10  
    /  
   5
```

The idea is to traverse the given Binary Tree. For every node, check if it is a leaf node. If yes, then return 1. If not leaf node then if left subtree is NULL, then recur for right subtree. And if right subtree is NULL, then recur for left subtree. If both left and right subtrees are not NULL, then take the minimum of two heights.

Below is implementation of the above idea.

C++

```
// C++ program to find minimum depth of a given Binary Tree
#include<bits/stdc++.h>
using namespace std;

// A BT Node
struct Node
{
    int data;
    struct Node* left, *right;
};

int minDepth(Node *root)
{
    // Corner case. Should never be hit unless the code is
    // called on root = NULL
    if (root == NULL)
        return 0;

    // Base case : Leaf Node. This accounts for height = 1.
    if (root->left == NULL && root->right == NULL)
        return 1;

    // If left subtree is NULL, recur for right subtree
    if (!root->left)
        return minDepth(root->right) + 1;

    // If right subtree is NULL, recur for left subtree
    if (!root->right)
        return minDepth(root->left) + 1;

    return min(minDepth(root->left), minDepth(root->right)) + 1;
}

// Utility function to create new Node
Node *newNode(int data)
{
    Node *temp = new Node;
    temp->data = data;
    temp->left = temp->right = NULL;
    return (temp);
}

// Driver program
int main()
```

```
{  
    // Let us construct the Tree shown in the above figure  
    Node *root      = newNode(1);  
    root->left     = newNode(2);  
    root->right    = newNode(3);  
    root->left->left = newNode(4);  
    root->left->right = newNode(5);  
    cout << minDepth(root);  
    return 0;  
}
```

**Java**

```
/* Java implementation to find minimum depth  
   of a given Binary tree */  
  
/* Class containing left and right child of current  
   node and key value*/  
class Node  
{  
    int data;  
    Node left, right;  
    public Node(int item)  
    {  
        data = item;  
        left = right = null;  
    }  
}  
public class BinaryTree  
{  
    //Root of the Binary Tree  
    Node root;  
  
    int minimumDepth()  
    {  
        return minimumDepth(root);  
    }  
  
    /* Function to calculate the minimum depth of the tree */  
    int minimumDepth(Node root)  
    {  
        // Corner case. Should never be hit unless the code is  
        // called on root = NULL  
        if (root == null)  
            return 0;  
  
        // Base case : Leaf Node. This accounts for height = 1.  
        if (root.left == null && root.right == null)
```

```
        return 1;

    // If left subtree is NULL, recur for right subtree
    if (root.left == null)
        return minimumDepth(root.right) + 1;

    // If right subtree is NULL, recur for left subtree
    if (root.right == null)
        return minimumDepth(root.left) + 1;

    return Math.min(minimumDepth(root.left),
                    minimumDepth(root.right)) + 1;
}

/* Driver program to test above functions */
public static void main(String args[])
{
    BinaryTree tree = new BinaryTree();
    tree.root = new Node(1);
    tree.root.left = new Node(2);
    tree.root.right = new Node(3);
    tree.root.left.left = new Node(4);
    tree.root.left.right = new Node(5);

    System.out.println("The minimum depth of "+
                       "binary tree is : " + tree.minimumDepth());
}
}
```

### Python

```
# Python program to find minimum depth of a given Binary Tree

# Tree node
class Node:
    def __init__(self , key):
        self.data = key
        self.left = None
        self.right = None

def minDepth(root):
    # Corner Case.Should never be hit unless the code is
    # called on root = NULL
    if root is None:
        return 0

    # Base Case : Leaf node.This acoounts for height = 1
    if root.left is None and root.right is None:
```

```
    return 1

    # If left subtree is Null, recur for right subtree
    if root.left is None:
        return minDepth(root.right)+1

    # If right subtree is Null , recur for left subtree
    if root.right is None:
        return minDepth(root.left) +1

    return min(minDepth(root.left), minDepth(root.right))+1

# Driver Program
root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)
print minDepth(root)

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

Output:

2

Time complexity of above solution is  $O(n)$  as it traverses the tree only once.  
Thanks to Gaurav Ahirwar for providing above solution.

The above method may end up with complete traversal of Binary Tree even when the topmost leaf is close to root. A **Better Solution** is to do Level Order Traversal. While doing traversal, returns depth of the first encountered leaf node. Below is implementation of this solution.

## C

```
// C++ program to find minimum depth of a given Binary Tree
#include<bits/stdc++.h>
using namespace std;

// A Binary Tree Node
struct Node
{
    int data;
    struct Node *left, *right;
};
```

```
// A queue item (Stores pointer to node and an integer)
struct qItem
{
    Node *node;
    int depth;
};

// Iterative method to find minimum depth of Binary Tree
int minDepth(Node *root)
{
    // Corner Case
    if (root == NULL)
        return 0;

    // Create an empty queue for level order traversal
    queue<qItem> q;

    // Enqueue Root and initialize depth as 1
    qItem qi = {root, 1};
    q.push(qi);

    // Do level order traversal
    while (q.empty() == false)
    {
        // Remove the front queue item
        qi = q.front();
        q.pop();

        // Get details of the remove item
        Node *node = qi.node;
        int depth = qi.depth;

        // If this is the first leaf node seen so far
        // Then return its depth as answer
        if (node->left == NULL && node->right == NULL)
            return depth;

        // If left subtree is not NULL, add it to queue
        if (node->left != NULL)
        {
            qi.node = node->left;
            qi.depth = depth + 1;
            q.push(qi);
        }

        // If right subtree is not NULL, add it to queue
        if (node->right != NULL)
        {
```

```
        qi.node  = node->right;
        qi.depth = depth+1;
        q.push(qi);
    }
}
return 0;
}

// Utility function to create a new tree Node
Node* newNode(int data)
{
    Node *temp = new Node;
    temp->data = data;
    temp->left = temp->right = NULL;
    return temp;
}

// Driver program to test above functions
int main()
{
    // Let us create binary tree shown in above diagram
    Node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);

    cout << minDepth(root);
    return 0;
}
```

### Python

```
# Python program to find minimum depth of a given Binary Tree

# A Binary Tree node
class Node:
    # Utility to create new node
    def __init__(self , data):
        self.data = data
        self.left = None
        self.right = None

def minDepth(root):
    # Corner Case
    if root is None:
        return 0
```

```
# Create an empty queue for level order traversal
q = []

# Enqueue root and initialize depth as 1
q.append({'node': root, 'depth': 1})

# Do level order traversal
while(len(q)>0):
    # Remove the front queue item
    queueItem = q.pop(0)

    # Get details of the removed item
    node = queueItem['node']
    depth = queueItem['depth']
    # If this is the first leaf node seen so far
    # then return its depth as answer
    if node.left is None and node.right is None:
        return depth

    # If left subtree is not None, add it to queue
    if node.left is not None:
        q.append({'node': node.left, 'depth': depth+1})

    # if right subtree is not None, add it to queue
    if node.right is not None:
        q.append({'node': node.right, 'depth': depth+1})

# Driver program to test above function
# Lets construct a binary tree shown in above diagram
root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)
print minDepth(root)

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

Output:

2

Thanks to Manish Chauhan for suggesting above idea and Ravi for providing implementation.

Improved By : [mrunalupadhyay](#)

**Source**

<https://www.geeksforgeeks.org/find-minimum-depth-of-a-binary-tree/>

## Chapter 157

# Find a number in minimum steps

Find a number in minimum steps - GeeksforGeeks

Given an infinite number line from -INFINITY to +INFINITY and we are on zero. We can move n steps either side at each n'th time.

1st time; we can move only 1 step to both ways, means -1 1;

2nd time we can move 2 steps from -1 and 1;

-1 : -3 (-1-2) 1(-1+2)  
1 : -1 ( 1-2) 3(1+2)

3rd time we can move 3 steps either way from -3, 1, -1, 3

-3: -6(-3-3) 0(-3+3)  
1: -2(1-3) 4(1+3)  
-1: -4(-1-3) 2(-1+3)  
3: 0(0-3) 6(3+3)

Find the minimum number of steps to reach a given number n.

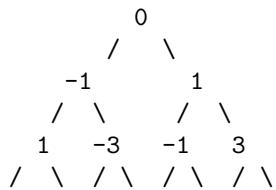
Examples:

Input : n = 10  
Output : 4  
We can reach 10 in 4 steps, 1, 3, 6, 10

Input : n = 13

Output : 5  
We can reach 10 in 4 steps, -1, 2, 5, 9, 14

This problem can be modeled as tree. We put initial point 0 at root, 1 and -1 as children of root. Next level contains values at distance 2 and so on.



The problem is now to find the closes node to root with value n. The idea is to do [Level Order Traversal](#) of tree to find the closest node. Note that using DFS for closest node is never a good idea (we may end up going down many unnecessary levels).

Below is C++ implementation of above idea.

```
// C++ program to find a number in minimum steps
#include <bits/stdc++.h>
using namespace std;
#define InF 99999

// To represent data of a node in tree
struct number
{
    int no;
    int level;
public:
    number() {}
    number(int n, int l):no(n),level(l) {}
};

// Prints level of node n
void findnthnumber(int n)
{
    // Create a queue and insert root
    queue<number> q;
    struct number r(0, 1);
    q.push(r);

    // Do level order traversal
    while (!q.empty())
    {
        // Remove a node from queue
        struct number temp = q.front();
```

```
q.pop();

// To avoid infinite loop
if (temp.no >= InF || temp.no <= -InF)
    break;

// Check if dequeued number is same as n
if (temp.no == n)
{
    cout << "Found number n at level "
        << temp.level-1;
    break;
}

// Insert children of dequeued node to queue
q.push(number(temp.no+temp.level, temp.level+1));
q.push(number(temp.no-temp.level, temp.level+1));
}

}

// Driver code
int main()
{
    findnthnumber(13);
    return 0;
}
```

Output :

```
Found number n at level 5
```

## Source

<https://www.geeksforgeeks.org/find-a-number-in-minimum-steps/>

## Chapter 158

# Find all possible binary trees with given Inorder Traversal

Find all possible binary trees with given Inorder Traversal - GeeksforGeeks

Given an array that represents Inorder Traversal, find all possible Binary Trees with the given Inorder traversal and print their preorder traversals.

Examples:

```
Input:  in[] = {3, 2};  
Output: Preorder traversals of different possible Binary Trees are:  
       3 2  
       2 3  
Below are different possible binary trees  
       3      2  
         \    /  
          2    3
```

```
Input:  in[] = {4, 5, 7};  
Output: Preorder traversals of different possible Binary Trees are:  
       4 5 7  
       4 7 5  
       5 4 7  
       7 4 5  
       7 5 4
```

```
Below are different possible binary trees  
       4      4      5      7      7  
         \      \      / \      /      /  
          5      7      4    7      4      5  
            \      /          \      /  
              7      5          5      4
```

We strongly recommend you to minimize your browser and try this yourself first.

Let given inorder traversal be **in[]**. In the given traversal, all nodes in left subtree of **in[i]** must appear before it and in right subtree must appear after it. So when we consider **in[i]** as root, all elements from **in[0]** to **in[i-1]** will be in left subtree and **in[i+1]** to **n-1** will be in right subtree. If **in[0]** to **in[i-1]** can form **x** different trees and **in[i+1]** to **in[n-1]** can form **y** different trees then we will have **x\*y** total trees when **in[i]** as root. So we simply iterate from 0 to **n-1** for root. For every node **in[i]**, recursively find different left and right subtrees. If we take a closer look, we can notice that the count is basically **n'th Catalan number**. We have discussed different approaches to find **n'th Catalan number** [here](#).

The idea is to maintain a list of roots of all Binary Trees. Recursively construct all possible left and right subtrees. Create a tree for every pair of left and right subtree and add the tree to list. Below is detailed algorithm.

- 1) Initialize list of Binary Trees as empty.
- 2) For every element **in[i]** where **i** varies from 0 to **n-1**,  
    do following
  - .....a) Create a new node with key as '**arr[i]**',  
        let this node be '**node**'
  - .....b) Recursively construct list of all left subtrees.
  - .....c) Recursively construct list of all right subtrees.
- 3) Iterate for all left subtrees
  - a) For current leftsubtree, iterate for all right subtrees  
            Add current left and right subtrees to '**node**' and add '**node**' to list.

C++

```
// C++ program to find binary tree with given inorder
// traversal
#include <bits/stdc++.h>
using namespace std;

// Node structure
struct Node
{
    int key;
    struct Node *left, *right;
};

// A utility function to create a new tree Node
struct Node *newNode(int item)
{
    struct Node *temp = new Node;
    temp->key = item;
    temp->left = temp->right = NULL;
```

```
    return temp;
}

// A utility function to do preorder traversal of BST
void preorder(Node *root)
{
    if (root != NULL)
    {
        printf("%d ", root->key);
        preorder(root->left);
        preorder(root->right);
    }
}

// Function for constructing all possible trees with
// given inorder traversal stored in an array from
// arr[start] to arr[end]. This function returns a
// vector of trees.
vector<Node *> getTrees(int arr[], int start, int end)
{
    // List to store all possible trees
    vector<Node *> trees;

    /* if start > end then subtree will be empty so
       returning NULL in the list */
    if (start > end)
    {
        trees.push_back(NULL);
        return trees;
    }

    /* Iterating through all values from start to end
       for constructing left and right subtree
       recursively */
    for (int i = start; i <= end; i++)
    {
        /* Constructing left subtree */
        vector<Node *> ltrees = getTrees(arr, start, i-1);

        /* Constructing right subtree */
        vector<Node *> rtrees = getTrees(arr, i+1, end);

        /* Now looping through all left and right subtrees
           and connecting them to ith root below */
        for (int j = 0; j < ltrees.size(); j++)
        {
            for (int k = 0; k < rtrees.size(); k++)
            {
```

```
// Making arr[i] as root
Node * node = newNode(arr[i]);

// Connecting left subtree
node->left = ltrees[j];

// Connecting right subtree
node->right = rtrees[k];

// Adding this tree to list
trees.push_back(node);
}

}

return trees;
}

// Driver Program to test above functions
int main()
{
    int in[] = {4, 5, 7};
    int n = sizeof(in) / sizeof(in[0]);

    vector<Node *> trees = getTrees(in, 0, n-1);

    cout << "Preorder traversals of different "
         << "possible Binary Trees are \n";
    for (int i = 0; i < trees.size(); i++)
    {
        preorder(trees[i]);
        printf("\n");
    }
    return 0;
}
```

### Java

```
// Java program to find binary tree with given inorder
// traversal
import java.util.Vector;

/* Class containing left and right child of current
   node and key value*/
class Node {
    int data;
    Node left, right;

    public Node(int item) {
```

```
        data = item;
        left = null;
        right = null;
    }
}

/* Class to print Level Order Traversal */
class BinaryTree {

    Node root;

    // A utility function to do preorder traversal of BST
    void preOrder(Node node) {
        if (node != null) {
            System.out.print(node.data + " ");
            preOrder(node.left);
            preOrder(node.right);
        }
    }

    // Function for constructing all possible trees with
    // given inorder traversal stored in an array from
    // arr[start] to arr[end]. This function returns a
    // vector of trees.
    Vector<Node> getTrees(int arr[], int start, int end) {

        // List to store all possible trees
        Vector<Node> trees= new Vector<Node>();

        /* if start > end then subtree will be empty so
           returning NULL in the list */
        if (start > end) {
            trees.add(null);
            return trees;
        }

        /* Iterating through all values from start to end
           for constructing left and right subtree
           recursively */
        for (int i = start; i <= end; i++) {
            /* Constructing left subtree */
            Vector<Node> ltrees = getTrees(arr, start, i - 1);

            /* Constructing right subtree */
            Vector<Node> rtrees = getTrees(arr, i + 1, end);

            /* Now looping through all left and right subtrees
               and connecting them to ith root below */
        }
    }
}
```

```
for (int j = 0; j < ltrees.size(); j++) {
    for (int k = 0; k < rtrees.size(); k++) {

        // Making arr[i] as root
        Node node = new Node(arr[i]);

        // Connecting left subtree
        node.left = ltrees.get(j);

        // Connecting right subtree
        node.right = rtrees.get(k);

        // Adding this tree to list
        trees.add(node);
    }
}
return trees;
}

public static void main(String args[]) {
    int in[] = {4, 5, 7};
    int n = in.length;
    BinaryTree tree = new BinaryTree();
    Vector<Node> trees = tree.getTrees(in, 0, n - 1);
    System.out.println("Preorder traversal of different "+
        " binary trees are:");
    for (int i = 0; i < trees.size(); i++) {
        tree.preOrder(trees.get(i));
        System.out.println("");
    }
}
```

### Python

```
# Python program to find binary tree with given
# inorder traversal

# Node Structure
class Node:

    # Utility to create a new node
    def __init__(self , item):
        self.key = item
        self.left = None
        self.right = None
```

```
# A utility function to do preorder traversal of BST
def preorder(root):
    if root is not None:
        print root.key,
        preorder(root.left)
        preorder(root.right)

# Function for constructing all possible trees with
# given inorder traversal stored in an array from
# arr[start] to arr[end]. This function returns a
# vector of trees.
def getTrees(arr , start , end):

    # List to store all possible trees
    trees = []

    """ if start > end then subtree will be empty so
    returning NULL in the list """
    if start > end :
        trees.append(None)
        return trees

    """ Iterating through all values from start to end
    for constructing left and right subtree
    recursively """
    for i in range(start , end+1):

        # Constructing left subtree
        ltrees = getTrees(arr , start , i-1)

        # Constructing right subtree
        rtrees = getTrees(arr , i+1 , end)

        """ Looping through all left and right subtrees
        and connecting to ith root below"""
        for j in ltrees :
            for k in rtrees :

                # Making arr[i] as root
                node  = Node(arr[i])

                # Connecting left subtree
                node.left = j

                # Connecting right subtree
                node.right = k
```

```
# Adding this tree to list
trees.append(node)

return trees

# Driver program to test above function
inp = [4 , 5, 7]
n = len(inp)

trees = getTrees(inp , 0 , n-1)

print "Preorder traversals of different possible\
Binary Trees are "
for i in trees :
    preorder(i);
    print ""

# This program is contributed by Nikhil Kumar Singh(nickzuck_007)
```

Output:

```
Preorder traversals of different possible Binary Trees are
4 5 7
4 7 5
5 4 7
7 4 5
7 5 4
```

Thanks to Utkarsh for suggesting above solution.

This problem is similar to the problem discussed [here](#).

## Source

<https://www.geeksforgeeks.org/find-all-possible-trees-with-given-inorder-traversal/>

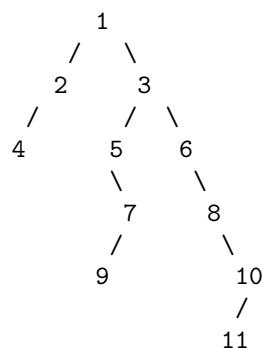
## Chapter 159

# Find depth of the deepest odd level leaf node

Find depth of the deepest odd level leaf node - GeeksforGeeks

Write a C code to get the depth of the deepest odd level leaf node in a binary tree. Consider that level starts with 1. Depth of a leaf node is number of nodes on the path from root to leaf (including both leaf and root).

For example, consider the following tree. The deepest odd level node is the node with value 9 and depth of this node is 5.



The idea is to recursively traverse the given binary tree and while traversing, maintain a variable “level” which will store the current node’s level in the tree. If current node is leaf then check “level” is odd or not. If level is odd then return it. If current node is not leaf, then recursively find maximum depth in left and right subtrees, and return maximum of the two depths.

C

```
// C program to find depth of the deepest odd level leaf node
#include <stdio.h>
#include <stdlib.h>

// A utility function to find maximum of two integers
int max(int x, int y) { return (x > y)? x : y; }

// A Binary Tree node
struct Node
{
    int data;
    struct Node *left, *right;
};

// A utility function to allocate a new tree node
struct Node* newNode(int data)
{
    struct Node* node = (struct Node*) malloc(sizeof(struct Node));
    node->data = data;
    node->left = node->right = NULL;
    return node;
}

// A recursive function to find depth of the deepest odd level leaf
int depthOfOddLeafUtil(struct Node *root,int level)
{
    // Base Case
    if (root == NULL)
        return 0;

    // If this node is a leaf and its level is odd, return its level
    if (root->left==NULL && root->right==NULL && level&1)
        return level;

    // If not leaf, return the maximum value from left and right subtrees
    return max(depthOfOddLeafUtil(root->left, level+1),
               depthOfOddLeafUtil(root->right, level+1));
}

/* Main function which calculates the depth of deepest odd level leaf.
This function mainly uses depthOfOddLeafUtil() */
int depthOfOddLeaf(struct Node *root)
{
    int level = 1, depth = 0;
    return depthOfOddLeafUtil(root, level);
}

// Driver program to test above functions
```

```
int main()
{
    struct Node* root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->right->left = newNode(5);
    root->right->right = newNode(6);
    root->right->left->right = newNode(7);
    root->right->right->right = newNode(8);
    root->right->left->right->left = newNode(9);
    root->right->right->right->right = newNode(10);
    root->right->right->right->left = newNode(11);

    printf("%d is the required depth\n", depthOfOddLeaf(root));
    getchar();
    return 0;
}
```

**Java**

```
// Java program to find depth of deepest odd level node

// A binary tree node
class Node
{
    int data;
    Node left, right;

    Node(int item)
    {
        data = item;
        left = right = null;
    }
}

class BinaryTree
{
    Node root;

    // A recursive function to find depth of the deepest odd level leaf
    int depthOfOddLeafUtil(Node node, int level)
    {
        // Base Case
        if (node == null)
            return 0;

        // If this node is a leaf and its level is odd, return its level
```

```
if (node.left == null && node.right == null && (level & 1) != 0)
    return level;

// If not leaf, return the maximum value from left and right subtrees
return Math.max(depthOfOddLeafUtil(node.left, level + 1),
                depthOfOddLeafUtil(node.right, level + 1));
}

/* Main function which calculates the depth of deepest odd level leaf.
   This function mainly uses depthOfOddLeafUtil() */
int depthOfOddLeaf(Node node)
{
    int level = 1, depth = 0;
    return depthOfOddLeafUtil(node, level);
}

public static void main(String args[])
{
    int k = 45;
    BinaryTree tree = new BinaryTree();
    tree.root = new Node(1);
    tree.root.left = new Node(2);
    tree.root.right = new Node(3);
    tree.root.left.left = new Node(4);
    tree.root.right.left = new Node(5);
    tree.root.right.right = new Node(6);
    tree.root.right.left.right = new Node(7);
    tree.root.right.right.right = new Node(8);
    tree.root.right.left.right.left = new Node(9);
    tree.root.right.right.right.right = new Node(10);
    tree.root.right.right.right.left = new Node(11);
    System.out.println(tree.depthOfOddLeaf(tree.root) +
                       " is the required depth");
}
}

// This code has been contributed by Mayank Jaiswal
```

### Python

```
# Python program to find depth of the deepest odd level
# leaf node

# A Binary tree node
class Node:

    # Constructor to create a new node
    def __init__(self, data):
```

```
        self.data = data
        self.left = None
        self.right = None

# A recursive function to find depth of the deepest
# odd level leaf node
def depthOfOddLeafUtil(root, level):

    # Base Case
    if root is None:
        return 0

    # If this node is leaf and its level is odd, return
    # its level
    if root.left is None and root.right is None and level&1:
        return level

    # If not leaf, return the maximum value from left
    # and right subtrees
    return (max(depthOfOddLeafUtil(root.left, level+1),
                depthOfOddLeafUtil(root.right, level+1)))

# Main function which calculates the depth of deepest odd
# level leaf .
# This function mainly uses depthOfOddLeafUtil()
def depthOfOddLeaf(root):
    level = 1
    depth = 0
    return depthOfOddLeafUtil(root, level)

# Driver program to test above function
root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.right.left = Node(5)
root.right.right = Node(6)
root.right.left.right = Node(7)
root.right.right.right = Node(8)
root.right.left.right.left = Node(9)
root.right.right.right.right = Node(10)
root.right.right.right.left = Node(11)

print "%d is the required depth" %(depthOfOddLeaf(root))

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

Output:

5 is the required depth

Time Complexity: The function does a simple traversal of the tree, so the complexity is  $O(n)$ .

### Iterative Approach

This approach is contributed by [Mandeep Singh](#). Traverse the tree in iterative fashion for each level, and whenever you encounter the leaf node, check if level is odd, if level is odd, then update the result.

```
// CPP program to find
// depth of the deepest
// odd level leaf node
// of binary tree
#include <bits/stdc++.h>
using namespace std;

// tree node
struct Node
{
    int data;
    Node *left, *right;
};

// returns a new
// tree Node
Node* newNode(int data)
{
    Node* temp = new Node();
    temp->data = data;
    temp->left = temp->right = NULL;
    return temp;
}

// return max odd number
// depth of leaf node
int maxOddLevelDepth(Node* root)
{
    if (!root)
        return 0;

    // create a queue
    // for level order
    // traversal
    queue<Node*> q;
    q.push(root);

    int result = INT_MAX;
```

```
int level = 0;

// traverse until the
// queue is empty
while (!q.empty())
{
    int size = q.size();
    level += 1;

    // traverse for
    // complete level
    while(size > 0)
    {
        Node* temp = q.front();
        q.pop();

        // check if the node is
        // leaf node and level
        // is odd if level is
        // odd, then update result
        if(!temp->left && !temp->right
           && (level % 2 != 0))
        {
            result = level;
        }

        // check for left child
        if (temp->left)
        {
            q.push(temp->left);
        }

        // check for right child
        if (temp->right)
        {
            q.push(temp->right);
        }
        size -= 1;
    }
}

return result;
}

// driver program
int main()
{
    // construct a tree
```

```
Node* root = newNode(1);
root->left = newNode(2);
root->right = newNode(3);
root->left->left = newNode(4);
root->right->left = newNode(5);
root->right->right = newNode(6);
root->right->left->right = newNode(7);
root->right->right->right = newNode(8);
root->right->left->right->left = newNode(9);
root->right->right->right->right = newNode(10);
root->right->right->right->left = newNode(11);

int result = maxOddLevelDepth(root);

if (result == INT_MAX)
    cout << "No leaf node at odd level\n";
else
    cout << result;
    cout << " is the required depth " << endl;
return 0;
}
```

Output:

```
5 is the required depth
```

Time Complexity: Time Complexity is O(n).

This article is contributed by **Chandra Prakash**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## Source

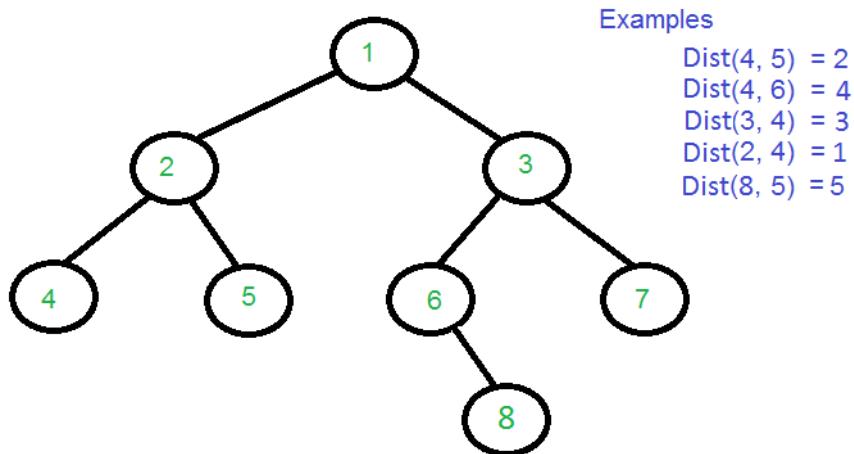
<https://www.geeksforgeeks.org/find-depth-of-the-deepest-odd-level-node/>

## Chapter 160

# Find distance between two nodes of a Binary Tree

Find distance between two nodes of a Binary Tree - GeeksforGeeks

Find the distance between two keys in a binary tree, no parent pointers are given. Distance between two nodes is the minimum number of edges to be traversed to reach one node from other.



The distance between two nodes can be obtained in terms of [lowest common ancestor](#). Following is the formula.

$\text{Dist}(n1, n2) = \text{Dist}(\text{root}, n1) + \text{Dist}(\text{root}, n2) - 2 * \text{Dist}(\text{root}, \text{lca})$

'n1' and 'n2' are the two given keys  
'root' is root of given Binary Tree.  
'lca' is lowest common ancestor of n1 and n2.  
 $\text{Dist}(n1, n2)$  is the distance between n1 and n2.

Following is the implementation of above approach. The implementation is adopted from last code provided in [Lowest Common Ancestor Post](#).

C++

```
/* Program to find distance between n1 and n2 using
   one traversal */
#include <iostream>
using namespace std;

// A Binary Tree Node
struct Node
{
    struct Node *left, *right;
    int key;
};

// Utility function to create a new tree Node
Node* newNode(int key)
{
    Node *temp = new Node;
    temp->key = key;
    temp->left = temp->right = NULL;
    return temp;
}

// Returns level of key k if it is present in tree,
// otherwise returns -1
int findLevel(Node *root, int k, int level)
{
    // Base Case
    if (root == NULL)
        return -1;

    // If key is present at root, or in left subtree
    // or right subtree, return true;
    if (root->key == k)
        return level;

    int l = findLevel(root->left, k, level+1);
    return (l != -1)? l : findLevel(root->right, k, level+1);
}

// This function returns pointer to LCA of two given
// values n1 and n2. It also sets d1, d2 and dist if
// one key is not ancestor of other
// d1 --> To store distance of n1 from root
// d2 --> To store distance of n2 from root
// lvl --> Level (or distance from root) of current node
```

```

// dist --> To store distance between n1 and n2
Node *findDistUtil(Node* root, int n1, int n2, int &d1,
                    int &d2, int &dist, int lvl)
{
    // Base case
    if (root == NULL) return NULL;

    // If either n1 or n2 matches with root's key, report
    // the presence by returning root (Note that if a key is
    // ancestor of other, then the ancestor key becomes LCA
    if (root->key == n1)
    {
        d1 = lvl;
        return root;
    }
    if (root->key == n2)
    {
        d2 = lvl;
        return root;
    }

    // Look for n1 and n2 in left and right subtrees
    Node *left_lca = findDistUtil(root->left, n1, n2,
                                    d1, d2, dist, lvl+1);
    Node *right_lca = findDistUtil(root->right, n1, n2,
                                    d1, d2, dist, lvl+1);

    // If both of the above calls return Non-NULL, then
    // one key is present in once subtree and other is
    // present in other. So this node is the LCA
    if (left_lca && right_lca)
    {
        dist = d1 + d2 - 2*lvl;
        return root;
    }

    // Otherwise check if left subtree or right subtree
    // is LCA
    return (left_lca != NULL)? left_lca: right_lca;
}

// The main function that returns distance between n1
// and n2. This function returns -1 if either n1 or n2
// is not present in Binary Tree.
int findDistance(Node *root, int n1, int n2)
{
    // Initialize d1 (distance of n1 from root), d2
    // (distance of n2 from root) and dist(distance

```

```
// between n1 and n2)
int d1 = -1, d2 = -1, dist;
Node *lca = findDistUtil(root, n1, n2, d1, d2,
                        dist, 1);

// If both n1 and n2 were present in Binary
// Tree, return dist
if (d1 != -1 && d2 != -1)
    return dist;

// If n1 is ancestor of n2, consider n1 as root
// and find level of n2 in subtree rooted with n1
if (d1 != -1)
{
    dist = findLevel(lca, n2, 0);
    return dist;
}

// If n2 is ancestor of n1, consider n2 as root
// and find level of n1 in subtree rooted with n2
if (d2 != -1)
{
    dist = findLevel(lca, n1, 0);
    return dist;
}

return -1;
}

// Driver program to test above functions
int main()
{
    // Let us create binary tree given in the
    // above example
    Node * root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->left = newNode(6);
    root->right->right = newNode(7);
    root->right->left->right = newNode(8);
    cout << "Dist(4, 5) = " << findDistance(root, 4, 5);
    cout << "nDist(4, 6) = " << findDistance(root, 4, 6);
    cout << "nDist(3, 4) = " << findDistance(root, 3, 4);
    cout << "nDist(2, 4) = " << findDistance(root, 2, 4);
    cout << "nDist(8, 5) = " << findDistance(root, 8, 5);
    return 0;
}
```

}

**Java**

```
/* A Java Program to find distance between n1 and n2
using one traversal */
public class DistanceBetweenTwoKey
{
    // (To the moderator) in c++ solution this variable
    // are declared as pointers hence changes made to them
    // reflects in the whole program

    // Global static variable
    static int d1 = -1;
    static int d2 = -1;
    static int dist = 0;

    // A Binary Tree Node
    static class Node{
        Node left, right;
        int key;

        // constructor
        Node(int key){
            this.key = key;
            left = null;
            right = null;
        }
    }

    // Returns level of key k if it is present in tree,
    // otherwise returns -1
    static int findLevel(Node root, int k, int level)
    {
        // Base Case
        if (root == null)
            return -1;

        // If key is present at root, or in left subtree or right subtree,
        // return true;
        if (root.key == k)
            return level;

        int l = findLevel(root.left, k, level + 1);
        return (l != -1)? l : findLevel(root.right, k, level + 1);
    }

    // This function returns pointer to LCA of two given values n1 and n2.
```

```

// It also sets d1, d2 and dist if one key is not ancestor of other
// d1 --> To store distance of n1 from root
// d2 --> To store distance of n2 from root
// lvl --> Level (or distance from root) of current node
// dist --> To store distance between n1 and n2
static Node findDistUtil(Node root, int n1, int n2, int lvl){

    // Base case
    if (root == null)
        return null;

    // If either n1 or n2 matches with root's key, report
    // the presence by returning root (Note that if a key is
    // ancestor of other, then the ancestor key becomes LCA
    if (root.key == n1){
        d1 = lvl;
        return root;
    }
    if (root.key == n2)
    {
        d2 = lvl;
        return root;
    }

    // Look for n1 and n2 in left and right subtrees
    Node left_lca = findDistUtil(root.left, n1, n2, lvl + 1);
    Node right_lca = findDistUtil(root.right, n1, n2, lvl + 1);

    // If both of the above calls return Non-NULL, then one key
    // is present in once subtree and other is present in other,
    // So this node is the LCA
    if (left_lca != null && right_lca != null)
    {
        dist = (d1 + d2) - 2*lvl;
        return root;
    }

    // Otherwise check if left subtree or right subtree is LCA
    return (left_lca != null)? left_lca : right_lca;
}

// The main function that returns distance between n1 and n2
// This function returns -1 if either n1 or n2 is not present in
// Binary Tree.
static int findDistance(Node root, int n1, int n2){
    d1 = -1;
    d2 = -1;
    dist = 0;
}

```

```
Node lca = findDistUtil(root, n1, n2, 1);

// If both n1 and n2 were present in Binary Tree, return dist
if (d1 != -1 && d2 != -1)
    return dist;

// If n1 is ancestor of n2, consider n1 as root and find level
// of n2 in subtree rooted with n1
if (d1 != -1)
{
    dist = findLevel(lca, n2, 0);
    return dist;
}

// If n2 is ancestor of n1, consider n2 as root and find level
// of n1 in subtree rooted with n2
if (d2 != -1)
{
    dist = findLevel(lca, n1, 0);
    return dist;
}

return -1;
}

// Driver program to test above functions
public static void main(String[] args) {

    // Let us create binary tree given in the above example
    Node root = new Node(1);
    root.left = new Node(2);
    root.right = new Node(3);
    root.left.left = new Node(4);
    root.left.right = new Node(5);
    root.right.left = new Node(6);
    root.right.right = new Node(7);
    root.right.left.right = new Node(8);

    System.out.println("Dist(4, 5) = "+findDistance(root, 4, 5));
    System.out.println("Dist(4, 6) = "+findDistance(root, 4, 6));
    System.out.println("Dist(3, 4) = "+findDistance(root, 3, 4));
    System.out.println("Dist(2, 4) = "+findDistance(root, 2, 4));
    System.out.println("Dist(8, 5) = " +findDistance(root, 8, 5));

}
}

// This code is contributed by Sumit Ghosh
```

## Python

```
# Python Program to find distance between
# n1 and n2 using one traversal

class Node:
    def __init__(self, data):
        self.data = data
        self.right = None
        self.left = None

def pathToNode(root, path, k):

    # base case handling
    if root is None:
        return False

    # append the node value in path
    path.append(root.data)

    # See if the k is same as root's data
    if root.data == k :
        return True

    # Check if k is found in left or right
    # sub-tree
    if ((root.left != None and pathToNode(root.left, path, k)) or
        (root.right!= None and pathToNode(root.right, path, k))):
        return True

    # If not present in subtree rooted with root,
    # remove root from path and return False
    path.pop()
    return False

def distance(root, data1, data2):
    if root:
        # store path corresponding to node: data1
        path1 = []
        pathToNode(root, path1, data1)

        # store path corresponding to node: data2
        path2 = []
        pathToNode(root, path2, data2)

        # iterate through the paths to find the
        # common path length
        i=0
```

```
while i<len(path1) and i<len(path2):
    # get out as soon as the path differs
    # or any path's length get exhausted
    if path1[i] != path2[i]:
        break
    i = i+1

    # get the path length by deducting the
    # intersecting path length (or till LCA)
    return (len(path1)+len(path2)-2*i)
else:
    return 0

# Driver Code to test above functions
root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.right.right= Node(7)
root.right.left = Node(6)
root.left.right = Node(5)
root.right.left.right = Node(8)

dist = distance(root, 4, 5)
print "Distance between node {} & {}: {}".format(4, 5, dist)

dist = distance(root, 4, 6)
print "Distance between node {} & {}: {}".format(4, 6, dist)

dist = distance(root, 3, 4)
print "Distance between node {} & {}: {}".format(3, 4, dist)

dist = distance(root, 2, 4)
print "Distance between node {} & {}: {}".format(2, 4, dist)

dist = distance(root, 8, 5)
print "Distance between node {} & {}: {}".format(8, 5, dist)

# This program is contributed by Aartee
```

Output:

```
Dist(4, 5) = 2
Dist(4, 6) = 4
Dist(3, 4) = 3
Dist(2, 4) = 1
Dist(8, 5) = 5
```

**Time Complexity:** Time complexity of the above solution is O(n) as the method does a single tree traversal.

Thanks to **Atul Singh** for providing the initial solution for this post.

**Better Solution :**

We first find LCA of two nodes. Then we find distance from LCA to two nodes.

C++

```
/* Program to find distance between n1 and n2
   using one traversal */
#include <iostream>
using namespace std;

// A Binary Tree Node
struct Node
{
    struct Node *left, *right;
    int key;
};

// Utility function to create a new tree Node
Node* newNode(int key)
{
    Node *temp = new Node;
    temp->key = key;
    temp->left = temp->right = NULL;
    return temp;
}

Node* LCA(Node * root, int n1,int n2)
{
    // Your code here
    if (root == NULL)
        return root;
    if (root->key == n1 || root->key == n2)
        return root;

    Node* left = LCA(root->left, n1, n2);
    Node* right = LCA(root->right, n1, n2);

    if (left != NULL && right != NULL)
        return root;
    if (left != NULL)
        return LCA(root->left, n1, n2);

    return LCA(root->right, n1, n2);
}
```

```
// Returns level of key k if it is present in
// tree, otherwise returns -1
int findLevel(Node *root, int k, int level)
{
    if(root == NULL) return -1;
    if(root->key == k) return level;

    int left = findLevel(root->left, k, level+1);
    if (left == -1)
        return findLevel(root->right, k, level+1);
    return left;
}

int findDistance(Node* root, int a, int b)
{
    // Your code here
    Node* lca = LCA(root, a , b);

    int d1 = findLevel(lca, a, 0);
    int d2 = findLevel(lca, b, 0);

    return d1 + d2;
}

// Driver program to test above functions
int main()
{
    // Let us create binary tree given in
    // the above example
    Node * root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->left = newNode(6);
    root->right->right = newNode(7);
    root->right->left->right = newNode(8);
    cout << "Dist(4, 5) = " << findDistance(root, 4, 5);
    cout << "\nDist(4, 6) = " << findDistance(root, 4, 6);
    cout << "\nDist(3, 4) = " << findDistance(root, 3, 4);
    cout << "\nDist(2, 4) = " << findDistance(root, 2, 4);
    cout << "\nDist(8, 5) = " << findDistance(root, 8, 5);
    return 0;
}
```

Java

```
/* Java Program to find distance between n1 and n2
   using one traversal */
public class GFG {

    public static class Node {
        int value;
        Node left;
        Node right;

        public Node(int value) {
            this.value = value;
        }
    }

    public static Node LCA(Node root, int n1, int n2)
    {
        if (root == null)
            return root;
        if (root.value == n1 || root.value == n2)
            return root;

        Node left = LCA(root.left, n1, n2);
        Node right = LCA(root.right, n1, n2);

        if (left != null && right != null)
            return root;
        if (left != null)
            return LCA(root.left, n1, n2);
        else
            return LCA(root.right, n1, n2);
    }

    // Returns level of key k if it is present in
    // tree, otherwise returns -1
    public static int findLevel(Node root, int a, int level)
    {
        if (root == null)
            return -1;
        if (root.value == a)
            return level;
        int left = findLevel(root.left, a, level + 1);
        if (left == -1)
            return findLevel(root.right, a, level + 1);
        return left;
    }

    public static int findDistance(Node root, int a, int b)
    {
```

```
Node lca = LCA(root, a, b);

int d1 = findLevel(lca, a, 0);
int d2 = findLevel(lca, b, 0);

return d1 + d2;
}

// Driver program to test above functions
public static void main(String[] args) {

    // Let us create binary tree given in
    // the above example
    Node root = new Node(1);
    root.left = new Node(2);
    root.right = new Node(3);
    root.left.left = new Node(4);
    root.left.right = new Node(5);
    root.right.left = new Node(6);
    root.right.right = new Node(7);
    root.right.left.right = new Node(8);
    System.out.println("Dist(4, 5) = "
                       + findDistance(root, 4, 5));

    System.out.println("Dist(4, 6) = "
                       + findDistance(root, 4, 6));

    System.out.println("Dist(3, 4) = "
                       + findDistance(root, 3, 4));

    System.out.println("Dist(2, 4) = "
                       + findDistance(root, 2, 4));

    System.out.println("Dist(8, 5) = "
                       + findDistance(root, 8, 5));

}
}

// This code is contributed by Srinivasan Jayaraman.
```

### Python3

```
"""
A python program to find distance between n1
and n2 in binary tree
"""
# binary tree node
```

```
class Node:  
    # Constructor to create new node  
    def __init__(self, data):  
        self.data = data  
        self.left = self.right = None  
  
    # This function returns pointer to LCA of  
    # two given values n1 and n2.  
    def LCA(root, n1, n2):  
  
        # Base case  
        if root is None:  
            return None  
  
        # If either n1 or n2 matches with root's  
        # key, report the presence by returning  
        # root  
        if root.data == n1 or root.data == n2:  
            return root  
  
        # Look for keys in left and right subtrees  
        left = LCA(root.left, n1, n2)  
        right = LCA(root.right, n1, n2)  
  
        if left is not None and right is not None:  
            return root  
  
        # Otherwise check if left subtree or  
        # right subtree is LCA  
        if left:  
            return left  
        else:  
            return right  
  
    # function to find distance of any node  
    # from root  
    def findLevel(root, data, d, level):  
  
        # Base case when tree is empty  
        if root is None:  
            return  
  
        # Node is found then append level  
        # value to list and return  
        if root.data == data:  
            d.append(level)
```

```
    return

    findLevel(root.left, data, d, level + 1)
    findLevel(root.right, data, d, level + 1)

# function to find distance between two
# nodes in a binary tree
def findDistance(root, n1, n2):

    lca = LCA(root, n1, n2)

    # to store distance of n1 from lca
    d1 = []

    # to store distance of n2 from lca
    d2 = []

    # if lca exist
    if lca:

        # distance of n1 from lca
        findLevel(lca, n1, d1, 0)

        # distance of n2 from lca
        findLevel(lca, n2, d2, 0)
        return d1[0] + d2[0]
    else:
        return -1

# Driver program to test above function
root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)
root.right.left = Node(6)
root.right.right = Node(7)
root.right.left.right = Node(8)

print("Dist(4,5) = ", findDistance(root, 4, 5))
print("Dist(4,6) = ", findDistance(root, 4, 6))
print("Dist(3,4) = ", findDistance(root, 3, 4))
print("Dist(2,4) = ", findDistance(root, 2, 4))
print("Dist(8,5) = ", findDistance(root, 8, 5))

# This article is contributed by Shweta Singh.
```

Output :

```
Dist(4, 5) = 2  
Dist(4, 6) = 4  
Dist(3, 4) = 3  
Dist(2, 4) = 1  
Dist(8, 5) = 5
```

Thanks to NILMADHAB MONDAL for suggesting this solution.

**Improved By :** [muthuram85](#), [shweta44](#)

### Source

<https://www.geeksforgeeks.org/find-distance-between-two-nodes-of-a-binary-tree/>

## Chapter 161

# Find distance from root to given node in a binary tree

Find distance from root to given node in a binary tree - GeeksforGeeks

Given root of a binary tree and a key x in it, find distance of the given key from root.  
Distance means number of edges between two nodes.

Examples:

```
Input : x = 45,
        Root of below tree
          5
         /   \
        10   15
       / \   / \
      20 25 30 35
         \
        45
Output : Distance = 3
There are three edges on path
from root to 45.
```

For more understanding of question,  
in above tree distance of 35 is two  
and distance of 10 is 1.

The idea is to traverse the tree from root. Check if x is present at root or in left subtree or in right subtree. We initialize distance as -1 and add 1 to distance for all three cases.

```
// C++ program to find distance of a given
```

```
// node from root.
#include <bits/stdc++.h>
using namespace std;

// A Binary Tree Node
struct Node
{
    int data;
    Node *left, *right;
};

// A utility function to create a new Binary
// Tree Node
Node *newNode(int item)
{
    Node *temp = new Node;
    temp->data = item;
    temp->left = temp->right = NULL;
    return temp;
}

// Returns -1 if x doesn't exist in tree. Else
// returns distance of x from root
int findDistance(Node *root, int x)
{
    // Base case
    if (root == NULL)
        return -1;

    // Initialize distance
    int dist = -1;

    // Check if x is present at root or in left
    // subtree or right subtree.
    if ((root->data == x) ||
        (dist = findDistance(root->left, x)) >= 0 ||
        (dist = findDistance(root->right, x)) >= 0)
        return dist + 1;

    return dist;
}

// Driver Program to test above functions
int main()
{
    Node *root = newNode(5);
    root->left = newNode(10);
    root->right = newNode(15);
```

```
root->left->left = newNode(20);
root->left->right = newNode(25);
root->left->right->right = newNode(45);
root->right->left = newNode(30);
root->right->right = newNode(35);

cout << findDistance(root, 45);
return 0;
}
```

Output:

3

## Source

<https://www.geeksforgeeks.org/find-distance-root-given-node-binary-tree/>

## Chapter 162

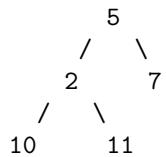
# Find first non matching leaves in two binary trees

Find first non matching leaves in two binary trees - GeeksforGeeks

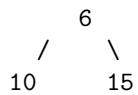
Given two binary trees, find first leaves of two trees that do not match. If there are no non-matching leaves, print nothing.

Examples:

Input : First Tree



Second Tree



Output : 11 15

If we consider leaves of two trees in order,  
we can see that 11 and 15 are the first leaves  
that do not match.

**Method 1 (Simple) :**

Do Inorder traversal of both trees one by one, store the leaves of both trees in two different lists. Finally find first values which are different in both lists. Time complexity is  $O(n_1 + n_2)$  where  $n_1$  and  $n_2$  are number of nodes in two trees. Auxiliary space requirement is  $O(n_1 + n_2)$ .

### Method 2 (Efficient)

This solution auxiliary space requirement as  $O(h_1 + h_2)$  where  $h_1$  and  $h_2$  are heights of trees. We do [Iterative Preorder traversal](#) of both the trees simultaneously using stacks. We maintain a stack for each tree. For every tree, keep pushing nodes in the stack till the top node is a leaf node. Compare the two top nodes of both the stack. If they are equal, do further traversal else return.

```
// C++ program to find first leaves that are
// not same.
#include<bits/stdc++.h>
using namespace std;

// Tree node
struct Node
{
    int data;
    Node *left, *right;
};

// Utility method to create a new node
Node *newNode(int x)
{
    Node * temp = new Node;
    temp->data = x;
    temp->left = temp->right = NULL;
    return temp;
}

bool isLeaf(Node * t)
{
    return ((t->left == NULL) && (t->right == NULL));
}

// Prints the first non-matching leaf node in
// two trees if it exists, else prints nothing.
void findFirstUnmatch(Node *root1, Node *root2)
{
    // If any of the tree is empty
    if (root1 == NULL || root2 == NULL)
        return;

    // Create two stacks for preorder traversals
    stack<Node*> s1, s2;
    s1.push(root1);
    s2.push(root2);

    while (!s1.empty() || !s2.empty())
    {
```

```
// If traversal of one tree is over
// and other tree still has nodes.
if (s1.empty() || s2.empty() )
    return;

// Do iterative traversal of first tree
// and find first lead node in it as "temp1"
Node *temp1 = s1.top();
s1.pop();
while (temp1 && !isLeaf(temp1))
{
    // pushing right childfirst so that
    // left child comes first while popping.
    s1.push(temp1->right);
    s1.push(temp1->left);
    temp1 = s1.top();
    s1.pop();
}

// Do iterative traversal of second tree
// and find first lead node in it as "temp2"
Node * temp2 = s2.top();
s2.pop();
while (temp2 && !isLeaf(temp2))
{
    s2.push(temp2->right);
    s2.push(temp2->left);
    temp2 = s2.top();
    s2.pop();
}

// If we found leaves in both trees
if (temp1 != NULL && temp2 != NULL )
{
    if (temp1->data != temp2->data )
    {
        cout << "First non matching leaves : "
            << temp1->data << " " << temp2->data
            << endl;
        return;
    }
}
}

// Driver code
int main()
{
```

```
struct Node *root1 = newNode(5);
root1->left = newNode(2);
root1->right = newNode(7);
root1->left->left = newNode(10);
root1->left->right = newNode(11);

struct Node * root2 = newNode(6);
root2->left = newNode(10);
root2->right = newNode(15);

findFirstUnmatch(root1,root2);

return 0;
}
```

Output:

11 15

**References:**

- <https://www.geeksforgeeks.org/amazon-interview-experience-set-337-sde-1/>
- <https://www.careercup.com/question?id=5673248478986240>

**Source**

<https://www.geeksforgeeks.org/find-first-non-matching-leaves-two-binary-trees/>

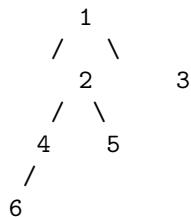
## Chapter 163

# Find height of a special binary tree whose leaf nodes are connected

Find height of a special binary tree whose leaf nodes are connected - GeeksforGeeks

Given a special binary tree whose leaf nodes are connected to form a circular doubly linked list, find its height.

For example,



In the above binary tree, 6, 5 and 3 are leaf nodes and they form a circular doubly linked list. Here, the left pointer of leaf node will act as a previous pointer of circular doubly linked list and its right pointer will act as next pointer of circular doubly linked list.

The idea is to follow similar approach as we do for finding height of a normal binary tree. We recursively calculate height of left and right subtrees of a node and assign height to the node as max of the heights of two children plus 1. But left and right child of a leaf node are null for normal binary trees. But, here leaf node is a circular doubly linked list node. So for a node to be a leaf node, we check if node's left's right is pointing to the node and its right's left is also pointing to the node itself.

Below is C++ implementation of above idea –

```
// C++ program to calculate height of a special tree
// whose leaf nodes forms a circular doubly linked list
#include <iostream>
using namespace std;

// A binary tree Node
struct Node
{
    int data;
    Node *left, *right;
};

// function to check if given node is a leaf node or node
bool isLeaf(Node* node)
{
    // If given node's left's right is pointing to given node
    // and its right's left is pointing to the node itself
    // then it's a leaf
    return node->left && node->left->right == node &&
           node->right && node->right->left == node;
}

/* Compute the height of a tree -- the number of
Nodes along the longest path from the root node
down to the farthest leaf node.*/
int maxDepth(Node* node)
{
    // if node is NULL, return 0
    if (node == NULL)
        return 0;

    // if node is a leaf node, return 1
    if (isLeaf(node))
        return 1;

    // compute the depth of each subtree and take maximum
    return 1 + max(maxDepth(node->left), maxDepth(node->right));
}

// Helper function that allocates a new tree node
Node* newNode(int data)
{
    Node* node = new Node;
    node->data = data;
    node->left = NULL;
    node->right = NULL;

    return node;
```

```
}

// Driver code
int main()
{
    Node* root = newNode(1);

    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->left->left->left = newNode(6);

    // Given tree contains 3 leaf nodes
    Node *L1 = root->left->left->left;
    Node *L2 = root->left->right;
    Node *L3 = root->right;

    // create circular doubly linked list out of
    // leaf nodes of the tree

    // set next pointer of linked list
    L1->right = L2, L2->right = L3, L3->right = L1;

    // set prev pointer of linked list
    L3->left = L2, L2->left = L1, L1->left = L3;

    // calculate height of the tree
    cout << "Height of tree is " << maxDepth(root);

    return 0;
}
```

Output:

```
Height of tree is 4
```

## Source

<https://www.geeksforgeeks.org/find-height-of-a-special-binary-tree-whose-leaf-nodes-are-connected/>

## Chapter 164

# Find if given vertical level of binary tree is sorted or not

Find if given vertical level of binary tree is sorted or not - GeeksforGeeks

Given a binary tree. Find if a given vertical level of the binary tree is sorted or not.  
(For the case when two nodes are overlapping, check if they form a sorted sequence in the level they lie.)

**Prerequisite:**[Vertical Order Traversal](#)

Examples:

```
Input : 1
        / \
       2   5
      / \
     7   4
    /
   6
Level l = -1
Output : Yes
Nodes in level -1 are 2 -> 6 which
forms a sorted sequence.
```

```
Input: 1
        / \
       2   6
      \ /
     3 4
Level l = 0
Output : Yes
Note that nodes with value 3 and 4
```

are overlapping in the binary tree.  
So we check if this form a sorted sequence level wise. The sequence formed at level 0 is 1 -> 3 -> 4 which is sorted.

A **simple** solution is to first do level order traversal of the binary tree and store each vertical level in different arrays. After this check, if array corresponding to level l is sorted or not. This solution has large memory requirements that can be reduced.

A **efficient** solution is to do vertical level order traversal of the binary tree and keep track of node values in vertical level l of the binary tree. A sequence is sorted if the previous element is less than or equal to the current element. While doing vertical level order traversal store previous value and compare current node in vertical level l with this previous value of level l. If current node value is greater than or equal to the previous value, then repeat the same procedure until the end of level l. If at any stage current node value is less than previous value then the level l is not sorted. If we reach at the end of level l then the level is sorted.

**Implementation:**

```
// CPP program to determine whether
// vertical level l of binary tree
// is sorted or not.
#include <bits/stdc++.h>
using namespace std;

// Structure of a tree node.
struct Node {
    int key;
    Node *left, *right;
};

// Function to create new tree node.
Node* newNode(int key)
{
    Node* temp = new Node;
    temp->key = key;
    temp->left = temp->right = NULL;
    return temp;
}

// Helper function to determine if
// vertical level l of given binary
// tree is sorted or not.
bool isSorted(Node* root, int level)
{
    // If root is null, then the answer is an
    // empty subset and an empty subset is
    // always considered to be sorted.
    if (root == NULL)
```

```
return true;

// Variable to store previous
// value in vertical level l.
int prevVal = INT_MIN;

// Variable to store current level
// while traversing tree vertically.
int currLevel;

// Variable to store current node
// while traversing tree vertically.
Node* currNode;

// Declare queue to do vertical order
// traversal. A pair is used as element
// of queue. The first element in pair
// represents the node and the second
// element represents vertical level
// of that node.
queue<pair<Node*, int> > q;

// Insert root in queue. Vertical level
// of root is 0.
q.push(make_pair(root, 0));

// Do vertical order traversal until
// all the nodes are not visited.
while (!q.empty()) {
    currNode = q.front().first;
    currLevel = q.front().second;
    q.pop();

    // Check if level of node extracted from
    // queue is required level or not. If it
    // is the required level then check if
    // previous value in that level is less
    // than or equal to value of node.
    if (currLevel == level) {
        if (prevVal <= currNode->key)
            prevVal = currNode->key;
        else
            return false;
    }

    // If left child is not NULL then push it
    // in queue with level reduced by 1.
    if (currNode->left)
```

```
q.push(make_pair(currNode->left, currLevel - 1));

// If right child is not NULL then push it
// in queue with level increased by 1.
if (currNode->right)
    q.push(make_pair(currNode->right, currLevel + 1));
}

// If the level asked is not present in the
// given binary tree, that means that level
// will contain an empty subset. Therefore answer
// will be true.
return true;
}

// Driver program
int main()
{
/*
      1
     / \
    2   5
   / \
  7   4
   /
  6
*/
Node* root = newNode(1);
root->left = newNode(2);
root->right = newNode(5);
root->left->left = newNode(7);
root->left->right = newNode(4);
root->left->right->left = newNode(6);

int level = -1;
if (isSorted(root, level) == true)
    cout << "Yes";
else
    cout << "No";
return 0;
}
```

**Output:**

Yes

**Time Complexity:**  $O(n)$

**Auxiliary Space:**  $O(n)$

### Source

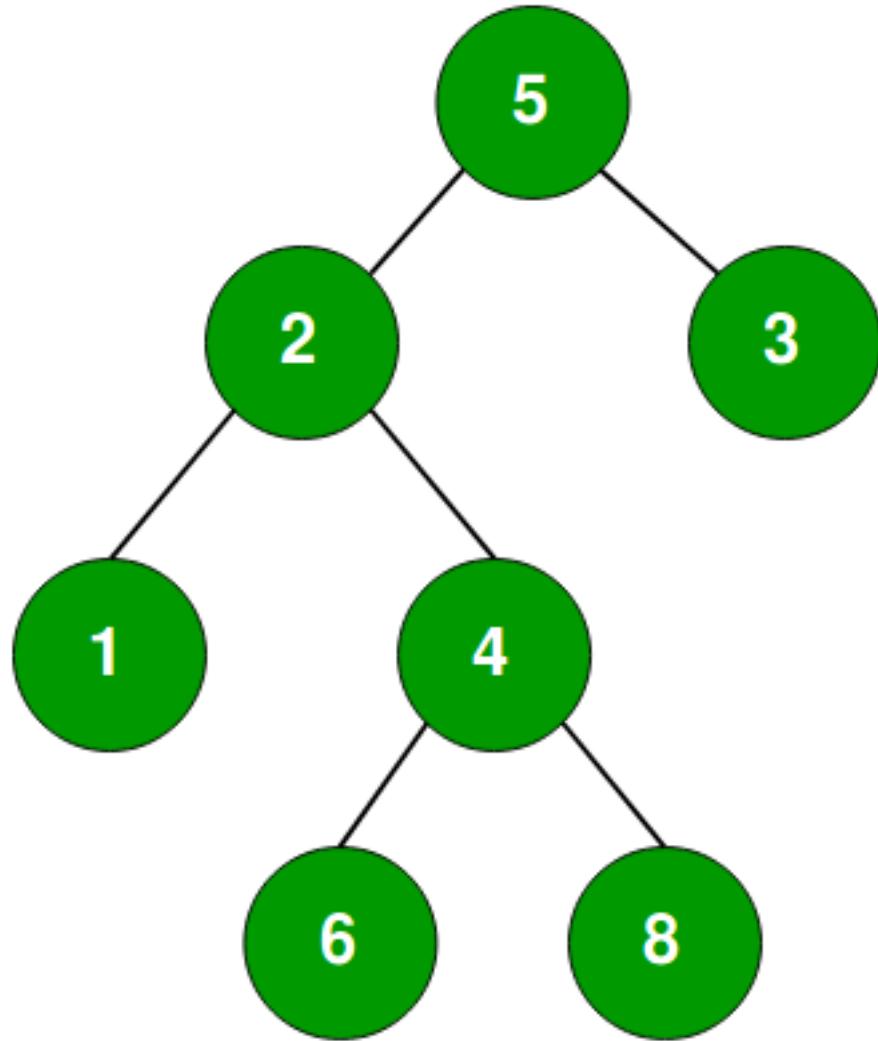
<https://www.geeksforgeeks.org/find-given-vertical-level-binary-tree-sorted-not/>

## Chapter 165

**Find if there is a pair in root to a leaf path with sum equals to root's data**

Find if there is a pair in root to a leaf path with sum equals to root's data - GeeksforGeeks

Given a binary tree, find if there is a pair in root to a leaf path such that sum of values in pair is equal to root's data. For example, in below tree (2, 3) and (4, 1) are pairs with sum equals to root's data.



The idea is based on hashing and tree traversal. The idea is similar to method 2 of array pair sum problem.

- Create an empty hash table.
- Start traversing tree in Preorder fashion.
- If we reach a leaf node, we return false.
- For every visited node, check if root's data minus current node's data exists in hash table or not. If yes, return true. Else insert current node in hash table.
- Recursively check in left and right subtrees.
- Remove current node from hash table so that it doesn't appear in other root to leaf paths.

Below is C++ implementation of above idea.

```
// C++ program to find if there is a pair in any root
// to leaf path with sum equals to root's key.
#include<bits/stdc++.h>
using namespace std;

/* A binary tree node has data, pointer to left child
and a pointer to right child */
struct Node
{
    int data;
    struct Node* left, *right;
};

/* utility that allocates a new node with the
given data and NULL left and right pointers. */
struct Node* newnode(int data)
{
    struct Node* node = new Node;
    node->data = data;
    node->left = node->right = NULL;
    return (node);
}

// Function to print root to leaf path which satisfies the condition
bool printPathUtil(Node *node, unordered_set<int> &s, int root_data)
{
    // Base condition
    if (node == NULL)
        return false;

    // Check if current node makes a pair with any of the
    // existing elements in set.
    int rem = root_data - node->data;
    if (s.find(rem) != s.end())
        return true;

    // Insert current node in set
    s.insert(node->data);

    // If result returned by either left or right child is
    // true, return true.
    bool res = printPathUtil(node->left, s, root_data) ||
               printPathUtil(node->right, s, root_data);

    // Remove current node from hash table
    s.erase(node->data);
}
```

```
    return res;
}

// A wrapper over printPathUtil()
bool isPathSum(Node *root)
{
    // create an empty hash table
    unordered_set<int> s;

    // Recursively check in left and right subtrees.
    return printPathUtil(root->left, s, root->data) ||
           printPathUtil(root->right, s, root->data);
}

// Driver program to run the case
int main()
{
    struct Node *root = newnode(8);
    root->left      = newnode(5);
    root->right     = newnode(4);
    root->left->left = newnode(9);
    root->left->right = newnode(7);
    root->left->right->left = newnode(1);
    root->left->right->right = newnode(12);
    root->left->right->right->right = newnode(2);
    root->right->right = newnode(11);
    root->right->right->left = newnode(3);
    isPathSum(root)? cout << "Yes" : cout << "No";
    return 0;
}
```

Output:

Yes

Time Complexity :  $O(n)$  under the assumption that hash search, insert and erase take  $O(1)$  time.

**Exercise :** Extend the above solution to print all root to leaf paths that have a pair with sum equals to root's data.

## Source

<https://www.geeksforgeeks.org/find-pair-root-leaf-path-sum-equals-roots-data/>

## Chapter 166

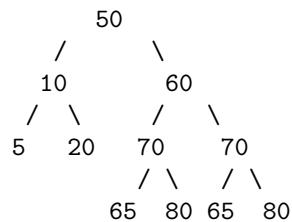
# Find largest subtree having identical left and right subtrees

Find largest subtree having identical left and right subtrees - GeeksforGeeks

Given a binary tree, find the largest subtree having identical left and right subtree. Expected complexity is  $O(n)$ .

For example,

Input:



Output:

Largest subtree is rooted at node 60

A simple solution is to consider every node, recursively check if left and right subtrees are identical using the approach discussed [here](#). Keep track of maximum size such node.

We can save recursive calls. The idea is to do a postorder traversal of given binary tree and for each node, we store structure of its left and right subtrees. In order to store the structure of left and right subtree, we use a string. We separate left and right subtree nodes from current node in the string by using a delimiter. For every encountered node, we update largest subtree found so far if its left and right subtree structure are similar.

Below is C++ implementation of above idea –

```
// C++ program to find the largest subtree
```

```
// having identical left and right subtree
#include <bits/stdc++.h>
using namespace std;

/* A binary tree node has data, pointer to left
   child and a pointer to right child */
struct Node
{
    int data;
    Node* left, * right;
};

/* Helper function that allocates a new node with
   the given data and NULL left and right pointers. */
Node* newNode(int data)
{
    Node* node = new Node;
    node->data = data;
    node->left = node->right = NULL;
    return (node);
}

// Sets maxSize to size of largest subtree with
// identical left and right. maxSize is set with
// size of the maximum sized subtree. It returns
// size of subtree rooted with current node. This
// size is used to keep track of maximum size.
int largestSubtreeUtil(Node* root, string& str,
                      int& maxSize, Node*& maxNode)
{
    if (root == NULL)
        return 0;

    // string to store structure of left and
    // right subtrees
    string left = "", right = "";

    // traverse left subtree and finds its size
    int ls = largestSubtreeUtil(root->left, left,
                                maxSize, maxNode);

    // traverse right subtree and finds its size
    int rs = largestSubtreeUtil(root->right, right,
                                maxSize, maxNode);

    // if left and right subtrees are similar
    // update maximum subtree if needed (Note that
    // left subtree may have a bigger value than
```

```
// right and vice versa)
int size = ls + rs + 1;
if (left.compare(right) == 0)
{
    if (size > maxSize)
    {
        maxSize = size;
        maxNode = root;
    }
}

// append left subtree data
str.append("|").append(left).append("|");

// append current node data
str.append("|").append(to_string(root->data)).append("|");

// append right subtree data
str.append("|").append(right).append("|");

return size;
}

// function to find the largest subtree
// having identical left and right subtree
int largestSubtree(Node* node, Node*& maxNode)
{
    int maxSize = 0;
    string str = "";
    largestSubtreeUtil(node, str, maxSize, maxNode);

    return maxSize;
}

/* Driver program to test above functions*/
int main()
{
    /* Let us construct the following Tree
           50
         /   \
        10   60
       / \   / \
      5 20  70  70
     / \ / \
    65 80 65 80 */
    Node* root = newNode(50);
    root->left = newNode(10);
    root->right = newNode(60);
```

```
root->left->left = newNode(5);
root->left->right = newNode(20);
root->right->left = newNode(70);
root->right->left->left = newNode(65);
root->right->left->right = newNode(80);
root->right->right = newNode(70);
root->right->right->left = newNode(65);
root->right->right->right = newNode(80);

Node *maxNode = NULL;
int maxSize = largestSubtree(root, maxNode);

cout << "Largest Subtree is rooted at node "
    << maxNode->data << "\nand its size is "
    << maxSize;

return 0;
}
```

Output :

```
Largest Subtree is rooted at node 60
and its size is 7
```

The worst case time complexity still remains  $O(n^2)$  as we need  $O(n)$  time to compare two strings.

**Further Optimization:**

We can optimized the space used in above program by using [Succinct Encoding of Binary Tree](#).

## Source

<https://www.geeksforgeeks.org/find-largest-subtree-having-identical-left-and-right-subtrees/>

## Chapter 167

# Find largest subtree sum in a tree

Find largest subtree sum in a tree - GeeksforGeeks

Given a binary tree, task is to find subtree with maximum sum in tree.

**Examples:**

Input :        1  
          /    \  
        2        3  
      / \      / \\\  
    4    5    6    7

Output : 28

As all the tree elements are positive,  
the largest subtree sum is equal to  
sum of all tree elements.

Input :        1  
          /    \  
        -2        3  
      / \      / \\\  
    4    5    -6    2

Output : 7

Subtree with largest sum is : -2

          /    \\\  
          4        5

Also, entire tree sum is also 7.

**Approach :** Do post order traversal of the binary tree. At every node, find left subtree value and right subtree value recursively. The value of subtree rooted at current node is equal to sum of current node value, left node subtree sum and right node subtree sum. Compare current subtree sum with overall maximum subtree sum so far.

**Implementation :**

C++

```
// C++ program to find largest subtree
// sum in a given binary tree.
#include <bits/stdc++.h>
using namespace std;

// Structure of a tree node.
struct Node {
    int key;
    Node *left, *right;
};

// Function to create new tree node.
Node* newNode(int key)
{
    Node* temp = new Node;
    temp->key = key;
    temp->left = temp->right = NULL;
    return temp;
}

// Helper function to find largest
// subtree sum recursively.
int findLargestSubtreeSumUtil(Node* root, int& ans)
{
    // If current node is null then
    // return 0 to parent node.
    if (root == NULL)
        return 0;

    // Subtree sum rooted at current node.
    int currSum = root->key +
        findLargestSubtreeSumUtil(root->left, ans)
        + findLargestSubtreeSumUtil(root->right, ans);

    // Update answer if current subtree
    // sum is greater than answer so far.
    ans = max(ans, currSum);

    // Return current subtree sum to
}
```

```
// its parent node.  
    return currSum;  
}  
  
// Function to find largest subtree sum.  
int findLargestSubtreeSum(Node* root)  
{  
    // If tree does not exist,  
    // then answer is 0.  
    if (root == NULL)  
        return 0;  
  
    // Variable to store maximum subtree sum.  
    int ans = INT_MIN;  
  
    // Call to recursive function to  
    // find maximum subtree sum.  
    findLargestSubtreeSumUtil(root, ans);  
  
    return ans;  
}  
  
// Driver function  
int main()  
{  
    /*  
       1  
      / \br/>     -2   3  
    / \   / \br/>   4   5 -6   2  
*/  
  
    Node* root = newNode(1);  
    root->left = newNode(-2);  
    root->right = newNode(3);  
    root->left->left = newNode(4);  
    root->left->right = newNode(5);  
    root->right->left = newNode(-6);  
    root->right->right = newNode(2);  
  
    cout << findLargestSubtreeSum(root);  
    return 0;  
}
```

Java

```
// Java program to find largest
// subtree sum in a given binary tree.
import java.util.*;
class GFG
{
    // Structure of a tree node.
    static class Node
    {
        int key;
        Node left, right;
    }

    static class INT
    {
        int v;
        INT(int a)
        {
            v = a;
        }
    }

    // Function to create new tree node.
    static Node newNode(int key)
    {
        Node temp = new Node();
        temp.key = key;
        temp.left = temp.right = null;
        return temp;
    }

    // Helper function to find largest
    // subtree sum recursively.
    static int findLargestSubtreeSumUtil(Node root,
                                         INT ans)
    {
        // If current node is null then
        // return 0 to parent node.
        if (root == null)
            return 0;

        // Subtree sum rooted
        // at current node.
        int currSum = root.key +
                     findLargestSubtreeSumUtil(root.left, ans) +
                     findLargestSubtreeSumUtil(root.right, ans);

        // Update answer if current subtree
        // sum is greater than answer so far.
        ans.v = Math.max(ans.v, currSum);

        // Return current subtree
```

```
// sum to its parent node.  
return currSum;  
}  
  
// Function to find  
// largest subtree sum.  
static int findLargestSubtreeSum(Node root)  
{  
    // If tree does not exist,  
    // then answer is 0.  
    if (root == null)  
        return 0;  
  
    // Variable to store  
    // maximum subtree sum.  
    INT ans = new INT(-9999999);  
  
    // Call to recursive function  
    // to find maximum subtree sum.  
    findLargestSubtreeSumUtil(root, ans);  
  
    return ans.v;  
}  
  
// Driver Code  
public static void main(String args[])  
{  
    /*  
     1  
     / \br/>     / \br/>    -2 3  
    / \ / \br/>    / \ / \br/>   4 5 -6 2  
    */  
  
    Node root = newNode(1);  
    root.left = newNode(-2);  
    root.right = newNode(3);  
    root.left.left = newNode(4);  
    root.left.right = newNode(5);  
    root.right.left = newNode(-6);  
    root.right.right = newNode(2);  
  
    System.out.println(findLargestSubtreeSum(root));  
}  
}  
  
// This code is contributed by Arnab Kundu
```

**Output:**

7

**Time Complexity:**  $O(n)$ , where  $n$  is number of nodes.

**Auxiliary Space:**  $O(n)$ , function call stack size.

**Improved By :** [andrew1234](#)

### Source

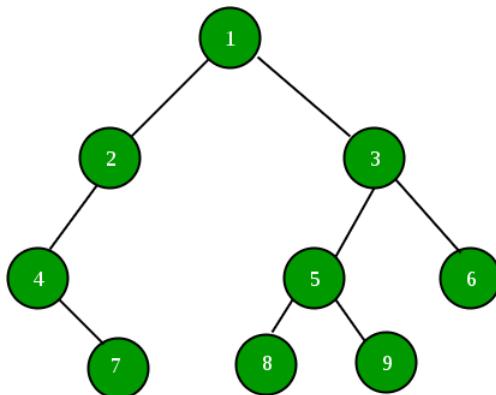
<https://www.geeksforgeeks.org/find-largest-subtree-sum-tree/>

## Chapter 168

# Find maximum (or minimum) in Binary Tree

Find maximum (or minimum) in Binary Tree - GeeksforGeeks

Given a Binary Tree, find maximum(or minimum) element in it. For example, maximum in the following Binary Tree is 9.



In Binary Search Tree, we can find maximum by traversing right pointers until we reach rightmost node. But in Binary Tree, we must visit every node to figure out maximum. So the idea is to traverse the given tree and for every node return maximum of 3 values.

- 1) Node's data.
- 2) Maximum in node's left subtree.
- 3) Maximum in node's right subtree.

Below is the implementation of above approach.

C

```
// C program to find maximum and minimum in a Binary Tree  
#include <stdio.h>
```

```
#include <stdlib.h>
#include <limits.h>

// A tree node
struct Node
{
    int data;
    struct Node* left, *right;
};

// A utility function to create a new node
struct Node* newNode(int data)
{
    struct Node* node = (struct Node*)
                           malloc(sizeof(struct Node));
    node->data = data;
    node->left = node->right = NULL;
    return(node);
}

// Returns maximum value in a given Binary Tree
int findMax(struct Node* root)
{
    // Base case
    if (root == NULL)
        return INT_MIN;

    // Return maximum of 3 values:
    // 1) Root's data 2) Max in Left Subtree
    // 3) Max in right subtree
    int res = root->data;
    int lres = findMax(root->left);
    int rres = findMax(root->right);
    if (lres > res)
        res = lres;
    if (rres > res)
        res = rres;
    return res;
}

// Driver program
int main(void)
{
    struct Node*NewRoot=NULL;
    struct Node *root = newNode(2);
    root->left      = newNode(7);
    root->right     = newNode(5);
    root->left->right = newNode(6);
```

```
root->left->right->left=newNode(1);
root->left->right->right=newNode(11);
root->right->right=newNode(9);
root->right->right->left=newNode(4);

printf ("Maximum element is %d \n", findMax(root));

return 0;
}
```

**Java**

```
// Java code to Find maximum (or minimum) in
// Binary Tree

// A binary tree node
class Node {
    int data;
    Node left, right;

    public Node(int data)
    {
        this.data = data;
        left = right = null;
    }
}

class BinaryTree {
    Node root;

    // Returns the max value in a binary tree
    static int findMax(Node node)
    {
        if (node == null)
            return Integer.MIN_VALUE;

        int res = node.data;
        int lres = findMax(node.left);
        int rres = findMax(node.right);

        if (lres > res)
            res = lres;
        if (rres > res)
            res = rres;
        return res;
    }

    /* Driver program to test above functions */
}
```

```
public static void main(String args[])
{
    BinaryTree tree = new BinaryTree();
    tree.root = new Node(2);
    tree.root.left = new Node(7);
    tree.root.right = new Node(5);
    tree.root.left.right = new Node(6);
    tree.root.left.right.left = new Node(1);
    tree.root.left.right.right = new Node(11);
    tree.root.right.right = new Node(9);
    tree.root.right.right.left = new Node(4);

    System.out.println("Maximum element is " +
                       tree.findMax(tree.root));
}
}

// This code is contributed by Kamal Rawal
```

Output:

```
Maximum element is 11
```

Similarly, we can find minimum element in Binary tree by comparing three values. Below is the function to find minimum in Binary Tree.

C

```
// Returns minimum value in a given Binary Tree
int findMin(struct Node* root)
{
    // Base case
    if (root == NULL)
        return INT_MAX;

    // Return minimum of 3 values:
    // 1) Root's data 2) Max in Left Subtree
    // 3) Max in right subtree
    int res = root->data;
    int lres = findMin(root->left);
    int rres = findMin(root->right);
    if (lres < res)
        res = lres;
    if (rres < res)
        res = rres;
    return res;
}
```

### Java

```
// Returns the min value in a binary tree
static int findMin(Node node)
{
    if (node == null)
        return Integer.MAX_VALUE;

    int res = node.data;
    int lres = findMin(node.left);
    int rres = findMin(node.right);

    if (lres < res)
        res = lres;
    if (rres < res)
        res = rres;
    return res;
}
```

This article is contributed by **Himanshu Gupta**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

### Source

<https://www.geeksforgeeks.org/find-maximum-or-minimum-in-binary-tree/>

## Chapter 169

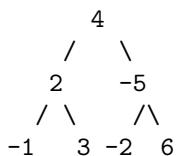
# Find maximum level product in Binary Tree

Find maximum level product in Binary Tree - GeeksforGeeks

Given a Binary Tree having positive and negative nodes, the task is to find maximum product level in it.

Examples:

Input :



Output: 36

Explanation :

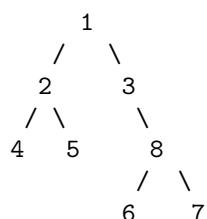
Product of all nodes of 0'th level is 4

Product of all nodes of 1'th level is -10

Product of all nodes of 0'th level is 36

Hence maximum product is 6

Input :



Output : 160

Explanation :

Product of all nodes of 0'th level is 1

```
Product of all nodes of 1'th level is 6
Product of all nodes of 0'th level is 160
Product of all nodes of 0'th level is 42
Hence maximum product is 160
```

**Prerequisites:** [Maximum Width of a Binary Tree](#)

**Approach :** The idea is to do level order traversal of tree. While doing traversal, process nodes of different level separately. For every level being processed, compute product of nodes in the level and keep track of maximum product.

```
// A queue based C++ program to find maximum product
// of a level in Binary Tree
#include <bits/stdc++.h>
using namespace std;

/* A binary tree node has data, pointer to left child
and a pointer to right child */
struct Node {
    int data;
    struct Node *left, *right;
};

// Function to find the maximum product of a level in tree
// using level order traversal
int maxLevelProduct(struct Node* root)
{
    // Base case
    if (root == NULL)
        return 0;

    // Initialize result
    int result = root->data;

    // Do Level order traversal keeping track of number
    // of nodes at every level.
    queue<Node*> q;
    q.push(root);
    while (!q.empty()) {

        // Get the size of queue when the level order
        // traversal for one level finishes
        int count = q.size();

        // Iterate for all the nodes in the queue currently
        int product = 1;
        while (count--) {
```

```
// Dequeue an node from queue
Node* temp = q.front();
q.pop();

// Multiply this node's value to current product.
product = product * temp->data;

// Enqueue left and right children of
// dequeued node
if (temp->left != NULL)
    q.push(temp->left);
if (temp->right != NULL)
    q.push(temp->right);
}

// Update the maximum node count value
result = max(product, result);
}

return result;
}

/* Helper function that allocates a new node with the
given data and NULL left and right pointers. */
struct Node* newNode(int data)
{
    struct Node* node = new Node;
    node->data = data;
    node->left = node->right = NULL;
    return (node);
}

// Driver code
int main()
{
    struct Node* root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->right = newNode(8);
    root->right->right->left = newNode(6);
    root->right->right->right = newNode(7);

    /* Constructed Binary tree is:
        1
       / \
      2   3
    
```

```
        / \   \
    4     5   8
        / \
    6     7 */
cout << "Maximum level product is "
      << maxLevelProduct(root) << endl;
return 0;
}
```

Output :

Maximum level product is 160

**Time Complexity :** O(n)

**Auxiliary Space :** O(n)

### Source

<https://www.geeksforgeeks.org/find-maximum-level-product-binary-tree/>

## Chapter 170

# Find maximum level sum in Binary Tree

Find maximum level sum in Binary Tree - GeeksforGeeks

Given a Binary Tree having positive and negative nodes, the task is to find maximum sum level in it.

Examples:

Input :                  4  
                  /    \  
              2      -5  
          / \    / \\\  
        -1    3   -2    6

Output: 6

Explanation :

Sum of all nodes of 0'th level is 4

Sum of all nodes of 1'th level is -3

Sum of all nodes of 0'th level is 6

Hence maximum sum is 6

Input :                  1  
                  /    \  
              2      3  
          / \    / \\\  
        4    5      8  
                  /    \  
              6      7

Output : 17

This problem is a variation of [maximum width problem](#). The idea is to do level order traversal of tree. While doing traversal, process nodes of different level separately. For

every level being processed, compute sum of nodes in the level and keep track of maximum sum.

```
// A queue based C++ program to find maximum sum
// of a level in Binary Tree
#include<bits/stdc++.h>
using namespace std ;

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct Node
{
    int data ;
    struct Node * left, * right ;
};

// Function to find the maximum sum of a level in tree
// using level order traversal
int maxLevelSum(struct Node * root)
{
    // Base case
    if (root == NULL)
        return 0;

    // Initialize result
    int result = root->data;

    // Do Level order traversal keeping track of number
    // of nodes at every level.
    queue<Node*> q;
    q.push(root);
    while (!q.empty())
    {
        // Get the size of queue when the level order
        // traversal for one level finishes
        int count = q.size() ;

        // Iterate for all the nodes in the queue currently
        int sum = 0;
        while (count--)
        {
            // Dequeue an node from queue
            Node *temp = q.front();
            q.pop();

            // Add this node's value to current sum.
            sum = sum + temp->data;
        }
    }
}
```

```

        // Enqueue left and right children of
        // dequeued node
        if (temp->left != NULL)
            q.push(temp->left);
        if (temp->right != NULL)
            q.push(temp->right);
    }

    // Update the maximum node count value
    result = max(sum, result);
}

return result;
}

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct Node * newNode(int data)
{
    struct Node * node = new Node;
    node->data = data;
    node->left = node->right = NULL;
    return (node);
}

int main()
{
    struct Node *root = newNode(1);
    root->left      = newNode(2);
    root->right     = newNode(3);
    root->left->left  = newNode(4);
    root->left->right = newNode(5);
    root->right->right = newNode(8);
    root->right->right->left  = newNode(6);
    root->right->right->right = newNode(7);

    /* Constructed Binary tree is:
           1
         /   \
        2     3
       / \   /
      4   5   8
         /   \
        6     7
    */

    cout << "Maximum level sum is "
        << maxLevelSum(root) << endl;
    return 0;
}

```

Output :

```
Maximum level sum is 17
```

Time Complexity :  $O(n)$   
Auxiliary Space :  $O(n)$

### Source

<https://www.geeksforgeeks.org/find-level-maximum-sum-binary-tree/>

## Chapter 171

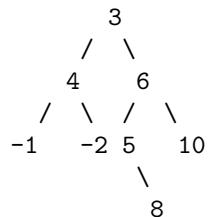
# Find maximum vertical sum in binary tree

Find maximum vertical sum in binary tree - GeeksforGeeks

Given a binary tree, find the maximum vertical level sum in binary tree.

Examples:

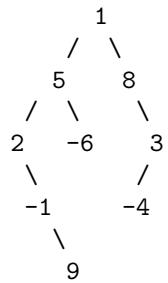
Input :



Output : 14

Vertical level having nodes 6 and 8 has maximum vertical sum 14.

Input :



Output : 4

A **simple** solution is to first find vertical level sum of each level starting from minimum vertical level to maximum vertical level. Finding sum of one vertical level takes  $O(n)$  time. In worst case time complexity of this solution is  $O(n^2)$ .

An **efficient** solution is to do level order traversal of given binary tree and update vertical level sum of each level while doing the traversal. After finding vertical sum of each level find maximum vertical sum from these values.

Below is the implementation of above approach:

```
// CPP program to find maximum vertical
// sum in binary tree.
#include <bits/stdc++.h>
using namespace std;

// A Binary Tree Node
struct Node {
    int data;
    struct Node *left, *right;
};

// A utility function to create a new
// Binary Tree Node
struct Node* newNode(int item)
{
    struct Node* temp = (struct Node*)malloc(sizeof(struct Node));
    temp->data = item;
    temp->left = temp->right = NULL;
    return temp;
}

// Function to find maximum vertical sum
// in binary tree.
int maxVerticalSum(Node* root)
{
    if (root == NULL) {
        return 0;
    }

    // To store sum of each vertical level.
    unordered_map<int, int> verSum;

    // To store maximum vertical level sum.
    int maxSum = INT_MIN;

    // To store vertical level of current node.
```

```
int currLev;

// Queue to perform level order traversal.
// Each element of queue is a pair of node
// and its vertical level.
queue<pair<Node*, int>> q;
q.push({ root, 0 });

while (!q.empty()) {

    // Extract node at front of queue
    // and its vertical level.
    root = q.front().first;
    currLev = q.front().second;
    q.pop();

    // Update vertical level sum of
    // vertical level to which
    // current node belongs to.
    verSum[currLev] += root->data;

    if (root->left)
        q.push({ root->left, currLev - 1 });

    if (root->right)
        q.push({ root->right, currLev + 1 });
}

// Find maximum vertical level sum.
for (auto it : verSum)
    maxSum = max(maxSum, it.second);

return maxSum;
}

// Driver Program to test above functions
int main()
{
    /*
        3
       / \
      4   6
     / \ / \
    -1 -2 5 10
           \
          8
    */
}
```

```
struct Node* root = newNode(3);
root->left = newNode(4);
root->right = newNode(6);
root->left->left = newNode(-1);
root->left->right = newNode(-2);
root->right->left = newNode(5);
root->right->right = newNode(10);
root->right->left->right = newNode(8);

cout << maxVerticalSum(root);
return 0;
}
```

**Output:**

14

**Time Complexity:**  $O(n)$

**Auxiliary Space:**  $O(n)$

**Source**

<https://www.geeksforgeeks.org/find-maximum-vertical-sum-in-binary-tree/>

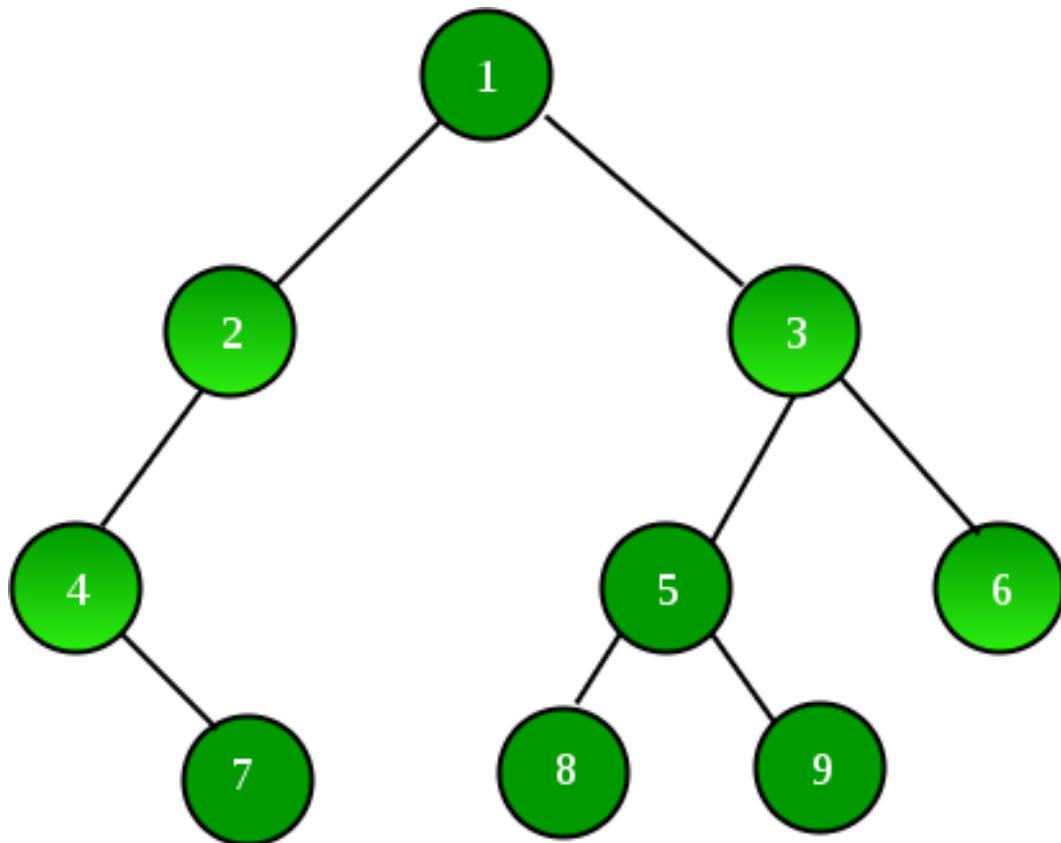
## Chapter 172

# Find mirror of a given node in Binary tree

Find mirror of a given node in Binary tree - GeeksforGeeks

Given a Binary tree, the problem is to find mirror of a given node. The mirror of a node is a node which exist at the mirror position of node in opposite subtree at the root.

Examples:



In above tree-

Node 2 and 3 are mirror nodes

Node 4 and 6 are mirror nodes.

We can have a recursive solution for finding mirror nodes. The algorithm is following –

- 1) Start from the root of the tree and recur nodes from both subtree simultaneously using two pointers for left and right nodes.
- 2) First recur all the external nodes and store returned value in mirror variable.
- 3) If current node value is equal to target node, return the value of opposite pointer else repeat step 2.
- 4) If no external node is left and mirror is none, recur internal nodes.

```
// C program to find the mirror Node in Binary tree
#include <stdio.h>
#include <stdlib.h>

/* A binary tree Node has data, pointer to left child
   and a pointer to right child */
struct Node
{
    int key;
    struct Node* left, *right;
};

// create new Node and initialize it
struct Node* newNode(int key)
{
    struct Node* n = (struct Node*)
                    malloc(sizeof(struct Node*));
    if (n != NULL)
    {
        n->key = key;
        n->left = NULL;
        n->right = NULL;
        return n;
    }
    else
    {
        printf("Memory allocation failed!");
        exit(1);
    }
}

// recursive function to find mirror of Node
int findMirrorRec(int target, struct Node* left,
                  struct Node* right)
{
    /* if any of the Node is none then Node itself
       and decendent have no mirror, so return
       none, no need to further explore! */
    if (left==NULL || right==NULL)
        return 0;

    /* if left Node is target Node, then return
       right's key (that is mirror) and vice
       versa */
    if (left->key == target)
        return right->key;

    if (right->key == target)
```

```
        return left->key;

    // first recur external Nodes
    int mirror_val = findMirrorRec(target,
                                    left->left,
                                    right->right);
    if (mirror_val)
        return mirror_val;

    // if no mirror found, recur internal Nodes
    findMirrorRec(target, left->right, right->left);
}

// interface for mirror search
int findMirror(struct Node* root, int target)
{
    if (root == NULL)
        return 0;
    if (root->key == target)
        return target;
    return findMirrorRec(target, root->left, root->right);
}

// Driver
int main()
{
    struct Node* root      = newNode(1);
    root->left            = newNode(2);
    root->left->left     = newNode(4);
    root->left->left->right = newNode(7);
    root->right           = newNode(3);
    root->right->left    = newNode(5);
    root->right->right   = newNode(6);
    root->right->left->left = newNode(8);
    root->right->left->right = newNode(9);

    // target Node whose mirror have to be searched
    int target = root->left->left->key;

    int mirror = findMirror(root, target);

    if (mirror)
        printf("Mirror of Node %d is Node %d\n",
               target, mirror);
    else
        printf("Mirror of Node %d is NULL!\n", target);
}
```

### Python

```
# Python3 program to find the mirror node in
# Binary tree

class Node:
    '''A binary tree node has data, reference to left child
       and a reference to right child'''

    def __init__(self, key, lchild=None, rchild=None):
        self.key = key
        self.lchild = None
        self.rchild = None

    # recursive function to find mirror
    def findMirrorRec(target, left, right):

        # If any of the node is none then node itself
        # and decendent have no mirror, so return
        # none, no need to further explore!
        if left == None or right == None:
            return None

        # if left node is target node, then return
        # right's key (that is mirror) and vice versa
        if left.key == target:
            return right.key
        if right.key == target:
            return left.key

        # first recur external nodes
        mirror_val = findMirrorRec(target, left.lchild, right.rchild)
        if mirror_val != None:
            return mirror_val

        # if no mirror found, recur internal nodes
        findMirrorRec(target, left.rchild, right.lchild)

    # interface for mirror search
    def findMirror(root, target):
        if root == None:
            return None

        if root.key == target:
            return target

        return findMirrorRec(target, root.lchild, root.rchild)
```

```
# Driver
def main():
    root = Node(1)
    n1 = Node(2)
    n2 = Node(3)
    root.lchild = n1
    root.rchild = n2
    n3 = Node(4)
    n4 = Node(5)
    n5 = Node(6)
    n1.lchild = n3
    n2.lchild = n4
    n2.rchild = n5
    n6 = Node(7)
    n7 = Node(8)
    n8 = Node(9)
    n3.rchild = n6
    n4.lchild = n7
    n4.rchild = n8

    # target node whose mirror have to be searched
    target = n3.key

    mirror = findMirror(root, target)
    print("Mirror of node {} is node {}".format(target, mirror))

if __name__ == '__main__':
    main()
```

Output:

Mirror of node 4 is node 6

Time Complexity:

## Source

<https://www.geeksforgeeks.org/find-mirror-given-node-binary-tree/>

## Chapter 173

# Find multiplication of sums of data of leaves at same levels

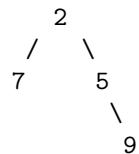
Find multiplication of sums of data of leaves at same levels - GeeksforGeeks

Given a Binary Tree, return following value for it.

- 1) For every level, compute sum of all leaves if there are leaves at this level. Otherwise ignore it.
- 2) Return multiplication of all sums.

Examples:

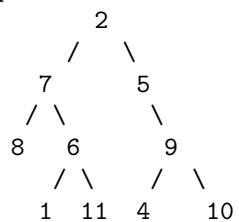
Input: Root of below tree



Output: 63

First levels doesn't have leaves. Second level has one leaf 7 and third level also has one leaf 9. Therefore result is  $7 * 9 = 63$

Input: Root of below tree



```
Output: 208
First two levels don't have leaves. Third
level has single leaf 8. Last level has four
leaves 1, 11, 4 and 10. Therefore result is
8 * (1 + 11 + 4 + 10)
```

We strongly recommend you to minimize your browser and try this yourself first.

One **Simple Solution** is to recursively compute leaf sum for all level starting from top to bottom. Then multiply sums of levels which have leaves. Time complexity of this solution would be  $O(n^2)$ .

An **Efficient Solution** is to use Queue based level order traversal. While doing the traversal, process all different levels separately. For every processed level, check if it has any leaves. If it has then compute sum of leaf nodes. Finally return product of all sums.

C++

```
/* Iterative C++ program to find sum of data of all leaves
   of a binary tree on same level and then multiply sums
   obtained of all levels. */
#include <bits/stdc++.h>
using namespace std;

// A Binary Tree Node
struct Node
{
    int data;
    struct Node *left, *right;
};

// helper function to check if a Node is leaf of tree
bool isLeaf(Node *root)
{
    return (!root->left && !root->right);
}

/* Calculate sum of all leaf Nodes at each level and returns
   multiplication of sums */
int sumAndMultiplyLevelData(Node *root)
{
    // Tree is empty
    if (!root)
        return 0;

    int mul = 1;      /* To store result */

    queue<Node*> q;
    q.push(root);

    while (!q.empty())
    {
        int size = q.size();
        int sum = 0;

        for (int i = 0; i < size; i++)
        {
            Node *temp = q.front();
            q.pop();

            if (isLeaf(temp))
                sum += temp->data;
        }

        mul *= sum;
    }
}
```

```
// Create an empty queue for level order traversal
queue<Node *> q;

// Enqueue Root and initialize height
q.push(root);

// Do level order traversal of tree
while (1)
{
    // NodeCount (queue size) indicates number of Nodes
    // at current level.
    int NodeCount = q.size();

    // If there are no Nodes at current level, we are done
    if (NodeCount == 0)
        break;

    // Initialize leaf sum for current level
    int levelSum = 0;

    // A boolean variable to indicate if found a leaf
    // Node at current level or not
    bool leafFound = false;

    // Dequeue all Nodes of current level and Enqueue all
    // Nodes of next level
    while (NodeCount > 0)
    {
        // Process next Node of current level
        Node *Node = q.front();

        /* if Node is a leaf, update sum at the level */
        if (isLeaf(Node))
        {
            leafFound = true;
            levelSum += Node->data;
        }
        q.pop();

        // Add children of Node
        if (Node->left != NULL)
            q.push(Node->left);
        if (Node->right != NULL)
            q.push(Node->right);
        NodeCount--;
    }

    // If we found at least one leaf, we multiply
}
```

```
// result with level sum.
if (leafFound)
    mul *= levelSum;
}

return mul; // Return result
}

// Utility function to create a new tree Node
Node* newNode(int data)
{
    Node *temp = new Node;
    temp->data = data;
    temp->left = temp->right = NULL;
    return temp;
}

// Driver program to test above functions
int main()
{
    Node *root = newNode(2);
    root->left = newNode(7);
    root->right = newNode(5);
    root->left->right = newNode(6);
    root->left->left = newNode(8);
    root->left->right->left = newNode(1);
    root->left->right->right = newNode(11);
    root->right->right = newNode(9);
    root->right->right->left = newNode(4);
    root->right->right->right = newNode(10);

    cout << "Final product value = "
        << sumAndMultiplyLevelData(root) << endl;

    return 0;
}
```

### Java

```
/* Iterative Java program to find sum of data of all leaves
   of a binary tree on same level and then multiply sums
   obtained of all levels. */

/* importing the necessary class */
import java.util.LinkedList;
import java.util.Queue;
import java.util.Stack;
```

```
/* Class containing left and right child of current
node and key value*/
class Node {

    int data;
    Node left, right;

    public Node(int item) {
        data = item;
        left = right = null;
    }
}

class BinaryTree {

    Node root;

    // helper function to check if a Node is leaf of tree
    boolean isLeaf(Node node)
    {
        return ((node.left == null) && (node.right == null));
    }

    /* Calculate sum of all leaf Nodes at each level and returns
       multiplication of sums */
    int sumAndMultiplyLevelData()
    {
        return sumAndMultiplyLevelData(root);
    }

    int sumAndMultiplyLevelData(Node node)
    {
        // Tree is empty
        if (node == null) {
            return 0;
        }

        int mul = 1; /* To store result */

        // Create an empty queue for level order traversal
        LinkedList<Node> q = new LinkedList<Node>();

        // Enqueue Root and initialize height
        q.add(node);

        // Do level order traversal of tree
        while (true) {

            // NodeCount (queue size) indicates number of Nodes
```

```
// at current level.
int NodeCount = q.size();

// If there are no Nodes at current level, we are done
if (NodeCount == 0) {
    break;
}

// Initialize leaf sum for current level
int levelSum = 0;

// A boolean variable to indicate if found a leaf
// Node at current level or not
boolean leafFound = false;

// Dequeue all Nodes of current level and Enqueue all
// Nodes of next level
while (NodeCount > 0) {
    Node node1;
    node1 = q.poll();

    /* if Node is a leaf, update sum at the level */
    if (isLeaf(node1)) {
        leafFound = true;
        levelSum += node1.data;
    }

    // Add children of Node
    if (node1.left != null) {
        q.add(node1.left);
    }
    if (node1.right != null) {
        q.add(node1.right);
    }
    NodeCount--;
}

// If we found at least one leaf, we multiply
// result with level sum.
if (leafFound) {
    mul *= levelSum;
}
}

return mul; // Return result
}
```

```
public static void main(String args[]) {  
  
    /* creating a binary tree and entering  
       the nodes */  
    BinaryTree tree = new BinaryTree();  
    tree.root = new Node(2);  
    tree.root.left = new Node(7);  
    tree.root.right = new Node(5);  
    tree.root.left.left = new Node(8);  
    tree.root.left.right = new Node(6);  
    tree.root.left.right.left = new Node(1);  
    tree.root.left.right.right = new Node(11);  
    tree.root.right.right = new Node(9);  
    tree.root.right.right.left = new Node(4);  
    tree.root.right.right.right = new Node(10);  
    System.out.println("The final product value : "  
                      + tree.sumAndMultiplyLevelData());  
}  
}  
  
// This code is contributed by Mayank Jaiswal
```

Output:

Final product value = 208

This article is contributed by [Mohammed Raqeeb](#). Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## Source

<https://www.geeksforgeeks.org/find-multiplication-of-sums-of-data-of-all-leaves-at-same-levels/>

## Chapter 174

# Find n-th node in Postorder traversal of a Binary Tree

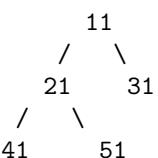
Find n-th node in Postorder traversal of a Binary Tree - GeeksforGeeks

Given a Binary tree and a number N, write a program to find the N-th node in the Postorder traversal of the given Binary tree.

**Prerequisite:** [Tree Traversal](#)

**Examples:**

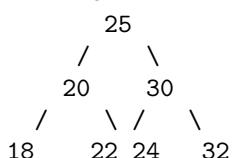
Input : N = 4



Output : 31

Explanation: Postorder Traversal of given Binary Tree is 41 51 21 31 11, so 4th node will be 31.

Input : N = 5



Output : 32

The idea to solve this problem is to do [postorder traversal](#) of the given binary tree and keep track of the count of nodes visited while traversing the tree and print the current node when the count becomes equal to N.

Below is the implementation of the above approach:

```
// C++ program to find n-th node of
// Postorder Traversal of Binary Tree
#include <bits/stdc++.h>
using namespace std;

// node of tree
struct Node {
    int data;
    Node *left, *right;
};

// function to create a new node
struct Node* createNode(int item)
{
    Node* temp = new Node;
    temp->data = item;
    temp->left = NULL;
    temp->right = NULL;

    return temp;
}

// function to find the N-th node in the postorder
// traversal of a given binary tree
void NthPostordernode(struct Node* root, int N)
{
    static int flag = 0;

    if (root == NULL)
        return;

    if (flag <= N) {

        // left recursion
        NthPostordernode(root->left, N);

        // right recursion
        NthPostordernode(root->right, N);

        flag++;
    }

    // prints the n-th node of preorder traversal
    if (flag == N)
        cout << root->data;
}
```

```
// driver code
int main()
{
    struct Node* root = createNode(25);
    root->left = createNode(20);
    root->right = createNode(30);
    root->left->left = createNode(18);
    root->left->right = createNode(22);
    root->right->left = createNode(24);
    root->right->right = createNode(32);

    int N = 6;

    // prints n-th node found
    NthPostordernode(root, N);

    return 0;
}
```

**Output:**

30

**Time Complexity:** O(n), where n is the number of nodes in the given binary tree.  
**Auxiliary Space:** O(1)

**Source**

<https://www.geeksforgeeks.org/find-n-th-node-in-postorder-traversal-of-a-binary-tree/>

## Chapter 175

# Find n-th node in Preorder traversal of a Binary Tree

Find n-th node in Preorder traversal of a Binary Tree - GeeksforGeeks

Given a Binary tree and a number N, write a program to find the N-th node in the Preorder traversal of the given Binary tree.

**Prerequisite:** [Tree Traversal](#)

**Examples:**

Input:      N = 4  
              11  
              / \  
            21  31  
          / \  
        41  51

Output: 51

Explanation: Preorder Traversal of given Binary Tree is 11 21 41 51 31, so 4th node will be 51.

Input:      N = 5  
              25  
              / \  
            20  30  
          / \ / \  
        18  22 24  32

Output: 30

The idea to solve this problem is to do [preorder traversal](#) of the given binary tree and keep track of the count of nodes visited while traversing the tree and print the current node when the count becomes equal to N.

```
// C++ program to find n-th node of
// Preorder Traversal of Binary Tree
#include <bits/stdc++.h>
using namespace std;

// Tree node
struct Node {
    int data;
    Node *left, *right;
};

// function to create new node
struct Node* createNode(int item)
{
    Node* temp = new Node;
    temp->data = item;
    temp->left = NULL;
    temp->right = NULL;

    return temp;
}

// function to find the N-th node in the preorder
// traversal of a given binary tree
void NthPreordernode(struct Node* root, int N)
{
    static int flag = 0;

    if (root == NULL)
        return;

    if (flag <= N) {
        flag++;

        // prints the n-th node of preorder traversal
        if (flag == N)
            cout << root->data;

        // left recursion
        NthPreordernode(root->left, N);

        // right recursion
        NthPreordernode(root->right, N);
    }
}

// Driver code
int main()
```

```
{  
    // construction of binary tree  
    struct Node* root = createNode(25);  
    root->left = createNode(20);  
    root->right = createNode(30);  
    root->left->left = createNode(18);  
    root->left->right = createNode(22);  
    root->right->left = createNode(24);  
    root->right->right = createNode(32);  
  
    // nth node  
    int N = 6;  
  
    // prints n-th found found  
    NthPreordernode(root, N);  
  
    return 0;  
}
```

**Output:**

24

**Time Complexity:** O(n), where n is the number of nodes in the given binary tree.

**Auxiliary Space:** O(1)

**Source**

<https://www.geeksforgeeks.org/find-n-th-node-in-preorder-traversal-of-a-binary-tree/>

## Chapter 176

# Find n-th node of inorder traversal

Find n-th node of inorder traversal - GeeksforGeeks

iven the binary tree and you have to find out the n-th node of [inorder traversal](#).

Examples:

Input : n = 4  
        10  
        / \  
      20  30  
      / \  
    40  50  
Output : 10  
Inorder Traversal is : 40 20 50 10 30

Input : n = 3  
        7  
        / \  
      2  3  
      / \  
    8  5  
Output : 8  
Inorder: 2 7 8 3 5  
3th node is 8

We do simple Inorder Traversal. While doing the traversal, we keep track of count of nodes visited so far. When count becomes n, we print the node.

```
// C program for nth nodes of inorder traversals
```

```
#include <stdio.h>
#include <stdlib.h>

/* A binary tree node has data, pointer to left child
and a pointer to right child */
struct Node {
    int data;
    struct Node* left;
    struct Node* right;
};

/* Helper function that allocates a new node with the
given data and NULL left and right pointers. */
struct Node* newNode(int data)
{
    struct Node* node =
        (struct Node*)malloc(sizeof(struct Node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;

    return (node);
}

/* Given a binary tree, print its nth nodes of inorder*/
void NthInorder(struct Node* node, int n)
{
    static int count = 0;
    if (node == NULL)
        return;

    if (count <= n) {

        /* first recur on left child */
        NthInorder(node->left, n);
        count++;

        // when count = n then print element
        if (count == n)
            printf("%d ", node->data);

        /* now recur on right child */
        NthInorder(node->right, n);
    }
}

/* Driver program to test above functions*/
int main()
```

```
{  
    struct Node* root = newNode(10);  
    root->left = newNode(20);  
    root->right = newNode(30);  
    root->left->left = newNode(40);  
    root->left->right = newNode(50);  
  
    int n = 4;  
  
    NthInorder(root, n);  
    return 0;  
}
```

Output:

1

Time Complexity: O(n)

## Source

<https://www.geeksforgeeks.org/find-n-th-node-inorder-traversal/>

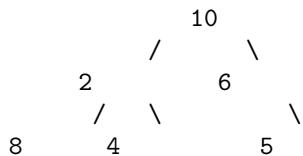
## Chapter 177

# Find next right node of a given key

Find next right node of a given key - GeeksforGeeks

Given a Binary tree and a key in the binary tree, find the node right to the given key. If there is no node on right side, then return NULL. Expected time complexity is O(n) where n is the number of nodes in the given binary tree.

For example, consider the following Binary Tree. Output for 2 is 6, output for 4 is 5. Output for 10, 6 and 5 is NULL.



**Solution:** The idea is to do [level order traversal](#) of given Binary Tree. When we find the given key, we just check if the next node in level order traversal is of same level, if yes, we return the next node, otherwise return NULL.

C++

```
/* Program to find next right of a given key */
#include <iostream>
#include <queue>
using namespace std;

// A Binary Tree Node
struct node
{
```

```
struct node *left, *right;
int key;
};

// Method to find next right of given key k, it returns NULL if k is
// not present in tree or k is the rightmost node of its level
node* nextRight(node *root, int k)
{
    // Base Case
    if (root == NULL)
        return 0;

    // Create an empty queue for level order traversal
    queue<node *> qn; // A queue to store node addresses
    queue<int> ql;    // Another queue to store node levels

    int level = 0; // Initialize level as 0

    // Enqueue Root and its level
    qn.push(root);
    ql.push(level);

    // A standard BFS loop
    while (qn.size())
    {
        // dequeue an node from qn and its level from ql
        node *node = qn.front();
        level = ql.front();
        qn.pop();
        ql.pop();

        // If the dequeued node has the given key k
        if (node->key == k)
        {
            // If there are no more items in queue or given node is
            // the rightmost node of its level, then return NULL
            if (ql.size() == 0 || ql.front() != level)
                return NULL;

            // Otherwise return next node from queue of nodes
            return qn.front();
        }

        // Standard BFS steps: enqueue children of this node
        if (node->left != NULL)
        {
            qn.push(node->left);
            ql.push(level+1);
        }
    }
}
```

```
        }
        if (node->right != NULL)
        {
            qn.push(node->right);
            ql.push(level+1);
        }
    }

    // We reach here if given key x doesn't exist in tree
    return NULL;
}

// Utility function to create a new tree node
node* newNode(int key)
{
    node *temp = new node;
    temp->key = key;
    temp->left = temp->right = NULL;
    return temp;
}

// A utility function to test above functions
void test(node *root, int k)
{
    node *nr = nextRight(root, k);
    if (nr != NULL)
        cout << "Next Right of " << k << " is " << nr->key << endl;
    else
        cout << "No next right node found for " << k << endl;
}

// Driver program to test above functions
int main()
{
    // Let us create binary tree given in the above example
    node *root = newNode(10);
    root->left = newNode(2);
    root->right = newNode(6);
    root->right->right = newNode(5);
    root->left->left = newNode(8);
    root->left->right = newNode(4);

    test(root, 10);
    test(root, 2);
    test(root, 6);
    test(root, 5);
    test(root, 8);
    test(root, 4);
```

```
    return 0;
}
```

**Java**

```
// Java program to find next right of a given key

import java.util.LinkedList;
import java.util.Queue;

// A binary tree node
class Node
{
    int data;
    Node left, right;

    Node(int item)
    {
        data = item;
        left = right = null;
    }
}

class BinaryTree
{
    Node root;

    // Method to find next right of given key k, it returns NULL if k is
    // not present in tree or k is the rightmost node of its level
    Node nextRight(Node first, int k)
    {
        // Base Case
        if (first == null)
            return null;

        // Create an empty queue for level order traversal
        // A queue to store node addresses
        Queue<Node> qn = new LinkedList<Node>();

        // Another queue to store node levels
        Queue<Integer> ql = new LinkedList<Integer>();

        int level = 0; // Initialize level as 0

        // Enqueue Root and its level
        qn.add(first);
        ql.add(level);
```

```
// A standard BFS loop
while (qn.size() != 0)
{
    // dequeue an node from qn and its level from ql
    Node node = qn.peek();
    level = ql.peek();
    qn.remove();
    ql.remove();

    // If the dequeued node has the given key k
    if (node.data == k)
    {
        // If there are no more items in queue or given node is
        // the rightmost node of its level, then return NULL
        if (ql.size() == 0 || ql.peek() != level)
            return null;

        // Otherwise return next node from queue of nodes
        return qn.peek();
    }

    // Standard BFS steps: enqueue children of this node
    if (node.left != null)
    {
        qn.add(node.left);
        ql.add(level + 1);
    }
    if (node.right != null)
    {
        qn.add(node.right);
        ql.add(level + 1);
    }
}

// We reach here if given key x doesn't exist in tree
return null;
}

// A utility function to test above functions
void test(Node node, int k)
{
    Node nr = nextRight(root, k);
    if (nr != null)
        System.out.println("Next Right of " + k + " is " + nr.data);
    else
        System.out.println("No next right node found for " + k);
}
```

```
// Driver program to test above functions
public static void main(String args[])
{
    BinaryTree tree = new BinaryTree();
    tree.root = new Node(10);
    tree.root.left = new Node(2);
    tree.root.right = new Node(6);
    tree.root.right.right = new Node(5);
    tree.root.left.left = new Node(8);
    tree.root.left.right = new Node(4);

    tree.test(tree.root, 10);
    tree.test(tree.root, 2);
    tree.test(tree.root, 6);
    tree.test(tree.root, 5);
    tree.test(tree.root, 8);
    tree.test(tree.root, 4);

}
}

// This code has been contributed by Mayank Jaiswal
```

### Python

```
# Python program to find next right node of given key

# A Binary Tree Node
class Node:

    # Constructor to create a new node
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None

    # Method to find next right of a given key k, it returns
    # None if k is not present in tree or k is the rightmost
    # node of its level
    def nextRight(root, k):

        # Base Case
        if root is None:
            return 0

        # Create an empty queue for level order traversal
        qn = [] # A queue to store node addresses
        q1 = [] # Another queue to store node levels
```

```
level = 0

# Enqueue root and its level
qn.append(root)
q1.append(level)

# Standard BFS loop
while(len(qn) > 0):

    # Dequeue a node from qn and its level from q1
    node = qn.pop(0)
    level = q1.pop(0)

    # If the dequeued node has the given key k
    if node.key == k :

        # If there are no more items in queue or given
        # node is the rightmost node of its level,
        # then return None
        if (len(q1) == 0 or q1[0] != level):
            return None

        # Otherwise return next node from queue of nodes
        return qn[0]

    # Standard BFS steps: enqueue children of this node
    if node.left is not None:
        qn.append(node.left)
        q1.append(level+1)

    if node.right is not None:
        qn.append(node.right)
        q1.append(level+1)

# We reach here if given key x doesn't exist in tree
return None

def test(root, k):
    nr = nextRight(root, k)
    if nr is not None:
        print "Next Right of " + str(k) + " is " + str(nr.key)
    else:
        print "No next right node found for " + str(k)

# Driver program to test above function
root = Node(10)
root.left = Node(2)
```

```
root.right = Node(6)
root.right.right = Node(5)
root.left.left = Node(8)
root.left.right = Node(4)

test(root, 10)
test(root, 2)
test(root, 6)
test(root, 5)
test(root, 8)
test(root, 4)

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

Output:

```
No next right node found for 10
Next Right of 2 is 6
No next right node found for 6
No next right node found for 5
Next Right of 8 is 4
Next Right of 4 is 5
```

**Time Complexity:** The above code is a simple BFS traversal code which visits every enqueue and dequeues a node at most once. Therefore, the time complexity is  $O(n)$  where  $n$  is the number of nodes in the given binary tree.

**Exercise:** Write a function to find left node of a given node. If there is no node on the left side, then return NULL.

## Source

<https://www.geeksforgeeks.org/find-next-right-node-of-a-given-key/>

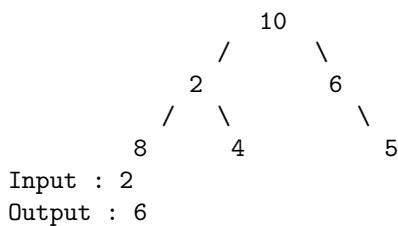
## Chapter 178

# Find next right node of a given key | Set 2

Find next right node of a given key | Set 2 - GeeksforGeeks

Given a Binary tree and a key in the binary tree, find the node right to the given key. If there is no node on right side, then return NULL. Expected time complexity is O(n) where n is the number of nodes in the given binary tree.

For example, consider the following Binary Tree. Output for 2 is 6, output for 4 is 5. Output for 10, 6 and 5 is NULL.



In our [previous](#) post we have discussed about a solution using [Level Order Traversal](#). In this post we will discuss about a solution based on Preorder traversal which takes constant auxiliary space.

The idea is to traverse the given tree using [preorder traversal](#) and search for the given key. Once we found the given key, we will mark the level number for this key. Now the next node we will find at the same level is the required node which is at the right of given key.

Below is the implementation of above idea:

```
/* Program to find next right of a given key
   using preorder traversal */
#include <iostream>
using namespace std;

// A Binary Tree Node
struct Node {
    struct Node *left, *right;
    int key;
};

// Utility function to create a new tree node
Node* newNode(int key)
{
    Node* temp = new Node;
    temp->key = key;
    temp->left = temp->right = NULL;
    return temp;
}

// Function to find next node for given node
// in same level in a binary tree by using
// pre-order traversal
Node* nextRightNode(Node* root, int k, int level,
                     int& value_level)
{
    // return null if tree is empty
    if (root == NULL)
        return NULL;

    // if desired node is found, set value_level
    // to current level
    if (root->key == k) {
        value_level = level;
        return NULL;
    }

    // if value_level is already set, then current
    // node is the next right node
    else if (value_level) {
        if (level == value_level)
            return root;
    }

    // recurse for left subtree by increasing level by 1
    Node* leftNode = nextRightNode(root->left, k,
                                   level + 1, value_level);
}
```

```
// if node is found in left subtree, return it
if (leftNode)
    return leftNode;

// recurse for right subtree by increasing level by 1
return nextRightNode(root->right, k, level + 1,
                     value_level);
}

// Function to find next node of given node in the
// same level in given binary tree
Node* nextRightNodeUtil(Node* root, int k)
{
    int value_level = 0;

    return nextRightNode(root, k, 1, value_level);
}

// A utility function to test above functions
void test(Node* root, int k)
{
    Node* nr = nextRightNodeUtil(root, k);
    if (nr != NULL)
        cout << "Next Right of " << k << " is "
            << nr->key << endl;
    else
        cout << "No next right node found for "
            << k << endl;
}

// Driver program to test above functions
int main()
{
    // Let us create binary tree given in the
    // above example
    Node* root = newNode(10);
    root->left = newNode(2);
    root->right = newNode(6);
    root->right->right = newNode(5);
    root->left->left = newNode(8);
    root->left->right = newNode(4);

    test(root, 10);
    test(root, 2);
    test(root, 6);
    test(root, 5);
    test(root, 8);
    test(root, 4);
```

```
    return 0;  
}
```

**Output:**

```
No next right node found for 10  
Next Right of 2 is 6  
No next right node found for 6  
No next right node found for 5  
Next Right of 8 is 4  
Next Right of 4 is 5
```

**Time Complexity:** O(n)

**Auxiliary Space:** O(1)

**Source**

<https://www.geeksforgeeks.org/find-next-right-node-given-key-set-2/>

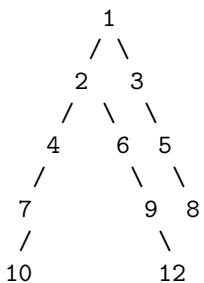
## Chapter 179

# Find right sibling of a binary tree with parent pointers

Find right sibling of a binary tree with parent pointers - GeeksforGeeks

Given a binary tree with parent pointers, find the right sibling of a given node(pointer to the node will be given), if it doesn't exist return null. Do it in O(1) space and O(n) time?

Examples:



Input : Given above tree with parent pointer and node 10

Output : 12

Idea is to find out first right child of nearest ancestor which is neither the current node nor parent of current node, keep track of level in those while going up. then, iterate through that node first left child, if left is not there then, right child and if level becomes 0, then, this is the next right sibling of the given node.

In above case if given node is 7, we will end up with 6 to find right child which doesn't have any child.

In this case we need to recursively call for right sibling with the current level, so that we can reach 8.

C++

```
// C program to print right sibling of a node
#include <stdio.h>
#include <stdlib.h>

// A Binary Tree Node
struct Node {
    int data;
    Node* left, *right, *parent;
};

// A utility function to create a new Binary
// Tree Node
Node* newNode(int item, Node* parent)
{
    Node* temp = new Node;
    temp->data = item;
    temp->left = temp->right = NULL;
    temp->parent = parent;
    return temp;
}

// Method to find right sibling
Node* findRightSibling(Node* node, int level)
{
    if (node == NULL || node->parent == NULL)
        return NULL;

    // GET Parent pointer whose right child is not
    // a parent or itself of this node. There might
    // be case when parent has no right child, but,
    // current node is left child of the parent
    // (second condition is for that).
    while (node->parent->right == node ||
           (node->parent->right == NULL &&
            node->parent->left == node)) {
        if (node->parent == NULL)
            return NULL;

        node = node->parent;
        level--;
    }

    // Move to the required child, where right sibling
    // can be present
    node = node->parent->right;

    // find right sibling in the given subtree(from current
```

```
// node), when level will be 0
while (level < 0) {

    // Iterate through subtree
    if (node->left != NULL)
        node = node->left;
    else if (node->right != NULL)
        node = node->right;
    else

        // if no child are there, we cannot have right
        // sibling in this path
        break;

    level++;
}

if (level == 0)
    return node;

// This is the case when we reach 9 node in the tree,
// where we need to again recursively find the right
// sibling
return findRightSibling(node, level);
}

// Driver Program to test above functions
int main()
{
    Node* root = newNode(1, NULL);
    root->left = newNode(2, root);
    root->right = newNode(3, root);
    root->left->left = newNode(4, root->left);
    root->left->right = newNode(6, root->left);
    root->left->left->left = newNode(7, root->left->left);
    root->left->left->left->left = newNode(10, root->left->left->left);
    root->left->right->right = newNode(9, root->left->right);
    root->right->right = newNode(5, root->right);
    root->right->right->right = newNode(8, root->right->right);
    root->right->right->right->right = newNode(12, root->right->right->right);

    // passing 10
    Node *res = findRightSibling(root->left->left->left->left, 0);
    if (res == NULL)
        printf("No right sibling");
    else
        printf("%d", res->data);
```

```
    return 0;
}
```

**Java**

```
// Java program to print right sibling of a node
public class Right_Sibling {

    // A Binary Tree Node
    static class Node {
        int data;
        Node left, right, parent;

        // Constructor
        public Node(int data, Node parent) {
            this.data = data;
            left = null;
            right = null;
            this.parent = parent;
        }
    };

    // Method to find right sibling
    static Node findRightSibling(Node node, int level)
    {
        if (node == null || node.parent == null)
            return null;

        // GET Parent pointer whose right child is not
        // a parent or itself of this node. There might
        // be case when parent has no right child, but,
        // current node is left child of the parent
        // (second condition is for that).
        while (node.parent.right == node ||
               (node.parent.right == null &&
                node.parent.left == node)) {
            if (node.parent == null)
                return null;

            node = node.parent;
            level--;
        }

        // Move to the required child, where right sibling
        // can be present
        node = node.parent.right;
```

```
// find right sibling in the given subtree(from current
// node), when level will be 0
while (level < 0) {

    // Iterate through subtree
    if (node.left != null)
        node = node.left;
    else if (node.right != null)
        node = node.right;
    else

        // if no child are there, we cannot have right
        // sibling in this path
        break;

    level++;
}

if (level == 0)
    return node;

// This is the case when we reach 9 node in the tree,
// where we need to again recursively find the right
// sibling
return findRightSibling(node, level);
}

// Driver Program to test above functions
public static void main(String args[])
{
    Node root = new Node(1, null);
    root.left = new Node(2, root);
    root.right = new Node(3, root);
    root.left.left = new Node(4, root.left);
    root.left.right = new Node(6, root.left);
    root.left.left.left = new Node(7, root.left.left);
    root.left.left.left.left = new Node(10, root.left.left.left);
    root.left.right.right = new Node(9, root.left.right);
    root.right.right = new Node(5, root.right);
    root.right.right.right = new Node(8, root.right.right);
    root.right.right.right.right = new Node(12, root.right.right.right);

    // passing 10
    System.out.println(findRightSibling(root.left.left.left, 0).data);
}
}

// This code is contributed by Sumit Ghosh
```

Output:

12

**Source**

<https://www.geeksforgeeks.org/find-right-sibling-binary-tree-parent-pointers/>

## Chapter 180

# Find root of the tree where children id sum for every node is given

Find root of the tree where children id sum for every node is given - GeeksforGeeks

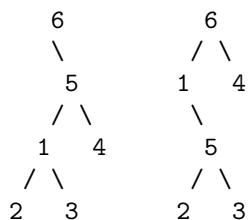
Consider a binary tree whose nodes have ids from 1 to n where n is number of nodes in the tree. The tree is given as a collection of n pairs, where every pair represents node id and sum of children ids.

Examples:

```
Input : 1 5  
        2 0  
        3 0  
        4 0  
        5 5  
        6 5
```

Output: 6

Explanation: In this case, two trees can be made as follows and 6 is the root node.



```
Input : 4 0
```

Output: 4

Explanation: Clearly 4 does not have any children and is the only node i.e., the root node.

At first sight this question appears to be a typical question of tree data structure but it can be solved as follows.

Every node id appears in children sum except root. So if we do sum of all ids and subtract it from sum of all children sums, we get root.

```
// Find root of tree where children
// sum for every node id is given.
#include<bits/stdc++.h>
using namespace std;

int findRoot(pair<int, int> arr[], int n)
{
    // Every node appears once as an id, and
    // every node except for the root appears
    // once in a sum. So if we subtract all
    // the sums from all the ids, we're left
    // with the root id.
    int root = 0;
    for (int i=0; i<n; i++)
        root += (arr[i].first - arr[i].second);

    return root;
}

// Driver code
int main()
{
    pair<int, int> arr[] = {{1, 5}, {2, 0},
                           {3, 0}, {4, 0}, {5, 5}, {6, 5}};
    int n = sizeof(arr)/sizeof(arr[0]);
    printf("%d\n", findRoot(arr, n));
    return 0;
}
```

Output:

6

## Source

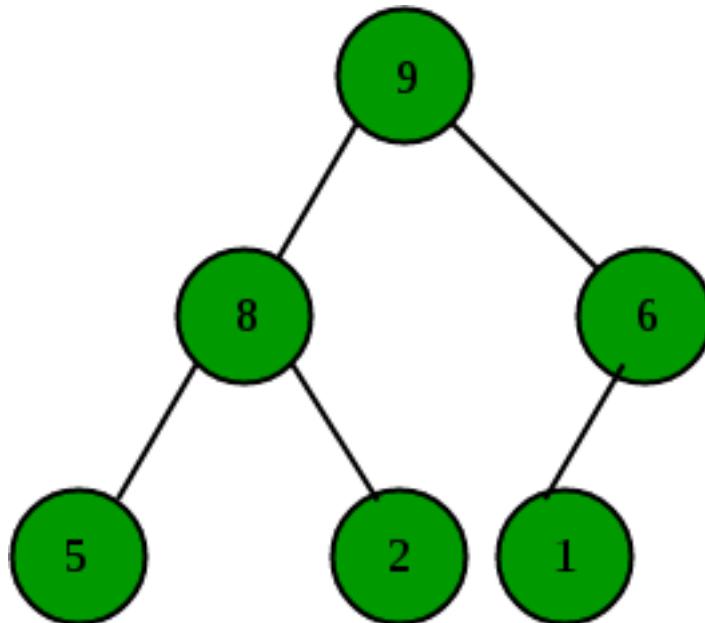
<https://www.geeksforgeeks.org/find-root-tree-children-id-sum-every-node-given/>

## Chapter 181

# Find sum of all left leaves in a given Binary Tree

Find sum of all left leaves in a given Binary Tree - GeeksforGeeks

Given a Binary Tree, find sum of all left leaves in it. For example, sum of all left leaves in below Binary Tree is  $5+1=6$ .



The idea is to traverse the tree, starting from root. For every node, check if its left subtree is a leaf. If it is, then add it to the result.

Following is the implementation of above idea.

C++

```
// A C++ program to find sum of all left leaves
#include <iostream>
using namespace std;

/* A binary tree Node has key, pointer to left and right
   children */
struct Node
{
    int key;
    struct Node* left, *right;
};

/* Helper function that allocates a new node with the
   given data and NULL left and right pointer. */
Node *newNode(char k)
{
    Node *node = new Node;
    node->key = k;
    node->right = node->left = NULL;
    return node;
}

// A utility function to check if a given node is leaf or not
bool isLeaf(Node *node)
{
    if (node == NULL)
        return false;
    if (node->left == NULL && node->right == NULL)
        return true;
    return false;
}

// This function returns sum of all left leaves in a given
// binary tree
int leftLeavesSum(Node *root)
{
    // Initialize result
    int res = 0;

    // Update result if root is not NULL
    if (root != NULL)
    {
        // If left of root is NULL, then add key of
        // left child
        if (isLeaf(root->left))
            res += root->left->key;
        else // Else recur for left child of root
            res += leftLeavesSum(root->left);
    }
}
```

```
// Recur for right child of root and update res
res += leftLeavesSum(root->right);
}

// return result
return res;
}

/* Driver program to test above functions*/
int main()
{
    // Let us a construct the Binary Tree
    struct Node *root      = newNode(20);
    root->left            = newNode(9);
    root->right           = newNode(49);
    root->right->left    = newNode(23);
    root->right->right   = newNode(52);
    root->right->right->left = newNode(50);
    root->left->left     = newNode(5);
    root->left->right    = newNode(12);
    root->left->right->right = newNode(12);
    cout << "Sum of left leaves is "
         << leftLeavesSum(root);
    return 0;
}
```

### Java

```
// Java program to find sum of all left leaves
class Node
{
    int data;
    Node left, right;

    Node(int item)
    {
        data = item;
        left = right = null;
    }
}

class BinaryTree
{
    Node root;

    // A utility function to check if a given node is leaf or not
    boolean isLeaf(Node node)
```

```
{  
    if (node == null)  
        return false;  
    if (node.left == null && node.right == null)  
        return true;  
    return false;  
}  
  
// This function returns sum of all left leaves in a given  
// binary tree  
int leftLeavesSum(Node node)  
{  
    // Initialize result  
    int res = 0;  
  
    // Update result if root is not NULL  
    if (node != null)  
    {  
        // If left of root is NULL, then add key of  
        // left child  
        if (isLeaf(node.left))  
            res += node.left.data;  
        else // Else recur for left child of root  
            res += leftLeavesSum(node.left);  
  
        // Recur for right child of root and update res  
        res += leftLeavesSum(node.right);  
    }  
  
    // return result  
    return res;  
}  
  
// Driver program  
public static void main(String args[])  
{  
    BinaryTree tree = new BinaryTree();  
    tree.root = new Node(20);  
    tree.root.left = new Node(9);  
    tree.root.right = new Node(49);  
    tree.root.left.right = new Node(12);  
    tree.root.left.left = new Node(5);  
    tree.root.right.left = new Node(23);  
    tree.root.right.right = new Node(52);  
    tree.root.left.right.right = new Node(12);  
    tree.root.right.right.left = new Node(50);  
  
    System.out.println("The sum of leaves is " +
```

```
        tree.leftLeavesSum(tree.root));  
    }  
}  
  
// This code is contributed by Mayank Jaiswal
```

### Python

```
# Python program to find sum of all left leaves  
  
# A Binary tree node  
class Node:  
    # Constructor to create a new Node  
    def __init__(self, key):  
        self.key = key  
        self.left = None  
        self.right = None  
  
    # A utility function to check if a given node is leaf or not  
def isLeaf(node):  
    if node is None:  
        return False  
    if node.left is None and node.right is None:  
        return True  
    return False  
  
# This function return sum of all left leaves in a  
# given binary tree  
def leftLeavesSum(root):  
  
    # Initialize result  
    res = 0  
  
    # Update result if root is not None  
    if root is not None:  
  
        # If left of root is None, then add key of  
        # left child  
        if isLeaf(root.left):  
            res += root.left.key  
        else:  
            # Else recur for left child of root  
            res += leftLeavesSum(root.left)  
  
        # Recur for right child of root and update res  
        res += leftLeavesSum(root.right)  
    return res
```

```
# Driver program to test above function

# Let us construct the Binary Tree shown in the above function
root = Node(20)
root.left = Node(9)
root.right = Node(49)
root.right.left = Node(23)
root.right.right = Node(52)
root.right.right.left = Node(50)
root.left.left = Node(5)
root.left.right = Node(12)
root.left.right.right = Node(12)
print "Sum of left leaves is", leftLeavesSum(root)

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

Output:

```
Sum of left leaves is 78
```

Time complexity of the above solution is  $O(n)$  where  $n$  is number of nodes in Binary Tree.

Following is Another Method to solve the above problem. This solution passes in a sum variable as an accumulator. When a left leaf is encountered, the leaf's data is added to sum. Time complexity of this method is also  $O(n)$ . Thanks to Xin Tong (geeksforgeeks userid trent.tong) for suggesting this method.

C++

```
// A C++ program to find sum of all left leaves
#include <iostream>
using namespace std;

/* A binary tree Node has key, pointer to left and right
   children */
struct Node
{
    int key;
    struct Node* left, *right;
};

/* Helper function that allocates a new node with the
   given data and NULL left and right pointer. */
Node *newNode(char k)
{
    Node *node = new Node;
    node->key = k;
    node->right = node->left = NULL;
```

```
    return node;
}

/* Pass in a sum variable as an accumulator */
void leftLeavesSumRec(Node *root, bool isleft, int *sum)
{
    if (!root) return;

    // Check whether this node is a leaf node and is left.
    if (!root->left && !root->right && isleft)
        *sum += root->key;

    // Pass 1 for left and 0 for right
    leftLeavesSumRec(root->left, 1, sum);
    leftLeavesSumRec(root->right, 0, sum);
}

// A wrapper over above recursive function
int leftLeavesSum(Node *root)
{
    int sum = 0; //Initialize result

    // use the above recursive function to evaluate sum
    leftLeavesSumRec(root, 0, &sum);

    return sum;
}

/* Driver program to test above functions*/
int main()
{
    // Let us construct the Binary Tree shown in the
    // above figure
    int sum = 0;
    struct Node *root      = newNode(20);
    root->left            = newNode(9);
    root->right           = newNode(49);
    root->right->left    = newNode(23);
    root->right->right   = newNode(52);
    root->right->right->left = newNode(50);
    root->left->left     = newNode(5);
    root->left->right    = newNode(12);
    root->left->right->right = newNode(12);

    cout << "Sum of left leaves is " << leftLeavesSum(root) << endl;
    return 0;
}
```

**Java**

```
// Java program to find sum of all left leaves
class Node
{
    int data;
    Node left, right;

    Node(int item) {
        data = item;
        left = right = null;
    }
}

// Passing sum as accumulator and implementing pass by reference
// of sum variable
class Sum
{
    int sum = 0;
}

class BinaryTree
{
    Node root;

    /* Pass in a sum variable as an accumulator */
    void leftLeavesSumRec(Node node, boolean isleft, Sum summ)
    {
        if (node == null)
            return;

        // Check whether this node is a leaf node and is left.
        if (node.left == null && node.right == null && isleft)
            summ.sum = summ.sum + node.data;

        // Pass true for left and false for right
        leftLeavesSumRec(node.left, true, summ);
        leftLeavesSumRec(node.right, false, summ);
    }

    // A wrapper over above recursive function
    int leftLeavesSum(Node node)
    {
        Sum suum = new Sum();

        // use the above recursive function to evaluate sum
        leftLeavesSumRec(node, false, suum);
    }
}
```

```
        return suum.sum;
    }

// Driver program
public static void main(String args[])
{
    BinaryTree tree = new BinaryTree();
    tree.root = new Node(20);
    tree.root.left = new Node(9);
    tree.root.right = new Node(49);
    tree.root.left.right = new Node(12);
    tree.root.left.left = new Node(5);
    tree.root.right.left = new Node(23);
    tree.root.right.right = new Node(52);
    tree.root.left.right.right = new Node(12);
    tree.root.right.right.left = new Node(50);

    System.out.println("The sum of leaves is " +
                       tree.leftLeavesSum(tree.root));
}
}

// This code is contributed by Mayank Jaiswal
```

### Python

```
# Python program to find sum of all left leaves

# A binary tree node
class Node:

    # A constructor to create a new Node
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None

def leftLeavesSumRec(root, isLeft, summ):
    if root is None:
        return

    # Check whether this node is a leaf node and is left
    if root.left is None and root.right is None and isLeft == True:
        summ[0] += root.key

    # Pass 1 for left and 0 for right
    leftLeavesSumRec(root.left, 1, summ)
    leftLeavesSumRec(root.right, 0, summ)
```

```
# A wrapper over above recursive function
def leftLeavesSum(root):
    summ = [0] # initialize result

    # Use the above recursive function to evaluate sum
    leftLeavesSumRec(root, 0, summ)

    return summ[0]

# Driver program to test above function

# Let us construct the Binary Tree shown in the
# above figure
root = Node(20);
root.left= Node(9);
root.right = Node(49);
root.right.left = Node(23);
root.right.right= Node(52);
root.right.right.left = Node(50);
root.left.left = Node(5);
root.left.right = Node(12);
root.left.right.right = Node(12);

print "Sum of left leaves is", leftLeavesSum(root)

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

Output:

```
Sum of left leaves is 78
```

#### **Iterative Approach :**

This is the Iterative Way to find the sum of the left leaves.

Idea is to perform Depth-First Traversal on the tree (either Inorder, Preorder or Postorder) using a stack and checking if the Left Child is a Leaf node. If it is, then add the nodes value to the sum variable

```
# Python program to find sum of all left leaves

# A binary tree node
class Node:

    # A constructor to create a new Node
    def __init__(self, key):
```

```
    self.key = key
    self.left = None
    self.right = None

# Return the sum of left leaf nodes
def sumOfLeftLeaves(root):
    if(root is None):
        return

    # Using a stack for Depth-First Traversal of the tree
    stack = []
    stack.append(root)

    # sum holds the sum of all the left leaves
    sum = 0

    while len(stack) > 0:
        currentNode = stack.pop()

        if currentNode.left is not None:
            stack.append(currentNode.left)

            # Check if currentNode's left child is a leaf node
            if currentNode.left.left is None and currentNode.left.right is None:

                # if currentNode is a leaf, add its data to the sum
                sum = sum + currentNode.left.data

        if currentNode.right is not None:
            stack.append(currentNode.right)

    return sum

# Driver Code
root = Tree(20);
root.left= Tree(9);
root.right = Tree(49);
root.right.left = Tree(23);
root.right.right= Tree(52);
root.right.right.left = Tree(50);
root.left.left = Tree(5);
root.left.right = Tree(12);
root.left.right.right = Tree(12);

print('Sum of left leaves is {}'.format(sumOfLeftLeaves(root)))
```

Output:

Sum of left leaves is 78

Thanks to [Shubham Tambere](#) for suggesting this approach.

This article is contributed by **Manish**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## Source

<https://www.geeksforgeeks.org/find-sum-left-leaves-given-binary-tree/>

## Chapter 182

# Find sum of all nodes of the given perfect binary tree

Find sum of all nodes of the given perfect binary tree - GeeksforGeeks

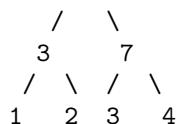
Given a positive integer L which represents the number of levels in a perfect binary tree. Given that the leaf nodes in this perfect binary tree are numbered starting from 1 to n, where n is the number of leaf nodes. And the parent node is the sum of the two child nodes. Our task is to write a program to print the sum of all of the nodes of this perfect binary tree.

**Examples:**

Input : L = 3

Output : 30

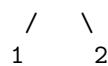
Explanation : Tree will be - 10



Input : L = 2

Output : 6

Explanation : Tree will be - 3



**Naive Approach:** The simplest solution is to first generate the value of all of the nodes of the perfect binary tree and then calculate the sum of all of the nodes. We can first generate all of the leaf nodes and then proceed in the bottom-up fashion to generate rest of the nodes. We know that in a perfect binary tree, the number of leaf nodes can be given by  $2^{L-1}$ , where

L is the number of levels. The number of nodes in a perfect binary tree as we move upward from the bottom will get decreased by half.

Below is the implementation of above idea:

C++

```
#include <bits/stdc++.h>
using namespace std;

// function to find sum of all of the nodes
// of given perfect binary tree
int sumNodes(int l)
{
    // no of leaf nodes
    int leafNodeCount = pow(2, l - 1);

    // list of vector to store nodes of
    // all of the levels
    vector<int> vec[l];

    // store the nodes of last level
    // i.e., the leaf nodes
    for (int i = 1; i <= leafNodeCount; i++)
        vec[l - 1].push_back(i);

    // store nodes of rest of the level
    // by moving in bottom-up manner
    for (int i = l - 2; i >= 0; i--) {
        int k = 0;

        // loop to calculate values of parent nodes
        // from the children nodes of lower level
        while (k < vec[i + 1].size() - 1) {

            // store the value of parent node as
            // sum of children nodes
            vec[i].push_back(vec[i + 1][k] +
                            vec[i + 1][k + 1]);
            k += 2;
        }
    }

    int sum = 0;

    // traverse the list of vector
    // and calculate the sum
    for (int i = 0; i < l; i++) {
```

```
        for (int j = 0; j < vec[i].size(); j++)
            sum += vec[i][j];
    }

    return sum;
}

// Driver Code
int main()
{
    int l = 3;

    cout << sumNodes(l);

    return 0;
}
```

**Java**

```
// Java program to implement
// the above approach
import java.util.*;
class GFG
{

    // function to find sum of
    // all of the nodes of given
    // perfect binary tree
    static int sumNodes(int l)
    {
        // no of leaf nodes
        int leafNodeCount = (int) Math.pow(2, l - 1);

        // list of vector to store
        // nodes of all of the levels
        Vector> vec = new Vector>();

        // initialize
        for (int i = 1; i <= l; i++) vec.add(new Vector());

        // store the nodes of last level
        // i.e., the leaf nodes
        for (int i = 1;
             i <= leafNodeCount; i++) vec.get(l - 1).add(i); // store nodes of rest of // the level by
        moving in // bottom-up manner for (int i = l - 2; i >= 0; i--)
        {
            int k = 0;

            // loop to calculate values
            // of parent nodes from the
            // children nodes of lower level
```

```
while (k < vec.get(i + 1).size() - 1) { // store the value of parent // node as sum of children
    nodes vec.get(i).add(vec.get(i + 1).get(k) + vec.get(i + 1).get(k + 1));
    k += 2;
}
int sum = 0; // traverse the list of vector // and calculate the sum for (int i = 0; i < l; i++) {
for (int j = 0; j < vec.get(i).size(); j++) {
    sum += vec.get(i).get(j);
}
return sum;
} // Driver Code
public static void main(String args[]) {
    int l = 3;
    System.out.println(sumNodes(l));
}
} // This code is contributed // by Arnab Kundu [tabbyending]
```

**Output:**

30

**Time Complexity:**  $O(n)$ , where  $n$  is the total number of nodes in the perfect binary tree.

**Efficient Approach:** An efficient approach is to observe that we only need to find the sum of all of the nodes. We can easily get the sum of all nodes at the last level using the formula of sum of first  $n$  natural numbers. Also, it can be seen that, as it is a perfect binary tree and parent nodes will be the sum of children nodes so the sum of nodes at all of the levels will be same. Therefore, we just need to find the sum of nodes at last level and multiply it by the total number of levels.

Below is the implementation of above idea:

**C++**

```
#include <bits/stdc++.h>
using namespace std;

// function to find sum of all of the nodes
// of given perfect binary tree
int sumNodes(int l)
{
    // no of leaf nodes
    int leafNodeCount = pow(2, l - 1);

    int sumLastLevel = 0;

    // sum of nodes at last level
    sumLastLevel = (leafNodeCount * (leafNodeCount + 1)) / 2;

    // sum of all nodes
    int sum = sumLastLevel * l;

    return sum;
}

// Driver Code
int main()
{
    int l = 3;
```

```
    cout << sumNodes(l);
    return 0;
}
```

**Java**

```
// Java code to find sum of all nodes
// of the given perfect binary tree
import java.io.*;
import java.lang.Math;

class GFG {

    // function to find sum of
    // all of the nodes of given
    // perfect binary tree
    static double sumNodes(int l)
    {

        // no of leaf nodes
        double leafNodeCount = Math.pow(2, l - 1);

        double sumLastLevel = 0;

        // sum of nodes at last level
        sumLastLevel = (leafNodeCount *
                        (leafNodeCount + 1)) / 2;

        // sum of all nodes
        double sum = sumLastLevel * l;

        return sum;
    }

    // Driver Code
    public static void main (String[] args) {

        int l = 3;
        System.out.println(sumNodes(l));
    }
}

// This code is contributed by
// Anuj_{AJ_67}
```

**Python3**

```
# function to find sum of all of the nodes
```

```
# of given perfect binary tree
import math

def sumNodes(l):

    # no of leaf nodes
    leafNodeCount = math.pow(2, l - 1);

    sumLastLevel = 0;

    # sum of nodes at last level
    sumLastLevel = ((leafNodeCount *
                      (leafNodeCount + 1)) / 2);

    # sum of all nodes
    sum = sumLastLevel * l;

    return int(sum);

# Driver Code
l = 3;
print (sumNodes(l));

# This code is contributed by manishshaw
```

C#

```
// C# code to find sum of all nodes
// of the given perfect binary tree
using System;
using System.Collections.Generic;

class GFG {

    // function to find sum of
    // all of the nodes of given
    // perfect binary tree
    static double sumNodes(int l)
    {

        // no of leaf nodes
        double leafNodeCount = Math.Pow(2, l - 1);

        double sumLastLevel = 0;

        // sum of nodes at last level
        sumLastLevel = (leafNodeCount *
                        (leafNodeCount + 1)) / 2;
```

```
// sum of all nodes
double sum = sumLastLevel * l;

return sum;
}

// Driver Code
public static void Main()
{
    int l = 3;
    Console.WriteLine(sumNodes(l));
}
}

// This code is contributed by
// Manish Shaw (manishshaw1)
```

### PHP

```
<?php
// PHP code to find sum of all nodes
// of the given perfect binary tree

// function to find sum of
// all of the nodes of given
// perfect binary tree
function sumNodes($l)
{

    // no of leaf nodes
    $leafNodeCount = ($l - 1) *
        ($l - 1);

    $sumLastLevel = 0;

    // sum of nodes at last level
    $sumLastLevel = ($leafNodeCount *
        ($leafNodeCount + 1)) / 2;

    // sum of all nodes
    $sum = $sumLastLevel * $l;

    return $sum;
}

// Driver Code
$l = 3;
```

```
echo (sumNodes($1));  
  
// This code is contributed by  
// Manish Shaw (manishshaw1)  
?>
```

Output:

30

**Time Complexity:** O(1)

**Improved By :** [manishshaw1](#), [vt\\_m](#), [andrew1234](#)

## Source

<https://www.geeksforgeeks.org/find-sum-nodes-given-perfect-binary-tree/>

## Chapter 183

# Find sum of all right leaves in a given Binary Tree

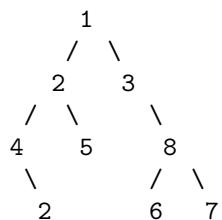
Find sum of all right leaves in a given Binary Tree - GeeksforGeeks

Given a Binary Tree, find sum of all right leaves in it.

Similar article : [Find sum of all left leaves in a given Binary Tree](#)

**Example :**

**Input :**



**Output :**

**sum = 2 + 5 + 7 = 14**

The idea is to traverse the tree starting from the root and check if the node is the leaf node or not. If the node is the right leaf than add data of right leaf to sum variable.

Following is the implementation for the same.

**C++**

```
// CPP program to find total sum
// of right leaf nodes
#include <bits/stdc++.h>
```

```
using namespace std;

// struct node of binary tree
struct Node{
    int data;
    Node *left, *right;
};

// return new node
Node *addNode(int data){
    Node *temp = new Node();
    temp->data = data;
    temp->left = temp->right = NULL;
}

// utility function to calculate sum
// of right leaf nodes
void rightLeafSum(Node *root, int *sum){
    if(!root)
        return;

    // check if the right child of root
    // is leaf node
    if(root->right)
        if(!root->right->left &&
           !root->right->right)
            *sum += root->right->data;

    rightLeafSum(root->left, sum);
    rightLeafSum(root->right, sum);
}

// driver program
int main(){

    //construct binary tree
    Node *root = addNode(1);
    root->left = addNode(2);
    root->left->left = addNode(4);
    root->left->right = addNode(5);
    root->left->left->right = addNode(2);
    root->right = addNode(3);
    root->right->right = addNode(8);
    root->right->right->left = addNode(6);
    root->right->right->right = addNode(7);

    // variable to store sum of right
    // leaves
}
```

```
    int sum = 0;
    rightLeafSum(root, &sum);
    cout << sum << endl;
    return 0;
}
```

**Java**

```
// Java program to find total
// sum of right leaf nodes
class GFG
{
    // sum
    static int sum = 0;

    // node of binary tree
    static class Node
    {
        int data;
        Node left, right;
    };

    // return new node
    static Node addNode(int data)
    {
        Node temp = new Node();
        temp.data = data;
        temp.left = temp.right = null;
        return temp;
    }

    // utility function to calculate
    // sum of right leaf nodes
    static void rightLeafSum(Node root)
    {
        if(root == null)
            return;

        // check if the right child
        // of root is leaf node
        if(root.right != null)
            if(root.right.left == null &&
               root.right.right == null)
                sum += root.right.data;

        rightLeafSum(root.left);
        rightLeafSum(root.right);
    }
}
```

```
}  
  
// Driver Code  
public static void main(String args[])  
{  
  
    //construct binary tree  
    Node root = addNode(1);  
    root.left = addNode(2);  
    root.left.left = addNode(4);  
    root.left.right = addNode(5);  
    root.left.left.right = addNode(2);  
    root.right = addNode(3);  
    root.right.right = addNode(8);  
    root.right.right.left = addNode(6);  
    root.right.right.right = addNode(7);  
  
    // variable to store sum  
    // of right leaves  
    sum = 0;  
    rightLeafSum(root);  
    System.out.println( sum );  
}  
}  
  
// This code is contributed by Arnab Kundu
```

**Output:**

14

Improved By : [andrew1234](#)

**Source**

<https://www.geeksforgeeks.org/find-sum-right-leaves-given-binary-tree/>

## Chapter 184

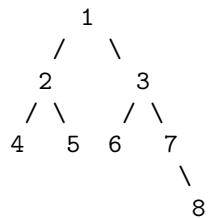
# Find the Deepest Node in a Binary Tree

Find the Deepest Node in a Binary Tree - GeeksforGeeks

Given a binary tree, find the deepest node in it.

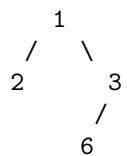
Examples:

Input : Root of below tree



Output : 8

Input : Root of below tree



Output : 6

**Method 1 :** The idea is to do Inorder traversal of given binary tree. While doing Inorder traversal, we pass level of current node also. We keep track of maximum level seen so far and value of deepest node seen so far.

```
// A C++ program to find value of the deepest node
```

```
// in a given binary tree
#include <bits/stdc++.h>
using namespace std;

// A tree node
struct Node
{
    int data;
    struct Node *left, *right;
};

// Utility function to create a new node
Node *newNode(int data)
{
    Node *temp = new Node;
    temp->data = data;
    temp->left = temp->right = NULL;
    return temp;
}

// maxLevel : keeps track of maximum level seen so far.
// res : Value of deepest node so far.
// level : Level of root
void find(Node *root, int level, int &maxLevel, int &res)
{
    if (root != NULL)
    {
        find(root->left, ++level, maxLevel, res);

        // Update level and resue
        if (level > maxLevel)
        {
            res = root->data;
            maxLevel = level;
        }

        find(root->right, level, maxLevel, res);
    }
}

// Returns value of deepest node
int deepestNode(Node *root)
{
    // Initialze result and max level
    int res = -1;
    int maxLevel = -1;

    // Updates value "res" and "maxLevel"
```

```
// Note that res and maxLen are passed
// by reference.
find(root, 0, maxLevel, res);
return res;
}

// Driver program
int main()
{
    Node* root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->right->left = newNode(5);
    root->right->right = newNode(6);
    root->right->left->right = newNode(7);
    root->right->right->right = newNode(8);
    root->right->left->right->left = newNode(9);
    cout << deepestNode(root);
    return 0;
}
```

Output:

9

Time Complexity : O(n)

**Method 2 :** The idea here is to find the height of the given tree and then print the node at the bottom-most level.

```
// A C++ program to find value of the
// deepest node in a given binary tree
#include <bits/stdc++.h>
using namespace std;

// A tree node with constructor
class Node
{
public:
    int data;
    Node *left, *right;

    // constructor
    Node(int key)
    {
        data = key;
```

```
        left = NULL;
        right = NULL;
    }
};

// Utility function to find height
// of a tree, rooted at 'root'.
int height(Node* root)
{
    if(!root) return 0;

    int leftHt = height(root->left);
    int rightHt = height(root->right);

    return max(leftHt, rightHt) + 1;
}

// levels : current Level
// Utility function to print all
// nodes at a given level.
void deepestNode(Node* root, int levels)
{
    if(!root) return;

    if(levels == 1)
        cout << root->data;

    else if(levels > 1)
    {
        deepestNode(root->left, levels - 1);
        deepestNode(root->right, levels - 1);
    }
}

// Driver program
int main()
{
    Node* root = new Node(1);
    root->left = new Node(2);
    root->right = new Node(3);
    root->left->left = new Node(4);
    root->right->left = new Node(5);
    root->right->right = new Node(6);
    root->right->left->right = new Node(7);
    root->right->right->right = new Node(8);
    root->right->left->right->left = new Node(9);

    // Calculating height of tree
```

```
int levels = height(root);

// Printing the deepest node
deepestNode(root, levels);

return 0;
}
```

Output:

9

Time Complexity : O(n)

Thanks to **Parth Patekar** for suggesting above method.

**Improved By :** [Parth Patekar](#)

## Source

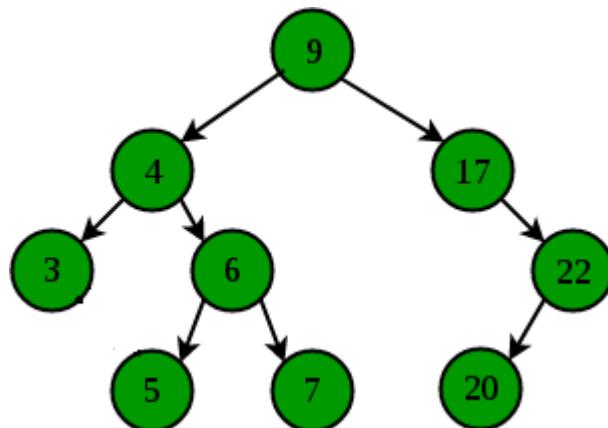
<https://www.geeksforgeeks.org/find-deepest-node-binary-tree/>

## Chapter 185

# Find the closest element in Binary Search Tree

Find the closest element in Binary Search Tree - GeeksforGeeks

Given a [binary search tree](#) and a target node K. The task is to find the node with minimum absolute difference with given target value K.



Examples:

```
// For above binary search tree
```

```
Input : k = 4
```

```
Output : 4
```

```
Input : k = 18
```

```
Output : 17
```

```
Input : k = 12
```

Output : 9

A **simple solution** for this problem is to store Inorder traversal of given binary search tree in an auxiliary array and then by taking absolute difference of each element find the node having minimum absolute difference with given target value K in linear time.

An **efficient solution** for this problem is to take advantage of characteristics of BST. Here is the algorithm to solve this problem :

- If target value K is present in given **BST**, then it's the node having minimum absolute difference.
- If target value K is less than the value of current node then move to the left child.
- If target value K is greater than the value of current node then move to the right child.

17

Time complexity :  $O(h)$  where h is height of given Binary Search Tree.

**Reference :**

<http://stackoverflow.com/questions/6209325/how-to-find-the-closest-element-to-a-given-key-value-in-a-binary-sea>

## Source

<https://www.geeksforgeeks.org/find-closest-element-binary-search-tree/>

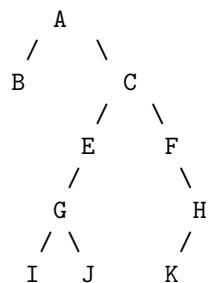
## Chapter 186

# Find the closest leaf in a Binary Tree

Find the closest leaf in a Binary Tree - GeeksforGeeks

Given a Binary Tree and a key 'k', find distance of the closest leaf from 'k'.

Examples:



Closest leaf to 'H' is 'K', so distance is 1 for 'H'

Closest leaf to 'C' is 'B', so distance is 2 for 'C'

Closest leaf to 'E' is either 'I' or 'J', so distance is 2 for 'E'

Closest leaf to 'B' is 'B' itself, so distance is 0 for 'B'

**We strongly recommend to minimize your browser and try this yourself first**

The main point to note here is that a closest key can either be a descendent of given key or can be reached through one of the ancestors.

The idea is to traverse the given tree in preorder and keep track of ancestors in an array. When we reach the given key, we evaluate distance of the closest leaf in subtree rooted with given key. We also traverse all ancestors one by one and find distance of the closest leaf in the subtree rooted with ancestor. We compare all distances and return minimum.

C++

```
// A C++ program to find the closer leaf of a given key in Binary Tree
#include <iostream>
#include <climits>
using namespace std;

/* A binary tree Node has key, pointer to left and right children */
struct Node
{
    char key;
    struct Node* left, *right;
};

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
Node *newNode(char k)
{
    Node *node = new Node;
    node->key = k;
    node->right = node->left = NULL;
    return node;
}

// A utility function to find minimum of x and y
int getMin(int x, int y)
{
    return (x < y)? x :y;
}

// A utility function to find distance of closest leaf of the tree
// rooted under given root
int closestDown(struct Node *root)
{
    // Base cases
    if (root == NULL)
        return INT_MAX;
    if (root->left == NULL && root->right == NULL)
        return 0;

    // Return minimum of left and right, plus one
    return 1 + getMin(closestDown(root->left), closestDown(root->right));
}

// Returns distance of the closest leaf to a given key 'k'. The array
// ancestors is used to keep track of ancestors of current node and
// 'index' is used to keep track of current index in 'ancestors[]'
int findClosestUtil(struct Node *root, char k, struct Node *ancestors[],
                    int index)
{
```

```

// Base case
if (root == NULL)
    return INT_MAX;

// If key found
if (root->key == k)
{
    // Find the cloest leaf under the subtree rooted with given key
    int res = closestDown(root);

    // Traverse all ancestors and update result if any parent node
    // gives smaller distance
    for (int i = index-1; i>=0; i--)
        res = getMin(res, index - i + closestDown(ancestors[i]));
    return res;
}

// If key node found, store current node and recur for left and
// right childrens
ancestors[index] = root;
return getMin(findClosestUtil(root->left, k, ancestors, index+1),
              findClosestUtil(root->right, k, ancestors, index+1));

}

// The main function that returns distance of the closest key to 'k'. It
// mainly uses recursive function findClosestUtil() to find the closes
// distance.
int findClosest(struct Node *root, char k)
{
    // Create an array to store ancestors
    // Assumption: Maximum height of tree is 100
    struct Node *ancestors[100];

    return findClosestUtil(root, k, ancestors, 0);
}

/* Driver program to test above functions*/
int main()
{
    // Let us construct the BST shown in the above figure
    struct Node *root      = newNode('A');
    root->left           = newNode('B');
    root->right          = newNode('C');
    root->right->left   = newNode('E');
    root->right->right  = newNode('F');
    root->right->left->left = newNode('G');
    root->right->left->left->left = newNode('I');
}

```

```
root->right->left->left->right = newNode('J');
root->right->right->right      = newNode('H');
root->right->right->right->left = newNode('K');

char k = 'H';
cout << "Distance of the closest key from " << k << " is "
    << findClosest(root, k) << endl;
k = 'C';
cout << "Distance of the closest key from " << k << " is "
    << findClosest(root, k) << endl;
k = 'E';
cout << "Distance of the closest key from " << k << " is "
    << findClosest(root, k) << endl;
k = 'B';
cout << "Distance of the closest key from " << k << " is "
    << findClosest(root, k) << endl;

return 0;
}
```

**Java**

```
// Java program to find closest leaf of a given key in Binary Tree

/* Class containing left and right child of current
   node and key value*/
class Node
{
    int data;
    Node left, right;

    public Node(int item)
    {
        data = item;
        left = right = null;
    }
}

class BinaryTree
{
    Node root;

    // A utility function to find minimum of x and y
    int getMin(int x, int y)
    {
        return (x < y) ? x : y;
    }
}
```

```

// A utility function to find distance of closest leaf of the tree
// rooted under given root
int closestDown(Node node)
{
    // Base cases
    if (node == null)
        return Integer.MAX_VALUE;
    if (node.left == null && node.right == null)
        return 0;

    // Return minimum of left and right, plus one
    return 1 + getMin(closestDown(node.left), closestDown(node.right));
}

// Returns distance of the cloest leaf to a given key 'k'. The array
// ancestors is used to keep track of ancestors of current node and
// 'index' is used to keep track of current index in 'ancestors[]'
int findClosestUtil(Node node, char k, Node ancestors[], int index)
{
    // Base case
    if (node == null)
        return Integer.MAX_VALUE;

    // If key found
    if (node.data == k)
    {
        // Find the cloest leaf under the subtree rooted with given key
        int res = closestDown(node);

        // Traverse all ancestors and update result if any parent node
        // gives smaller distance
        for (int i = index - 1; i >= 0; i--)
            res = getMin(res, index - i + closestDown(ancestors[i]));
        return res;
    }

    // If key node found, store current node and recur for left and
    // right childrens
    ancestors[index] = node;
    return getMin(findClosestUtil(node.left, k, ancestors, index + 1),
                  findClosestUtil(node.right, k, ancestors, index + 1));
}

// The main function that returns distance of the closest key to 'k'. It
// mainly uses recursive function findClosestUtil() to find the closes
// distance.
int findClosest(Node node, char k)

```

```
{  
    // Create an array to store ancestors  
    // Assumption: Maximum height of tree is 100  
    Node ancestors[] = new Node[100];  
  
    return findClosestUtil(node, k, ancestors, 0);  
}  
  
// Driver program to test for above functions  
public static void main(String args[])  
{  
    BinaryTree tree = new BinaryTree();  
    tree.root = new Node('A');  
    tree.root.left = new Node('B');  
    tree.root.right = new Node('C');  
    tree.root.right.left = new Node('E');  
    tree.root.right.right = new Node('F');  
    tree.root.right.left.left = new Node('G');  
    tree.root.right.left.left = new Node('I');  
    tree.root.right.left.right = new Node('J');  
    tree.root.right.right.right = new Node('H');  
    tree.root.right.right.left = new Node('H');  
  
    char k = 'H';  
    System.out.println("Distance of the closest key from " + k + " is "  
                      + tree.findClosest(tree.root, k));  
    k = 'C';  
    System.out.println("Distance of the closest key from " + k + " is "  
                      + tree.findClosest(tree.root, k));  
    k = 'E';  
    System.out.println("Distance of the closest key from " + k + " is "  
                      + tree.findClosest(tree.root, k));  
    k = 'B';  
    System.out.println("Distance of the closest key from " + k + " is "  
                      + tree.findClosest(tree.root, k));  
  
}  
}  
  
// This code has been contributed by Mayank Jaiswal
```

### Python

```
# Python program to find closest leaf of a  
# given key in binary tree  
  
INT_MAX = 2**32
```

```

# A binary tree node
class Node:
    # Constructor to create a binary tree
    def __init__(self ,key):
        self.key = key
        self.left = None
        self.right = None

def closestDown(root):
    #Base Case
    if root is None:
        return INT_MAX
    if root.left is None and root.right is None:
        return 0

    # Return minum of left and right plus one
    return 1 + min(closestDown(root.left),
                    closestDown(root.right))

# Returns destance of the closes leaf to a given key k
# The array ancestors us used to keep track of ancestors
# of current node and 'index' is used to keep track of
# current index in 'ancestors[i]'
def findClosestUtil(root, k, ancestors, index):
    # Base Case
    if root is None:
        return INT_MAX

    # if key found
    if root.key == k:
        # Find closest leaf under the subtree rooted
        # with given key
        res = closestDown(root)

    # Traverse ll ancestors and update result if any
    # parent node gives smaller distance
    for i in reversed(range(0,index)):
        res = min(res, index-i+closestDown(ancestors[i]))
    return res

    # if key node found, store current node and recur for left
    # and right childrens
    ancestors[index] = root
    return min(
        findClosestUtil(root.left, k,ancestors, index+1),
        findClosestUtil(root.right, k, ancestors, index+1))

# The main function that return distance of the cses key to

```

```
# 'key'. It mainly uses recursive function findClosestUtil()
# to find the closes distance
def findClosest(root, k):
    # Create an arrray to store ancestors
    # Assumption: Maximum height of tree is 100
    ancestors = [None for i in range(100)]

    return findClosestUtil(root, k, ancestors, 0)

# Driver program to test above function
root = Node('A')
root.left = Node('B')
root.right = Node('C');
root.right.left = Node('E');
root.right.right = Node('F');
root.right.left.left = Node('G');
root.right.left.left.left = Node('I');
root.right.left.left.right = Node('J');
root.right.right.right = Node('H');
root.right.right.right.left = Node('K');

k = 'H';
print "Distance of the closest key from " + k + " is",
print findClosest(root, k)

k = 'C'
print "Distance of the closest key from " + k + " is",
print findClosest(root, k)

k = 'E'
print "Distance of the closest key from " + k + " is",
print findClosest(root, k)

k = 'B'
print "Distance of the closest key from " + k + " is",
print findClosest(root, k)

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

Output:

```
Distace of the closest key from H is 1
Distace of the closest key from C is 2
Distace of the closest key from E is 2
Distace of the closest key from B is 0
```

The above code can be optimized by storing the left/right information also in ancestor

array. The idea is, if given key is in left subtree of an ancestors, then there is no point to call closestDown(). Also, the loop can that traverses ancestors array can be optimized to not traverse ancestors which are at more distance than current result.

**Exercise:**

Extend the above solution to print not only distance, but the key of closest leaf also.

This article is contributed by Shubham. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## Source

<https://www.geeksforgeeks.org/find-closest-leaf-binary-tree/>

## Chapter 187

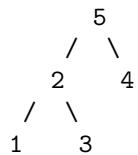
# Find the largest BST subtree in a given Binary Tree | Set 1

Find the largest BST subtree in a given Binary Tree | Set 1 - GeeksforGeeks

Given a Binary Tree, write a function that returns the size of the largest subtree which is also a Binary Search Tree (BST). If the complete Binary Tree is BST, then return the size of whole tree.

Examples:

Input:

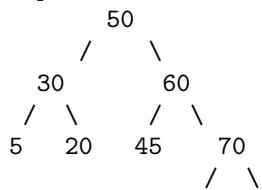


Output: 3

The following subtree is the maximum size BST subtree



Input:



```

65      80
Output: 5
The following subtree is the maximum size BST subtree
  60
  / \
 45   70
  / \
65   80

```

### Method 1 (Simple but inefficient)

Start from root and do an inorder traversal of the tree. For each node N, check whether the subtree rooted with N is BST or not. If BST, then return size of the subtree rooted with N. Else, recur down the left and right subtrees and return the maximum of values returned by left and right subtrees.

```

/*
See https://www.geeksforgeeks.org/write-a-c-program-to-calculate-size-of-a-tree/ for implementation

See Method 3 of https://www.geeksforgeeks.org/a-program-to-check-if-a-binary-tree-is-bst-or-not/ for implementation of isBST()

max() returns maximum of two integers
*/
int largestBST(struct node *root)
{
    if (isBST(root))
        return size(root);
    else
        return max(largestBST(root->left), largestBST(root->right));
}

```

Time Complexity: The worst case time complexity of this method will be  $O(n^2)$ . Consider a skewed tree for worst case analysis.

### Method 2 (Tricky and Efficient)

In method 1, we traverse the tree in top down manner and do BST test for every node. If we traverse the tree in bottom up manner, then we can pass information about subtrees to the parent. The passed information can be used by the parent to do BST test (for parent node) only in constant time (or  $O(1)$  time). A left subtree need to tell the parent whether it is BST or not and also need to pass maximum value in it. So that we can compare the maximum value with the parent's data to check the BST property. Similarly, the right subtree need to pass the minimum value up the tree. The subtrees need to pass the following information up the tree for the finding the largest BST.

- 1) Whether the subtree itself is BST or not (In the following code, `is_bst_ref` is used for this purpose)
- 2) If the subtree is left subtree of its parent, then maximum value in it. And if it is right

subtree then minimum value in it.

3) Size of this subtree if this subtree is BST (In the following code, return value of largestBSTUtil() is used for this purpose)

max\_ref is used for passing the maximum value up the tree and min\_ptr is used for passing minimum value up the tree.

C

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* node = (struct node*)
                           malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;

    return(node);
}

int largestBSTUtil(struct node* node, int *min_ref, int *max_ref,
                  int *max_size_ref, bool *is_bst_ref);

/* Returns size of the largest BST subtree in a Binary Tree
   (efficient version). */
int largestBST(struct node* node)
{
    // Set the initial values for calling largestBSTUtil()
    int min = INT_MAX; // For minimum value in right subtree
    int max = INT_MIN; // For maximum value in left subtree

    int max_size = 0; // For size of the largest BST
    bool is_bst = 0;
```

```

largestBSTUtil(node, &min, &max, &max_size, &is_bst);

return max_size;
}

/* largestBSTUtil() updates *max_size_ref for the size of the largest BST
   subtree. Also, if the tree rooted with node is non-empty and a BST,
   then returns size of the tree. Otherwise returns 0.*/
int largestBSTUtil(struct node* node, int *min_ref, int *max_ref,
                   int *max_size_ref, bool *is_bst_ref)
{

/* Base Case */
if (node == NULL)
{
    *is_bst_ref = 1; // An empty tree is BST
    return 0;        // Size of the BST is 0
}

int min = INT_MAX;

/* A flag variable for left subtree property
   i.e., max(root->left) < root->data */
bool left_flag = false;

/* A flag variable for right subtree property
   i.e., min(root->right) > root->data */
bool right_flag = false;

int ls, rs; // To store sizes of left and right subtrees

/* Following tasks are done by recursive call for left subtree
   a) Get the maximum value in left subtree (Stored in *max_ref)
   b) Check whether Left Subtree is BST or not (Stored in *is_bst_ref)
   c) Get the size of maximum size BST in left subtree (updates *max_size) */
*max_ref = INT_MIN;
ls = largestBSTUtil(node->left, min_ref, max_ref, max_size_ref, is_bst_ref);
if (*is_bst_ref == 1 && node->data > *max_ref)
    left_flag = true;

/* Before updating *min_ref, store the min value in left subtree. So that we
   have the correct minimum value for this subtree */
min = *min_ref;

/* The following recursive call does similar (similar to left subtree)
   task for right subtree */
*min_ref = INT_MAX;
rs = largestBSTUtil(node->right, min_ref, max_ref, max_size_ref, is_bst_ref);

```

```

if (*is_bst_ref == 1 && node->data < *min_ref)
    right_flag = true;

// Update min and max values for the parent recursive calls
if (min < *min_ref)
    *min_ref = min;
if (node->data < *min_ref) // For leaf nodes
    *min_ref = node->data;
if (node->data > *max_ref)
    *max_ref = node->data;

/* If both left and right subtrees are BST. And left and right
   subtree properties hold for this node, then this tree is BST.
   So return the size of this tree */
if(left_flag && right_flag)
{
    if (ls + rs + 1 > *max_size_ref)
        *max_size_ref = ls + rs + 1;
    return ls + rs + 1;
}
else
{
    //Since this subtree is not BST, set is_bst flag for parent calls
    *is_bst_ref = 0;
    return 0;
}
}

/* Driver program to test above functions*/
int main()
{
    /* Let us construct the following Tree
           50
         /   \
       10     60
      / \   / \
     5  20  55  70
        /   / \
      45   65  80
    */
    struct node *root = newNode(50);
    root->left = newNode(10);
    root->right = newNode(60);
    root->left->left = newNode(5);
    root->left->right = newNode(20);
    root->right->left = newNode(55);
    root->right->left->left = newNode(45);
}

```

```
root->right->right = newNode(70);
root->right->right->left = newNode(65);
root->right->right->right = newNode(80);

/* The complete tree is not BST as 45 is in right subtree of 50.
   The following subtree is the largest BST
      60
     / \
    55   70
   /   / \
  45  65   80
*/
printf(" Size of the largest BST is %d", largestBST(root));

getchar();
return 0;
}
```

**Java**

```
// Java program to find largest BST subtree in given Binary Tree

class Node {

    int data;
    Node left, right;

    Node(int d) {
        data = d;
        left = right = null;
    }
}

class Value {

    int max_size = 0; // for size of largest BST
    boolean is_bst = false;
    int min = Integer.MAX_VALUE; // For minimum value in right subtree
    int max = Integer.MIN_VALUE; // For maximum value in left subtree
}

class BinaryTree {

    static Node root;
    Value val = new Value();

    /* Returns size of the largest BST subtree in a Binary Tree
```

```

(efficient version). */
int largestBST(Node node) {

    largestBSTUtil(node, val, val, val, val);

    return val.max_size;
}

/* largestBSTUtil() updates *max_size_ref for the size of the largest BST
   subtree. Also, if the tree rooted with node is non-empty and a BST,
   then returns size of the tree. Otherwise returns 0.*/
int largestBSTUtil(Node node, Value min_ref, Value max_ref,
                   Value max_size_ref, Value is_bst_ref) {

    /* Base Case */
    if (node == null) {
        is_bst_ref.is_bst = true; // An empty tree is BST
        return 0;      // Size of the BST is 0
    }

    int min = Integer.MAX_VALUE;

    /* A flag variable for left subtree property
       i.e., max(root->left) < root->data */
    boolean left_flag = false;

    /* A flag variable for right subtree property
       i.e., min(root->right) > root->data */
    boolean right_flag = false;

    int ls, rs; // To store sizes of left and right subtrees

    /* Following tasks are done by recursive call for left subtree
     a) Get the maximum value in left subtree (Stored in *max_ref)
     b) Check whether Left Subtree is BST or not (Stored in *is_bst_ref)
     c) Get the size of maximum size BST in left subtree (updates *max_size) */
    max_ref.max = Integer.MIN_VALUE;
    ls = largestBSTUtil(node.left, min_ref, max_ref, max_size_ref, is_bst_ref);
    if (is_bst_ref.is_bst == true && node.data > max_ref.max) {
        left_flag = true;
    }

    /* Before updating *min_ref, store the min value in left subtree. So that we
       have the correct minimum value for this subtree */
    min = min_ref.min;

    /* The following recursive call does similar (similar to left subtree)
       task for right subtree */
}

```

```

min_ref.min = Integer.MAX_VALUE;
rs = largestBSTUtil(node.right, min_ref, max_ref, max_size_ref, is_bst_ref);
if (is_bst_ref.is_bst == true && node.data < min_ref.min) {
    right_flag = true;
}

// Update min and max values for the parent recursive calls
if (min < min_ref.min) {
    min_ref.min = min;
}
if (node.data < min_ref.min) // For leaf nodes
{
    min_ref.min = node.data;
}
if (node.data > max_ref.max) {
    max_ref.max = node.data;
}

/* If both left and right subtrees are BST. And left and right
   subtree properties hold for this node, then this tree is BST.
   So return the size of this tree */
if (left_flag && right_flag) {
    if (ls + rs + 1 > max_size_ref.max_size) {
        max_size_ref.max_size = ls + rs + 1;
    }
    return ls + rs + 1;
} else {
    //Since this subtree is not BST, set is_bst flag for parent calls
    is_bst_ref.is_bst = false;
    return 0;
}
}

public static void main(String[] args) {
    /* Let us construct the following Tree
        50
        /     \
       10      60
      / \      / \
     5  20    55  70
    /   / \
   45  65   80
    */
}

BinaryTree tree = new BinaryTree();
tree.root = new Node(50);
tree.root.left = new Node(10);
tree.root.right = new Node(60);

```

```
tree.root.left.left = new Node(5);
tree.root.left.right = new Node(20);
tree.root.right.left = new Node(55);
tree.root.right.left.left = new Node(45);
tree.root.right.right = new Node(70);
tree.root.right.right.left = new Node(65);
tree.root.right.right.right = new Node(80);

/* The complete tree is not BST as 45 is in right subtree of 50.
The following subtree is the largest BST
      60
     /   \
    55   70
   /     /   \
  45    65   80
*/
System.out.println("Size of largest BST is " + tree.largestBST(root));
}

}

// This code has been contributed by Mayank Jaiswal
```

Time Complexity: O(n) where n is the number of nodes in the given Binary Tree.

### Largest BST in a Binary Tree | Set 2

#### Source

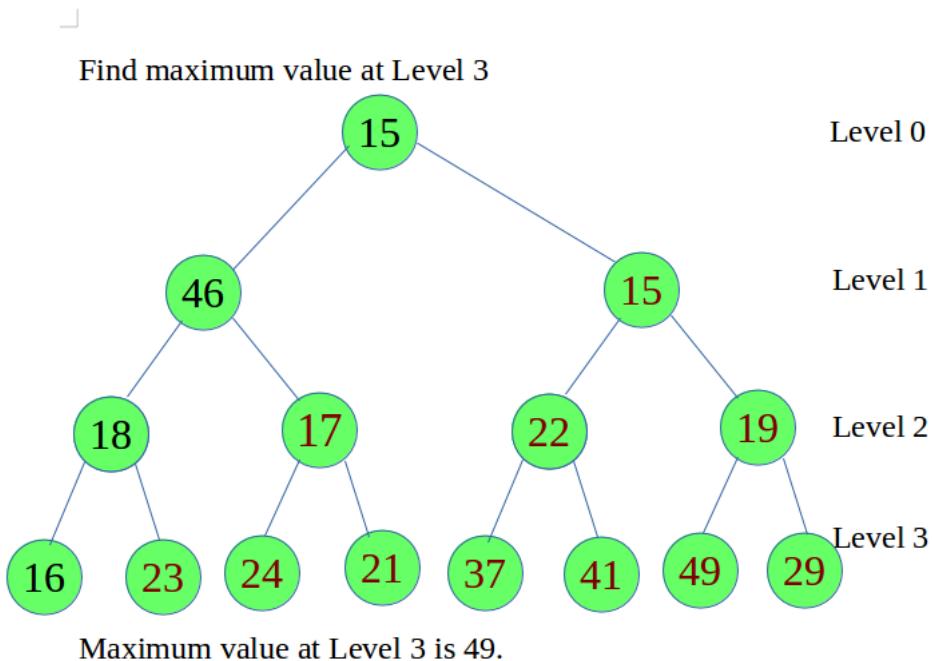
<https://www.geeksforgeeks.org/find-the-largest-subtree-in-a-tree-that-is-also-a-bst/>

## Chapter 188

# Find the maximum node at a given level in a binary tree

Find the maximum node at a given level in a binary tree - GeeksforGeeks

Given a **Binary Tree** and a **Level**. The task is to **find the node** with the maximum value at that given level.



The idea is to [traverse the tree along depth](#) recursively and return the nodes once the required level is reached and then return the maximum of left and right subtrees for each

subsequent call. So that the last call will return the node with maximum value among all nodes at the given level.

Below is the step by step algorithm:

1. Perform DFS traversal and every time decrease the value of *level* by 1 and keep traversing to the left and right subtrees recursively.
2. When value of *level* becomes 0, it means we are on the given level, then return *root->data*.
3. Find the maximum between the two values returned by left and right subtrees and return the maximum.

Below is the implementation of above approach:

C++

```
// CPP program to find the node with
// maximum value at a given level

#include <iostream>

using namespace std;

// Tree node
struct Node {
    int data;
    struct Node *left, *right;
};

// Utility function to create a new Node
struct Node* newNode(int val)
{
    struct Node* temp = new Node;
    temp->left = NULL;
    temp->right = NULL;
    temp->data = val;
    return temp;
}

// function to find the maximum value
// at given level
int maxAtLevel(struct Node* root, int level)
{
    // If the tree is empty
    if (root == NULL)
        return 0;
```

```
// if level becomes 0, it means we are on
// any node at the given level
if (level == 0)
    return root->data;

int x = maxAtLevel(root->left, level - 1);
int y = maxAtLevel(root->right, level - 1);

// return maximum of two
return max(x, y);
}

// Driver code
int main()
{
    // Creating the tree
    struct Node* root = NULL;
    root = newNode(45);
    root->left = newNode(46);
    root->left->left = newNode(18);
    root->left->left->left = newNode(16);
    root->left->left->right = newNode(23);
    root->left->right = newNode(17);
    root->left->right->left = newNode(24);
    root->left->right->right = newNode(21);
    root->right = newNode(15);
    root->right->left = newNode(22);
    root->right->left->left = newNode(37);
    root->right->left->right = newNode(41);
    root->right->right = newNode(19);
    root->right->right->left = newNode(49);
    root->right->right->right = newNode(29);

    int level = 3;

    cout << maxAtLevel(root, level);

    return 0;
}
```

### Java

```
// Java program to find the
// node with maximum value
// at a given level
import java.util.*;
class GFG
{
```

```
// Tree node
static class Node
{
    int data;
    Node left, right;
}

// Utility function to
// create a new Node
static Node newNode(int val)
{
    Node temp = new Node();
    temp.left = null;
    temp.right = null;
    temp.data = val;
    return temp;
}

// function to find
// the maximum value
// at given level
static int maxAtLevel(Node root, int level)
{
    // If the tree is empty
    if (root == null)
        return 0;

    // if level becomes 0,
    // it means we are on
    // any node at the given level
    if (level == 0)
        return root.data;

    int x = maxAtLevel(root.left, level - 1);
    int y = maxAtLevel(root.right, level - 1);

    // return maximum of two
    return Math.max(x, y);
}

// Driver code
public static void main(String args[])
{
    // Creating the tree
    Node root = null;
    root = newNode(45);
    root.left = newNode(46);
```

```
root.left.left = newNode(18);
root.left.left.left = newNode(16);
root.left.left.right = newNode(23);
root.left.right = newNode(17);
root.left.right.left = newNode(24);
root.left.right.right = newNode(21);
root.right = newNode(15);
root.right.left = newNode(22);
root.right.left.left = newNode(37);
root.right.left.right = newNode(41);
root.right.right = newNode(19);
root.right.right.left = newNode(49);
root.right.right.right = newNode(29);

int level = 3;

System.out.println(maxAtLevel(root, level));
}

}

// This code is contributed
// by Arnab Kundu
```

**Output:**

49

Improved By : [andrew1234](#)

**Source**

<https://www.geeksforgeeks.org/find-the-maximum-node-at-a-given-level-in-a-binary-tree/>

## Chapter 189

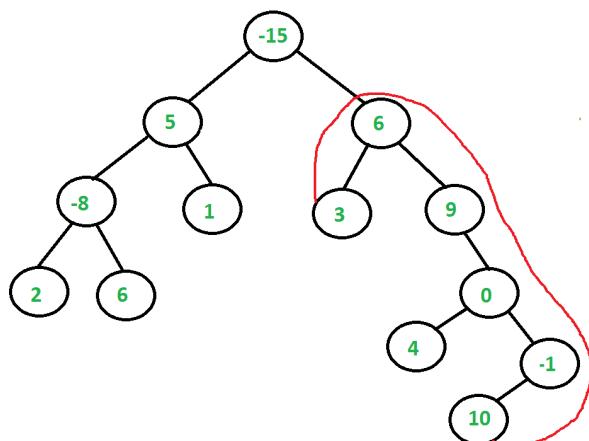
# Find the maximum path sum between two leaves of a binary tree

Find the maximum path sum between two leaves of a binary tree - GeeksforGeeks

Given a binary tree in which each node element contains a number. Find the maximum possible sum from one leaf node to another.

The maximum sum path may or may not go through root. For example, in the following binary tree, the maximum sum is **27**( $3 + 6 + 9 + 0 - 1 + 10$ ). Expected time complexity is  $O(n)$ .

If one side of root is empty, then function should return minus infinite (INT\_MIN in case of C/C++)



A simple solution is to traverse the tree and do following for every traversed node X.

- 1) Find maximum sum from leaf to root in left subtree of X (we can use [this post](#) for this and next steps)

- 2) Find maximum sum from leaf to root in right subtree of X.
- 3) Add the above two calculated values and X->data and compare the sum with the maximum value obtained so far and update the maximum value.
- 4) Return the maximum value.

The time complexity of above solution is  $O(n^2)$

**We can find the maximum sum using single traversal of binary tree.** The idea is to maintain two values in recursive calls

- 1) Maximum root to leaf path sum for the subtree rooted under current node.
- 2) The maximum path sum between leaves (desired output).

For every visited node X, we find the maximum root to leaf sum in left and right subtrees of X. We add the two values with X->data, and compare the sum with maximum path sum found so far.

Following is the implementation of the above  $O(n)$  solution.

C++

```
// C++ program to find maximum path sum between two leaves of
// a binary tree
#include <bits/stdc++.h>
using namespace std;

// A binary tree node
struct Node
{
    int data;
    struct Node* left, *right;
};

// Utility function to allocate memory for a new node
struct Node* newNode(int data)
{
    struct Node* node = new(struct Node);
    node->data = data;
    node->left = node->right = NULL;
    return (node);
}

// Utility function to find maximum of two integers
int max(int a, int b)
{ return (a >= b)? a: b; }

// A utility function to find the maximum sum between any
// two leaves. This function calculates two values:
// 1) Maximum path sum between two leaves which is stored
//     in res.
// 2) The maximum root to leaf path sum which is returned.
```

```
// If one side of root is empty, then it returns INT_MIN
int maxPathSumUtil(struct Node *root, int &res)
{
    // Base cases
    if (root==NULL) return 0;
    if (!root->left && !root->right) return root->data;

    // Find maximum sum in left and right subtree. Also
    // find maximum root to leaf sums in left and right
    // subtrees and store them in ls and rs
    int ls = maxPathSumUtil(root->left, res);
    int rs = maxPathSumUtil(root->right, res);

    // If both left and right children exist
    if (root->left && root->right)
    {
        // Update result if needed
        res = max(res, ls + rs + root->data);

        // Return maximum possible value for root being
        // on one side
        return max(ls, rs) + root->data;
    }

    // If any of the two children is empty, return
    // root sum for root being on one side
    return (!root->left)? rs + root->data:
                           ls + root->data;
}

// The main function which returns sum of the maximum
// sum path between two leaves. This function mainly
// uses maxPathSumUtil()
int maxPathSum(struct Node *root)
{
    int res = INT_MIN;
    maxPathSumUtil(root, res);
    return res;
}

// driver program to test above function
int main()
{
    struct Node *root = newNode(-15);
    root->left = newNode(5);
    root->right = newNode(6);
    root->left->left = newNode(-8);
```

```
root->left->right = newNode(1);
root->left->left->left = newNode(2);
root->left->left->right = newNode(6);
root->right->left = newNode(3);
root->right->right = newNode(9);
root->right->right->right= newNode(0);
root->right->right->right->left= newNode(4);
root->right->right->right->right= newNode(-1);
root->right->right->right->right->left= newNode(10);
cout << "Max pathSum of the given binary tree is "
     << maxPathSum(root);
return 0;
}
```

**Java**

```
// Java program to find maximum path sum between two leaves
// of a binary tree
class Node {

    int data;
    Node left, right;

    Node(int item) {
        data = item;
        left = right = null;
    }
}

// An object of Res is passed around so that the
// same value can be used by multiple recursive calls.
class Res {
    int val;
}

class BinaryTree {

    static Node root;

    // A utility function to find the maximum sum between any
    // two leaves.This function calculates two values:
    // 1) Maximum path sum between two leaves which is stored
    //     in res.
    // 2) The maximum root to leaf path sum which is returned.
    // If one side of root is empty, then it returns INT_MIN
    int maxPathSumUtil(Node node, Res res) {

        // Base cases
```

```
if (node == null)
    return 0;
if (node.left == null && node.right == null)
    return node.data;

// Find maximum sum in left and right subtree. Also
// find maximum root to leaf sums in left and right
// subtrees and store them in ls and rs
int ls = maxPathSumUtil(node.left, res);
int rs = maxPathSumUtil(node.right, res);

// If both left and right children exist
if (node.left != null && node.right != null) {

    // Update result if needed
    res.val = Math.max(res.val, ls + rs + node.data);

    // Return maximum possible value for root being
    // on one side
    return Math.max(ls, rs) + node.data;
}

// If any of the two children is empty, return
// root sum for root being on one side
return (node.left == null) ? rs + node.data
                           : ls + node.data;
}

// The main function which returns sum of the maximum
// sum path between two leaves. This function mainly
// uses maxPathSumUtil()
int maxPathSum(Node node)
{
    Res res = new Res();
    res.val = Integer.MIN_VALUE;
    maxPathSumUtil(root, res);
    return res.val;
}

//Driver program to test above functions
public static void main(String args[]) {
    BinaryTree tree = new BinaryTree();
    tree.root = new Node(-15);
    tree.root.left = new Node(5);
    tree.root.right = new Node(6);
    tree.root.left.left = new Node(-8);
    tree.root.left.right = new Node(1);
    tree.root.left.left.left = new Node(2);
```

```
tree.root.left.left.right = new Node(6);
tree.root.right.left = new Node(3);
tree.root.right.right = new Node(9);
tree.root.right.right.right = new Node(0);
tree.root.right.right.left = new Node(4);
tree.root.right.right.right = new Node(-1);
tree.root.right.right.right.left = new Node(10);
System.out.println("Max pathSum of the given binary tree is "
+ tree.maxPathSum(root));
}
}

// This code is contributed by Mayank Jaiswal
```

### Python

```
# Python program to find maximumpath sum between two leaves
# of a binary tree

INT_MIN = -2**32

# A binary tree node
class Node:
    # Constructor to create a new node
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

    # Utility function to find maximum sum between any
    # two leaves. This function calculates two values:
    # 1) Maximum path sum between two leaves which are stored
    #     in res
    # 2) The maximum root to leaf path sum which is returned
    # If one side of root is empty, then it returns INT_MIN

def maxPathSumUtil(root, res):

    # Base Case
    if root is None:
        return 0

    if root.left is None and root.right is None:
        return root.data

    # Find maximumsum in left and righ subtree. Also
    # find maximum root to leaf sums in left and righ
    # subtrees ans store them in ls and rs
```

```
ls = maxPathSumUtil(root.left, res)
rs = maxPathSumUtil(root.right, res)

# If both left and right children exist
if root.left is not None and root.right is not None:

    # update result if needed
    res[0] = max(res[0], ls + rs + root.data)

    # Return maximum possible value for root being
    # on one side
    return max(ls, rs) + root.data

# If any of the two children is empty, return
# root sum for root being on one side
if root.left is None:
    return rs + root.data
else:
    return ls + root.data

# The main function which returns sum of the maximum
# sum path between two leaves. This function mainly
# uses maxPathSumUtil()
def maxPathSum(root):
    res = [INT_MIN]
    maxPathSumUtil(root, res)
    return res[0]

# Driver program to test above function
root = Node(-15)
root.left = Node(5)
root.right = Node(6)
root.left.left = Node(-8)
root.left.right = Node(1)
root.left.left.left = Node(2)
root.left.left.right = Node(6)
root.right.left = Node(3)
root.right.right = Node(9)
root.right.right.right= Node(0)
root.right.right.right.left = Node(4)
root.right.right.right.right = Node(-1)
root.right.right.right.left = Node(10)

print "Max pathSum of the given binary tree is", maxPathSum(root)

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

Output:

```
Max pathSum of the given binary tree is 27.
```

Thanks to Saurabh Vats for suggesting corrections in original approach.

This article is contributed by **Kripal Gaurav**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## Source

<https://www.geeksforgeeks.org/find-maximum-path-sum-two-leaves-binary-tree/>

## Chapter 190

# Find the maximum sum leaf to root path in a Binary Tree

Find the maximum sum leaf to root path in a Binary Tree - GeeksforGeeks

### Source

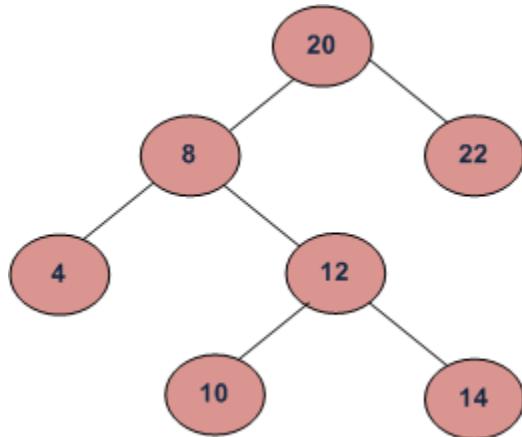
<https://www.geeksforgeeks.org/find-the-maximum-sum-path-in-a-binary-tree/>

## Chapter 191

# Find the node with minimum value in a Binary Search Tree

Find the node with minimum value in a Binary Search Tree - GeeksforGeeks

This is quite simple. Just traverse the node from root to left recursively until left is NULL. The node whose left is NULL is the node with minimum value.



For the above tree, we start with 20, then we move left 8, we keep on moving to left until we see NULL. Since left of 4 is NULL, 4 is the node with minimum value.

C

```
#include <stdio.h>
#include<stdlib.h>

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
```

```
{  
    int data;  
    struct node* left;  
    struct node* right;  
};  
  
/* Helper function that allocates a new node  
with the given data and NULL left and right  
pointers. */  
struct node* newNode(int data)  
{  
    struct node* node = (struct node*)  
        malloc(sizeof(struct node));  
    node->data = data;  
    node->left = NULL;  
    node->right = NULL;  
  
    return(node);  
}  
  
/* Give a binary search tree and a number,  
inserts a new node with the given number in  
the correct place in the tree. Returns the new  
root pointer which the caller should then use  
(the standard trick to avoid using reference  
parameters). */  
struct node* insert(struct node* node, int data)  
{  
    /* 1. If the tree is empty, return a new,  
       single node */  
    if (node == NULL)  
        return(newNode(data));  
    else  
    {  
        /* 2. Otherwise, recur down the tree */  
        if (data <= node->data)  
            node->left = insert(node->left, data);  
        else  
            node->right = insert(node->right, data);  
  
        /* return the (unchanged) node pointer */  
        return node;  
    }  
}  
  
/* Given a non-empty binary search tree,  
return the minimum data value found in that  
tree. Note that the entire tree does not need
```

```
to be searched. */
int minValue(struct node* node) {
    struct node* current = node;

    /* loop down to find the leftmost leaf */
    while (current->left != NULL) {
        current = current->left;
    }
    return(current->data);
}

/* Driver program to test sameTree function*/
int main()
{
    struct node* root = NULL;
    root = insert(root, 4);
    insert(root, 2);
    insert(root, 1);
    insert(root, 3);
    insert(root, 6);
    insert(root, 5);

    printf("\n Minimum value in BST is %d", minValue(root));
    getchar();
    return 0;
}
```

### Java

```
// Java program to find minimum value node in Binary Search Tree

// A binary tree node
class Node {

    int data;
    Node left, right;

    Node(int d) {
        data = d;
        left = right = null;
    }
}

class BinaryTree {

    static Node head;

    /* Given a binary search tree and a number,
```

```
inserts a new node with the given number in
the correct place in the tree. Returns the new
root pointer which the caller should then use
(the standard trick to avoid using reference
parameters). */
Node insert(Node node, int data) {

    /* 1. If the tree is empty, return a new,
       single node */
    if (node == null) {
        return (new Node(data));
    } else {

        /* 2. Otherwise, recur down the tree */
        if (data <= node.data) {
            node.left = insert(node.left, data);
        } else {
            node.right = insert(node.right, data);
        }

        /* return the (unchanged) node pointer */
        return node;
    }
}

/* Given a non-empty binary search tree,
   return the minimum data value found in that
   tree. Note that the entire tree does not need
   to be searched. */
int minvalue(Node node) {
    Node current = node;

    /* loop down to find the leftmost leaf */
    while (current.left != null) {
        current = current.left;
    }
    return (current.data);
}

// Driver program to test above functions
public static void main(String[] args) {
    BinaryTree tree = new BinaryTree();
    Node root = null;
    root = tree.insert(root, 4);
    tree.insert(root, 2);
    tree.insert(root, 1);
    tree.insert(root, 3);
    tree.insert(root, 6);
```

```
        tree.insert(root, 5);

        System.out.println("The minimum value of BST is " + tree.minvalue(root));
    }
}

// This code has been contributed by Mayank Jaiswal
```

### Python

```
# Python program to find the node with minimum value in bst

# A binary tree node
class Node:

    # Constructor to create a new node
    def __init__(self, key):
        self.data = key
        self.left = None
        self.right = None

    """ Give a binary search tree and a number,
    inserts a new node with the given number in
    the correct place in the tree. Returns the new
    root pointer which the caller should then use
    (the standard trick to avoid using reference
    parameters). """
    def insert(node, data):

        # 1. If the tree is empty, return a new,
        # single node
        if node is None:
            return (Node(data))

        else:
            # 2. Otherwise, recur down the tree
            if data <= node.data:
                node.left = insert(node.left, data)
            else:
                node.right = insert(node.right, data)

        # Return the (unchanged) node pointer
        return node

    """ Given a non-empty binary search tree,
    return the minimum data value found in that
    tree. Note that the entire tree does not need
    to be searched. """

```

```
def minValue(node):
    current = node

    # loop down to find the leftmost leaf
    while(current.left is not None):
        current = current.left

    return current.data

# Driver program
root = None
root = insert(root,4)
insert(root,2)
insert(root,1)
insert(root,3)
insert(root,6)
insert(root,5)

print "\nMinimum value in BST is %d" %(minValue(root))

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

**Time Complexity:** O(n) Worst case happens for left skewed trees.

Similarly we can get the maximum value by recursively traversing the right node of a binary search tree.

**References:**

<http://cslibrary.stanford.edu/110/BinaryTrees.html>

**Source**

<https://www.geeksforgeeks.org/find-the-minimum-element-in-a-binary-search-tree/>

## Chapter 192

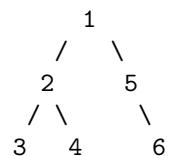
# Flatten a binary tree into linked list

Flatten a binary tree into linked list - GeeksforGeeks

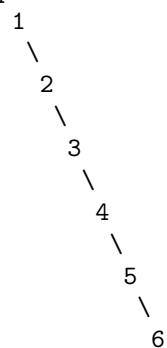
Given a binary tree, flatten it into linked list in-place. Usage of auxiliary data structure is not allowed. After flattening, left of each node should point to NULL and right should contain next node in level order.

Examples:

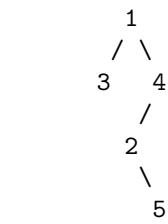
Input :



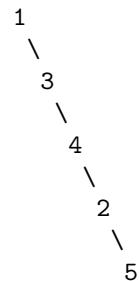
Output :



Input :



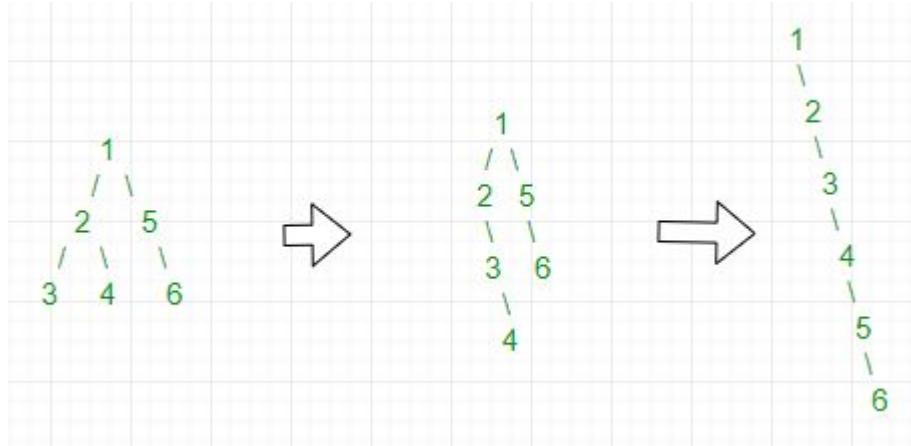
Output :



**Simple Approach:** A simple solution is to use [Level Order Traversal using Queue](#). In level order traversal, keep track of previous node. Make current node as right child of previous and left of previous node as NULL. This solution requires queue, but question asks to solve without additional data structure.

**Efficient Without Additional Data Structure** Recursively look for the node with no grandchildren and both left and right child in the left sub-tree. Then store node->right in temp and make node->right=node->left. Insert temp in first node NULL on right of node by node=node->right. Repeat until it is converted to linked list.

For Example,



```

/* Program to flatten a given Binary
Tree into linked list */
#include <iostream>
using namespace std;
  
```

```
struct Node {  
    int key;  
    Node *left, *right;  
};  
  
/* utility that allocates a new Node  
   with the given key */  
Node* newNode(int key)  
{  
    Node* node = new Node;  
    node->key = key;  
    node->left = node->right = NULL;  
    return (node);  
}  
  
// Function to convert binary tree into  
// linked list by altering the right node  
// and making left node point to NULL  
void flatten(struct Node* root)  
{  
    // base condition- return if root is NULL  
    // or if it is a leaf node  
    if (root == NULL || root->left == NULL &&  
        root->right == NULL) {  
        return;  
    }  
  
    // if root->left exists then we have  
    // to make it root->right  
    if (root->left != NULL) {  
  
        // move left recursively  
        flatten(root->left);  
  
        // store the node root->right  
        struct Node* tmpRight = root->right;  
        root->right = root->left;  
        root->left = NULL;  
  
        // find the position to insert  
        // the stored value  
        struct Node* t = root->right;  
        while (t->right != NULL) {  
            t = t->right;  
        }  
  
        // insert the stored value  
        t->right = tmpRight;  
    }  
}
```

```
}

// now call the same function
// for root->right
flatten(root->right);
}

// To find the inorder traversal
void inorder(struct Node* root)
{
    // base condition
    if (root == NULL)
        return;
    inorder(root->left);
    cout << root->key << " ";
    inorder(root->right);
}

/* Driver program to test above functions*/
int main()
{
    /*
        1
       /   \
      2     5
     / \   \
    3   4   6 */
    Node* root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(5);
    root->left->left = newNode(3);
    root->left->right = newNode(4);
    root->right->right = newNode(6);

    flatten(root);

    cout << "The Inorder traversal after "
        "flattening binary tree ";
    inorder(root);
    return 0;
}
```

**Output:**

```
The Inorder traversal after flattening
binary tree 1 2 3 4 5 6
```

**Source**

<https://www.geeksforgeeks.org/flatten-a-binary-tree-into-linked-list/>

## Chapter 193

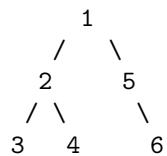
# Flatten a binary tree into linked list | Set-2

Flatten a binary tree into linked list | Set-2 - GeeksforGeeks

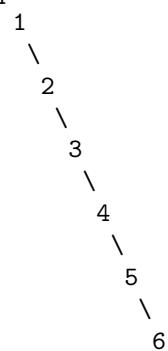
Given a binary tree, flatten it into a linked list. After flattening, the left of each node should point to NULL and right should contain next node in level order.

**Example:**

**Input:**

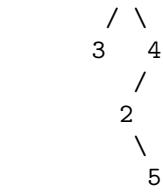


**Output:**

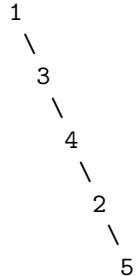


**Input:**

1



Output:



**Approach:** An approach using recursion has already been discussed in the [previous post](#). A pre-order traversal of the binary tree using stack has been implied in this approach. In this traversal, every time a right child is pushed in the stack, the right child is made equal to the left child and left child is made equal to NULL. If the right child of the node becomes NULL, the stack is popped and the right child becomes the popped value from the stack. The above steps are repeated until the size of the stack is zero or root is NULL.

Below is the implementation of the above approach:

```
// C++ program to flatten the linked
// list using stack | set-2
#include <iostream>
#include <stack>
using namespace std;

struct Node {
    int key;
    Node *left, *right;
};

/* utility that allocates a new Node
   with the given key */
Node* newNode(int key)
{
    Node* node = new Node;
    node->key = key;
    node->left = node->right = NULL;
    return (node);
}

// To find the inorder traversal
```

```
void inorder(struct Node* root)
{
    // base condition
    if (root == NULL)
        return;
    inorder(root->left);
    cout << root->key << " ";
    inorder(root->right);
}

// Function to convert binary tree into
// linked list by altering the right node
// and making left node point to NULL
Node* solution(Node* A)
{

    // Declare a stack
    stack<Node*> st;
    Node* ans = A;

    // Iterate till the stack is not empty
    // and till root is Null
    while (A != NULL || st.size() != 0) {

        // Check for NULL
        if (A->right != NULL) {
            st.push(A->right);
        }

        // Make the Right Left and
        // left NULL
        A->right = A->left;
        A->left = NULL;

        // Check for NULL
        if (A->right == NULL && st.size() != 0) {
            A->right = st.top();
            st.pop();
        }

        // Iterate
        A = A->right;
    }
    return ans;
}

// Driver Code
int main()
```

```
{  
    /*      1  
         /   \\  
        2     5  
       / \     \  
      3   4     6 */  
  
    // Build the tree  
    Node* root = newNode(1);  
    root->left = newNode(2);  
    root->right = newNode(5);  
    root->left->left = newNode(3);  
    root->left->right = newNode(4);  
    root->right->right = newNode(6);  
  
    // Call the function to  
    // flatten the tree  
    root = solution(root);  
  
    cout << "The Inorder traversal after "  
         "flattening binary tree ";  
  
    // call the function to print  
    // inorder after flattening  
    inorder(root);  
    return 0;  
  
    return 0;  
}
```

**Output:**

The Inorder traversal after flattening binary tree 1 2 3 4 5 6

**Time Complexity:** O(N)  
**Auxiliary Space:** O(Log N)

**Source**

<https://www.geeksforgeeks.org/flatten-a-binary-tree-into-linked-list-set-2/>

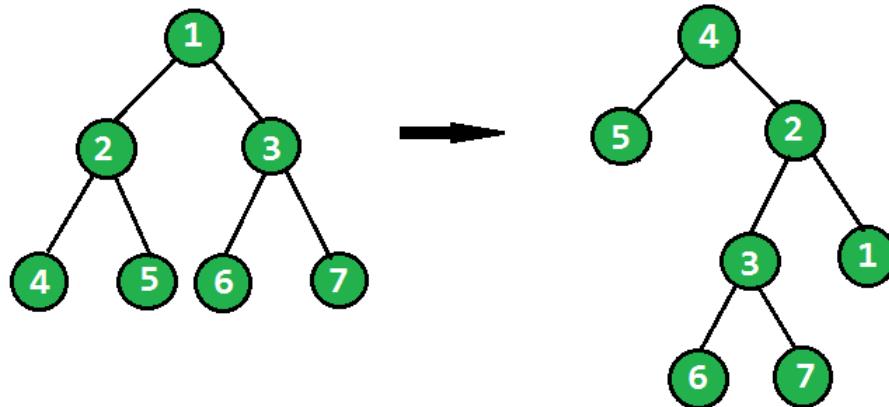
## Chapter 194

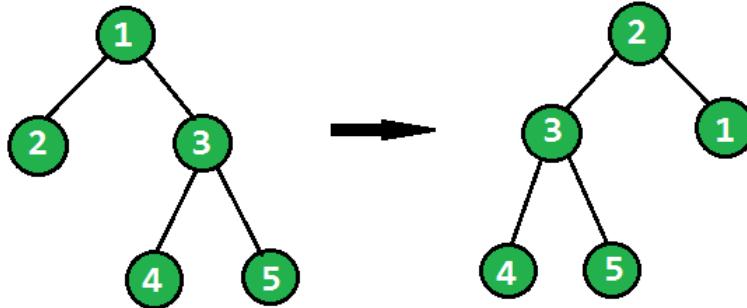
# Flip Binary Tree

Flip Binary Tree - GeeksforGeeks

Given a binary tree, the task is to flip the binary tree towards right direction that is clockwise. See below examples to see the transformation.

In the flip operation, left most node becomes the root of flipped tree and its parent become its right child and the right sibling become its left child and same should be done for all left most nodes recursively.



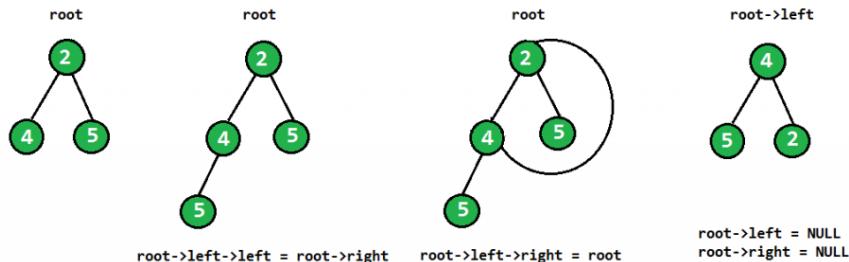


Below is main rotation code of a subtree

```

root->left->left = root->right;
root->left->right = root;
root->left = NULL;
root->right = NULL;
    
```

The above code can be understood by following diagram –



as we are storing `root->left` in flipped root, flipped subtree gets stored in each recursive call.

C++

```

/* C/C++ program to flip a binary tree */
#include <bits/stdc++.h>
using namespace std;

/* A binary tree node structure */
struct Node
{
    
```

```
int data;
Node *left, *right;
};

/* Utility function to create a new Binary
Tree Node */
struct Node* newNode(int data)
{
    struct Node *temp = new struct Node;
    temp->data = data;
    temp->left = temp->right = NULL;
    return temp;
}

// method to flip the binary tree
Node* flipBinaryTree(Node* root)
{
    // Base cases
    if (root == NULL)
        return root;
    if (root->left == NULL && root->right == NULL)
        return root;

    // recursively call the same method
    Node* flippedRoot = flipBinaryTree(root->left);

    // rearranging main root Node after returning
    // from recursive call
    root->left->left = root->right;
    root->left->right = root;
    root->left = root->right = NULL;

    return flippedRoot;
}

// Iterative method to do level order traversal
// line by line
void printLevelOrder(Node *root)
{
    // Base Case
    if (root == NULL)  return;

    // Create an empty queue for level order traversal
    queue<Node *> q;

    // Enqueue Root and initialize height
    q.push(root);
```

```
while (1)
{
    // nodeCount (queue size) indicates number
    // of nodes at current level.
    int nodeCount = q.size();
    if (nodeCount == 0)
        break;

    // Dequeue all nodes of current level and
    // Enqueue all nodes of next level
    while (nodeCount > 0)
    {
        Node *node = q.front();
        cout << node->data << " ";
        q.pop();
        if (node->left != NULL)
            q.push(node->left);
        if (node->right != NULL)
            q.push(node->right);
        nodeCount--;
    }
    cout << endl;
}

// Driver code
int main()
{
    Node* root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->right->left = newNode(4);
    root->right->right = newNode(5);

    cout << "Level order traversal of given tree\n";
    printLevelOrder(root);

    root = flipBinaryTree(root);

    cout << "\nLevel order traversal of the flipped"
         " tree\n";
    printLevelOrder(root);
    return 0;
}
```

### Python

```
# Python program to flip a binary tree
```

```
# A binary tree node
class Node:

    # Constructor to create a new node
    def __init__(self, data):
        self.data = data
        self.right = None
        self.left = None

def flipBinaryTree(root):

    # Base Cases
    if root is None:
        return root

    if root.left is None and root.right is None:
        return root

    # Recursively call the same method
    flippedRoot = flipBinaryTree(root.left)

    # Rearranging main root Node after returning
    # from recursive call
    root.left.left = root.right
    root.left.right = root
    root.left = root.right = None

    return flippedRoot

# Iterative method to do the level order traversal
# line by line
def printLevelOrder(root):

    # Base Case
    if root is None:
        return

    # Create an empty queue for level order traversal
    from Queue import Queue
    q = Queue()

    # Enqueue root and initialize height
    q.put(root)

    while(True):

        # nodeCount (queue size) indicates number
```

```
# of nodes at current level
nodeCount = q.qsize()
if nodeCount == 0:
    break

# Dequeue all nodes of current level and
# Enqueue all nodes of next level
while nodeCount > 0:
    node = q.get()
    print node.data,
    if node.left is not None:
        q.put(node.left)
    if node.right is not None:
        q.put(node.right)
    nodeCount -= 1

print

# Driver code
root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.right.left = Node(4)
root.right.right = Node(5)

print "Level order traversal of given tree"
printLevelOrder(root)

root = flipBinaryTree(root)

print "\nLevel order traversal of the flipped tree"
printLevelOrder(root)

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

Output:

```
Level order traversal of given tree
1
2 3
4 5

Level order traversal of the flipped tree
2
3 1
4 5
```

### Iterative Approach

This approach is contributed by **Pal13**.

The iterative solution follows the same approach as the recursive one, the only thing we need to pay attention to is to save the node information that will be overwritten.

#### CPP

```
// C/C++ program to flip a binary tree
#include <bits/stdc++.h>
using namespace std;

// A binary tree node structure
struct Node
{
    int data;
    Node *left, *right;
};

// Utility function to create a new Binary
// Tree Node
struct Node* newNode(int data)
{
    struct Node *temp = new struct Node;
    temp->data = data;
    temp->left = temp->right = NULL;
    return temp;
}

// method to flip the binary tree
Node* flipBinaryTree(Node* root)
{
    // Initialization of pointers
    Node *curr = root;
    Node *next = NULL;
    Node *temp = NULL;
    Node *prev = NULL;

    // Iterate through all left nodes
    while(curr)
    {
        next = curr->left;

        // Swapping nodes now, need temp to keep the previous right child
        // Making prev's right as curr's left child
        curr->left = temp;

        // Storing curr's right child
        temp = curr->right;
        curr->right = next;
    }
}
```

```

temp = curr->right;

// Making prev as curr's right child
curr->right = prev;

prev = curr;
curr = next;
}
return prev;
}

// Iterative method to do level order traversal
// line by line
void printLevelOrder(Node *root)
{
    // Base Case
    if (root == NULL) return;

    // Create an empty queue for level order traversal
    queue<Node *> q;

    // Enqueue Root and initialize height
    q.push(root);

    while (1)
    {
        // nodeCount (queue size) indicates number
        // of nodes at current level.
        int nodeCount = q.size();
        if (nodeCount == 0)
            break;

        // Dequeue all nodes of current level and
        // Enqueue all nodes of next level
        while (nodeCount > 0)
        {
            Node *node = q.front();
            cout << node->data << " ";
            q.pop();

            if (node->left != NULL)
                q.push(node->left);

            if (node->right != NULL)
                q.push(node->right);
            nodeCount--;
        }
        cout << endl;
    }
}

```

```
        }
    }

// Driver code
int main()
{
    Node* root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->right->left = newNode(4);
    root->right->right = newNode(5);

    cout << "Level order traversal of given tree\n";
    printLevelOrder(root);

    root = flipBinaryTree(root);

    cout << "\nLevel order traversal of the flipped"
         " tree\n";
    printLevelOrder(root);
    return 0;
}

// This article is contributed by Pal13
```

### **Java**

```
// Java program to flip a binary tree
import java.util.*;
class GFG
{
    // A binary tree node
    static class Node
    {
        int data;
        Node left, right;
    };

    // Utility function to create
    // a new Binary Tree Node

    static Node newNode(int data)
    {
        Node temp = new Node();
        temp.data = data;
        temp.left = temp.right = null;
        return temp;
    }
```

```
// method to flip the binary tree
static Node flipBinaryTree(Node root)
{
    // Initialization of pointers
    Node curr = root;
    Node next = null;
    Node temp = null;
    Node prev = null;

    // Iterate through all left nodes
    while(curr != null)
    {
        next = curr.left;

        // Swapping nodes now, need
        // temp to keep the previous
        // right child

        // Making prev's right
        // as curr's left child
        curr.left = temp;

        // Storing curr's right child
        temp = curr.right;

        // Making prev as curr's
        // right child
        curr.right = prev;

        prev = curr;
        curr = next;
    }
    return prev;
}

// Iterative method to do
// level order traversal
// line by line
static void printLevelOrder(Node root)
{
    // Base Case
    if (root == null) return;

    // Create an empty queue for
    // level order traversal
    Queue q = new LinkedList();

    // Enqueue Root and
    // initialize height
    q.add(root);

    while (true)
    {
```

```
// nodeCount (queue size)
// indicates number of nodes
// at current level.
int nodeCount = q.size();
if (nodeCount == 0)
    break;

// Dequeue all nodes of current
// level and Enqueue all nodes
// of next level
while (nodeCount > 0)
{
    Node node = q.peek();
    System.out.print(node.data + " ");
    q.remove();

    if (node.left != null)
        q.add(node.left);

    if (node.right != null)
        q.add(node.right);
    nodeCount--;
}
System.out.println();
}

// Driver code
public static void main(String args[])
{
    Node root = newNode(1);
    root.left = newNode(2);
    root.right = newNode(3);
    root.right.left = newNode(4);
    root.right.right = newNode(5);

    System.out.print("Level order traversal " +
    "of given tree\n");
    printLevelOrder(root);

    root = flipBinaryTree(root);

    System.out.print("\nLevel order traversal " +
    "of the flipped tree\n");
    printLevelOrder(root);
}

// This code is contributed
// by Arnab Kundu
```

**Output:**

Level order traversal of given tree

```
1  
2 3  
4 5
```

Level order traversal of the flipped tree

```
2  
3 1  
4 5
```

**Complexity Analysis:**

**Time complexity:**  $O(n)$  as in the worst case, depth of binary tree will be  $n$ .

**Auxiliary Space :**  $O(1)$ .

**Improved By :** [andrew1234](#)

**Source**

<https://www.geeksforgeeks.org/flip-binary-tree/>

## Chapter 195

# Foldable Binary Trees

Foldable Binary Trees - GeeksforGeeks

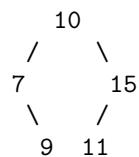
Question: Given a binary tree, find out if the tree can be folded or not.

A tree can be folded if left and right subtrees of the tree are structure wise mirror image of each other. An empty tree is considered as foldable.

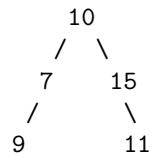
Consider the below trees:

- (a) and (b) can be folded.
- (c) and (d) cannot be folded.

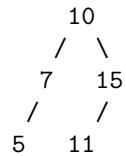
(a)



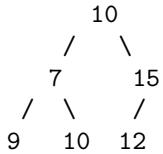
(b)



(c)



(d)



**Method 1 (Change Left subtree to its Mirror and compare it with Right subtree)**

Algorithm: isFoldable(root)

- 1) If tree is empty, then return true.
- 2) Convert the left subtree to its mirror image  
    mirror(root->left); /\* See this post \*/
- 3) Check if the structure of left subtree and right subtree is same  
    and store the result.  
    res = isStructSame(root->left, root->right); /\*isStructSame()  
        recursively compares structures of two subtrees and returns  
        true if structures are same \*/
- 4) Revert the changes made in step (2) to get the original tree.  
    mirror(root->left);
- 5) Return result res stored in step 2.

Thanks to ajaym for suggesting this approach.

C

```

#include<stdio.h>
#include<stdlib.h>

/* You would want to remove below 3 lines if your compiler
   supports bool, true and false */
#define bool int
#define true 1
#define false 0

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

/* converts a tree to its mirror image */
void mirror(struct node* node);
    
```

```

/* returns true if structure of two trees a and b is same
   Only structure is considered for comparison, not data! */
bool isStructSame(struct node *a, struct node *b);

/* Returns true if the given tree is foldable */
bool isFoldable(struct node *root)
{
    bool res;

    /* base case */
    if(root == NULL)
        return true;

    /* convert left subtree to its mirror */
    mirror(root->left);

    /* Compare the structures of the right subtree and mirrored
       left subtree */
    res = isStructSame(root->left, root->right);

    /* Get the original tree back */
    mirror(root->left);

    return res;
}

bool isStructSame(struct node *a, struct node *b)
{
    if (a == NULL && b == NULL)
        { return true; }
    if ( a != NULL && b != NULL &&
        isStructSame(a->left, b->left) &&
        isStructSame(a->right, b->right)
        )
        { return true; }

    return false;
}

/* UTILITY FUNCTIONS */
/* Change a tree so that the roles of the left and
   right pointers are swapped at every node.
   See https://www.geeksforgeeks.org/?p=662 for details */
void mirror(struct node* node)
{
    if (node==NULL)

```

```

    return;
else
{
    struct node* temp;

    /* do the subtrees */
    mirror(node->left);
    mirror(node->right);

    /* swap the pointers in this node */
    temp      = node->left;
    node->left  = node->right;
    node->right = temp;
}
}

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* node = (struct node*)
                           malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;

    return(node);
}

/* Driver program to test mirror() */
int main(void)
{
    /* The constructed binary tree is
       1
      / \
     2   3
    \   /
     4  5
    */
    struct node *root = newNode(1);
    root->left      = newNode(2);
    root->right     = newNode(3);
    root->right->left = newNode(4);
    root->left->right = newNode(5);

    if(isFoldable(root) == 1)
    { printf("\n tree is foldable"); }
    else

```

```
{ printf("\n tree is not foldable"); }

getchar();
return 0;
}
```

**Java**

```
// Java program to check foldable binary tree

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
class Node
{
    int data;
    Node left, right;

    Node(int item)
    {
        data = item;
        left = right = null;
    }
}

class BinaryTree
{
    Node root;

    /* Returns true if the given tree is foldable */
    boolean isFoldable(Node node)
    {
        boolean res;

        /* base case */
        if (node == null)
            return true;

        /* convert left subtree to its mirror */
        mirror(node.left);

        /* Compare the structures of the right subtree and mirrored
           left subtree */
        res = isStructSame(node.left, node.right);

        /* Get the original tree back */
        mirror(node.left);

        return res;
    }
}
```

```
}

boolean isStructSame(Node a, Node b)
{
    if (a == null && b == null)
        return true;
    if (a != null && b != null
        && isStructSame(a.left, b.left)
        && isStructSame(a.right, b.right))
        return true;

    return false;
}

/* UTILITY FUNCTIONS */

/* Change a tree so that the roles of the left and
right pointers are swapped at every node.
See https://www.geeksforgeeks.org/?p=662 for details */
void mirror(Node node)
{
    if (node == null)
        return;
    else
    {
        Node temp;

        /* do the subtrees */
        mirror(node.left);
        mirror(node.right);

        /* swap the pointers in this node */
        temp = node.left;
        node.left = node.right;
        node.right = temp;
    }
}

/* Driver program to test above functions */
public static void main(String args[])
{
    BinaryTree tree = new BinaryTree();

    /* The constructed binary tree is
           1
         /   \
        2     3
          \   /
    */
}
```

```

        4  5
    */
tree.root = new Node(1);
tree.root.left = new Node(2);
tree.root.right = new Node(3);
tree.root.right.left = new Node(4);
tree.root.left.right = new Node(5);

if (tree.isFoldable(tree.root))
    System.out.println("tree is foldable");
else
    System.out.println("Tree is not foldable");
}

// This code has been contributed by Mayank Jaiswal

```

Time complexity: O(n)

### Method 2 (Check if Left and Right subtrees are Mirror)

There are mainly two functions:

// Checks if tree can be folded or not

```

IsFoldable(root)
1) If tree is empty then return true
2) Else check if left and right subtrees are structure wise mirrors of
   each other. Use utility function IsFoldableUtil(root->left,
   root->right) for this.

```

// Checks if n1 and n2 are mirror of each other.

```

IsFoldableUtil(n1, n2)
1) If both trees are empty then return true.
2) If one of them is empty and other is not then return false.
3) Return true if following conditions are met
   a) n1->left is mirror of n2->right
   b) n1->right is mirror of n2->left

```

C

```

#include<stdio.h>
#include<stdlib.h>

```

```
/* You would want to remove below 3 lines if your compiler
   supports bool, true and false */
#define bool int
#define true 1
#define false 0

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

/* A utility function that checks if trees with roots as n1 and n2
   are mirror of each other */
bool IsFoldableUtil(struct node *n1, struct node *n2);

/* Returns true if the given tree can be folded */
bool IsFoldable(struct node *root)
{
    if (root == NULL)
    {
        return true; }

    return IsFoldableUtil(root->left, root->right);
}

/* A utility function that checks if trees with roots as n1 and n2
   are mirror of each other */
bool IsFoldableUtil(struct node *n1, struct node *n2)
{
    /* If both left and right subtrees are NULL,
       then return true */
    if (n1 == NULL && n2 == NULL)
    {
        return true; }

    /* If one of the trees is NULL and other is not,
       then return false */
    if (n1 == NULL || n2 == NULL)
    {
        return false; }

    /* Otherwise check if left and right subtrees are mirrors of
       their counterparts */
    return IsFoldableUtil(n1->left, n2->right) &&
           IsFoldableUtil(n1->right, n2->left);
}
```

```

/*UTILITY FUNCTIONS */
/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* node = (struct node*)
                           malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;

    return(node);
}

/* Driver program to test mirror() */
int main(void)
{
    /* The constructed binary tree is
       1
      / \
     2   3
    \   /
     4 5
    */
    struct node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->right = newNode(4);
    root->right->left = newNode(5);

    if(IsFoldable(root) == true)
    { printf("\n tree is foldable"); }
    else
    { printf("\n tree is not foldable"); }

    getchar();
    return 0;
}

```

**Java**

```

// Java program to check foldable binary tree

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
class Node
{
    int data;

```

```
Node left, right;

Node(int item)
{
    data = item;
    left = right = null;
}
}

class BinaryTree
{
    Node root;

    /* Returns true if the given tree can be folded */
    boolean IsFoldable(Node node)
    {
        if (node == null)
            return true;

        return IsFoldableUtil(node.left, node.right);
    }

    /* A utility function that checks if trees with roots as n1 and n2
     * are mirror of each other */
    boolean IsFoldableUtil(Node n1, Node n2)
    {

        /* If both left and right subtrees are NULL,
         * then return true */
        if (n1 == null && n2 == null)
            return true;

        /* If one of the trees is NULL and other is not,
         * then return false */
        if (n1 == null || n2 == null)
            return false;

        /* Otherwise check if left and right subtrees are mirrors of
         * their counterparts */
        return IsFoldableUtil(n1.left, n2.right)
            && IsFoldableUtil(n1.right, n2.left);
    }

    /* Driver program to test above functions */
    public static void main(String args[])
    {
        BinaryTree tree = new BinaryTree();

        /* The constructed binary tree is

```

```
    1
   / \
  2   3
  \   /
   4  5
*/
tree.root = new Node(1);
tree.root.left = new Node(2);
tree.root.right = new Node(3);
tree.root.right.left = new Node(4);
tree.root.left.right = new Node(5);

if (tree.IsFoldable(tree.root))
    System.out.println("tree is foldable");
else
    System.out.println("Tree is not foldable");

}

}

// This code has been contributed by Mayank Jaiswal
```

Thanks to *Dzmitry Huba* for suggesting this approach.

## Source

<https://www.geeksforgeeks.org/foldable-binary-trees/>

## Chapter 196

# General Tree (Each node can have arbitrary number of children) Level Order Traversal

General Tree (Each node can have arbitrary number of children) Level Order Traversal - GeeksforGeeks

Given a generic tree, perform a Level order traversal and print all of its nodes

Input :                10  
          /    /      \    \  
          2    34      56    100  
         / \          |    / | \\  
         77   88      1    7   8   9

Output : 10  
2 34 56 100  
77 88 1 7 8 9

Input :                1  
          /    /      \    \  
          2    3      4    5  
         / \          |    / | \\  
         6    7      8    9   10   11  
Output : 1  
2 3 4 5  
6 7 8 9 10 11

The approach to this problem is similar to [Level Order traversal in a binary tree](#). We Start with pushing root node in a queue and for each node we pop it,print it and push all its child in the queue.

In case of a generic tree we store child nodes in a vector. Thus we put all elements of the vector in the queue.

```
// CPP program to do level order traversal
// of a generic tree
#include <bits/stdc++.h>
using namespace std;

// Represents a node of an n-ary tree
struct Node
{
    int key;
    vector<Node *> child;
};

// Utility function to create a new tree node
Node *newNode(int key)
{
    Node *temp = new Node;
    temp->key = key;
    return temp;
}

// Prints the n-ary tree level wise
void LevelOrderTraversal(Node * root)
{
    if (root==NULL)
        return;

    // Standard level order traversal code
    // using queue
    queue<Node *> q; // Create a queue
    q.push(root); // Enqueue root
    while (!q.empty())
    {
        int n = q.size();

        // If this node has children
        while (n > 0)
        {
            // Dequeue an item from queue and print it
            Node * p = q.front();
            q.pop();
            cout << p->key << " ";

            // Enqueue all children of the dequeued item
            for (int i=0; i<p->child.size(); i++)
                q.push(p->child[i]);
        }
    }
}
```

```
        n--;
    }

    cout << endl; // Print new line between two levels
}
}

// Driver program
int main()
{
    /* Let us create below tree
     *          10
     *         /   \
     *        2   34   56   100
     *        / \       |   /   |
     *      77 88       1   7   8   9
     */
    Node *root = newNode(10);
    (root->child).push_back(newNode(2));
    (root->child).push_back(newNode(34));
    (root->child).push_back(newNode(56));
    (root->child).push_back(newNode(100));
    (root->child[0]->child).push_back(newNode(77));
    (root->child[0]->child).push_back(newNode(88));
    (root->child[2]->child).push_back(newNode(1));
    (root->child[3]->child).push_back(newNode(7));
    (root->child[3]->child).push_back(newNode(8));
    (root->child[3]->child).push_back(newNode(9));

    cout << "Level order traversal Before Mirroring\n";
    LevelOrderTraversal(root);

    return 0;
}
```

Output:

```
10
2 34 56 100
77 88 1 7 8 9
```

## Source

<https://www.geeksforgeeks.org/generic-tree-level-order-traversal/>

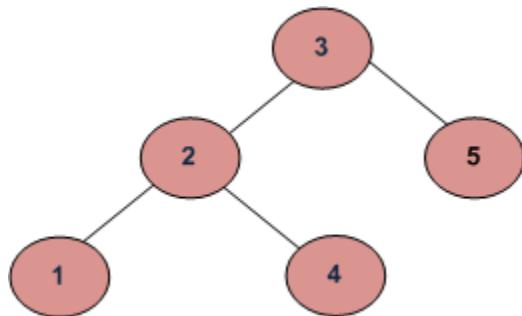
## Chapter 197

# Get Level of a node in a Binary Tree

Get Level of a node in a Binary Tree - GeeksforGeeks

Given a Binary Tree and a key, write a function that returns level of the key.

For example, consider the following tree. If the input key is 3, then your function should return 1. If the input key is 4, then your function should return 3. And for key which is not present in key, then your function should return 0.



The idea is to start from the root and level as 1. If the key matches with root's data, return level. Else recursively call for left and right subtrees with level as level + 1.

C

```
#include<stdio.h>

/* A tree node structure */
struct node
{
    int data;
    struct node *left;
    struct node *right;
```

```
};

/* Helper function for getLevel(). It returns level of the data if data is
   present in tree, otherwise returns 0.*/
int getLevelUtil(struct node *node, int data, int level)
{
    if (node == NULL)
        return 0;

    if (node->data == data)
        return level;

    int downlevel = getLevelUtil(node->left, data, level+1);
    if (downlevel != 0)
        return downlevel;

    downlevel = getLevelUtil(node->right, data, level+1);
    return downlevel;
}

/* Returns level of given data value */
int getLevel(struct node *node, int data)
{
    return getLevelUtil(node,data,1);
}

/* Utility function to create a new Binary Tree node */
struct node* newNode(int data)
{
    struct node *temp = new struct node;
    temp->data = data;
    temp->left = NULL;
    temp->right = NULL;

    return temp;
}

/* Driver function to test above functions */
int main()
{
    struct node *root = new struct node;
    int x;

    /* Constructing tree given in the above figure */
    root = newNode(3);
    root->left = newNode(2);
    root->right = newNode(5);
    root->left->left = newNode(1);
```

```
root->left->right = newNode(4);

for (x = 1; x <=5; x++)
{
    int level = getLevel(root, x);
    if (level)
        printf(" Level of %d is %d\n", x, getLevel(root, x));
    else
        printf(" %d is not present in tree \n", x);

}

getchar();
return 0;
}
```

**Java**

```
/* A tree node structure */
class Node
{
    int data;
    Node left, right;

    public Node(int d)
    {
        data = d;
        left = right = null;
    }
}

class BinaryTree
{

    Node root;

    /* Helper function for getLevel(). It returns level of the data
     * if data is present in tree, otherwise returns 0.*/
    int getLevelUtil(Node node, int data, int level)
    {
        if (node == null)
            return 0;

        if (node.data == data)
            return level;

        int downlevel = getLevelUtil(node.left, data, level + 1);
        if (downlevel != 0)
```

```
        return downlevel;

    downlevel = getLevelUtil(node.right, data, level + 1);
    return downlevel;
}

/* Returns level of given data value */
int getLevel(Node node, int data)
{
    return getLevelUtil(node, data, 1);
}

/* Driver function to test above functions */
public static void main(String[] args)
{
    BinaryTree tree = new BinaryTree();

    /* Constructing tree given in the above figure */
    tree.root = new Node(3);
    tree.root.left = new Node(2);
    tree.root.right = new Node(5);
    tree.root.left.left = new Node(1);
    tree.root.left.right = new Node(4);
    for (int x = 1; x <= 5; x++)
    {
        int level = tree.getLevel(tree.root, x);
        if (level != 0)
            System.out.println("Level of " + x + " is "
                               + tree.getLevel(tree.root, x));
        else
            System.out.println(x + " is not present in tree");
    }
}

// This code has been contributed by Mayank Jaiswal(mayank_24)
```

Output:

```
Level of 1 is 3
Level of 2 is 2
Level of 3 is 1
Level of 4 is 3
Level of 5 is 2
```

Time Complexity of getLevel() is O(n) where n is the number of nodes in the given Binary Tree.

**Source**

<https://www.geeksforgeeks.org/get-level-of-a-node-in-a-binary-tree/>

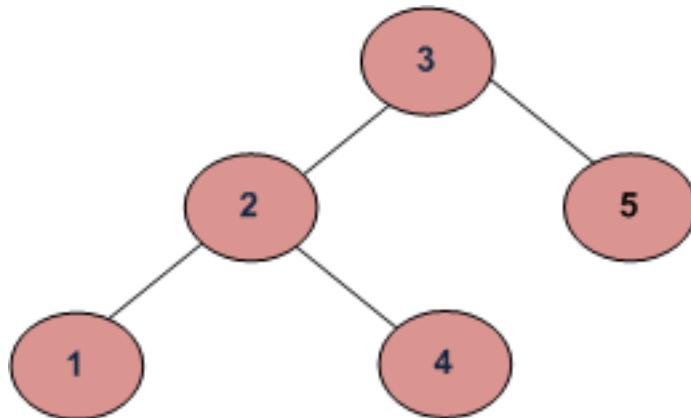
## Chapter 198

# Get level of a node in binary tree | iterative approach

Get level of a node in binary tree | iterative approach - GeeksforGeeks

Given a Binary Tree and a key, write a function that returns level of the key.

For example, consider the following tree. If the input key is 3, then your function should return 1. If the input key is 4, then your function should return 3. And for key which is not present in key, then your function should return 0.



Recursive approach to this problem is discussed here

<https://www.geeksforgeeks.org/get-level-of-a-node-in-a-binary-tree/>

The iterative approach is discussed below :

The iterative approach is modified version of [Level Order Tree Traversal Algorithm](#)

```
create a empty queue q
```

```
push root and then NULL to q
loop till q is not empty
    get the front node into temp node
    if it is NULL, it means all nodes of
        one level are traversed, so increment
        level
    else
        check if temp data is equal to data
        to be searched
        if yes then return level
        if temp->left is not NULL,
            enqueue temp->left
        if temp->right is not NULL,
            enqueue temp->right
if value not found, then return 0

// CPP program to print level of given node
// in binary tree iterative approach
/* Example binary tree
root is at level 1

          20
         /   \
        10   30
       / \   / \
      5  15 25  40
     /
    12 */

#include <iostream>
#include <queue>
using namespace std;

// node of binary tree
struct node {
    int data;
    node* left;
    node* right;
};

// utility function to create
// a new node
node* getnode(int data)
{
    node* newnode = new node();
    newnode->data = data;
    newnode->left = NULL;
    newnode->right = NULL;
}
```

```
// utility function to return level of given node
int getlevel(node* root, int data)
{
    queue<node*> q;
    int level = 1;
    q.push(root);

    // extra NULL is pushed to keep track
    // of all the nodes to be pushed before
    // level is incremented by 1
    q.push(NULL);
    while (!q.empty()) {
        node* temp = q.front();
        q.pop();
        if (temp == NULL) {
            if (q.front() != NULL) {
                q.push(NULL);
            }
            level += 1;
        } else {
            if (temp->data == data) {
                return level;
            }
            if (temp->left) {
                q.push(temp->left);
            }
            if (temp->right) {
                q.push(temp->right);
            }
        }
    }
    return 0;
}

int main()
{
    // create a binary tree
    node* root = getnode(20);
    root->left = getnode(10);
    root->right = getnode(30);
    root->left->left = getnode(5);
    root->left->right = getnode(15);
    root->left->right->left = getnode(12);
    root->right->left = getnode(25);
    root->right->right = getnode(40);

    // return level of node
```

```
int level = getlevel(root, 30);
(level != 0) ? (cout << "level of node 30 is " << level << endl) :
               (cout << "node 30 not found" << endl);

level = getlevel(root, 12);
(level != 0) ? (cout << "level of node 12 is " << level << endl) :
               (cout << "node 12 not found" << endl);

level = getlevel(root, 25);
(level != 0) ? (cout << "level of node 25 is " << level << endl) :
               (cout << "node 25 not found" << endl);

level = getlevel(root, 27);
(level != 0) ? (cout << "level of node 27 is " << level << endl) :
               (cout << "node 27 not found" << endl);
return 0;
}
```

Output:

```
level of node 30 is 2
level of node 12 is 4
level of node 25 is 3
node 27 not found
```

## Source

<https://www.geeksforgeeks.org/get-level-node-binary-tree-iterative-approach/>

## Chapter 199

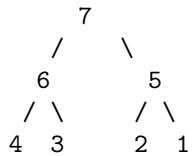
# Get maximum left node in binary tree

Get maximum left node in binary tree - GeeksforGeeks

Given a tree, the task is to find the maximum in an only left node of the binary tree.

Examples:

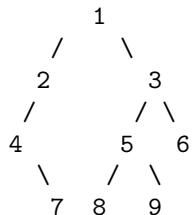
Input :



Output :

6

Input :



Output :

8

Traverse with inorder traversal and Apply the condition for the left node only and get maximum of left node.

Let's try to understand with code.

```
// CPP program to print maximum element
// in left node.
#include <bits/stdc++.h>
using namespace std;

// A Binary Tree Node
struct Node {
    int data;
    struct Node *left, *right;
};

// Get max of left element using
// Inorder traversal
int maxOfLeftElement(Node* root)
{
    int res = INT_MIN;
    if (root == NULL)
        return res;

    if (root->left != NULL)
        res = root->left->data;

    // Return maximum of three values
    // 1) Recursive max in left subtree
    // 2) Value in left node
    // 3) Recursive max in right subtree
    return max({ maxOfLeftElement(root->left),
                res,
                maxOfLeftElement(root->right) });
}

// Utility function to create a new tree node
Node* newNode(int data)
{
    Node* temp = new Node;
    temp->data = data;
    temp->left = temp->right = NULL;
    return temp;
}

// Driver program to test above functions
int main()
{
    // Let us create binary tree shown in above diagram
    Node* root = newNode(7);
    root->left = newNode(6);
    root->right = newNode(5);
    root->left->left = newNode(4);
```

```
root->left->right = newNode(3);
root->right->left = newNode(2);
root->right->right = newNode(1);

/*
      7
     /   \
    6     5
   / \   / \
  4  3   2  1
*/
cout << maxOfLeftElement(root);
return 0;
}
```

**Output:**

6

**Source**

<https://www.geeksforgeeks.org/get-maximum-left-node-binary-tree/>

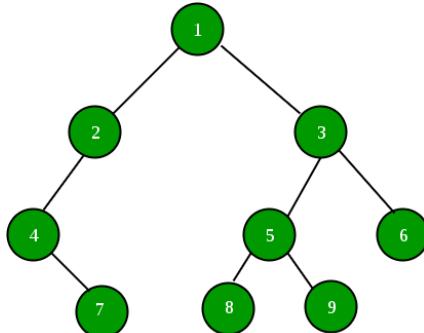
## Chapter 200

# Given a binary tree, how do you remove all the half nodes?

Given a binary tree, how do you remove all the half nodes? - GeeksforGeeks

Given A binary Tree, how do you remove all the half nodes (which has only one child)?  
Note leaves should not be touched as they have both children as NULL.

For example consider the below tree.



Nodes 2 and 4 are half nodes as one of their child is Null.

The idea is to use post-order traversal to solve this problem efficiently. We first process the left children, then right children, and finally the node itself. So we form the new tree bottom up, starting from the leaves towards the root. By the time we process the current node, both its left and right subtrees were already processed. Below is the implementation of this idea.

C

```
// C program to remove all half nodes
#include <stdio.h>
#include <stdlib.h>
```

```
struct node
{
    int data;
    struct node* left, *right;
};

struct node* newNode(int data)
{
    struct node* node = (struct node*)
        malloc(sizeof(struct node));
    node->data = data;
    node->left = node->right = NULL;
    return(node);
}

void printInoder(struct node*root)
{
    if (root != NULL)
    {
        printInoder(root->left);
        printf("%d ",root->data);
        printInoder(root->right);
    }
}

// Removes all nodes with only one child and returns
// new root (note that root may change)
struct node* RemoveHalfNodes(struct node* root)
{
    if (root==NULL)
        return NULL;

    root->left  = RemoveHalfNodes(root->left);
    root->right = RemoveHalfNodes(root->right);

    if (root->left==NULL && root->right==NULL)
        return root;

    /* if current nodes is a half node with left
       child NULL left, then it's right child is
       returned and replaces it in the given tree */
    if (root->left==NULL)
    {
        struct node *new_root = root->right;
        free(root); // To avoid memory leak
        return new_root;
    }
}
```

```
/* if current nodes is a half node with right
   child NULL right, then it's right child is
   returned and replaces it in the given tree */
if (root->right==NULL)
{
    struct node *new_root = root->left;
    free(root); // To avoid memory leak
    return new_root;
}

return root;
}

// Driver program
int main(void)
{
    struct node*NewRoot=NULL;
    struct node *root = newNode(2);
    root->left      = newNode(7);
    root->right     = newNode(5);
    root->left->right = newNode(6);
    root->left->right->left=newNode(1);
    root->left->right->right=newNode(11);
    root->right->right=newNode(9);
    root->right->right->left=newNode(4);

    printf("Inorder traversal of given tree \n");
    printInoder(root);

    NewRoot = RemoveHalfNodes(root);

    printf("\nInorder traversal of the modified tree \n");
    printInoder(NewRoot);
    return 0;
}
```

### Java

```
// Java program to remove half nodes
class Node
{
    int data;
    Node left, right;

    Node(int item)
    {
        data = item;
```

```
        left = right = null;
    }
}

class BinaryTree
{
    Node root;

    void printInorder(Node node)
    {
        if (node != null)
        {
            printInorder(node.left);
            System.out.print(node.data + " ");
            printInorder(node.right);
        }
    }

    // Removes all nodes with only one child and returns
    // new root (note that root may change)
    Node RemoveHalfNodes(Node node)
    {
        if (node == null)
            return null;

        node.left = RemoveHalfNodes(node.left);
        node.right = RemoveHalfNodes(node.right);

        if (node.left == null && node.right == null)
            return node;

        /* if current nodes is a half node with left
           child NULL left, then it's right child is
           returned and replaces it in the given tree */
        if (node.left == null)
        {
            Node new_root = node.right;
            return new_root;
        }

        /* if current nodes is a half node with right
           child NULL right, then it's left child is
           returned and replaces it in the given tree */
        if (node.right == null)
        {
            Node new_root = node.left;
            return new_root;
        }
    }
}
```

```
        return node;
    }

// Driver program
public static void main(String args[])
{
    BinaryTree tree = new BinaryTree();
    Node NewRoot = null;
    tree.root = new Node(2);
    tree.root.left = new Node(7);
    tree.root.right = new Node(5);
    tree.root.left.right = new Node(6);
    tree.root.left.right.left = new Node(1);
    tree.root.left.right.right = new Node(11);
    tree.root.right.right = new Node(9);
    tree.root.right.right.left = new Node(4);

    System.out.println("the inorder traversal of tree is ");
    tree.printInorder(tree.root);

    NewRoot = tree.RemoveHalfNodes(tree.root);

    System.out.print("\nInorder traversal of the modified tree \n");
    tree.printInorder(NewRoot);
}
}

// This code has been contributed by Mayank Jaiswal
```

### Python

```
# Python program to remove all half nodes

# A binary tree node
class Node:
    # Constructor for creating a new node
    def __init__(self , data):
        self.data = data
        self.left = None
        self.right = None

    # For inorder traversal
def printInorder(root):
    if root is not None:
        printInorder(root.left)
        print root.data,
        printInorder(root.right)
```

```
# Removes all nodes with only one child and returns
# new root(note that root may change)
def RemoveHalfNodes(root):
    if root is None:
        return None

    # Recur to left tree
    root.left = RemoveHalfNodes(root.left)

    # Recur to right tree
    root.right = RemoveHalfNodes(root.right)

    # if both left and right child is None
    # the node is not a Half node
    if root.left is None and root.right is None:
        return root

    # If current nodes is a half node with left child
    # None then it's right child is returned and
    # replaces it in the given tree
    if root.left is None:
        new_root = root.right
        temp = root
        root = None
        del(temp)
        return new_root

    if root.right is None:
        new_root = root.left
        temp = root
        root = None
        del(temp)
        return new_root

    return root

# Driver Program
root = Node(2)
root.left = Node(7)
root.right = Node(5)
root.left.right = Node(6)
root.left.right.left = Node(1)
root.left.right.right = Node(11)
root.right.right = Node(9)
root.right.right.left = Node(4)

print "Inorder traversal of given tree"
```

```
printInorder(root)  
  
NewRoot = RemoveHalfNodes(root)  
  
print "\nInorder traversal of the modified tree"  
printInorder(NewRoot)  
  
# This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

Output:

```
Inorder traversal of given tree  
7 1 6 11 2 5 4 9  
Inorder traversal of the modified tree  
1 6 11 2 4
```

Time complexity of the above solution is O(n) as it does a simple traversal of binary tree.

This article is contributed by **Jyoti Saini**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## Source

<https://www.geeksforgeeks.org/given-a-binary-tree-how-do-you-remove-all-the-half-nodes/>

# Chapter 201

## Given a binary tree, print all root-to-leaf paths

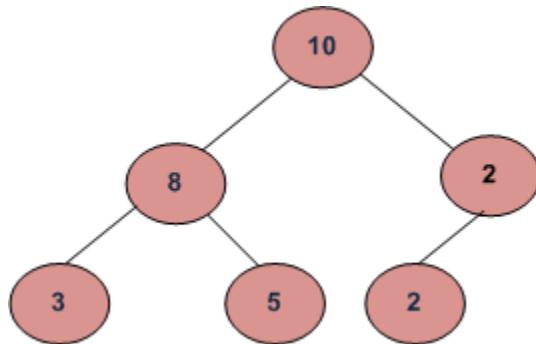
Given a binary tree, print all root-to-leaf paths - GeeksforGeeks

For the below example tree, all root-to-leaf paths are:

10 -> 8 -> 3

10 -> 8 -> 5

10 -> 2 -> 2



Algorithm:

Use a path array path[] to store current root to leaf path. Traverse from root to all leaves in top-down fashion. While traversing, store data of all nodes in current path in array path[]. When we reach a leaf node, print the path array.

C

```
#include<stdio.h>
#include<stdlib.h>

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
```

```
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

/* Prototypes for functions needed in printPaths() */
void printPathsRecur(struct node* node, int path[], int pathLen);
void printArray(int ints[], int len);

/* Given a binary tree, print out all of its root-to-leaf
   paths, one per line. Uses a recursive helper to do the work.*/
void printPaths(struct node* node)
{
    int path[1000];
    printPathsRecur(node, path, 0);
}

/* Recursive helper function -- given a node, and an array containing
   the path from the root node up to but not including this node,
   print out all the root-leaf paths.*/
void printPathsRecur(struct node* node, int path[], int pathLen)
{
    if (node==NULL)
        return;

    /* append this node to the path array */
    path[pathLen] = node->data;
    pathLen++;

    /* it's a leaf, so print the path that led to here */
    if (node->left==NULL && node->right==NULL)
    {
        printArray(path, pathLen);
    }
    else
    {
        /* otherwise try both subtrees */
        printPathsRecur(node->left, path, pathLen);
        printPathsRecur(node->right, path, pathLen);
    }
}

/* UTILITY FUNCTIONS */
/* Utility that prints out an array on a line. */
void printArray(int ints[], int len)
```

```
{  
    int i;  
    for (i=0; i<len; i++)  
    {  
        printf("%d ", ints[i]);  
    }  
    printf("\n");  
}  
  
/* utility that allocates a new node with the  
   given data and NULL left and right pointers. */  
struct node* newnode(int data)  
{  
    struct node* node = (struct node*)  
                      malloc(sizeof(struct node));  
    node->data = data;  
    node->left = NULL;  
    node->right = NULL;  
  
    return(node);  
}  
  
/* Driver program to test above functions*/  
int main()  
{  
  
    /* Constructed binary tree is  
       10  
      /   \  
     8     2  
    / \   /  
   3   5  2  
 */  
    struct node *root = newnode(10);  
    root->left = newnode(8);  
    root->right = newnode(2);  
    root->left->left = newnode(3);  
    root->left->right = newnode(5);  
    root->right->left = newnode(2);  
  
    printPaths(root);  
  
    getchar();  
    return 0;  
}
```

Java

```
// Java program to print all the node to leaf path

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
class Node
{
    int data;
    Node left, right;

    Node(int item)
    {
        data = item;
        left = right = null;
    }
}

class BinaryTree
{
    Node root;

    /*Given a binary tree, print out all of its root-to-leaf
     paths, one per line. Uses a recursive helper to do
     the work.*/
    void printPaths(Node node)
    {
        int path[] = new int[1000];
        printPathsRecur(node, path, 0);
    }

    /* Recursive helper function -- given a node, and an array
       containing the path from the root node up to but not
       including this node, print out all the root-leaf paths.*/
    void printPathsRecur(Node node, int path[], int pathLen)
    {
        if (node == null)
            return;

        /* append this node to the path array */
        path[pathLen] = node.data;
        pathLen++;

        /* it's a leaf, so print the path that led to here */
        if (node.left == null && node.right == null)
            printArray(path, pathLen);
        else
        {
            /* otherwise try both subtrees */
            printPathsRecur(node.left, path, pathLen);
            printPathsRecur(node.right, path, pathLen);
        }
    }
}
```

```
        printPathsRecur(node.right, path, pathLen);
    }
}

/* Utility function that prints out an array on a line. */
void printArray(int ints[], int len)
{
    int i;
    for (i = 0; i < len; i++)
    {
        System.out.print(ints[i] + " ");
    }
    System.out.println("");
}

// driver program to test above functions
public static void main(String args[])
{
    BinaryTree tree = new BinaryTree();
    tree.root = new Node(10);
    tree.root.left = new Node(8);
    tree.root.right = new Node(2);
    tree.root.left.left = new Node(3);
    tree.root.left.right = new Node(5);
    tree.root.right.left = new Node(2);

    /* Let us test the built tree by printing Inorder traversal */
    tree.printPaths(tree.root);
}
}

// This code has been contributed by Mayank Jaiswal
```

### Python3

```
"""
Python program to print all path from root to
leaf in a binary tree
"""

# binary tree node contains data field ,
# left and right pointer
class Node:
    # constructor to create tree node
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None
```

```
# function to print all path from root
# to leaf in binary tree
def printPaths(root):
    # list to store path
    path = []
    printPathsRec(root, path, 0)

# Helper function to print path from root
# to leaf in binary tree
def printPathsRec(root, path, pathLen):

    # Base condition - if binary tree is
    # empty return
    if root is None:
        return

    # add current root's data into
    # path_ar list

    # if length of list is gre
    if(len(path) > pathLen):
        path[pathLen] = root.data
    else:
        path.append(root.data)

    # increment pathLen by 1
    pathLen = pathLen + 1

    if root.left is None and root.right is None:

        # leaf node then print the list
        printArray(path, pathLen)
    else:
        # try for left and right subtree
        printPathsRec(root.left, path, pathLen)
        printPathsRec(root.right, path, pathLen)

# Helper function to print list in which
# root-to-leaf path is stored
def printArray(ints, len):
    for i in ints[0 : len]:
        print(i, ",end="")
    print()

# Driver program to test above function
"""
Constructed binary tree is
```

```
          10
         / \
        8   2
       / \ /
      3 5 2
"""
root = Node(10)
root.left = Node(8)
root.right = Node(2)
root.left.left = Node(3)
root.left.right = Node(5)
root.right.left = Node(2)
printPaths(root)

# This code has been contributed by Shweta Singh.
```

Output :

```
10 8 3
10 8 5
10 2 2
```

Time Complexity:  $O(n^2)$  where n is number of nodes.

References:

<http://cslibrary.stanford.edu/110/BinaryTrees.html>

**Improved By :** shweta44

## Source

<https://www.geeksforgeeks.org/given-a-binary-tree-print-all-root-to-leaf-paths/>

## Chapter 202

**Given a binary tree, print out all of its root-to-leaf paths one per line.**

Given a binary tree, print out all of its root-to-leaf paths one per line. - GeeksforGeeks

Asked by Varun Bhatia

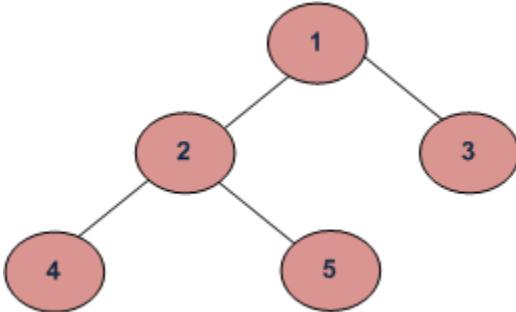
Here is the solution.

**Algorithm:**

```
initialize: pathlen = 0, path[1000]
/*1000 is some max limit for paths, it can change*/

/*printPathsRecur traverses nodes of tree in preorder */
printPathsRecur(tree, path[], pathlen)
    1) If node is not NULL then
        a) push data to path array:
            path[pathlen] = node->data.
        b) increment pathlen
            pathlen++
    2) If node is a leaf node then print the path array.
    3) Else
        a) Call printPathsRecur for left subtree
            printPathsRecur(node->left, path, pathLen)
        b) Call printPathsRecur for right subtree.
            printPathsRecur(node->right, path, pathLen)
```

**Example:**



Example Tree

Output for the above example will be

```
1 2 4
1 2 5
1 3
```

**Implementation:**

C

```
/*program to print all of its root-to-leaf paths for a tree*/
#include <stdio.h>
#include <stdlib.h>

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

void printArray(int [], int);
void printPathsRecur(struct node*, int [], int);
struct node* newNode(int );
void printPaths(struct node*);

/* Given a binary tree, print out all of its root-to-leaf
   paths, one per line. Uses a recursive helper to do the work.*/
void printPaths(struct node* node)
{
    int path[1000];
```

```
printPathsRecur(node, path, 0);
}

/* Recursive helper function -- given a node, and an array containing
   the path from the root node up to but not including this node,
   print out all the root-leaf paths. */
void printPathsRecur(struct node* node, int path[], int pathLen)
{
    if (node==NULL) return;

    /* append this node to the path array */
    path[pathLen] = node->data;
    pathLen++;

    /* it's a leaf, so print the path that led to here */
    if (node->left==NULL && node->right==NULL)
    {
        printArray(path, pathLen);
    }
    else
    {
        /* otherwise try both subtrees */
        printPathsRecur(node->left, path, pathLen);
        printPathsRecur(node->right, path, pathLen);
    }
}

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* node = (struct node*)
                           malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;

    return(node);
}

/* Utility that prints out an array on a line */
void printArray(int ints[], int len)
{
    int i;
    for (i=0; i<len; i++) {
        printf("%d ", ints[i]);
    }
    printf("\n");
}
```

```
}

/* Driver program to test mirror() */
int main()
{
    struct node *root = newNode(1);
    root->left      = newNode(2);
    root->right     = newNode(3);
    root->left->left  = newNode(4);
    root->left->right = newNode(5);

    /* Print all root-to-leaf paths of the input tree */
    printPaths(root);

    getchar();
    return 0;
}
```

**Java**

```
// Java program to print all root to leaf paths

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
class Node
{
    int data;
    Node left, right;

    Node(int item)
    {
        data = item;
        left = right = null;
    }
}

class BinaryTree
{
    Node root;

    /* Given a binary tree, print out all of its root-to-leaf
       paths, one per line. Uses a recursive helper to do the work.*/
    void printPaths(Node node)
    {
        int path[] = new int[1000];
        printPathsRecur(node, path, 0);
    }
}
```

```
/* Recursive helper function -- given a node, and an array containing
   the path from the root node up to but not including this node,
   print out all the root-leaf paths. */
void printPathsRecur(Node node, int path[], int pathLen)
{
    if (node == null)
        return;

    /* append this node to the path array */
    path[pathLen] = node.data;
    pathLen++;

    /* it's a leaf, so print the path that led to here */
    if (node.left == null && node.right == null)
        printArray(path, pathLen);
    else
    {
        /* otherwise try both subtrees */
        printPathsRecur(node.left, path, pathLen);
        printPathsRecur(node.right, path, pathLen);
    }
}

/* Utility that prints out an array on a line */
void printArray(int ints[], int len)
{
    int i;
    for (i = 0; i < len; i++)
        System.out.print(ints[i] + " ");
    System.out.println("");
}

/* Driver program to test all above functions */
public static void main(String[] args)
{
    BinaryTree tree = new BinaryTree();

    tree.root = new Node(1);
    tree.root.left = new Node(2);
    tree.root.right = new Node(3);
    tree.root.left.left = new Node(4);
    tree.root.left.right = new Node(5);

    /* Print all root-to-leaf paths of the input tree */
    tree.printPaths(tree.root);
}
```

### Python3

```
# Program to print all of its
# root-to-leaf paths for a tree
class Node:

    # A binary tree node has data,
    # pointer to left child and a
    # pointer to right child
    def __init__(self, data):
        self.data = data
        self.right = None
        self.left = None

def printRoute(stack, root):
    if root == None:
        return

    # append this node to the path array
    stack.append(root.data)
    if(root.left == None and root.right == None):

        # print out all of its
        # root - to - leaf
        print(' '.join([str(i) for i in stack]))

    # otherwise try both subtrees
    printRoute(stack, root.left)
    printRoute(stack, root.right)
    stack.pop()

# Driver Code
root = Node(1);
root.left = Node(2);
root.right = Node(3);
root.left.left = Node(4);
root.left.right = Node(5);
printRoute([], root)

# This code is contributed
# by Farheen Nilofer
```

### Output:

```
1 2 4
1 2 5
1 3
```

*Chapter 202. Given a binary tree, print out all of its root-to-leaf paths one per line.*

---

**References:**

<http://cslibrary.stanford.edu/110/BinaryTrees.html>

**Improved By :** Farheen Nilofer

**Source**

<https://www.geeksforgeeks.org/given-a-binary-tree-print-out-all-of-its-root-to-leaf-paths-one-per-line/>

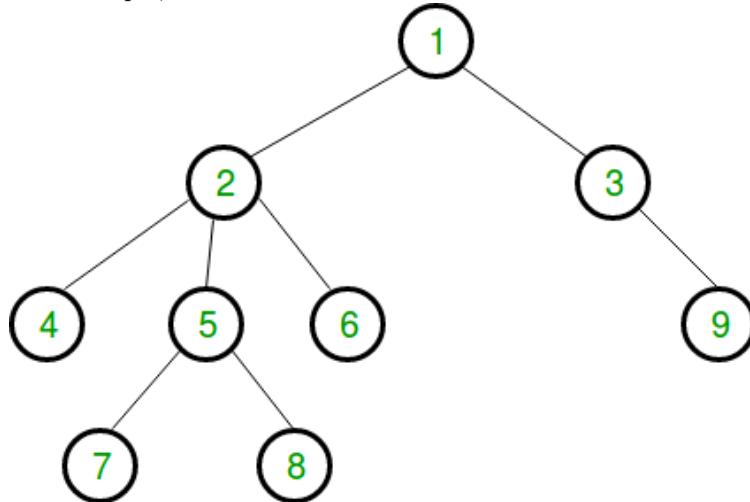
## Chapter 203

**Given a n-ary tree, count number of nodes which have more number of children than parents**

Given a n-ary tree, count number of nodes which have more number of children than parents  
- GeeksforGeeks

Given a N-ary tree represented as adjacency list, we need to write a program to count all such nodes in this tree which has more number of children than its parent.

For Example,



In the above tree, count will be 1 as there is only one such node which is '2' which has more number of children than its parent. 2 has three children (4, 5 and 6) where as its parent, 1

has only two childrens (2 and 3).

We can solve this problem using both **BFS** and **DFS** algorithms. We will explain here in details about how to solve this problem using BFS algorithm.

As the tree is represented using adjacency list representation. So, for any node say ‘u’ the number of children of this node can be given as `adj[u].size()`.

Now the idea is to apply BFS on the given tree and while traversing the children of a node ‘u’ say ‘v’ we will simply check is `adj[v].size() > adj[u].size()`.

Below is the C++ implementation of above idea:

```
// C++ program to count number of nodes
// which has more children than its parent

#include<bits/stdc++.h>
using namespace std;

// function to count number of nodes
// which has more children than its parent
int countNodes(vector<int> adj[], int root)
{
    int count = 0;

    // queue for applying BFS
    queue<int> q;

    // BFS algorithm
    q.push(root);

    while (!q.empty())
    {
        int node = q.front();
        q.pop();

        // traverse children of node
        for( int i=0;i<adj[node].size();i++)
        {
            // children of node
            int children = adj[node][i];

            // if number of childs of children
            // is greater than number of childs
            // of node, then increment count
            if (adj[children].size() > adj[node].size())
                count++;
            q.push(children);
        }
    }
}
```

```
    return count;
}

// Driver program to test above functions
int main()
{
    // adjacency list for n-ary tree
    vector<int> adj[10];

    // construct n ary tree as shown
    // in above diagram
    adj[1].push_back(2);
    adj[1].push_back(3);
    adj[2].push_back(4);
    adj[2].push_back(5);
    adj[2].push_back(6);
    adj[3].push_back(9);
    adj[5].push_back(7);
    adj[5].push_back(8);

    int root = 1;

    cout << countNodes(adj, root);
    return 0;
}
```

Output:

1

**Time Complexity:** O( n ) , where n is the number of nodes in the tree.

## Source

<https://www.geeksforgeeks.org/given-n-ary-tree-count-number-nodes-number-children-parent/>

## Chapter 204

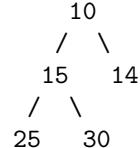
# Given level order traversal of a Binary Tree, check if the Tree is a Min-Heap

Given level order traversal of a Binary Tree, check if the Tree is a Min-Heap - GeeksforGeeks

Given the level order traversal of a [Complete Binary Tree](#), determine whether the Binary Tree is a valid [Min-Heap](#)

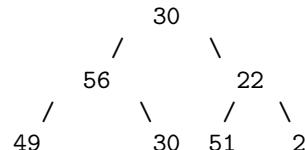
Examples:

```
Input : level = [10, 15, 14, 25, 30]
Output : True
The tree of the given level order traversal is
```



We see that each parent has a value less than its child, and hence satisfies the min-heap property

```
Input : level = [30, 56, 22, 49, 30, 51, 2, 67]
Output : False
The tree of the given level order traversal is
```



/

67

We observe that at level 0,  $30 > 22$ , and hence min-heap property is not satisfied

We need to check whether each non-leaf node (parent) satisfies the [heap property](#). For this, we check whether each parent (at index i) is smaller than its children (at indices  $2*i+1$  and  $2*i+2$ , if the parent has two children). If only one child, we only check the parent against index  $2*i+1$ .

C++

```
// C++ program to check if a given tree is
// Binary Heap or not
#include <bits/stdc++.h>
using namespace std;

// Returns true if given level order traversal
// is Min Heap.
bool isMinHeap(int level[], int n)
{
    // First non leaf node is at index (n/2-1).
    // Check whether each parent is greater than child
    for (int i=(n/2-1) ; i>=0 ; i--)
    {
        // Left child will be at index 2*i+1
        // Right child will be at index 2*i+2
        if (level[i] > level[2 * i + 1])
            return false;

        if (2*i + 2 < n)
        {
            // If parent is greater than right child
            if (level[i] > level[2 * i + 2])
                return false;
        }
    }
    return true;
}

// Driver code
int main()
{
    int level[] = {10, 15, 14, 25, 30};
    int n = sizeof(level)/sizeof(level[0]);
    if (isMinHeap(level, n))
        cout << "True";
    else
```

```
        cout << "False";
    return 0;
}
```

**Java**

```
// Java program to check if a given tree is
// Binary Heap or not
import java.io.*;
import java.util.*;

public class detheap
{
    // Returns true if given level order traversal
    // is Min Heap.
    static boolean isMinHeap(int []level)
    {
        int n = level.length - 1;

        // First non leaf node is at index (n/2-1).
        // Check whether each parent is greater than child
        for (int i=(n/2-1) ; i>=0 ; i--)
        {
            // Left child will be at index 2*i+1
            // Right child will be at index 2*i+2
            if (level[i] > level[2 * i + 1])
                return false;

            if (2*i + 2 < n)
            {
                // If parent is greater than right child
                if (level[i] > level[2 * i + 2])
                    return false;
            }
        }
        return true;
    }

    // Driver code
    public static void main(String[] args)
                    throws IOException
    {
        // Level order traversal
        int[] level = new int[]{10, 15, 14, 25, 30};

        if (isMinHeap(level))
            System.out.println("True");
        else
```

Chapter 204. Given level order traversal of a Binary Tree, check if the Tree is a Min-Heap

```
        System.out.println("False");
    }
}
```

Output:

True

These algorithms run with worse case **O(n)** complexity

### Source

<https://www.geeksforgeeks.org/given-level-order-traversal-binary-tree-check-tree-min-heap/>

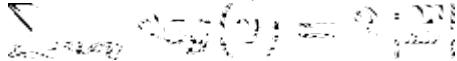
## Chapter 205

# Handshaking Lemma and Interesting Tree Properties

Handshaking Lemma and Interesting Tree Properties - GeeksforGeeks

### What is Handshaking Lemma?

[Handshaking lemma](#) is about undirected graph. In every finite undirected graph number of vertices with odd degree is always even. The handshaking lemma is a consequence of the degree sum formula (also sometimes called the handshaking lemma)



### How is Handshaking Lemma useful in Tree Data structure?

Following are some interesting facts that can be proved using Handshaking lemma.

- 1) *In a k-ary tree where every node has either 0 or k children, following property is always true.*

$$L = (k - 1)*I + 1$$

Where L = Number of leaf nodes  
I = Number of internal nodes

### Proof:

Proof can be divided in two cases.

**Case 1** (Root is Leaf): There is only one node in tree. The above formula is true for single node as  $L = 1$ ,  $I = 0$ .

**Case 2** (Root is Internal Node): For trees with more than 1 nodes, root is always internal node. The above formula can be proved using Handshaking Lemma for this case. A tree is an undirected acyclic graph.

Total number of edges in Tree is number of nodes minus 1, i.e.,  $|E| = L + I - 1$ .

All internal nodes except root in the given type of tree have degree  $k + 1$ . Root has degree  $k$ . All leaves have degree 1. Applying the Handshaking lemma to such trees, we get following relation.

$$\text{Sum of all degrees} = 2 * (\text{Sum of Edges})$$

$$\begin{aligned} & \text{Sum of degrees of leaves} + \\ & \text{Sum of degrees for Internal Node except root} + \\ & \text{Root's degree} = 2 * (\text{No. of nodes} - 1) \end{aligned}$$

$$\begin{aligned} & \text{Putting values of above terms,} \\ & L + (I-1)*(k+1) + k = 2 * (L + I - 1) \\ & L + k*I - k + I - 1 + k = 2*L + 2*I - 2 \\ & L + K*I + I - 1 = 2*L + 2*I - 2 \\ & K*I + 1 - I = L \\ & (K-1)*I + 1 = L \end{aligned}$$

So the above property is proved using Handshaking Lemma, let us discuss one more interesting property.

**2) In Binary tree, number of leaf nodes is always one more than nodes with two children.**

$$\begin{aligned} L &= T + 1 \\ \text{Where } L &= \text{Number of leaf nodes} \\ T &= \text{Number of internal nodes with two children} \end{aligned}$$

**Proof:**

Let number of nodes with 2 children be  $T$ . Proof can be divided in three cases.

**Case 1:** There is only one node, the relationship holds as  $T = 0, L = 1$ .

**Case 2:** Root has two children, i.e., degree of root is 2.

$$\begin{aligned} & \text{Sum of degrees of nodes with two children except root} + \\ & \text{Sum of degrees of nodes with one child} + \\ & \text{Sum of degrees of leaves} + \text{Root's degree} = 2 * (\text{No. of Nodes} - 1) \end{aligned}$$

$$\begin{aligned} & \text{Putting values of above terms,} \\ & (T-1)*3 + S*2 + L + 2 = (S + T + L - 1)*2 \end{aligned}$$

Cancelling 2S from both sides.

$$(T-1)*3 + L + 2 = (S + L - 1)*2$$

$$T - 1 = L - 2$$

$$T = L - 1$$

**Case 3:** Root has one child, i.e., degree of root is 1.

Sum of degrees of nodes with two children +

Sum of degrees of nodes with one child except root +

Sum of degrees of leaves + Root's degree = 2 \* (No. of Nodes - 1)

Putting values of above terms,

$$T*3 + (S-1)*2 + L + 1 = (S + T + L - 1)*2$$

Cancelling 2S from both sides.

$$3*T + L - 1 = 2*T + 2*L - 2$$

$$T - 1 = L - 2$$

$$T = L - 1$$

Therefore, in all three cases, we get  $T = L-1$ .

We have discussed proof of two important properties of Trees using Handshaking Lemma. Many GATE questions have been asked on these properties, following are few links.

[GATE-CS-2015 \(Set 3\) | Question 35](#)

[GATE-CS-2015 \(Set 2\) | Question 20](#)

[GATE-CS-2005 | Question 36](#)

[GATE-CS-2002 | Question 34](#)

[GATE-CS-2007 | Question 43](#)

## Source

<https://www.geeksforgeeks.org/handshaking-lemma-and-interesting-tree-properties/>

## Chapter 206

# HashSet vs TreeSet in Java

HashSet vs TreeSet in Java - GeeksforGeeks

- **Speed and internal implementation**

[HashSet](#) : For operations like search, insert and delete. It takes constant time for these operations on average. HashSet is faster than TreeSet. HashSet is Implemented using a [hash table](#).

[TreeSet](#) : TreeSet takes  $O(\log n)$  for search, insert and delete which is higher than HashSet. But TreeSet keeps sorted data. Also, it supports operations like higher() (Returns least higher element), floor(), ceiling(), etc. These operations are also  $O(\log n)$  in TreeSet and not supported in HashSet. TreeSet is implemented using a Self Balancing Binary Search Tree ([Red-Black Tree](#)). TreeSet is backed by TreeMap in Java.

- **Ordering**

Elements in HashSet are not ordered. TreeSet maintains objects in Sorted order defined by either Comparable or Comparator method in Java. TreeSet elements are sorted in ascending order by default. It offers several methods to deal with the ordered set like first(), last(), headSet(), tailSet(), etc.

- **Null Object**

HashSet allows null object. TreeSet doesn't allow null Object and throw NullPointerException, Why, because TreeSet uses compareTo() method to compare keys and compareTo() will throw `java.lang.NullPointerException`.

- **Comparison**

HashSet uses [equals\(\)](#) method to compare two object in Set and for detecting duplicates. TreeSet uses compareTo() method for same purpose.

If equals() and compareTo() are not consistent, i.e. for two equal object equals should return true while compareTo() should return zero, than it will break contract of Set interface and will allow duplicates in Set implementations like TreeSet

If you want a sorted Set then it is better to add elements to HashSet and then convert it into TreeSet rather than creating a TreeSet and adding elements to it.

### HashSet example

```
// Java program to demonstrate working of
// HashSet
import java.util.HashSet;
class HashSetDemo {
    public static void main(String[] args)
    {

        // Create a HashSet
        HashSet<String> hset = new HashSet<String>();

        // add elements to HashSet
        hset.add("geeks");
        hset.add("for");
        hset.add("practice");
        hset.add("contribute");

        // Duplicate removed
        hset.add("geeks");

        // Displaying HashSet elements
        System.out.println("HashSet contains: ");
        for (String temp : hset) {
            System.out.println(temp);
        }
    }
}
```

#### Output:

```
HashSet contains:
practice
geeks
for
contribute
```

### TreeSet example

```
// Java program to demonstrate working of
// TreeSet.
import java.util.TreeSet;
class TreeSetDemo {

    public static void main(String[] args)
    {
```

```
// Create a TreeSet
TreeSet<String> tset = new TreeSet<String>();

// add elements to HashSet
tset.add("geeks");
tset.add("for");
tset.add("practice");
tset.add("contribute");

// Duplicate removed
tset.add("geeks");

// Displaying TreeSet elements
System.out.println("TreeSet contains: ");
for (String temp : tset) {
    System.out.println(temp);
}
```

**Output:**

```
TreeSet contains:
contribute
for
geeks
practice
```

**When to prefer TreeSet over HashSet**

1. Sorted unique elements are required instead of unique elements. The sorted list given by TreeSet is always in ascending order.
2. TreeSet has greater locality than HashSet. If two entries are near by in the order, then TreeSet places them near each other in data structure and hence in memory, while HashSet spreads the entries all over memory regardless of the keys they are associated to.
3. TreeSet uses Red- Black tree algorithm underneath to sort out the elements. When one need to perform read/write operations frequently, then TreeSet is a good choice.
4. [LinkedHashSet](#) is another data structure that is between these two. It provides time complexities like HashSet and maintains order of insertion (Note that this is not sorted order, but the order in which elements are inserted).

**Source**

<https://www.geeksforgeeks.org/hashset-vs-treeset-in-java/>

## Chapter 207

# Height of a complete binary tree (or Heap) with N nodes

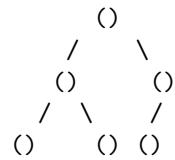
Height of a complete binary tree (or Heap) with N nodes - GeeksforGeeks

Consider a [Binary Heap](#) of size N. We need to find height of it.

Examples :

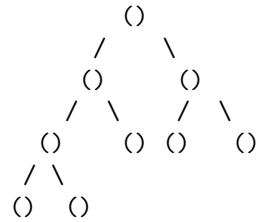
Input : N = 6

Output : 2



Input : N = 9

Output :



Let the size of heap be **N** and height be **h**

If we take few examples, we can notice that the value of h in a [complete binary tree](#) is  $\text{ceil}(\log_2(N+1)) - 1$ .

Examples :

N	h
<hr/>	
1	0
2	1
3	1
4	2
5	2
.....	
.....	

C++

```
// CPP program to find height of complete
// binary tree from total nodes.
#include <bits/stdc++.h>
using namespace std;

int height(int N)
{
    return ceil(log2(N + 1)) - 1;
}

// driver node
int main()
{
    int N = 6;
    cout << height(N);
    return 0;
}
```

Java

```
// Java program to find height
// of complete binary tree
// from total nodes.
import java.lang.*;

class GFG {

    // Function to calculate height
    static int height(int N)
    {
        return (int) Math.ceil(Math.log(N +
            1) / Math.log(2)) - 1;
    }

    // Driver Code
}
```

```
public static void main(String[] args)
{
    int N = 6;
    System.out.println(height(N));
}
}

// This code is contributed by
// Smitha Dinesh Semwal
```

### Python 3

```
# Python 3 program to find
# height of complete binary
# tree from total nodes.
import math
def height(N):
    return math.ceil(math.log2(N + 1)) - 1

# driver node
N = 6
print(height(N))

# This code is contributed by
# Smitha Dinesh Semwal
```

### C#

```
// C# program to find height
// of complete binary tree
// from total nodes.
using System;

class GFG {
    static int height(int N)
    {
        return (int)Math.Ceiling(Math.Log(N
            + 1) / Math.Log(2)) - 1;
    }

    // Driver node
    public static void Main()
    {
        int N = 6;
        Console.WriteLine(height(N));
    }
}
```

```
// This code is contributed by  
// Smitha Dinesh Semwal
```

### PHP

```
<?php  
// PHP program to find height  
// of complete binary tree  
// from total nodes.  
  
function height($N)  
{  
    return ceil(log($N + 1, 2)) - 1;  
}  
  
// Driver Code  
$N = 6;  
echo height($N);  
  
// This code is contributed by aj_36  
?>
```

### Output :

2

Improved By : [Smitha Dinesh Semwal, jit\\_t](#)

### Source

<https://www.geeksforgeeks.org/height-complete-binary-tree-heap-n-nodes/>

## Chapter 208

# Height of a generic tree from parent array

Height of a generic tree from parent array - GeeksforGeeks

We are given a tree of size n as array parent[0..n-1] where every index i in parent[] represents a node and the value at i represents the immediate parent of that node. For root node value will be -1. Find the height of the generic tree given the parent links.

Examples:

Input : parent[] = {-1, 0, 0, 0, 3, 1, 1, 2}  
Output : 2

Input : parent[] = {-1, 0, 1, 2, 3}  
Output : 4

Here, generic tree is sometimes also called as N-ary tree or N-way tree where N denotes the maximum number of child a node can have. In this problem array represents n number of nodes in the tree.

### Approach 1:

One solution is to traverse up the tree from node till root node is reached with node value -1. While Traversing for each node store maximum path length.

Time Complexity of this solution is  $O(n^2)$ .

### Approach 2:

Build graph for N-ary Tree in  $O(n)$  time and apply BFS on the stored graph in  $O(n)$  time and while doing BFS store maximum reached level. This solution does two iterations to find the height of N-ary tree.

```
// C++ code to find height of N-ary
```

```
// tree in O(n)
#include <bits/stdc++.h>
#define MAX 1001
using namespace std;

// Adjacency list to
// store N-ary tree
vector<int> adj[MAX];

// Build tree in tree in O(n)
int build_tree(int arr[], int n)
{
    int root_index = 0;

    // Iterate for all nodes
    for (int i = 0; i < n; i++) {

        // if root node, store index
        if (arr[i] == -1)
            root_index = i;

        else {
            adj[i].push_back(arr[i]);
            adj[arr[i]].push_back(i);
        }
    }
    return root_index;
}

// Applying BFS
int BFS(int start)
{
    // map is used as visited array
    map<int, int> vis;

    queue<pair<int, int>> q;
    int max_level_reached = 0;

    // height of root node is zero
    q.push({start, 0});

    // p.first denotes node in adjacency list
    // p.second denotes level of p.first
    pair<int, int> p;

    while (!q.empty()) {

        p = q.front();
        q.pop();

        if (vis.find(p.first) != vis.end())
            continue;

        vis[p.first] = p.second;

        for (int child : adj[p.first]) {
            if (vis.find(child) == vis.end())
                q.push({child, p.second + 1});
        }
    }
}
```

```

vis[p.first] = 1;

// store the maximum level reached
max_level_reached = max(max_level_reached,
                        p.second);

q.pop();

for (int i = 0; i < adj[p.first].size(); i++)

    // adding 1 to previous level
    // stored on node p.first
    // which is parent of node adj[p.first][i]
    // if adj[p.first][i] is not visited
    if (!vis[adj[p.first][i]])
        q.push({adj[p.first][i], p.second + 1});
}

return max_level_reached;
}

// Driver Function
int main()
{
    // node 0 to node n-1
    int parent[] = { -1, 0, 1, 2, 3 };

    // Number of nodes in tree
    int n = sizeof(parent) / sizeof(parent[0]);

    int root_index = build_tree(parent, n);

    int ma = BFS(root_index);
    cout << "Height of N-ary Tree=" << ma;
    return 0;
}

```

#### Output:

Height of N-ary Tree=4

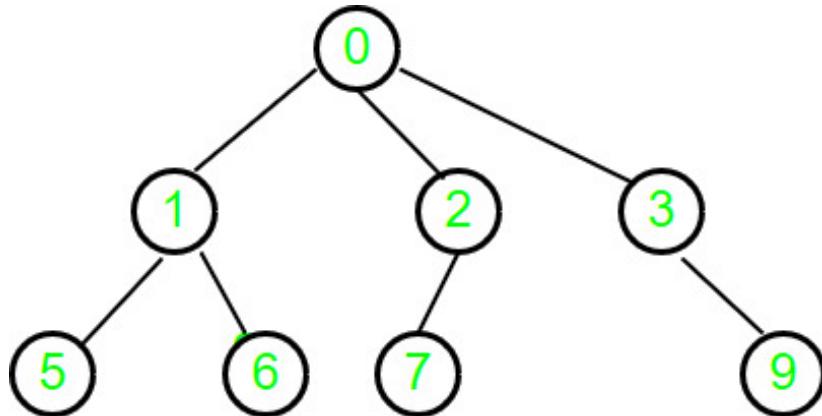
Time Complexity of this solution is **O(2n)** which converges to O(n) for very large n.

#### Approach 3:

We can find the height of N-ary Tree in only one iteration. We visit nodes from 0 to n-1 iteratively and mark the unvisited ancestors recursively if they are not visited before till we reach a node which is visited or we reach root node. If we reach visited node while

traversing up the tree using parent links, then we use its height and will not go further in recursion.

**Explanation For Example 1::**



For **node 0** : Check for Root node is true,  
Return 0 as height, Mark node 0 as visited

For **node 1** : Recur for immediate ancestor, i.e 0, which is already visited  
So, Use it's height and return height(node 0) +1  
Mark node 1 as visited

For **node 2** : Recur for immediate ancestor, i.e 0, which is already visited  
So, Use it's height and return height(node 0) +1  
Mark node 2 as visited

For **node 3** : Recur for immediate ancestor, i.e 0, which is already visited  
So, Use it's height and return height(node 0) +1  
Mark node 3 as visited

For **node 4** : Recur for immediate ancestor, i.e 3, which is already visited  
So, Use it's height and return height(node 3) +1  
Mark node 3 as visited

For **node 5** : Recur for immediate ancestor, i.e 1, which is already visited  
So, Use it's height and return height(node 1) +1  
Mark node 5 as visited

For **node 6** : Recur for immediate ancestor, i.e 1, which is already visited  
So, Use it's height and return height(node 1) +1  
Mark node 6 as visited

For **node 7** : Recur for immediate ancestor, i.e 2, which is already visited  
So, Use it's height and return height(node 2) +1  
Mark node 7 as visited

Hence, we processed each node in N-ary tree only once.

```

// C++ code to find height of N-ary
// tree in O(n) (Efficient Approach)
#include <bits/stdc++.h>
using namespace std;
  
```

```
// Recur For Ancestors of node and
// store height of node at last
int fillHeight(int p[], int node, int visited[],
               int height[])
{
    // If root node
    if (p[node] == -1) {

        // mark root node as visited
        visited[node] = 1;
        return 0;
    }

    // If node is already visited
    if (visited[node])
        return height[node];

    // Visit node and calculate its height
    visited[node] = 1;

    // recur for the parent node
    height[node] = 1 + fillHeight(p, p[node],
                                  visited, height);

    // return calculated height for node
    return height[node];
}

int findHeight(int parent[], int n)
{
    // To store max height
    int ma = 0;

    // To check whether or not node is visited before
    int visited[n];

    // For Storing Height of node
    int height[n];

    memset(visited, 0, sizeof(visited));
    memset(height, 0, sizeof(height));

    for (int i = 0; i < n; i++) {

        // If not visited before
        if (!visited[i])
            height[i] = fillHeight(parent, i,
                                   visited, height);
    }
}
```

```
// store maximum height so far
ma = max(ma, height[i]);
}

return ma;
}

// Driver Function
int main()
{
    int parent[] = { -1, 0, 0, 0, 3, 1, 1, 2 };
    int n = sizeof(parent) / sizeof(parent[0]);

    cout << "Height of N-ary Tree = "
        << findHeight(parent, n);
    return 0;
}
```

**Output:**

Height of N-ary Tree = 2

**Time Complexity:** O(n)

**Source**

<https://www.geeksforgeeks.org/height-generic-tree-parent-array/>

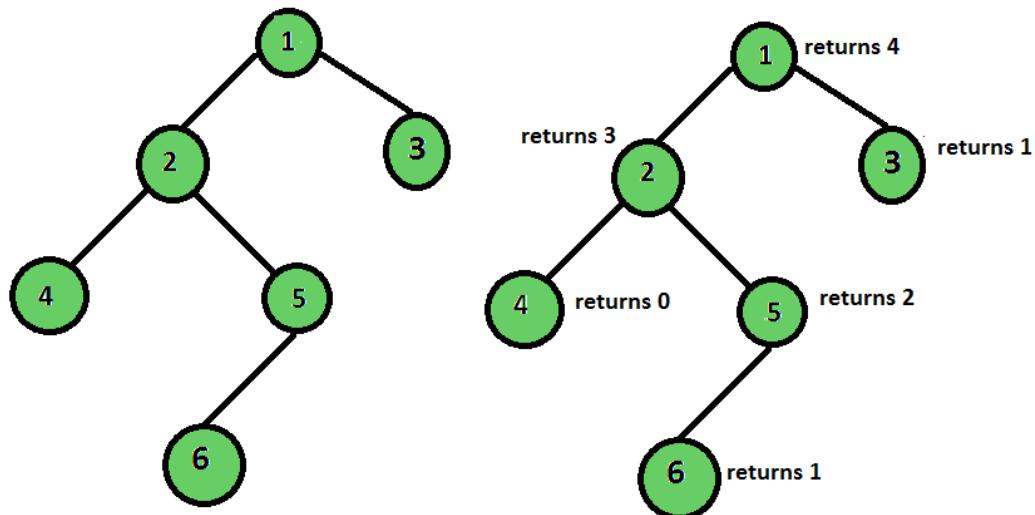
## Chapter 209

# Height of binary tree considering even level leaves only

Height of binary tree considering even level leaves only - GeeksforGeeks

Find the height of the binary tree given that only the nodes on the even levels are considered as the valid leaf nodes.

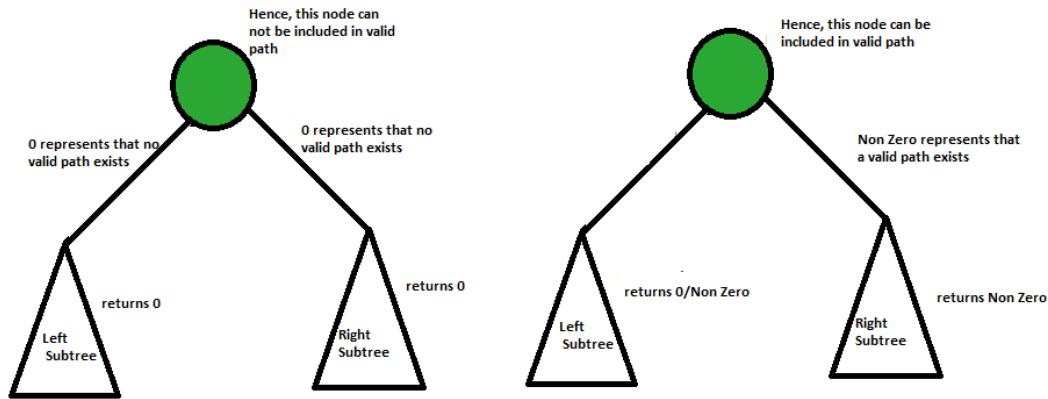
The height of a binary tree is the number of edges between the tree's root and its furthest leaf. But what if we bring a twist and change the definition of a leaf node. Let us define a valid leaf node as the node that has no children and is at an even level (considering root node as an odd level node).



**Output :** Height of tree is 4

**Solution :** The approach to this problem is slightly different from the normal height finding

approach. In the return step, we check if the node is a valid root node or not. If it is valid, return 1, else we return 0. Now in the recursive step- if the left and the right sub-tree both yield 0, the current node yields 0 too, because in that case there is no path from current node to a valid leaf node. But in case at least one of the values returned by the children is non-zero, it means the leaf node on that path is a valid leaf node, and hence that path can contribute to the final result, so we return max of the values returned + 1 for the current node.



```
/*
 * Program to find height of the tree considering
 * only even level leaves. */
#include <bits/stdc++.h>
using namespace std;

/* A binary tree node has data, pointer to
   left child and a pointer to right child */
struct Node {
    int data;
    struct Node* left;
    struct Node* right;
};

int heightOfTreeUtil(Node* root, bool isEven)
{
    // Base Case
    if (!root)
        return 0;

    if (!root->left && !root->right) {
        if (isEven)
            return 1;
        else
            return 0;
    }
}
```

```
/*left stores the result of left subtree,
   and right stores the result of right subtree*/
int left = heightOfTreeUtil(root->left, !isEven);
int right = heightOfTreeUtil(root->right, !isEven);

/*If both left and right returns 0, it means
   there is no valid path till leaf node*/
if (left == 0 && right == 0)
    return 0;

    return (1 + max(left, right));
}

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct Node* newNode(int data)
{
    struct Node* node =
        (struct Node*)malloc(sizeof(struct Node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;

    return (node);
}

int heightOfTree(Node* root)
{
    return heightOfTreeUtil(root, false);
}

/* Driver program to test above functions*/
int main()
{
    // Let us create binary tree shown in above diagram
    struct Node* root = newNode(1);

    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->left->right->left = newNode(6);
    cout << "Height of tree is " << heightOfTree(root);
    return 0;
}
```

Output:

Height of tree is 4

**Time Complexity:** $O(n)$  where n is number of nodes in given binary tree.

### Source

<https://www.geeksforgeeks.org/height-binary-tree-considering-even-level-leaves/>

## Chapter 210

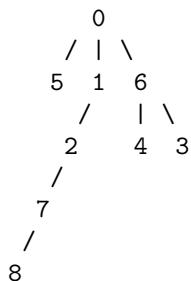
# Height of n-ary tree if parent array is given

Height of n-ary tree if parent array is given - GeeksforGeeks

Given a parent array P, where P[i] indicates the parent of ith node in the tree (assume parent of root node id indicated with -1). Find the height of the tree.

Examples:

```
Input : array[] = [-1 0 1 6 6 0 0 2 7]
Output : height = 5
Tree formed is:
```



1. Start at each node and keep going to its parent until we reach -1.
2. Also keep track of the maximum height among all nodes.

C++

```
// C++ program to find the height of the generic
// tree(n-ary tree) if parent array is given
#include <bits/stdc++.h>
```

```
using namespace std;

// function to find the height of tree
int findHeight(int* parent, int n)
{
    int res = 0;

    // Traverse each node
    for (int i = 0; i < n; i++) {

        // traverse to parent until -1
        // is reached
        int p = i, current = 1;
        while (parent[p] != -1) {
            current++;
            p = parent[p];
        }

        res = max(res, current);
    }
    return res;
}

// Driver program
int main()
{
    int parent[] = {-1, 0, 1, 6, 6, 0, 0, 2, 7};
    int n = sizeof(parent)/sizeof(parent[0]);
    int height = findHeight(parent, n);
    cout << "Height of the given tree is: "
         << height << endl;
    return 0;
}
```

### Java

```
// java program to find the height of
// the generic tree(n-ary tree) if
// parent array is given
import java.io.*;

public class GFG {

    // function to find the height of tree
    static int findHeight(int []parent, int n)
    {
        int res = 0;
```

```
// Traverse each node
for (int i = 0; i < n; i++)
{
    // traverse to parent until -1
    // is reached
    int p = i, current = 1;
    while (parent[p] != -1)
    {
        current++;
        p = parent[p];
    }

    res = Math.max(res, current);
}
return res;
}

// Driver program
static public void main (String[] args)
{
    int []parent = {-1, 0, 1, 6, 6, 0,
                   0, 2, 7};
    int n = parent.length;

    int height = findHeight(parent, n);

    System.out.println("Height of the "
                       + "given tree is: " + height);
}
}

// This code is contributed by vt_m.
```

C#

```
// C# program to find the height of
// the generic tree(n-ary tree) if
// parent array is given
using System;

public class GFG {

    // function to find the height of tree
    static int findHeight(int []parent, int n)
    {
        int res = 0;
```

```
// Traverse each node
for (int i = 0; i < n; i++)
{
    // traverse to parent until -1
    // is reached
    int p = i, current = 1;
    while (parent[p] != -1)
    {
        current++;
        p = parent[p];
    }

    res = Math.Max(res, current);
}

return res;
}

// Driver program
static public void Main ()
{
    int []parent = {-1, 0, 1, 6, 6, 0,
                    0, 2, 7};
    int n = parent.Length;

    int height = findHeight(parent, n);

    Console.WriteLine ("Height of the "
                      + "given tree is: " + height);
}
}

// This code is contributed by vt_m.
```

Output:

```
Height of the given tree is: 5
```

Improved By : [vt\\_m](#)

## Source

<https://www.geeksforgeeks.org/height-n-ary-tree-parent-array-given/>

## Chapter 211

# How to determine if a binary tree is height-balanced?

How to determine if a binary tree is height-balanced? - GeeksforGeeks

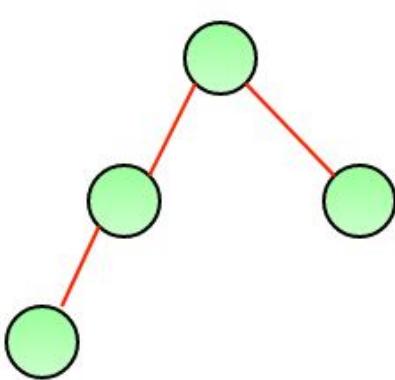
A tree where no leaf is much farther away from the root than any other leaf. Different balancing schemes allow different definitions of “much farther” and different amounts of work to keep them balanced.

Consider a height-balancing scheme where following conditions should be checked to determine if a binary tree is balanced.

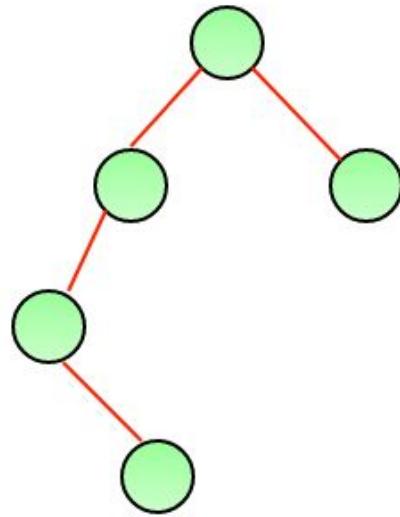
An empty tree is height-balanced. A non-empty binary tree T is balanced if:

- 1) Left subtree of T is balanced
- 2) Right subtree of T is balanced
- 3) The difference between heights of left subtree and right subtree is not more than 1.

The above height-balancing scheme is used in AVL trees. The diagram below shows two trees, one of them is height-balanced and other is not. The second tree is not height-balanced because height of left subtree is 2 more than height of right subtree.



A height balanced tree



Not a height balanced tree

To check if a tree is height-balanced, get the height of left and right subtrees. Return true if difference between heights is not more than 1 and left and right subtrees are balanced, otherwise return false.

C

```
/* C program to check if a tree is height-balanced or not */
#include<stdio.h>
#include<stdlib.h>
#define bool int

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

/* Returns the height of a binary tree */
int height(struct node* node);

/* Returns true if binary tree with root as root is height-balanced */
bool isBalanced(struct node *root)
{
    int lh; /* for height of left subtree */
```

```
int rh; /* for height of right subtree */

/* If tree is empty then return true */
if(root == NULL)
    return 1;

/* Get the height of left and right sub trees */
lh = height(root->left);
rh = height(root->right);

if( abs(lh-rh) <= 1 &&
    isBalanced(root->left) &&
    isBalanced(root->right))
    return 1;

/* If we reach here then tree is not height-balanced */
return 0;
}

/* UTILITY FUNCTIONS TO TEST isBalanced() FUNCTION */

/* returns maximum of two integers */
int max(int a, int b)
{
    return (a >= b)? a: b;
}

/* The function Compute the "height" of a tree. Height is the
   number of nodes along the longest path from the root node
   down to the farthest leaf node.*/
int height(struct node* node)
{
    /* base case tree is empty */
    if(node == NULL)
        return 0;

    /* If tree is not empty then height = 1 + max of left
       height and right heights */
    return 1 + max(height(node->left), height(node->right));
}

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* node = (struct node*)
                           malloc(sizeof(struct node));
    node->data = data;
```

```
    node->left = NULL;
    node->right = NULL;

    return(node);
}

int main()
{
    struct node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->left->left->left = newNode(8);

    if(isBalanced(root))
        printf("Tree is balanced");
    else
        printf("Tree is not balanced");

    getchar();
    return 0;
}
```

**Java**

```
/* Java program to determine if binary tree is
height balanced or not */

/* A binary tree node has data, pointer to left child,
   and a pointer to right child */
class Node
{
    int data;
    Node left, right;
    Node(int d)
    {
        data = d;
        left = right = null;
    }
}

class BinaryTree
{
    Node root;

    /* Returns true if binary tree with root as root is height-balanced */
    boolean isBalanced(Node node)
```

```
{  
    int lh; /* for height of left subtree */  
  
    int rh; /* for height of right subtree */  
  
    /* If tree is empty then return true */  
    if (node == null)  
        return true;  
  
    /* Get the height of left and right sub trees */  
    lh = height(node.left);  
    rh = height(node.right);  
  
    if (Math.abs(lh - rh) <= 1  
        && isBalanced(node.left)  
        && isBalanced(node.right))  
        return true;  
  
    /* If we reach here then tree is not height-balanced */  
    return false;  
}  
  
/* UTILITY FUNCTIONS TO TEST isBalanced() FUNCTION */  
/* The function Compute the "height" of a tree. Height is the  
   number of nodes along the longest path from the root node  
   down to the farthest leaf node.*/  
int height(Node node)  
{  
    /* base case tree is empty */  
    if (node == null)  
        return 0;  
  
    /* If tree is not empty then height = 1 + max of left  
       height and right heights */  
    return 1 + Math.max(height(node.left), height(node.right));  
}  
  
public static void main(String args[])  
{  
    BinaryTree tree = new BinaryTree();  
    tree.root = new Node(1);  
    tree.root.left = new Node(2);  
    tree.root.right = new Node(3);  
    tree.root.left.left = new Node(4);  
    tree.root.left.right = new Node(5);  
    tree.root.left.left.left = new Node(8);  
  
    if(tree.isBalanced(tree.root))
```

```
        System.out.println("Tree is balanced");
    else
        System.out.println("Tree is not balanced");
}
}

// This code has been contributed by Mayank Jaiswal(mayank_24)
```

**Python3**

```
"""
Python program to check if a tree is height-balanced
"""

# A binary tree Node
class Node:
    # Constructor to create a new Node
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

    # function to find height of binary tree
    def height(root):

        # base condition when binary tree is empty
        if root is None:
            return 0
        return max(height(root.left), height(root.right)) + 1

    # function to check if tree is height-balanced or not
    def isBalanced(root):

        # Base condition
        if root is None:
            return True

        # for left and right subtree height
        lh = height(root.left)
        rh = height(root.right)

        # allowed values for (lh - rh) are 1, -1, 0
        if (abs(lh - rh) <= 1) and isBalanced(
            root.left) is True and isBalanced( root.right) is True:
            return True

        # if we reach here means tree is not
        # height-balanced tree
        return False
```

```
# Driver function to test the above function
root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)
root.left.left.left = Node(8)
if isBalanced(root):
    print("Tree is balanced")
else:
    print("Tree is not balanced")

# This code is contributed by Shweta Singh
```

Output:

```
Tree is not balanced
```

Time Complexity:  $O(n^2)$  Worst case occurs in case of skewed tree.

**Optimized implementation:** Above implementation can be optimized by calculating the height in the same recursion rather than calling a height() function separately. Thanks to Amar for suggesting this optimized version. This optimization reduces time complexity to  $O(n)$ .

C

```
/* program to check if a tree is height-balanced or not */
#include<stdio.h>
#include<stdlib.h>
#define bool int

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

/* The function returns true if root is balanced else false
   The second parameter is to store the height of tree.
   Initially, we need to pass a pointer to a location with value
   as 0. We can also write a wrapper over this function */
bool isBalanced(struct node *root, int* height)
```

```
{  
    /* lh --> Height of left subtree  
       rh --> Height of right subtree */  
    int lh = 0, rh = 0;  
  
    /* l will be true if left subtree is balanced  
       and r will be true if right subtree is balanced */  
    int l = 0, r = 0;  
  
    if(root == NULL)  
    {  
        *height = 0;  
        return 1;  
    }  
  
    /* Get the heights of left and right subtrees in lh and rh  
       And store the returned values in l and r */  
    l = isBalanced(root->left, &lh);  
    r = isBalanced(root->right,&rh);  
  
    /* Height of current node is max of heights of left and  
       right subtrees plus 1*/  
    *height = (lh > rh? lh: rh) + 1;  
  
    /* If difference between heights of left and right  
       subtrees is more than 2 then this node is not balanced  
       so return 0 */  
    if((lh - rh >= 2) || (rh - lh >= 2))  
        return 0;  
  
    /* If this node is balanced and left and right subtrees  
       are balanced then return true */  
    else return l&&r;  
}  
  
/* UTILITY FUNCTIONS TO TEST isBalanced() FUNCTION */  
  
/* Helper function that allocates a new node with the  
   given data and NULL left and right pointers. */  
struct node* newNode(int data)  
{  
    struct node* node = (struct node*)  
                      malloc(sizeof(struct node));  
    node->data = data;  
    node->left = NULL;  
    node->right = NULL;
```

```
    return(node);
}

int main()
{
    int height = 0;

    /* Constructed binary tree is
        1
       /   \
      2     3
     / \   /
    4   5   6
   /
  7
*/
    struct node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->left = newNode(6);
    root->left->left->left = newNode(7);

    if(isBalanced(root, &height))
        printf("Tree is balanced");
    else
        printf("Tree is not balanced");

    getchar();
    return 0;
}
```

### Java

```
/* Java program to determine if binary tree is
height balanced or not */

/* A binary tree node has data, pointer to left child,
and a pointer to right child */
class Node {

    int data;
    Node left, right;

    Node(int d) {
        data = d;
        left = right = null;
    }
}
```

```
        }
    }

// A wrapper class used to modify height across
// recursive calls.
class Height
{
    int height = 0;
}

class BinaryTree {

    Node root;

    /* Returns true if binary tree with root as root is height-balanced */
    boolean isBalanced(Node root, Height height)
    {
        /* If tree is empty then return true */
        if (root == null)
        {
            height.height = 0;
            return true;
        }

        /* Get heights of left and right sub trees */
        Height lheight = new Height(), rheight = new Height();
        boolean l = isBalanced(root.left, lheight);
        boolean r = isBalanced(root.right, rheight);
        int lh = lheight.height, rh = rheight.height;

        /* Height of current node is max of heights of
           left and right subtrees plus 1*/
        height.height = (lh > rh? lh: rh) + 1;

        /* If difference between heights of left and right
           subtrees is more than 2 then this node is not balanced
           so return 0 */
        if ((lh - rh >= 2) ||
            (rh - lh >= 2))
            return false;

        /* If this node is balanced and left and right subtrees
           are balanced then return true */
        else return l && r;
    }

    /* The function Compute the "height" of a tree. Height is the
```

```
number of nodes along the longest path from the root node
down to the farthest leaf node.*/
int height(Node node)
{
    /* base case tree is empty */
    if (node == null)
        return 0;

    /* If tree is not empty then height = 1 + max of left
    height and right heights */
    return 1 + Math.max(height(node.left), height(node.right));
}

public static void main(String args[])
{
    Height height = new Height();

    /* Constructed binary tree is
       1
      /   \
     2     3
    / \   /
   4   5  6
  /
 7
*/
    BinaryTree tree = new BinaryTree();
    tree.root = new Node(1);
    tree.root.left = new Node(2);
    tree.root.right = new Node(3);
    tree.root.left.left = new Node(4);
    tree.root.left.right = new Node(5);
    tree.root.right.right = new Node(6);
    tree.root.left.left.left = new Node(7);

    if (tree.isBalanced(tree.root, height))
        System.out.println("Tree is balanced");
    else
        System.out.println("Tree is not balanced");
}
}

// This code has been contributed by Mayank Jaiswal(mayank_24)
```

### Python3

```
"""
Python program to check if Binary tree is
height-balanced
```

```
"""
# A binary tree node
class Node:

    # constructor to create node of
    # binary tree
    def __init__(self, data):
        self.data = data
        self.left = self.right = None

# utility class to pass height object
class Height:
    def __init__(self):
        self.height = 0

# helper function to check if binary
# tree is height balanced
def isBalanced(root, height):

    # lh and rh to store height of
    # left and right subtree
    lh = Height()
    rh = Height()

    # Base condition when tree is
    # empty return true
    if root is None:
        return True

    # l and r are used to check if left
    # and right subtree are balanced
    l = isBalanced(root.left, lh)
    r = isBalanced(root.right, rh)

    # height of tree is maximum of
    # left subtree height and
    # right subtree height plus 1
    height.height = max(lh.height, rh.height) + 1

    if abs(lh.height - rh.height) <= 1:
        return l and r

    # if we reach here then the tree
    # is not balanced
    return False

# Driver function to test the above function
```

```
"""
Constructed binary tree is
      1
     / \
    2   3
   / \ /
  4 5 6
 /
7
"""

# to store the height of tree during traversal
height = Height()

root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)
root.right.left = Node(6)
root.left.left.left = Node(7)

if isBalanced(root, height):
    print('Tree is balanced')
else:
    print('Tree is not balanced')

# This code is contributed by Shweta Singh
```

Time Complexity: O(n)

Improved By : [shweta44](#)

## Source

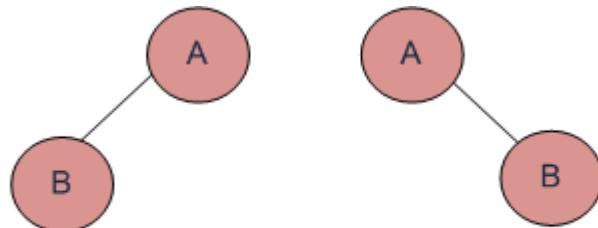
<https://www.geeksforgeeks.org/how-to-determine-if-a-binary-tree-is-balanced/>

## Chapter 212

If you are given two traversal sequences, can you construct the binary tree?

If you are given two traversal sequences, can you construct the binary tree? - GeeksforGeeks

It depends on what traversals are given. If one of the traversal methods is Inorder then the tree can be constructed, otherwise not.



Trees having Preorder, Postorder and Level-Order and traversals

Therefore, following combination can uniquely identify a tree.

Inorder and Preorder.

Inorder and Postorder.

Inorder and Level-order.

**And following do not.**

Postorder and Preorder.

Preorder and Level-order.

Postorder and Level-order.

For example, Preorder, Level-order and Postorder traversals are same for the trees given in above diagram.

*Chapter 212. If you are given two traversal sequences, can you construct the binary tree?*

---

Preorder Traversal = AB

Postorder Traversal = BA

Level-Order Traversal = AB

So, even if three of them (Pre, Post and Level) are given, the tree can not be constructed.

## Source

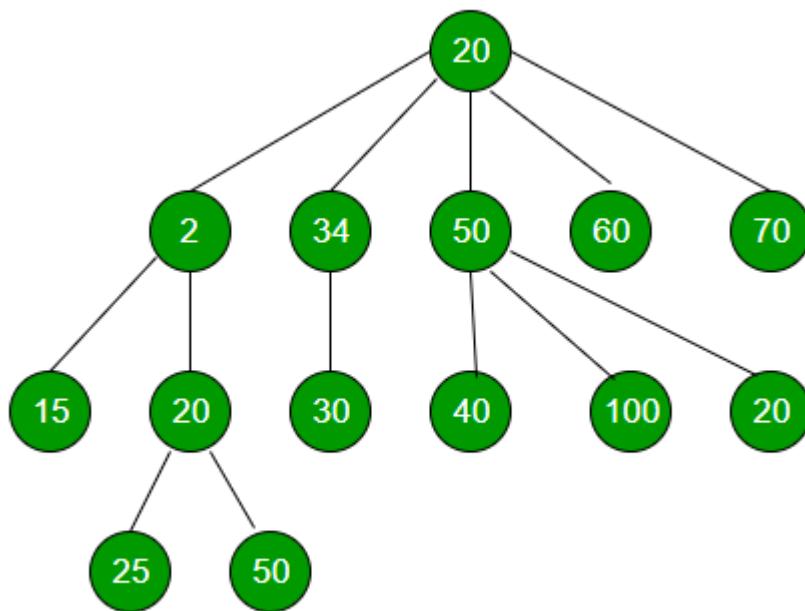
<https://www.geeksforgeeks.org/if-you-are-given-two-traversal-sequences-can-you-construct-the-binary-tree/>

## Chapter 213

# Immediate Smaller element in an N-ary Tree

Immediate Smaller element in an N-ary Tree - GeeksforGeeks

Given an element x, task is to find the value of its immediate smaller element.



Example :

```
Input : x = 30 (for above tree)
Output : Immediate smaller element is 25
```

**Explanation :** Elements 2, 15, 20 and 25 are smaller than x i.e, 30, but 25 is the immediate smaller element and hence the answer.

**Approach :**

- Let **res** be the resultant node.
- Initialize the resultant Node as NULL.
- For every Node, check if data of root is greater than res, but less than x. if yes, update res.
- Recursively do the same for all nodes of the given Generic Tree.
- Return res, and res->key would be the immediate smaller element.

Below is the implementation of above approach :

```
// C++ program to find immediate Smaller
// Element of a given element in a n-ary tree.
#include <bits/stdc++.h>
using namespace std;

// class of a node of an n-ary tree
class Node {

public:
    int key;
    vector<Node*> child;

    // constructor
    Node(int data)
    {
        key = data;
    }
};

// Function to find immediate Smaller Element
// of a given number x
void immediateSmallerElementUtil(Node* root,
                                  int x, Node** res)
{
    if (root == NULL)
        return;

    // if root is greater than res, but less
    // than x, then update res
    if (root->key < x)
        if (!(*res) || (*res)->key < root->key)
```

```

*res = root; // Updating res

// Number of children of root
int numChildren = root->child.size();

// Recursive calling for every child
for (int i = 0; i < numChildren; i++)
    immediateSmallerElementUtil(root->child[i], x, res);

return;
}

// Function to return immediate Smaller
// Element of x in tree
Node* immediateSmallerElement(Node* root, int x)
{
    // resultant node
    Node* res = NULL;

    // calling helper function and using
    // pass by reference
    immediateSmallerElementUtil(root, x, &res);

    return res;
}

// Driver program
int main()
{
    // Creating a generic tree
    Node* root = new Node(20);
    (root->child).push_back(new Node(2));
    (root->child).push_back(new Node(34));
    (root->child).push_back(new Node(50));
    (root->child).push_back(new Node(60));
    (root->child).push_back(new Node(70));
    (root->child[0]->child).push_back(new Node(15));
    (root->child[0]->child).push_back(new Node(20));
    (root->child[1]->child).push_back(new Node(30));
    (root->child[2]->child).push_back(new Node(40));
    (root->child[2]->child).push_back(new Node(100));
    (root->child[2]->child).push_back(new Node(20));
    (root->child[0]->child[1]->child).push_back(new Node(25));
    (root->child[0]->child[1]->child).push_back(new Node(50));

    int x = 30;

    cout << "Immediate smaller element of " << x << " is ";
}

```

```
cout << immediateSmallerElement(root, x)->key << endl;  
return 0;  
}
```

Output :

```
Immediate smaller element of 30 is 25
```

**Time Complexity :** O(N), where N is the number of nodes in N-ary Tree.

**Auxiliary Space :** O(N), for recursive call(worst case when a node has N number of childs)

### Source

<https://www.geeksforgeeks.org/immediate-smaller-element-n-ary-tree/>

## Chapter 214

# Implementation of Binary Search Tree in Javascript

Implementation of Binary Search Tree in Javascript - GeeksforGeeks

In this article, we would be implementing the [Binary Search Tree](#) data structure in Javascript. A tree is a collection of nodes connected by some edges. A [tree](#) is a non linear data structure. A Binary Search tree is a binary tree in which nodes which have lesser value are stored on the left while the nodes with higher value are stored at the right.

Now lets see an example of an Binary Search Tree node:

```
// Node class
class Node
{
    constructor(data)
    {
        this.data = data;
        this.left = null;
        this.right = null;
    }
}
```

As in the above code snippet we define a node class having three properties *data*, *left* and *right*. *Left* and *right* are pointers to the left and right node in a Binary Search Tree. *Data* is initialized with *data* which is passed when object for this node is created and *left* and *right* is set to null.

Now lets see an example of an Binary Search Tree class.

```
// Binary Search tree class
class BinarySearchTree
{
    constructor()
```

```
{  
    // root of a binary search tree  
    this.root = null;  
}  
  
// function to be implemented  
// insert(data)  
// remove(data)  
  
// Helper function  
// findMinNode()  
// getRootNode()  
// inorder(node)  
// preorder(node)  
// postorder(node)  
// search(node, data)  
}
```

The above example shows a framework of a Binary Search tree class, which contains a private variable *root* which holds the root of a tree, it is initialized to null.

Now lets implement each of this function:

1. **insert(data)** – It inserts a new node in a tree with a value *data*

```
// helper method which creates a new node to  
// be inserted and calls insertNode  
insert(data)  
{  
    // Creating a node and initailising  
    // with data  
    var newNode = new Node(data);  
  
    // root is null then node will  
    // be added to the tree and made root.  
    if(this.root === null)  
        this.root = newNode;  
    else  
  
        // find the correct position in the  
        // tree and add the node  
        this.insertNode(this.root, newNode);  
}  
  
// Method to insert a node in a tree  
// it moves over the tree to find the location  
// to insert a node with a given data  
insertNode(node, newNode)
```

```

{
    // if the data is less than the node
    // data move left of the tree
    if(newNode.data < node.data)
    {
        // if left is null insert node here
        if(node.left === null)
            node.left = newNode;
        else

            // if left is not null recur until
            // null is found
            this.insertNode(node.left, newNode);
    }

    // if the data is more than the node
    // data move right of the tree
    else
    {
        // if right is null insert node here
        if(node.right === null)
            node.right = newNode;
        else

            // if right is not null recur until
            // null is found
            this.insertNode(node.right,newNode);
    }
}

```

In the above code we have two methods *insert(data)* and *insertNode(node, newNode)*. Lets understand them one by one:-

- **insert(data)** – It creates a new node with a value data, if the tree is empty it add this node to tree and make it a root, otherwise it calls *insert(node, data)*.
- **insert(node, data)** – It compares the given *data* with the data of current node and moves left or right accordingly and recur until it finds a correct node with a null value where new node can be added.

2. **remove(data)** – This function removes a node with a given data.

```

// helper method that calls the
// removeNode with a given data
remove(data)
{
    // root is re-initialized with
    // root of a modified tree.
    this.root = this.removeNode(this.root, data);
}

```

```
}

// Method to remove node with a
// given data
// it recurss over the tree to find the
// data and removes it
removeNode(node, key)
{
    // if the root is null then tree is
    // empty
    if(node === null)
        return null;

    // if data to be delete is less than
    // roots data then move to left subtree
    else if(key < node.data)
    {
        node.left = this.removeNode(node.left, key);
        return node;
    }

    // if data to be delete is greater than
    // roots data then move to right subtree
    else if(key > node.data)
    {
        node.right = this.removeNode(node.right, key);
        return node;
    }

    // if data is similar to the root's data
    // then delete this node
    else
    {
        // deleting node with no children
        if(node.left === null && node.right === null)
        {
            node = null;
            return node;
        }

        // deleting node with one children
        if(node.left === null)
        {
            node = node.right;
            return node;
        }
    }
}
```

```
        else if(node.right === null)
        {
            node = node.left;
            return node;
        }

        // Deleting node with two children
        // minimum node of the right subtree
        // is stored in aux
        var aux = this.findMinNode(node.right);
        node.data = aux.data;

        node.right = this.removeNode(node.right, aux.data);
        return node;
    }

}
```

In the above code we have two method `remove(data)` and `removeNode(node, data)`, let understand them one by one:

- **remove(data)** – It is helper methods which calls removeNode by passing root node and given data and updates the root of the tree with the value returned by the function
- **removeNode(node, data)** – It searches for a node with a given data and then perform certain steps to delete it.

While deleting a node from the tree their are three different scenarios as follows:-

- **Deleting the leaf node** – As leaf node do not have any children hence they can be easily removed and null is returned to the parent node
- **Deleting a node with one child** – If a node have a left child then we update the pointer of the parent node to the left child of the node to be deleted and similarly if a node have a right child then we update the pointer of the parent node to the right child of the node to be deleted
- **Deleting a node with two children** – In order to delete a node with two children we find the node with minimum value in its right subtree and replace this node with the minimum valued node and remove the minimum valued node from the tree

In the above code we have used `findMinNode(node)`, it is defined in a helper method section.

**Recommendation:** [Binary Search Tree | Set 2 \(Delete\)](#) It contains a detail explanation and a video tutorial for deleting a node from binary search tree.

## Tree Traversal

Now Lets understand different ways of traversing a Binary Search Tree.

1. **inorder(node)** – It performs inorder traversal of a tree starting from a given *node*.  
Algorithm for inorder:

Traverse the left subtree i.e perform inorder on left subtreeVisit the rootTraverse the right subtree

```
// Performs inorder traversal of a tree
inorder(node)
{
    if(node !== null)
    {
        this.inorder(node.left);
        console.log(node.data);
        this.inorder(node.right);
    }
}
```

2. **preorder(node)** – It performs preorder traversal of a tree starting from a given *node*.  
Algorithm for preorder:

Visit the rootTraverse the left subtree i.e perform inorder on left subtreeTraverse the right subtree

```
// Performs preorder traversal of a tree
preorder(node)
{
    if(node != null)
    {
        console.log(node.data);
        this.preorder(node.left);
        this.preorder(node.right);
    }
}
```

3. **postorder(node)** – It performs postorder traversal of a tree starting from a given *node*.  
Algorithm for postorder:

Traverse the left subtree i.e perform inorder on left subtreeTraverse the right subtree i.e

```
// Performs postorder traversal of a tree
postorder(node)
{
    if(node != null)
    {
        this.postorder(node.left);
```

```
        this.postorder(node.right);
        console.log(node.data);
    }
}
```

### Helper Methods

Lets declare some helper method which are useful while working with Binary Search Tree.

1. **findMinNode(node)** – It searches for a node with a minimum value starting from *node*.

```
// finds the minimum node in tree
// searching starts from given node
findMinNode(node)
{
    // if left of a node is null
    // then it must be minimum node
    if(node.left === null)
        return node;
    else
        return this.findMinNode(node.left);
}
```

As seen in the above method we start from a *node* and keeping moving to the left subtree until we find a node whose left child is null, once we find such node we return it.

2. **get rootNode()** – It returns the root node of a tree.

```
// returns root of the tree
getRootNode()
{
    return this.root;
}
```

3. **search(data)** – It searches the node with a value *data* in the entire tree.

```
// search for a node with given data
search(node, data)
{
    // if trees is empty return null
    if(node === null)
        return null;

    // if data is less than node's data
    // move left
    else if(data < node.data)
```

```
        return this.search(node.left, data);

        // if data is less than node's data
        // move left
        else if(data > node.data)
            return this.search(node.right, data);

        // if data is equal to the node data
        // return node
        else
            return node;
    }
```

**Note :** Different helper method can be declared in the *BinarySearchTree* class as per the requirement.

### Implementation

Now lets use the *BinarySearchTree* class and its different methods described above.

```
// create an object for the BinarySearchTree
var BST = new BinarySearchTree();

// Inserting nodes to the BinarySearchTree
BST.insert(15);
BST.insert(25);
BST.insert(10);
BST.insert(7);
BST.insert(22);
BST.insert(17);
BST.insert(13);
BST.insert(5);
BST.insert(9);
BST.insert(27);

//      15
//      / \
//      10   25
//      / \   / \
//      7   13 22  27
//      / \   /
//      5   9   17

var root = BST.getRootNode();

// prints 5 7 9 10 13 15 17 22 25 27
BST.inorder(root);

// Removing node with no children
BST.remove(5);
```

```
//          15
//          / \
//          10   25
//         / \   / \
//        7   13 22  27
//        \   /
//        9   17

var root = BST.getNode();

// prints 7 9 10 13 15 17 22 25 27
BST.inorder(root);

// Removing node with one children
BST.remove(7);

//          15
//          / \
//          10   25
//         / \   / \
//        9   13 22  27
//           /
//           17

var root = BST.getNode();

// prints 9 10 13 15 17 22 25 27
BST.inorder(root);

// Removing node with two children
BST.remove(15);

//          17
//          / \
//          10   25
//         / \   / \
//        9   13 22  27

var root = BST.getNode();
console.log("inorder traversal");

// prints 9 10 13 17 22 25 27
BST.inorder(root);
```

```
console.log("postorder traversal");
BST.postorder(root);
console.log("preorder traversal");
BST.preorder(root);
```

For more on binary trees please refer to the following article: [Binary tree Data Structure](#)

## Source

<https://www.geeksforgeeks.org/implementation-binary-search-tree-javascript/>

## Chapter 215

# Inorder Non-threaded Binary Tree Traversal without Recursion or Stack

Inorder Non-threaded Binary Tree Traversal without Recursion or Stack - GeeksforGeeks

We have discussed [Thread based Morris Traversal](#). Can we do inorder traversal without threads if we have parent pointers available to us?

Input: Root of Below Tree [Every node of tree has parent pointer also]

```
    10
   /   \
  5    100
   /   \
  80  120
```

Output: 5 10 80 100 120  
The code should not extra space (No Recursion and stack)

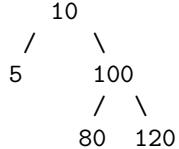
In inorder traversal, we follow “left root right”. We can move to children using left and right pointers. Once a node is visited, we need to move to parent also. For example, in the above tree, we need to move to 10 after printing 5. For this purpose, we use parent pointer. Below is algorithm.

1. Initialize current node as root
2. Initialize a flag: `leftdone = false;`
3. Do following while root is not NULL
  - a) If `leftdone` is false, set current node as leftmost

- child of node.
- b) Mark leftdone as true and print current node.
  - c) If right child of current nodes exists, set current as right child and set leftdone as false.
  - d) Else If parent exists, If current node is left child of its parent, set current node as parent.  
If current node is right child, keep moving to ancestors using parent pointer while current node is right child of its parent.
  - e) Else break (We have reached back to root)

**Illustration:**

Let us consider below tree for illustration.



Initialize: Current node = 10, leftdone = false

Since leftdone is false, we move to 5 (3.a), print it and set leftdone = true.

Now we move to parent of 5 (3.d). Node 10 is printed because leftdone is true.

We move to right of 10 and set leftdone as false (3.c)

Now current node is 100. Since leftdone is false, we move to 80 (3.a) and set leftdone as true. We print current node 80 and move back to parent 100 (3.d). Since leftdone is true, we print current node 100.

Right of 100 exists, so we move to 120 (3.c). We print current node 120.

Since 120 is right child of its parent we keep moving to parent while parent is right child of its parent. We reach root. So we break the loop and stop

Below is C++ implementation of above algorithm. Note that the implementation uses Binary Search Tree instead of Binary Tree. We can use the same function **inorder()** for Binary Tree also. The reason for using Binary Search Tree in below code is, it is easy to construct a Binary Search Tree with parent pointers and easy to test the outcome (In BST inorder traversal is always sorted).

C++

```
// C++ program to print inorder traversal of a Binary Search
// Tree (BST) without recursion and stack
#include <bits/stdc++.h>
using namespace std;

// BST Node
struct Node
{
    Node *left, *right, *parent;
    int key;
};

// A utility function to create a new BST node
Node *newNode(int item)
{
    Node *temp = new Node;
    temp->key = item;
    temp->parent = temp->left = temp->right = NULL;
    return temp;
}

/* A utility function to insert a new node with
   given key in BST */
Node *insert(Node *node, int key)
{
    /* If the tree is empty, return a new node */
    if (node == NULL) return newNode(key);

    /* Otherwise, recur down the tree */
    if (key < node->key)
    {
        node->left = insert(node->left, key);
        node->left->parent = node;
    }
    else if (key > node->key)
    {
        node->right = insert(node->right, key);
        node->right->parent = node;
    }

    /* return the (unchanged) node pointer */
    return node;
}

// Function to print inorder traversal using parent
// pointer
```

```
void inorder(Node *root)
{
    bool leftdone = false;

    // Start traversal from root
    while (root)
    {
        // If left child is not traversed, find the
        // leftmost child
        if (!leftdone)
        {
            while (root->left)
                root = root->left;
        }

        // Print root's data
        printf("%d ", root->key);

        // Mark left as done
        leftdone = true;

        // If right child exists
        if (root->right)
        {
            leftdone = false;
            root = root->right;
        }

        // If right child doesn't exist, move to parent
        else if (root->parent)
        {
            // If this node is right child of its parent,
            // visit parent's parent first
            while (root->parent &&
                   root == root->parent->right)
                root = root->parent;
            if (!root->parent)
                break;
            root = root->parent;
        }
        else break;
    }

    int main(void)
{
    Node * root = NULL;
```

```
root = insert(root, 24);
root = insert(root, 27);
root = insert(root, 29);
root = insert(root, 34);
root = insert(root, 14);
root = insert(root, 4);
root = insert(root, 10);
root = insert(root, 22);
root = insert(root, 13);
root = insert(root, 3);
root = insert(root, 2);
root = insert(root, 6);

printf("Inorder traversal is \n");
inorder(root);

return 0;
}
```

**Java**

```
/* Java program to print inorder traversal of a Binary Search Tree
   without recursion and stack */

// BST node
class Node
{
    int key;
    Node left, right, parent;

    public Node(int key)
    {
        this.key = key;
        left = right = parent = null;
    }
}

class BinaryTree
{
    Node root;

    /* A utility function to insert a new node with
       given key in BST */
    Node insert(Node node, int key)
    {
        /* If the tree is empty, return a new node */
        if (node == null)
            return new Node(key);
```

```
/* Otherwise, recur down the tree */
if (key < node.key)
{
    node.left = insert(node.left, key);
    node.left.parent = node;
}
else if (key > node.key)
{
    node.right = insert(node.right, key);
    node.right.parent = node;
}

/* return the (unchanged) node pointer */
return node;
}

// Function to print inorder traversal using parent
// pointer
void inorder(Node root)
{
    boolean leftdone = false;

    // Start traversal from root
    while (root != null)
    {
        // If left child is not traversed, find the
        // leftmost child
        if (!leftdone)
        {
            while (root.left != null)
            {
                root = root.left;
            }
        }

        // Print root's data
        System.out.print(root.key + " ");

        // Mark left as done
        leftdone = true;

        // If right child exists
        if (root.right != null)
        {
            leftdone = false;
            root = root.right;
        }
    }
}
```

```
// If right child doesn't exist, move to parent
else if (root.parent != null)
{
    // If this node is right child of its parent,
    // visit parent's parent first
    while (root.parent != null
        && root == root.parent.right)
        root = root.parent;

    if (root.parent == null)
        break;
    root = root.parent;
}
else
    break;
}

public static void main(String[] args)
{
    BinaryTree tree = new BinaryTree();
    tree.root = tree.insert(tree.root, 24);
    tree.root = tree.insert(tree.root, 27);
    tree.root = tree.insert(tree.root, 29);
    tree.root = tree.insert(tree.root, 34);
    tree.root = tree.insert(tree.root, 14);
    tree.root = tree.insert(tree.root, 4);
    tree.root = tree.insert(tree.root, 10);
    tree.root = tree.insert(tree.root, 22);
    tree.root = tree.insert(tree.root, 13);
    tree.root = tree.insert(tree.root, 3);
    tree.root = tree.insert(tree.root, 2);
    tree.root = tree.insert(tree.root, 6);

    System.out.println("Inorder traversal is ");
    tree.inorder(tree.root);
}
}

// This code has been contributed by Mayank Jaiswal(mayank_24)
```

Output:

```
Inorder traversal is
2 3 4 6 10 13 14 22 24 27 29 34
```

This article is contributed by **Rishi Chhibber**. Please write comments if you find anything

incorrect, or you want to share more information about the topic discussed above

## **Source**

<https://www.geeksforgeeks.org/inorder-non-threaded-binary-tree-traversal-without-recursion-or-stack/>

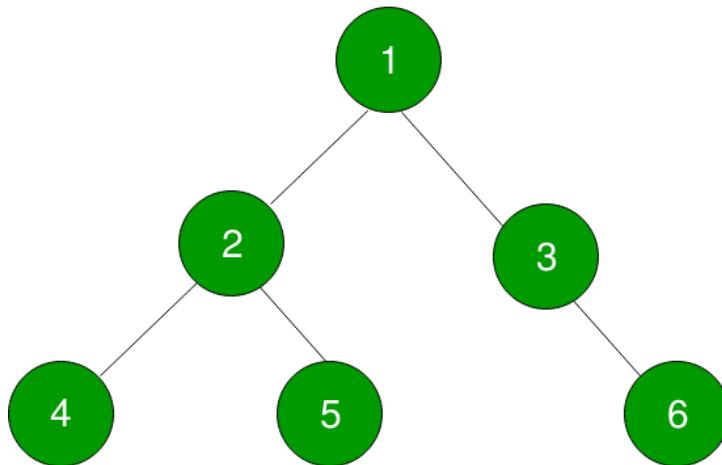
## Chapter 216

# Inorder Successor of a node in Binary Tree

Inorder Successor of a node in Binary Tree - GeeksforGeeks

Given a binary tree and a node, we need to write a program to find inorder successor of this node.

**Inorder Successor** of a node in binary tree is the next node in Inorder traversal of the Binary Tree. Inorder Successor is NULL for the last node in Inorder traversal.



In the above diagram, inorder successor of node **4** is **2** and node **5** is **1**.

We have already discussed how to find the [inorder successor of a node in Binary Search Tree](#). We can not use the same approach to find the inorder successor in general Binary trees.

We need to take care of 3 cases for any node to find its inorder successor as described below:

1. Right child of node is not NULL. If the right child of the node is not NULL then the inorder successor of this node will be the leftmost node in it's right subtree.
2. Right Child of the node is NULL. If the right child of node x, say p such that p->left = x. For example in the above given tree, inorder successor of node 5 will be 1. First parent of 5 is 2 but 2->left != 5. So next parent of 2 is 1, now 1->left = 2. Therefore, inorder successor of 5 is 1.

Below is the algorithm for this case:

- Suppose the given node is **x**. Start traversing the tree from **root** node to find **x** recursively.
  - If **root == x**, stop recursion otherwise find **x** recursively for left and right subtrees.
  - Now after finding the node **x**, recursion will backtrack to the **root**. Every recursive call will return the node itself to the calling function, we will store this in a temporary node say **temp**.Now, when it backtracked to its parent which will be **root** now, check whether **root.left = temp**, if not , keep going up
3. If node is the rightmost node. If the node is the rightmost node in the given tree. For example, in the above tree node 6 is the right most node. In this case, there will be no inorder successor of this node. i.e. Inorder Successor of the rightmost node in a tree is NULL.

Below is C++ implementation of above approach:

```
// CPP program to find inorder successor of a node
#include<bits/stdc++.h>
using namespace std;

// A Binary Tree Node
struct Node
{
    int data;
    struct Node *left, *right;
};

// Temporary node for case 2
Node* temp = new Node;

// Utility function to create a new tree node
Node* newNode(int data)
{
    Node *temp = new Node;
    temp->data = data;
    temp->left = temp->right = NULL;
    return temp;
}

// function to find left most node in a tree
```

```
Node* leftMostNode(Node* node)
{
    while (node != NULL && node->left != NULL)
        node = node->left;
    return node;
}

// function to find right most node in a tree
Node* rightMostNode(Node* node)
{
    while (node != NULL && node->right != NULL)
        node = node->right;
    return node;
}

// recursive function to find the Inorder Successor
// when the right child of node x is NULL
Node* findInorderRecursive(Node* root, Node* x )
{
    if (!root)
        return NULL;

    if (root==x || (temp = findInorderRecursive(root->left,x)) ||
        (temp = findInorderRecursive(root->right,x)))
    {
        if (temp)
        {
            if (root->left == temp)
            {
                cout << "Inorder Successor of " << x->data;
                cout << " is " << root->data << "\n";
                return NULL;
            }
        }
    }

    return root;
}

return NULL;
}

// function to find inorder successor of
// a node
void inorderSuccessor(Node* root, Node* x)
{
    // Case1: If right child is not NULL
    if (x->right != NULL)
    {
```

```
Node* inorderSucc = leftMostNode(x->right);
cout<<"Inorder Successor of "<<x->data<<" is ";
cout<<inorderSucc->data<<"\n";
}

// Case2: If right child is NULL
if (x->right == NULL)
{
    int f = 0;

    Node* rightMost = rightMostNode(root);

    // case3: If x is the right most node
    if (rightMost == x)
        cout << "No inorder successor! Right most node.\n";
    else
        findInorderRecursive(root, x);
}
}

// Driver program to test above functions
int main()
{
    // Let's construct the binary tree
    // as shown in above diagram

    Node* root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->right = newNode(6);

    // Case 1
    inorderSuccesor(root, root->right);

    // case 2
    inorderSuccesor(root, root->left->left);

    // case 3
    inorderSuccesor(root, root->right->right);

    return 0;
}
```

Output:

Inorder Successor of 3 is 6  
Inorder Successor of 4 is 2  
No inorder successor! Right most node.

## Source

<https://www.geeksforgeeks.org/inorder-successor-node-binary-tree/>

## Chapter 217

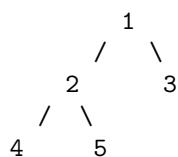
# Inorder Tree Traversal without Recursion

Inorder Tree Traversal without Recursion - GeeksforGeeks

Using [Stack](#) is the obvious way to traverse tree without recursion. Below is an algorithm for traversing binary tree using stack. See [this](#) for step wise step execution of the algorithm.

- 1) Create an empty stack S.
- 2) Initialize current node as root
- 3) Push the current node to S and set current = current->left until current is NULL
- 4) If current is NULL and stack is not empty then
  - a) Pop the top item from stack.
  - b) Print the popped item, set current = popped\_item->right
  - c) Go to step 3.
- 5) If current is NULL and stack is empty then we are done.

Let us consider the below tree for example



Step 1 Creates an empty stack: S = NULL

Step 2 sets current as address of root: current -> 1

Step 3 Pushes the current node and set current = current->left until current is NULL

```
current -> 1
push 1: Stack S -> 1
current -> 2
push 2: Stack S -> 2, 1
current -> 4
push 4: Stack S -> 4, 2, 1
current = NULL
```

Step 4 pops from S  
a) Pop 4: Stack S -> 2, 1  
b) print "4"  
c) current = NULL /\*right of 4 \*/ and go to step 3  
Since current is NULL step 3 doesn't do anything.

Step 4 pops again.  
a) Pop 2: Stack S -> 1  
b) print "2"  
c) current -> 5/\*right of 2 \*/ and go to step 3

Step 3 pushes 5 to stack and makes current NULL  
Stack S -> 5, 1  
current = NULL

Step 4 pops from S  
a) Pop 5: Stack S -> 1  
b) print "5"  
c) current = NULL /\*right of 5 \*/ and go to step 3  
Since current is NULL step 3 doesn't do anything

Step 4 pops again.  
a) Pop 1: Stack S -> NULL  
b) print "1"  
c) current -> 3 /\*right of 5 \*/

Step 3 pushes 3 to stack and makes current NULL  
Stack S -> 3  
current = NULL

Step 4 pops from S  
a) Pop 3: Stack S -> NULL  
b) print "3"  
c) current = NULL /\*right of 3 \*/

Traversal is done now as stack S is empty and current is NULL.

C++

```
// C++ program to print inorder traversal
```

```
// using stack.
#include<bits/stdc++.h>
using namespace std;

/* A binary tree Node has data, pointer to left child
   and a pointer to right child */
struct Node
{
    int data;
    struct Node* left;
    struct Node* right;
    Node (int data)
    {
        this->data = data;
        left = right = NULL;
    }
};

/* Iterative function for inorder tree
   traversal */
void inOrder(struct Node *root)
{
    stack<Node *> s;
    Node *curr = root;

    while (curr != NULL || s.empty() == false)
    {
        /* Reach the left most Node of the
           curr Node */
        while (curr != NULL)
        {
            /* place pointer to a tree node on
               the stack before traversing
               the node's left subtree */
            s.push(curr);
            curr = curr->left;
        }

        /* Current must be NULL at this point */
        curr = s.top();
        s.pop();

        cout << curr->data << " ";

        /* we have visited the node and its
           left subtree. Now, it's right
           subtree's turn */
        curr = curr->right;
    }
}
```

```
    } /* end of while */  
}  
  
/* Driver program to test above functions*/  
int main()  
{  
  
    /* Constructed binary tree is  
        1  
       /   \  
      2     3  
     /   \  
    4     5  
*/  
    struct Node *root = new Node(1);  
    root->left = new Node(2);  
    root->right = new Node(3);  
    root->left->left = new Node(4);  
    root->left->right = new Node(5);  
  
    inOrder(root);  
    return 0;  
}
```

C

```
#include<stdio.h>  
#include<stdlib.h>  
#define bool int  
  
/* A binary tree tNode has data, pointer to left child  
   and a pointer to right child */  
struct tNode  
{  
    int data;  
    struct tNode* left;  
    struct tNode* right;  
};  
  
/* Structure of a stack node. Linked List implementation is used for  
   stack. A stack node contains a pointer to tree node and a pointer to  
   next stack node */  
struct sNode  
{  
    struct tNode *t;  
    struct sNode *next;  
};
```

```

/* Stack related functions */
void push(struct sNode** top_ref, struct tNode *t);
struct tNode *pop(struct sNode** top_ref);
bool isEmpty(struct sNode *top);

/* Iterative function for inorder tree traversal */
void inOrder(struct tNode *root)
{
    /* set current to root of binary tree */
    struct tNode *current = root;
    struct sNode *s = NULL; /* Initialize stack s */
    bool done = 0;

    while (!done)
    {
        /* Reach the left most tNode of the current tNode */
        if(current != NULL)
        {
            /* place pointer to a tree node on the stack before traversing
               the node's left subtree */
            push(&s, current);
            current = current->left;
        }

        /* backtrack from the empty subtree and visit the tNode
           at the top of the stack; however, if the stack is empty,
           you are done */
        else
        {
            if (!isEmpty(s))
            {
                current = pop(&s);
                printf("%d ", current->data);

                /* we have visited the node and its left subtree.
                   Now, it's right subtree's turn */
                current = current->right;
            }
            else
                done = 1;
        }
    } /* end of while */
}

/* UTILITY FUNCTIONS */
/* Function to push an item to sNode*/
void push(struct sNode** top_ref, struct tNode *t)

```

```
{  
    /* allocate tNode */  
    struct sNode* new_tNode =  
        (struct sNode*) malloc(sizeof(struct sNode));  
  
    if(new_tNode == NULL)  
    {  
        printf("Stack Overflow \n");  
        getchar();  
        exit(0);  
    }  
  
    /* put in the data */  
    new_tNode->t = t;  
  
    /* link the old list off the new tNode */  
    new_tNode->next = (*top_ref);  
  
    /* move the head to point to the new tNode */  
    (*top_ref) = new_tNode;  
}  
  
/* The function returns true if stack is empty, otherwise false */  
bool isEmpty(struct sNode *top)  
{  
    return (top == NULL)? 1 : 0;  
}  
  
/* Function to pop an item from stack*/  
struct tNode *pop(struct sNode** top_ref)  
{  
    struct tNode *res;  
    struct sNode *top;  
  
    /*If sNode is empty then error */  
    if(isEmpty(*top_ref))  
    {  
        printf("Stack Underflow \n");  
        getchar();  
        exit(0);  
    }  
    else  
    {  
        top = *top_ref;  
        res = top->t;  
        *top_ref = top->next;  
        free(top);  
        return res;  
    }  
}
```

```
    }

}

/* Helper function that allocates a new tNode with the
   given data and NULL left and right pointers. */
struct tNode* newtNode(int data)
{
    struct tNode* tNode = (struct tNode*)
                           malloc(sizeof(struct tNode));
    tNode->data = data;
    tNode->left = NULL;
    tNode->right = NULL;

    return(tNode);
}

/* Driver program to test above functions*/
int main()
{

    /* Constructed binary tree is
       1
      / \
     2   3
    / \
   4   5
*/
    struct tNode *root = newtNode(1);
    root->left      = newtNode(2);
    root->right     = newtNode(3);
    root->left->left  = newtNode(4);
    root->left->right = newtNode(5);

    inOrder(root);

    getchar();
    return 0;
}
```

Java

```
// non-recursive java program for inorder traversal
import java.util.Stack;

/* Class containing left and right child of
current node and key value*/
class Node
{
```

```
int data;
Node left, right;

public Node(int item)
{
    data = item;
    left = right = null;
}
}

/* Class to print the inorder traversal */
class BinaryTree
{
    Node root;
    void inorder()
    {
        if (root == null)
            return;

        Stack<Node> s = new Stack<Node>();
        Node curr = root;

        // traverse the tree
        while (curr != null || s.size() > 0)
        {

            /* Reach the left most Node of the
               curr Node */
            while (curr != null)
            {
                /* place pointer to a tree node on
                   the stack before traversing
                   the node's left subtree */
                s.push(curr);
                curr = curr.left;
            }

            /* Current must be NULL at this point */
            curr = s.pop();

            System.out.print(curr.data + " ");

            /* we have visited the node and its
               left subtree. Now, it's right
               subtree's turn */
            curr = curr.right;
        }
    }
}
```

```
}

public static void main(String args[])
{
    /* creating a binary tree and entering
       the nodes */
    BinaryTree tree = new BinaryTree();
    tree.root = new Node(1);
    tree.root.left = new Node(2);
    tree.root.right = new Node(3);
    tree.root.left.left = new Node(4);
    tree.root.left.right = new Node(5);
    tree.inorder();
}
}
```

### Python

```
# Python program to do inorder traversal without recursion

# A binary tree node
class Node:

    # Constructor to create a new node
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

# Iterative function for inorder tree traversal
def inOrder(root):

    # Set current to root of binary tree
    current = root
    s = [] # initialize stack
    done = 0

    while(not done):

        # Reach the left most Node of the current Node
        if current is not None:

            # Place pointer to a tree node on the stack
            # before traversing the node's left subtree
            s.append(current)

            current = current.left

        else:
            # Current must be None
            # Pop an item from the stack
            current = s.pop()
            print current.data,
```

```
# BackTrack from the empty subtree and visit the Node
# at the top of the stack; however, if the stack is
# empty you are done
else:
    if(len(s) >0 ):
        current = s.pop()
        print current.data,

        # We have visited the node and its left
        # subtree. Now, it's right subtree's turn
        current = current.right

    else:
        done = 1

# Driver program to test above function

""" Constructed binary tree is
      1
     /   \
    2     3
   /   \
  4     5
"""
root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)

inOrder(root)

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

Time Complexity: O(n)

Output:

```
4 2 5 1 3
```

References:

<http://web.cs.wpi.edu/~cs2005/common/iterative.inorder>

<http://neural.cs.nthu.edu.tw/jang/courses/cs2351/slides/animation/Iterative%20Inorder%20Traversal.pps>

See [this post](#) for another approach of Inorder Tree Traversal without recursion and without stack!

**Improved By :** [Rishabh Jindal 2](#)

## Source

<https://www.geeksforgeeks.org/inorder-tree-traversal-without-recursion/>

## Chapter 218

# Inorder Tree Traversal without recursion and without stack!

Inorder Tree Traversal without recursion and without stack! - GeeksforGeeks

Using Morris Traversal, we can traverse the tree without using stack and recursion. The idea of Morris Traversal is based on [Threaded Binary Tree](#). In this traversal, we first create links to Inorder successor and print the data using these links, and finally revert the changes to restore original tree.

1. Initialize current as root
2. While current is not NULL
  - If current does not have left child
    - a) Print current's data
    - b) Go to the right, i.e., current = current->right
  - Else
    - a) Make current as right child of the rightmost node in current's left subtree
    - b) Go to this left child, i.e., current = current->left

Although the tree is modified through the traversal, it is reverted back to its original shape after the completion. Unlike [Stack based traversal](#), no extra space is required for this traversal.

C++

```
#include <stdio.h>
#include <stdlib.h>

/* A binary tree tNode has data, pointer to left child
   and a pointer to right child */
struct tNode {
```

```
int data;
struct tNode* left;
struct tNode* right;
};

/* Function to traverse binary tree without recursion and
   without stack */
void MorrisTraversal(struct tNode* root)
{
    struct tNode *current, *pre;

    if (root == NULL)
        return;

    current = root;
    while (current != NULL) {

        if (current->left == NULL) {
            printf("%d ", current->data);
            current = current->right;
        }
        else {

            /* Find the inorder predecessor of current */
            pre = current->left;
            while (pre->right != NULL && pre->right != current)
                pre = pre->right;

            /* Make current as right child of its inorder
               predecessor */
            if (pre->right == NULL) {
                pre->right = current;
                current = current->left;
            }

            /* Revert the changes made in if part to restore
               the original tree i.e., fix the right child
               of predecessor */
            else {
                pre->right = NULL;
                printf("%d ", current->data);
                current = current->right;
            } /* End of if condition pre->right == NULL */
        } /* End of if condition current->left == NULL*/
    } /* End of while */
}

/* UTILITY FUNCTIONS */
```

```
/* Helper function that allocates a new tNode with the
   given data and NULL left and right pointers. */
struct tNode* newtNode(int data)
{
    struct tNode* node = new tNode;
    node->data = data;
    node->left = NULL;
    node->right = NULL;

    return (node);
}

/* Driver program to test above functions*/
int main()
{
    /* Constructed binary tree is
       1
      / \
     2   3
    / \
   4   5
*/
    struct tNode* root = newtNode(1);
    root->left = newtNode(2);
    root->right = newtNode(3);
    root->left->left = newtNode(4);
    root->left->right = newtNode(5);

    MorrisTraversal(root);

    return 0;
}
```

**Java**

```
// Java program to print inorder traversal without recursion and stack

/* A binary tree tNode has data, pointer to left child
   and a pointer to right child */
class tNode {
    int data;
    tNode left, right;

    tNode(int item)
    {
        data = item;
        left = right = null;
```

```
    }
}

class BinaryTree {
    tNode root;

    /* Function to traverse binary tree without recursion and
       without stack */
    void MorrisTraversal(tNode root)
    {
        tNode current, pre;

        if (root == null)
            return;

        current = root;
        while (current != null) {
            if (current.left == null) {
                System.out.print(current.data + " ");
                current = current.right;
            }
            else {
                /* Find the inorder predecessor of current */
                pre = current.left;
                while (pre.right != null && pre.right != current)
                    pre = pre.right;

                /* Make current as right child of its inorder predecessor */
                if (pre.right == null) {
                    pre.right = current;
                    current = current.left;
                }

                /* Revert the changes made in if part to restore the
                   original tree i.e., fix the right child of predecessor*/
                else {
                    pre.right = null;
                    System.out.print(current.data + " ");
                    current = current.right;
                } /* End of if condition pre->right == NULL */

            } /* End of if condition current->left == NULL*/

        } /* End of while */
    }

    public static void main(String args[])
    {
```

```
/* Constructed binary tree is
      1
     /   \
    2     3
   /   \
  4     5
*/
BinaryTree tree = new BinaryTree();
tree.root = new tNode(1);
tree.root.left = new tNode(2);
tree.root.right = new tNode(3);
tree.root.left.left = new tNode(4);
tree.root.left.right = new tNode(5);

tree.MorrisTraversal(tree.root);
}
}

// This code has been contributed by Mayank Jaiswal(mayank_24)
```

### Python

```
# Python program to do inorder traversal without recursion and
# without stack Morris inOrder Traversal

# A binary tree node
class Node:

    # Constructor to create a new node
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

    # Iterative function for inorder tree traversal
    def MorrisTraversal(root):

        # Set current to root of binary tree
        current = root

        while(current is not None):

            if current.left is None:
                print current.data,
                current = current.right
            else:
                # Find the inorder predecessor of current
                pre = current.left
```

```
while(pre.right is not None and pre.right != current):
    pre = pre.right

    # Make current as right child of its inorder predecessor
    if(pre.right is None):
        pre.right = current
        current = current.left

    # Revert the changes made in if part to restore the
    # original tree i.e., fix the right child of predecessor
    else:
        pre.right = None
        print current.data,
        current = current.right

# Driver program to test above function
"""
Constructed binary tree is
      1
     /   \
    2     3
   /   \
  4     5
"""
root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)

MorrisTraversal(root)

# This code is contributed by Naveen Aili
```

Output:

4 2 5 1 3

Time Complexity : O(n) If we take a closer look, we can notice that every edge of the tree is traversed at-most two times. And in worst case same number of extra edges (as input tree) are created and removed.

References:

[www.liacs.nl/~deutz/DS/september28.pdf](http://www.liacs.nl/~deutz/DS/september28.pdf)

[www.scss.tcd.ie/disciplines/software\\_systems/.../HughGibbonsSlides.pdf](http://www.scss.tcd.ie/disciplines/software_systems/.../HughGibbonsSlides.pdf)

## Source

<https://www.geeksforgeeks.org/inorder-tree-traversal-without-recursion-and-without-stack/>

## Chapter 219

# Inorder predecessor and successor for a given key in BST

Inorder predecessor and successor for a given key in BST - GeeksforGeeks

I recently encountered with a question in an interview at e-commerce company. The interviewer asked the following question:

There is BST given with root node with key part as integer only. The structure of each node is as follows:

```
struct Node
{
    int key;
    struct Node *left, *right ;
};
```

You need to find the inorder successor and predecessor of a given key. In case the given key is not found in BST, then return the two values within which this key will lie.

Following is the algorithm to reach the desired result. Its a recursive method:

Input: root node, key  
output: predecessor node, successor node

1. If root is NULL  
    then return
2. if key is found then
  - a. If its left subtree is not null  
        Then predecessor will be the right most  
        child of left subtree or left child itself.
  - b. If its right subtree is not null

```
The successor will be the left most child
of right subtree or right child itself.

return
3. If key is smaller then root node
    set the successor as root
    search recursively into left subtree
else
    set the predecessor as root
    search recursively into right subtree
```

Following is C++ implementation of the above algorithm:

C++

```
// C++ program to find predecessor and successor in a BST
#include <iostream>
using namespace std;

// BST Node
struct Node
{
    int key;
    struct Node *left, *right;
};

// This function finds predecessor and successor of key in BST.
// It sets pre and suc as predecessor and successor respectively
void findPreSuc(Node* root, Node*& pre, Node*& suc, int key)
{
    // Base case
    if (root == NULL)  return ;

    // If key is present at root
    if (root->key == key)
    {
        // the maximum value in left subtree is predecessor
        if (root->left != NULL)
        {
            Node* tmp = root->left;
            while (tmp->right)
                tmp = tmp->right;
            pre = tmp ;
        }

        // the minimum value in right subtree is successor
        if (root->right != NULL)
        {
            Node* tmp = root->right ;
            while (tmp->left)
                tmp = tmp->left;
            suc = tmp ;
        }
    }
}
```

```
        while (tmp->left)
            tmp = tmp->left ;
        suc = tmp ;
    }
    return ;
}

// If key is smaller than root's key, go to left subtree
if (root->key > key)
{
    suc = root ;
    findPreSuc(root->left, pre, suc, key) ;
}
else // go to right subtree
{
    pre = root ;
    findPreSuc(root->right, pre, suc, key) ;
}
}

// A utility function to create a new BST node
Node *newNode(int item)
{
    Node *temp = new Node;
    temp->key = item;
    temp->left = temp->right = NULL;
    return temp;
}

/* A utility function to insert a new node with given key in BST */
Node* insert(Node* node, int key)
{
    if (node == NULL) return newNode(key);
    if (key < node->key)
        node->left = insert(node->left, key);
    else
        node->right = insert(node->right, key);
    return node;
}

// Driver program to test above function
int main()
{
    int key = 65;      //Key to be searched in BST

    /* Let us create following BST
           50
         /     \

```

```
      30      70
     / \   / \
    20 40 60 80 */
Node *root = NULL;
root = insert(root, 50);
insert(root, 30);
insert(root, 20);
insert(root, 40);
insert(root, 70);
insert(root, 60);
insert(root, 80);

Node* pre = NULL, *suc = NULL;

findPreSuc(root, pre, suc, key);
if (pre != NULL)
    cout << "Predecessor is " << pre->key << endl;
else
    cout << "No Predecessor";

if (suc != NULL)
    cout << "Successor is " << suc->key;
else
    cout << "No Successor";
return 0;
}
```

### Python

```
# Python program to find predecessor and successor in a BST

# A BST node
class Node:

    # Constructor to create a new node
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None

    # This function finds predecessor and successor of key in BST
    # It sets pre and suc as predecessor and successor respectively
    def findPreSuc(self, key):

        # Base Case
        if root is None:
            return
```

```
# If key is present at root
if root.key == key:

    # the maximum value in left subtree is predecessor
    if root.left is not None:
        tmp = root.left
        while(tmp.right):
            tmp = tmp.right
        findPreSuc.pre = tmp

    # the minimum value in right subtree is successor
    if root.right is not None:
        tmp = root.right
        while(temp.left):
            tmp = tmp.left
        findPreSuc.suc = tmp

    return

# If key is smaller than root's key, go to left subtree
if root.key > key :
    findPreSuc.suc = root
    findPreSuc(root.left, key)

else: # go to right subtree
    findPreSuc.pre = root
    findPreSuc(root.right, key)

# A utility function to insert a new node in with given key in BST
def insert(node , key):
    if node is None:
        return Node(key)

    if key < node.key:
        node.left = insert(node.left, key)

    else:
        node.right = insert(node.right, key)

    return node

# Driver program to test above function
key = 65 #Key to be searched in BST

""" Let us create following BST
```

```
      50
     /   \
    30   70
   / \   / \
  20 40 60 80
"""
root = None
root = insert(root, 50)
insert(root, 30);
insert(root, 20);
insert(root, 40);
insert(root, 70);
insert(root, 60);
insert(root, 80);

# Static variables of the function findPreSuc
findPreSuc.pre = None
findPreSuc.suc = None

findPreSuc(root, key)

if findPreSuc.pre is not None:
    print "Predecessor is", findPreSuc.pre.key

else:
    print "No Predecessor"

if findPreSuc.suc is not None:
    print "Successor is", findPreSuc.suc.key
else:
    print "No Successor"

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

Output:

```
Predecessor is 60
Successor is 70
```

#### Another Approach :

We can also find the inorder successor and inorder predecessor using inorder traversal . Check if the current node is smaller than the given key for predecessor and for successor, check if it is greater than the given key . If it is greater than the given key then, check if it is smaller than the already stored value in successor then, update it . At last, get the predecessor and successor stored in q(successor) and p(predecessor).

```
// CPP code for inorder successor
```

```
// and predecessor of tree
#include<iostream>
#include<stdlib.h>

using namespace std;

struct Node
{
    int data;
    Node* left,*right;
};

// Function to return data
Node* getnode(int info)
{
    Node* p = (Node*)malloc(sizeof(Node));
    p->data = info;
    p->right = NULL;
    p->left = NULL;
    return p;
}

/*
since inorder traversal results in
ascending order visit to node , we
can store the values of the largest
no which is smaller than a (predecessor)
and smallest no which is large than
a (succesor) using inorder traversal
*/
void find_p_s(Node* root,int a,
               Node** p, Node** q)
{
    // If root is null return
    if(!root)
        return ;

    // traverse the left subtree
    find_p_s(root->left, a, p);

    // root data is greater than a
    if(root&&root->data > a)
    {

        // q stores the node whose data is greater
        // than a and is smaller than the previously
        // stored data in *q which is sucessor
        if((!*q) || (*q) && (*q)->data > root->data)
```

```
*q = root;
}

// if the root data is smaller than
// store it in p which is predecessor
else if(root && root->data < a)
{
    *p = root;
}

// traverse the right subtree
find_p_s(root->right, a, p, q);
}

// Driver code
int main()
{
    Node* root1 = getnode(50);
    root1->left = getnode(20);
    root1->right = getnode(60);
    root1->left->left = getnode(10);
    root1->left->right = getnode(30);
    root1->right->left = getnode(55);
    root1->right->right = getnode(70);
    Node* p = NULL, *q = NULL;

    find_p_s(root1, 55, &p, &q);

    if(p)
        cout << p->data;
    if(q)
        cout << " " << q->data;
    return 0;
}
```

Output :

50 60

Thanks **Shweta** for suggesting this method.

This article is contributed by **algoLover**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## Source

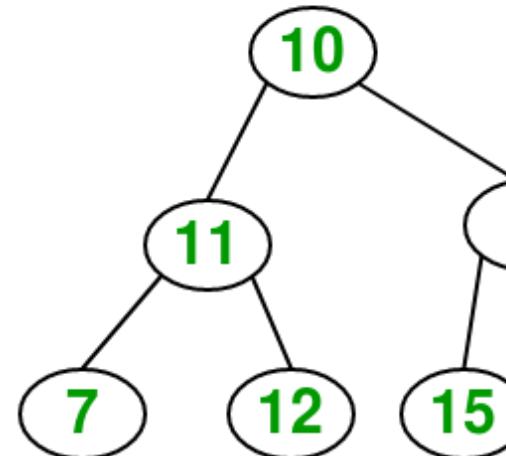
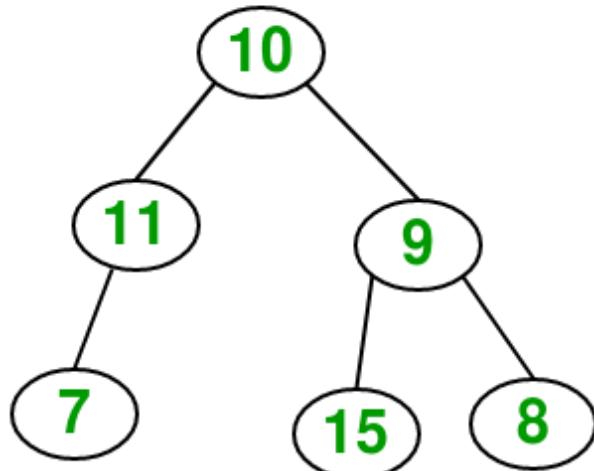
<https://www.geeksforgeeks.org/inorder-predecessor-successor-given-key-bst/>

## Chapter 220

### Insertion in a Binary Tree

Insertion in a Binary Tree - GeeksforGeeks

Given a binary tree and a key, insert the key into the binary tree at first position available in [level order](#).



After inserting

The idea is to do iterative level order traversal of the given tree using [queue](#). If we find a node whose left child is empty, we make new key as left child of the node. Else if we find a node whose right child is empty, we make new key as right child. We keep traversing the tree until we find a node whose either left or right is empty.

C++

```
// C++ program to insert element in binary tree
#include <iostream>
#include <queue>
using namespace std;

/* A binary tree node has key, pointer to left child
and a pointer to right child */
struct Node {
    int key;
    struct Node* left, *right;
};

/* function to create a new node of tree and r
eturns pointer */
struct Node* newNode(int key)
{
    struct Node* temp = new Node;
    temp->key = key;
    temp->left = temp->right = NULL;
    return temp;
}

/* Inorder traversal of a binary tree*/
void inorder(struct Node* temp)
{
    if (!temp)
        return;

    inorder(temp->left);
    cout << temp->key << " ";
    inorder(temp->right);
}

/*function to insert element in binary tree */
void insert(struct Node* temp, int key)
{
    queue<struct Node*> q;
    q.push(temp);

    // Do level order traversal until we find
    // an empty place.
    while (!q.empty()) {
        struct Node* temp = q.front();
        q.pop();

        if (!temp->left) {
            temp->left = newNode(key);
            break;
        }
    }
}
```

```
    } else
        q.push(temp->left);

    if (!temp->right) {
        temp->right = newNode(key);
        break;
    } else
        q.push(temp->right);
}
}

// Driver code
int main()
{
    struct Node* root = newNode(10);
    root->left = newNode(11);
    root->left->left = newNode(7);
    root->right = newNode(9);
    root->right->left = newNode(15);
    root->right->right = newNode(8);

    cout << "Inorder traversal before insertion:";
    inorder(root);

    int key = 12;
    insert(root, key);

    cout << endl;
    cout << "Inorder traversal after insertion:";
    inorder(root);

    return 0;
}
```

**Java**

```
// Java program to insert element in binary tree
import java.util.LinkedList;
import java.util.Queue;
public class GFG {

    /* A binary tree node has key, pointer to
    left child and a pointer to right child */
    static class Node {
        int key;
        Node left, right;

        // constructor
```

```
Node(int key){  
    this.key = key;  
    left = null;  
    right = null;  
}  
}  
static Node root;  
static Node temp = root;  
  
/* Inorder traversal of a binary tree*/  
static void inorder(Node temp)  
{  
    if (temp == null)  
        return;  
  
    inorder(temp.left);  
    System.out.print(temp.key+" ");  
    inorder(temp.right);  
}  
  
/*function to insert element in binary tree */  
static void insert(Node temp, int key)  
{  
    Queue<Node> q = new LinkedList<Node>();  
    q.add(temp);  
  
    // Do level order traversal until we find  
    // an empty place.  
    while (!q.isEmpty()) {  
        temp = q.peek();  
        q.remove();  
  
        if (temp.left == null) {  
            temp.left = new Node(key);  
            break;  
        } else  
            q.add(temp.left);  
  
        if (temp.right == null) {  
            temp.right = new Node(key);  
            break;  
        } else  
            q.add(temp.right);  
    }  
}  
  
// Driver code  
public static void main(String args[])
```

```
{  
    root = new Node(10);  
    root.left = new Node(11);  
    root.left.left = new Node(7);  
    root.right = new Node(9);  
    root.right.left = new Node(15);  
    root.right.right = new Node(8);  
  
    System.out.print("Inorder traversal before insertion:");  
    inorder(root);  
  
    int key = 12;  
    insert(root, key);  
  
    System.out.print("\nInorder traversal after insertion:");  
    inorder(root);  
}  
}  
// This code is contributed by Sumit Ghosh
```

Output:

```
Inorder traversal before insertion: 7 11 10 15 9 8  
Inorder traversal after insertion: 7 11 12 10 15 9 8
```

## Source

<https://www.geeksforgeeks.org/insertion-binary-tree/>

## Chapter 221

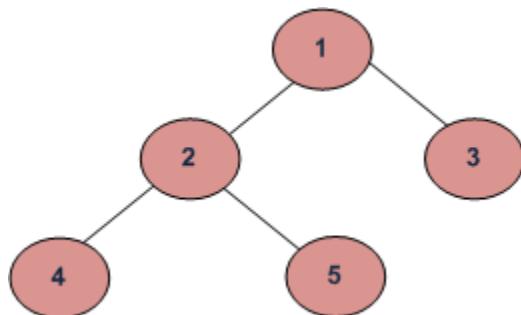
# Iterative Method to find Height of Binary Tree

Iterative Method to find Height of Binary Tree - GeeksforGeeks

There are two conventions to define height of Binary Tree

- 1) Number of nodes on longest path from root to the deepest node.
- 2) Number of edges on longest path from root to the deepest node.

In this post, the first convention is followed. For example, height of the below tree is 3.



Example Tree

Recursive method to find height of Binary Tree is discussed [here](#). How to find height without recursion? We can use level order traversal to find height without recursion. The idea is to traverse level by level. Whenever move down to a level, increment height by 1 (height is initialized as 0). Count number of nodes at each level, stop traversing when count of nodes at next level is 0.

Following is detailed algorithm to find level order traversal using queue.

Create a queue.

```
Push root into the queue.  
height = 0  
Loop  
    nodeCount = size of queue  
  
    // If number of nodes at this level is 0, return height  
    if nodeCount is 0  
        return Height;  
    else  
        increase Height  
  
    // Remove nodes of this level and add nodes of  
    // next level  
    while (nodeCount > 0)  
        pop node from front  
        push its children to queue  
        decrease nodeCount  
    // At this point, queue has nodes of next level
```

Following is the implementation of above algorithm.

C++

```
/* Program to find height of the tree by Iterative Method */  
#include <iostream>  
#include <queue>  
using namespace std;  
  
// A Binary Tree Node  
struct node  
{  
    struct node *left;  
    int data;  
    struct node *right;  
};  
  
// Iterative method to find height of Binary Tree  
int treeHeight(node *root)  
{  
    // Base Case  
    if (root == NULL)  
        return 0;  
  
    // Create an empty queue for level order traversal  
    queue<node *> q;  
  
    // Enqueue Root and initialize height  
    q.push(root);
```

```
int height = 0;

while (1)
{
    // nodeCount (queue size) indicates number of nodes
    // at current level.
    int nodeCount = q.size();
    if (nodeCount == 0)
        return height;

    height++;

    // Dequeue all nodes of current level and Enqueue all
    // nodes of next level
    while (nodeCount > 0)
    {
        node *node = q.front();
        q.pop();
        if (node->left != NULL)
            q.push(node->left);
        if (node->right != NULL)
            q.push(node->right);
        nodeCount--;
    }
}

// Utility function to create a new tree node
node* newNode(int data)
{
    node *temp = new node;
    temp->data = data;
    temp->left = NULL;
    temp->right = NULL;
    return temp;
}

// Driver program to test above functions
int main()
{
    // Let us create binary tree shown in above diagram
    node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);

    cout << "Height of tree is " << treeHeight(root);
```

```
    return 0;
}
```

**Java**

```
// An iterative java program to find height of binary tree

import java.util.LinkedList;
import java.util.Queue;

// A binary tree node
class Node
{
    int data;
    Node left, right;

    Node(int item)
    {
        data = item;
        left = right;
    }
}

class BinaryTree
{
    Node root;

    // Iterative method to find height of Bianry Tree
    int treeHeight(Node node)
    {
        // Base Case
        if (node == null)
            return 0;

        // Create an empty queue for level order tarversal
        Queue<Node> q = new LinkedList();

        // Enqueue Root and initialize height
        q.add(node);
        int height = 0;

        while (1 == 1)
        {
            // nodeCount (queue size) indicates number of nodes
            // at current level.
            int nodeCount = q.size();
            if (nodeCount == 0)
                return height;

            ... (remaining code for level-order traversal and height calculation)
        }
    }
}
```

```
height++;

// Dequeue all nodes of current level and Enqueue all
// nodes of next level
while (nodeCount > 0)
{
    Node newnode = q.peek();
    q.remove();
    if (newnode.left != null)
        q.add(newnode.left);
    if (newnode.right != null)
        q.add(newnode.right);
    nodeCount--;
}
}

// Driver program to test above functions
public static void main(String args[])
{
    BinaryTree tree = new BinaryTree();

    // Let us create a binary tree shown in above diagram
    tree.root = new Node(1);
    tree.root.left = new Node(2);
    tree.root.right = new Node(3);
    tree.root.left.left = new Node(4);
    tree.root.left.right = new Node(5);
    System.out.println("Height of tree is " + tree.treeHeight(tree.root));
}
}

// This code has been contributed by Mayank Jaiswal
```

### Python

```
# Program to find height of tree by Iteration Method

# A binary tree node
class Node:

    # Constructor to create new node
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

# Iterative method to find height of Binary Tree
```

```
def treeHeight(root):

    # Base Case
    if root is None:
        return 0

    # Create a empty queue for level order traversal
    q = []

    # Enqueue Root and Initialize Height
    q.append(root)
    height = 0

    while(True):

        # nodeCount(queue size) indicates number of nodes
        # at current level
        nodeCount = len(q)
        if nodeCount == 0 :
            return height

        height += 1

        # Dequeue all nodes of current level and Enqueue
        # all nodes of next level
        while(nodeCount > 0):
            node = q[0]
            q.pop(0)
            if node.left is not None:
                q.append(node.left)
            if node.right is not None:
                q.append(node.right)

            nodeCount -= 1

    # Driver program to test above function
    # Let us create binary tree shown in above diagram
    root = Node(1)
    root.left = Node(2)
    root.right = Node(3)
    root.left.left = Node(4)
    root.left.right = Node(5)

    print "Height of tree is", treeHeight(root)

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

Output:

```
Height of tree is 3
```

**Time Complexity:**  $O(n)$  where  $n$  is number of nodes in given binary tree.

This article is contributed by [Rahul Kumar](#). Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## Source

<https://www.geeksforgeeks.org/iterative-method-to-find-height-of-binary-tree/>

## Chapter 222

# Iterative Postorder Traversal | Set 1 (Using Two Stacks)

[Iterative Postorder Traversal | Set 1 \(Using Two Stacks\) - GeeksforGeeks](#)

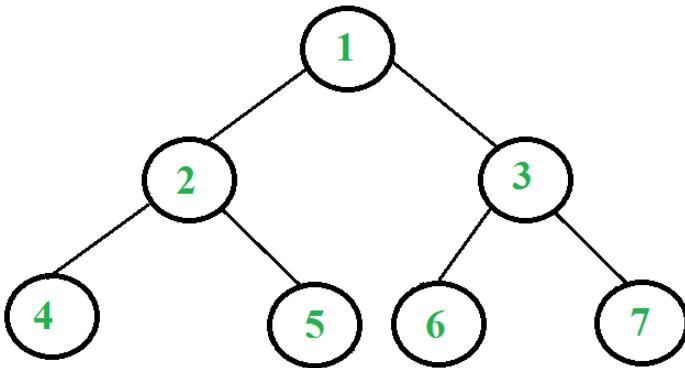
We have discussed [iterative inorder](#) and [iterative preorder](#) traversals. In this post, iterative postorder traversal is discussed, which is more complex than the other two traversals (due to its nature of non-tail recursion, there is an extra statement after the final recursive call to itself). Postorder traversal can easily be done using two stacks, though. The idea is to push reverse postorder traversal to a stack. Once we have the reversed postorder traversal in a stack, we can just pop all items one by one from the stack and print them; this order of printing will be in postorder because of the LIFO property of stacks. Now the question is, how to get reversed postorder elements in a stack – the second stack is used for this purpose. For example, in the following tree, we need to get 1, 3, 7, 6, 2, 5, 4 in a stack. If take a closer look at this sequence, we can observe that this sequence is very similar to the preorder traversal. The only difference is that the right child is visited before left child, and therefore the sequence is “root right left” instead of “root left right”. So, we can do something like [iterative preorder traversal](#) with the following differences:

- a) Instead of printing an item, we push it to a stack.
- b) We push the left subtree before the right subtree.

Following is the complete algorithm. After step 2, we get the reverse of a postorder traversal in the second stack. We use the first stack to get the correct order.

1. Push root to first stack.
2. Loop while first stack is not empty
  - 2.1 Pop a node from first stack and push it to second stack
  - 2.2 Push left and right children of the popped node to first stack
3. Print contents of second stack

Let us consider the following tree



Following are the steps to print postorder traversal of the above tree using two stacks.

1. Push 1 to first stack.  
First stack: 1  
Second stack: Empty
2. Pop 1 from first stack and push it to second stack.  
Push left and right children of 1 to first stack  
First stack: 2, 3  
Second stack: 1
3. Pop 3 from first stack and push it to second stack.  
Push left and right children of 3 to first stack  
First stack: 2, 6, 7  
Second stack: 1, 3
4. Pop 7 from first stack and push it to second stack.  
First stack: 2, 6  
Second stack: 1, 3, 7
5. Pop 6 from first stack and push it to second stack.  
First stack: 2  
Second stack: 1, 3, 7, 6
6. Pop 2 from first stack and push it to second stack.  
Push left and right children of 2 to first stack  
First stack: 4, 5  
Second stack: 1, 3, 7, 6, 2
7. Pop 5 from first stack and push it to second stack.  
First stack: 4  
Second stack: 1, 3, 7, 6, 2, 5
8. Pop 4 from first stack and push it to second stack.

```
First stack: Empty
Second stack: 1, 3, 7, 6, 2, 5, 4
```

The algorithm stops here since there are no more items in the first stack.  
Observe that the contents of second stack are in postorder fashion. Print them.

Following is C implementation of iterative postorder traversal using two stacks.

C

```
#include <stdio.h>
#include <stdlib.h>

// Maximum stack size
#define MAX_SIZE 100

// A tree node
struct Node {
    int data;
    struct Node *left, *right;
};

// Stack type
struct Stack {
    int size;
    int top;
    struct Node** array;
};

// A utility function to create a new tree node
struct Node* newNode(int data)
{
    struct Node* node = (struct Node*)malloc(sizeof(struct Node));
    node->data = data;
    node->left = node->right = NULL;
    return node;
}

// A utility function to create a stack of given size
struct Stack* createStack(int size)
{
    struct Stack* stack = (struct Stack*)malloc(sizeof(struct Stack));
    stack->size = size;
    stack->top = -1;
    stack->array = (struct Node**)malloc(stack->size * sizeof(struct Node*));
    return stack;
}
```

```
// BASIC OPERATIONS OF STACK
int isFull(struct Stack* stack)
{
    return stack->top - 1 == stack->size;
}

int isEmpty(struct Stack* stack)
{
    return stack->top == -1;
}

void push(struct Stack* stack, struct Node* node)
{
    if (isFull(stack))
        return;
    stack->array[+stack->top] = node;
}

struct Node* pop(struct Stack* stack)
{
    if (isEmpty(stack))
        return NULL;
    return stack->array[stack->top--];
}

// An iterative function to do post order traversal of a given binary tree
void postOrderIterative(struct Node* root)
{
    if (root == NULL)
        return;

    // Create two stacks
    struct Stack* s1 = createStack(MAX_SIZE);
    struct Stack* s2 = createStack(MAX_SIZE);

    // push root to first stack
    push(s1, root);
    struct Node* node;

    // Run while first stack is not empty
    while (!isEmpty(s1)) {
        // Pop an item from s1 and push it to s2
        node = pop(s1);
        push(s2, node);

        // Push left and right children of removed item to s1
        if (node->left)
```

```
        push(s1, node->left);
        if (node->right)
            push(s1, node->right);
    }

    // Print all elements of second stack
    while (!isEmpty(s2)) {
        node = pop(s2);
        printf("%d ", node->data);
    }
}

// Driver program to test above functions
int main()
{
    // Let us construct the tree shown in above figure
    struct Node* root = NULL;
    root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->left = newNode(6);
    root->right->right = newNode(7);

    postOrderIterative(root);

    return 0;
}
```

**Java**

```
// Java program for iterative post
// order using two stacks

import java.util.*;
public class IterativePostorder {

    static class node {
        int data;
        node left, right;

        public node(int data)
        {
            this.data = data;
        }
    }
}
```

```
// Two stacks as used in explanation
static Stack<node> s1, s2;

static void postOrderIterative(node root)
{
    // Create two stacks
    s1 = new Stack<>();
    s2 = new Stack<>();

    if (root == null)
        return;

    // push root to first stack
    s1.push(root);

    // Run while first stack is not empty
    while (!s1.isEmpty()) {
        // Pop an item from s1 and push it to s2
        node temp = s1.pop();
        s2.push(temp);

        // Push left and right children of
        // removed item to s1
        if (temp.left != null)
            s1.push(temp.left);
        if (temp.right != null)
            s1.push(temp.right);
    }

    // Print all elements of second stack
    while (!s2.isEmpty()) {
        node temp = s2.pop();
        System.out.print(temp.data + " ");
    }
}

public static void main(String[] args)
{
    // Let us construct the tree
    // shown in above figure

    node root = null;
    root = new node(1);
    root.left = new node(2);
    root.right = new node(3);
    root.left.left = new node(4);
    root.left.right = new node(5);
    root.right.left = new node(6);
```

```
    root.right.right = new node(7);

    postOrderIterative(root);
}
}

// This code is contributed by Rishabh Mahrsee
```

### Python

```
# Python program for iterative postorder traversal using
# two stacks

# A binary tree node
class Node:

    # Constructor to create a new node
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

# An iterative function to do postorder traversal of a
# given binary tree
def postOrderIterative(root):

    if root is None:
        return

    # Create two stacks
    s1 = []
    s2 = []

    # Push root to first stack
    s1.append(root)

    # Run while first stack is not empty
    while s1:

        # Pop an item from s1 and append it to s2
        node = s1.pop()
        s2.append(node)

        # Push left and right children of removed item to s1
        if node.left:
            s1.append(node.left)
        if node.right:
            s1.append(node.right)
```

```
# Print all elements of second stack
while s2:
    node = s2.pop()
    print node.data,

# Driver program to test above function
root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)
root.right.left = Node(6)
root.right.right = Node(7)
postOrderIterative(root)
```

Output:

4 5 2 6 7 3 1

Following is an overview of the above post.

Iterative preorder traversal can be easily implemented using two stacks. The first stack is used to get the reverse postorder traversal. The steps to get a reverse postorder are similar to [iterative preorder](#).

You may also like to see [a method which uses only one stack](#).

This article is compiled by [Aashish Barnwal](#). Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

**Improved By :** [IshitaTripathi](#)

## Source

<https://www.geeksforgeeks.org/iterative-postorder-traversal/>

## Chapter 223

# Iterative Postorder Traversal | Set 2 (Using One Stack)

[Iterative Postorder Traversal | Set 2 \(Using One Stack\) - GeeksforGeeks](#)

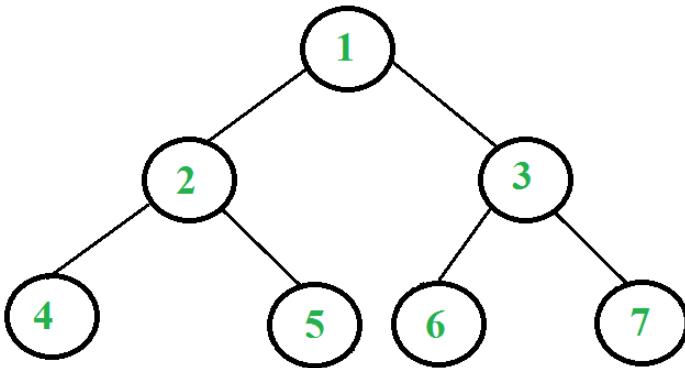
We have discussed a simple [iterative postorder traversal using two stacks](#) in the previous post. In this post, an approach with only one stack is discussed.

The idea is to move down to leftmost node using left pointer. While moving down, push root and root's right child to stack. Once we reach leftmost node, print it if it doesn't have a right child. If it has a right child, then change root so that the right child is processed before.

Following is detailed algorithm.

- 1.1 Create an empty stack
- 2.1 Do following while root is not NULL
  - a) Push root's right child and then root to stack.
  - b) Set root as root's left child.
- 2.2 Pop an item from stack and set it as root.
  - a) If the popped item has a right child and the right child is at top of stack, then remove the right child from stack, push the root back and set root as root's right child.
  - b) Else print root's data and set root as NULL.
- 2.3 Repeat steps 2.1 and 2.2 while stack is not empty.

Let us consider the following tree



Following are the steps to print postorder traversal of the above tree using one stack.

1. Right child of 1 exists.  
Push 3 to stack. Push 1 to stack. Move to left child.  
Stack: 3, 1
2. Right child of 2 exists.  
Push 5 to stack. Push 2 to stack. Move to left child.  
Stack: 3, 1, 5, 2
3. Right child of 4 doesn't exist.  
Push 4 to stack. Move to left child.  
Stack: 3, 1, 5, 2, 4
4. Current node is NULL.  
Pop 4 from stack. Right child of 4 doesn't exist.  
Print 4. Set current node to NULL.  
Stack: 3, 1, 5, 2
5. Current node is NULL.  
Pop 2 from stack. Since right child of 2 equals stack top element,  
pop 5 from stack. Now push 2 to stack.  
Move current node to right child of 2 i.e. 5  
Stack: 3, 1, 2
6. Right child of 5 doesn't exist. Push 5 to stack. Move to left child.  
Stack: 3, 1, 2, 5
7. Current node is NULL. Pop 5 from stack. Right child of 5 doesn't exist.  
Print 5. Set current node to NULL.  
Stack: 3, 1, 2
8. Current node is NULL. Pop 2 from stack.  
Right child of 2 is not equal to stack top element.

- Print 2. Set current node to NULL.  
Stack: 3, 1
9. Current node is NULL. Pop 1 from stack.  
Since right child of 1 equals stack top element, pop 3 from stack.  
Now push 1 to stack. Move current node to right child of 1 i.e. 3  
Stack: 1
10. Repeat the same as above steps and Print 6, 7 and 3.  
Pop 1 and Print 1.

C

```
// C program for iterative postorder traversal using one stack
#include <stdio.h>
#include <stdlib.h>

// Maximum stack size
#define MAX_SIZE 100

// A tree node
struct Node
{
    int data;
    struct Node *left, *right;
};

// Stack type
struct Stack
{
    int size;
    int top;
    struct Node* *array;
};

// A utility function to create a new tree node
struct Node* newNode(int data)
{
    struct Node* node = (struct Node*) malloc(sizeof(struct Node));
    node->data = data;
    node->left = node->right = NULL;
    return node;
}

// A utility function to create a stack of given size
struct Stack* createStack(int size)
{
    struct Stack* stack = (struct Stack*) malloc(sizeof(struct Stack));
```

```
stack->size = size;
stack->top = -1;
stack->array = (struct Node**) malloc(stack->size * sizeof(struct Node*));
return stack;
}

// BASIC OPERATIONS OF STACK
int isFull(struct Stack* stack)
{ return stack->top - 1 == stack->size; }

int isEmpty(struct Stack* stack)
{ return stack->top == -1; }

void push(struct Stack* stack, struct Node* node)
{
    if (isFull(stack))
        return;
    stack->array[+stack->top] = node;
}

struct Node* pop(struct Stack* stack)
{
    if (isEmpty(stack))
        return NULL;
    return stack->array[stack->top--];
}

struct Node* peek(struct Stack* stack)
{
    if (isEmpty(stack))
        return NULL;
    return stack->array[stack->top];
}

// An iterative function to do postorder traversal of a given binary tree
void postOrderIterative(struct Node* root)
{
    // Check for empty tree
    if (root == NULL)
        return;

    struct Stack* stack = createStack(MAX_SIZE);
    do
    {
        // Move to leftmost node
        while (root)
        {
            // Push root's right child and then root to stack.

```

```
if (root->right)
    push(stack, root->right);
push(stack, root);

// Set root as root's left child
root = root->left;
}

// Pop an item from stack and set it as root
root = pop(stack);

// If the popped item has a right child and the right child is not
// processed yet, then make sure right child is processed before root
if (root->right && peek(stack) == root->right)
{
    pop(stack); // remove right child from stack
    push(stack, root); // push root back to stack
    root = root->right; // change root so that the right
                        // child is processed next
}
else // Else print root's data and set root as NULL
{
    printf("%d ", root->data);
    root = NULL;
}
} while (!isEmpty(stack));
}

// Driver program to test above functions
int main()
{
    // Let us construct the tree shown in above figure
    struct Node* root = NULL;
    root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->left = newNode(6);
    root->right->right = newNode(7);
    printf("Post order traversal of binary tree is :\n");
    printf("[");
    postOrderIterative(root);
    printf("]");
}

return 0;
}
```

**Java**

```
// A java program for iterative postorder traversal using stack

import java.util.ArrayList;
import java.util.Stack;

// A binary tree node
class Node
{
    int data;
    Node left, right;

    Node(int item)
    {
        data = item;
        left = right;
    }
}

class BinaryTree
{
    Node root;
    ArrayList<Integer> list = new ArrayList<Integer>();

    // An iterative function to do postorder traversal
    // of a given binary tree
    ArrayList<Integer> postOrderIterative(Node node)
    {
        Stack<Node> S = new Stack<Node>();

        // Check for empty tree
        if (node == null)
            return list;
        S.push(node);
        Node prev = null;
        while (!S.isEmpty())
        {
            Node current = S.peek();

            /* go down the tree in search of a leaf an if so process it
            and pop stack otherwise move down */
            if (prev == null || prev.left == current ||
                prev.right == current)
            {
                if (current.left != null)
                    S.push(current.left);
                else if (current.right != null)
```

```
        S.push(current.right);
    else
    {
        S.pop();
        list.add(current.data);
    }

    /* go up the tree from left node, if the child is right
       push it onto stack otherwise process parent and pop
       stack */
}
else if (current.left == prev)
{
    if (current.right != null)
        S.push(current.right);
    else
    {
        S.pop();
        list.add(current.data);
    }

    /* go up the tree from right node and after coming back
       from right node process parent and pop stack */
}
else if (current.right == prev)
{
    S.pop();
    list.add(current.data);
}

prev = current;
}

return list;
}

// Driver program to test above functions
public static void main(String args[])
{
    BinaryTree tree = new BinaryTree();

    // Let us create trees shown in above diagram
    tree.root = new Node(1);
    tree.root.left = new Node(2);
    tree.root.right = new Node(3);
    tree.root.left.left = new Node(4);
    tree.root.left.right = new Node(5);
    tree.root.right.left = new Node(6);
```

```
tree.root.right.right = new Node(7);

ArrayList<Integer> mylist = tree.postOrderIterative(tree.root);

System.out.println("Post order traversal of binary tree is :");
System.out.println(mylist);
}

}

// This code has been contributed by Mayank Jaiswal
```

### Python

```
# Python program for iterative postorder traversal
# using one stack

# Stores the answer
ans = []

# A Binary tree node
class Node:

    # Constructor to create a new node
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

def peek(stack):
    if len(stack) > 0:
        return stack[-1]
    return None

# A iterative function to do postorder traversal of
# a given binary tree
def postOrderIterative(root):

    # Check for empty tree
    if root is None:
        return

    stack = []

    while(True):

        while (root):
            # Push root's right child and then root to stack
            if root.right is not None:
                stack.append(root.right)

            root = root.left
```

```
stack.append(root)

# Set root as root's left child
root = root.left

# Pop an item from stack and set it as root
root = stack.pop()

# If the popped item has a right child and the
# right child is not processed yet, then make sure
# right child is processed before root
if (root.right is not None and
    peek(stack) == root.right):
    stack.pop() # Remove right child from stack
    stack.append(root) # Push root back to stack
    root = root.right # change root so that the
                      # right child is processed next

# Else print root's data and set root as None
else:
    ans.append(root.data)
    root = None

if (len(stack) <= 0):
    break

# Driver program to test above function
root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)
root.right.left = Node(6)
root.right.right = Node(7)

print "Post Order traversal of binary tree is"
postOrderIterative(root)
print ans

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

Output:

```
Post Order traversal of binary tree is
[4, 5, 2, 6, 7, 3, 1]
```

This article is compiled by [Aashish Barnwal](#). Please write comments if you find anything

incorrect, or you want to share more information about the topic discussed above

## **Source**

<https://www.geeksforgeeks.org/iterative-postorder-traversal-using-stack/>

## Chapter 224

# Iterative Preorder Traversal

Iterative Preorder Traversal - GeeksforGeeks

Given a Binary Tree, write an iterative function to print Preorder traversal of the given binary tree.

Refer [this](#) for recursive preorder traversal of Binary Tree. To convert an inherently recursive procedures to iterative, we need an explicit stack. Following is a simple stack based iterative process to print Preorder traversal.

- 1) Create an empty stack *nodeStack* and push root node to stack.
- 2) Do following while *nodeStack* is not empty.
  - a) Pop an item from stack and print it.
  - b) Push right child of popped item to stack
  - c) Push left child of popped item to stack

Right child is pushed before left child to make sure that left subtree is processed first.

C++

```
#include <stdlib.h>
#include <stdio.h>
#include <iostream>
#include <stack>

using namespace std;

/* A binary tree node has data, left child and right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

/* Helper function that allocates a new node with the given data and
```

```

    NULL left and right pointers.*/
struct node* newNode(int data)
{
    struct node* node = new struct node;
    node->data = data;
    node->left = NULL;
    node->right = NULL;
    return(node);
}

// An iterative process to print preorder traversal of Binary tree
void iterativePreorder(node *root)
{
    // Base Case
    if (root == NULL)
        return;

    // Create an empty stack and push root to it
    stack<node *> nodeStack;
    nodeStack.push(root);

    /* Pop all items one by one. Do following for every popped item
     a) print it
     b) push its right child
     c) push its left child
    Note that right child is pushed first so that left is processed first */
    while (nodeStack.empty() == false)
    {
        // Pop the top item from stack and print it
        struct node *node = nodeStack.top();
        printf ("%d ", node->data);
        nodeStack.pop();

        // Push right and left children of the popped node to stack
        if (node->right)
            nodeStack.push(node->right);
        if (node->left)
            nodeStack.push(node->left);
    }
}

// Driver program to test above functions
int main()
{
    /* Constructed binary tree is
           10
          /   \
         8     2
    */
}

```

```
    / \ /
  3   5  2
*/
struct node *root = newNode(10);
root->left      = newNode(8);
root->right     = newNode(2);
root->left->left = newNode(3);
root->left->right = newNode(5);
root->right->left = newNode(2);
iterativePreorder(root);
return 0;
}
```

Java

```
// Java program to implement iterative preorder traversal
import java.util.Stack;

// A binary tree node
class Node {

    int data;
    Node left, right;

    Node(int item) {
        data = item;
        left = right = null;
    }
}

class BinaryTree {

    Node root;

    void iterativePreorder()
    {
        iterativePreorder(root);
    }

    // An iterative process to print preorder traversal of Binary tree
    void iterativePreorder(Node node) {

        // Base Case
        if (node == null) {
            return;
        }

        // Create an empty stack and push root to it
```

```

Stack<Node> nodeStack = new Stack<Node>();
nodeStack.push(root);

/* Pop all items one by one. Do following for every popped item
   a) print it
   b) push its right child
   c) push its left child
   Note that right child is pushed first so that left is processed first */
while (nodeStack.empty() == false) {

    // Pop the top item from stack and print it
    Node mynode = nodeStack.peek();
    System.out.print(mynode.data + " ");
    nodeStack.pop();

    // Push right and left children of the popped node to stack
    if (mynode.right != null) {
        nodeStack.push(mynode.right);
    }
    if (mynode.left != null) {
        nodeStack.push(mynode.left);
    }
}
}

// driver program to test above functions
public static void main(String args[]) {
    BinaryTree tree = new BinaryTree();
    tree.root = new Node(10);
    tree.root.left = new Node(8);
    tree.root.right = new Node(2);
    tree.root.left.left = new Node(3);
    tree.root.left.right = new Node(5);
    tree.root.right.left = new Node(2);
    tree.iterativePreorder();

}
}

// This code has been contributed by Mayank Jaiswal

```

**Python**

```

# Python program to perform iterative preorder traversal

# A binary tree node
class Node:

```

```
# Constructor to create a new node
def __init__(self, data):
    self.data = data
    self.left = None
    self.right = None

# An iterative process to print preorder traversal of BT
def iterativePreorder(root):

    # Base Case
    if root is None:
        return

    # Create an empty stack and push root to it
    nodeStack = []
    nodeStack.append(root)

    # Pop all items one by one. Do following for every popped item
    #   a) print it
    #   b) push its right child
    #   c) push its left child
    # Note that right child is pushed first so that left
    # is processed first */
    while(len(nodeStack) > 0):

        # Pop the top item from stack and print it
        node = nodeStack.pop()
        print node.data,

        # Push right and left children of the popped node
        # to stack
        if node.right is not None:
            nodeStack.append(node.right)
        if node.left is not None:
            nodeStack.append(node.left)

# Driver program to test above function
root = Node(10)
root.left = Node(8)
root.right = Node(2)
root.left.left = Node(3)
root.left.right = Node(5)
root.right.left = Node(2)
iterativePreorder(root)

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

Output:

10 8 3 5 2 2

This article is compiled by Saurabh Sharma and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## Source

<https://www.geeksforgeeks.org/iterative-preorder-traversal/>

## Chapter 225

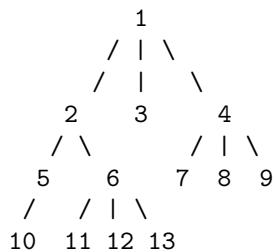
# Iterative Preorder Traversal of an N-ary Tree

Iterative Preorder Traversal of an N-ary Tree - GeeksforGeeks

Given a K-ary Tree. The task is to write an iterative program to perform the [preorder traversal](#) of the given n-ary tree.

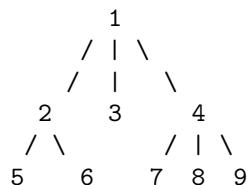
**Examples:**

Input: 3-Array Tree



Output: 1 2 5 10 6 11 12 13 3 4 7 8 9

Input: 3-Array Tree



Output: 1 2 5 6 3 4 7 8 9

Preorder Traversal of an N-ary Tree is similar to the preorder traversal of Binary Search Tree or Binary Tree with the only difference that is, all the child nodes of a parent are traversed from left to right in a sequence.

[Iterative Preorder Traversal of Binary Tree.](#)

**Cases to handle during traversal:** Two Cases have been taken care of in this Iterative Preorder Traversal Algorithm:

1. If Top of the stack is a leaf node then remove it from the stack
2. If Top of the stack is Parent with children:
  - As soon as an unvisited child is found(left to right sequence), Push it to Stack and Store it in Auxillary List and mark the following child as visited.Then, start again from Case-1, to explore this newly visited child.
  - If all Child nodes from left to right of a Parent has been visited then remove the Parent from the stack.

**Note:** In the below python implementation, a “dequeue” is used to implement the stack instead of a list because of its efficient append and pop operations.

Below is the implementation of the above approach:

```
# Python program for Iterative Preorder
# Traversal of N-ary Tree.
# Preorder: Root, print children
# from left to right.

from collections import deque

# Node Structure of K-ary Tree
class NewNode():

    def __init__(self, val):
        self.key = val
        # all children are stored in a list
        self.child = []

    # Utility function to print the
    # preorder of the given K-Ary Tree
    def preorderTraversal(root):

        Stack = deque([])
        # 'Preorder'-> contains all the
        # visited nodes.
        Preorder = []
```

```

Preorder.append(root.key)
Stack.append(root)
while len(Stack)>0:
    # 'Flag' checks whether all the child
    # nodes have been visited.
    flag = 0
    # CASE 1- If Top of the stack is a leaf
    # node then remove it from the stack:
    if len((Stack[len(Stack)-1]).child)== 0:
        X = Stack.pop()
        # CASE 2- If Top of the stack is
        # Parent with children:
    else:
        Par = Stack[len(Stack)-1]
        # a)As soon as an unvisited child is
        # found(left to right sequence),
        # Push it to Stack and Store it in
        # Auxillary List(Marked Visited)
        # Start Again from Case-1, to explore
        # this newly visited child
        for i in range(0, len(Par.child)):
            if Par.child[i].key not in Preorder:
                flag = 1
                Stack.append(Par.child[i])
                Preorder.append(Par.child[i].key)
                break;
            # b)If all Child nodes from left to right
            # of a Parent have been visited
            # then remove the parent from the stack.
        if flag == 0:
            Stack.pop()
print(Preorder)

# Execution Start From here
if __name__=='__main__':
# input nodes
    ...
    1
    / | \
    / | \
    2 3 4
    / \ / | \
    5 6 7 8 9
    / / | \
    10 11 12 13
    ...

```

```
root = NewNode(1)
root.child.append(NewNode(2))
root.child.append(NewNode(3))
root.child.append(NewNode(4))
root.child[0].child.append(NewNode(5))
root.child[0].child[0].child.append(NewNode(10))
root.child[0].child.append(NewNode(6))
root.child[0].child[1].child.append(NewNode(11))
root.child[0].child[1].child.append(NewNode(12))
root.child[0].child[1].child.append(NewNode(13))
root.child[2].child.append(NewNode(7))
root.child[2].child.append(NewNode(8))
root.child[2].child.append(NewNode(9))

preorderTraversal(root)
```

**Output:**

```
[1, 2, 5, 10, 6, 11, 12, 13, 3, 4, 7, 8, 9]
```

**Source**

<https://www.geeksforgeeks.org/iterative-preorder-traversal-of-a-n-ary-tree/>

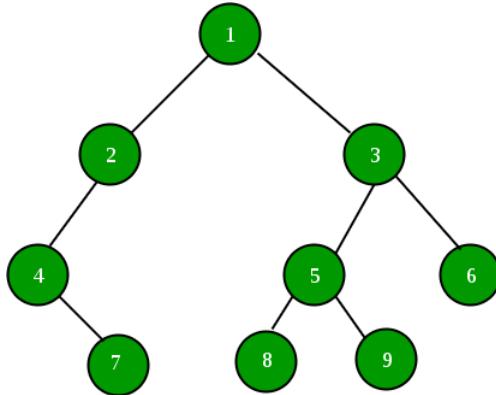
## Chapter 226

# Iterative Search for a key 'x' in Binary Tree

Iterative Search for a key 'x' in Binary Tree - GeeksforGeeks

Given a Binary Tree and a key to be searched in it, write an iterative method that returns true if key is present in Binary Tree, else false.

For example, in the following tree, if the searched key is 3, then function should return true and if the searched key is 12, then function should return false.



One thing is sure that we need to traverse complete tree to decide whether key is present or not. We can use any of the following traversals to iteratively search a key in a given binary tree.

- 1) Iterative [Level Order Traversal](#).
- 2) [Iterative Inorder Traversal](#)
- 3) [Iterative Preorder Traversal](#)
- 4) [Iterative Postorder Traversal](#)

Below is iterative [Level Order Traversal](#) based solution to search an item x in binary tree.

```
// Iterative level order traversal based method to search in Binary Tree
#include <iostream>
#include <queue>
using namespace std;

/* A binary tree node has data, left child and right child */
struct node
{
    int data;
    struct node* left, *right;
};

/* Helper function that allocates a new node with the given data and
NULL left and right pointers.*/
struct node* newNode(int data)
{
    struct node* node = new struct node;
    node->data = data;
    node->left = node->right = NULL;
    return(node);
}

// An iterative process to search an element x in a given binary tree
bool iterativeSearch(node *root, int x)
{
    // Base Case
    if (root == NULL)
        return false;

    // Create an empty queue for level order traversal
    queue<node *> q;

    // Enqueue Root and initialize height
    q.push(root);

    // Queue based level order traversal
    while (q.empty() == false)
    {
        // See if current node is same as x
        node *node = q.front();
        if (node->data == x)
            return true;

        // Remove current node and enqueue its children
        q.pop();
        if (node->left != NULL)
            q.push(node->left);
        if (node->right != NULL)
```

```
        q.push(node->right);
    }

    return false;
}

// Driver program
int main(void)
{
    struct node*NewRoot=NULL;
    struct node *root = newNode(2);
    root->left      = newNode(7);
    root->right     = newNode(5);
    root->left->right = newNode(6);
    root->left->right->left=newNode(1);
    root->left->right->right=newNode(11);
    root->right->right=newNode(9);
    root->right->right->left=newNode(4);

    iterativeSearch(root, 6)? cout << "Found\n": cout << "Not Found\n";
    iterativeSearch(root, 12)? cout << "Found\n": cout << "Not Found\n";
    return 0;
}
```

Output:

```
Found
Not Found
```

Below implementation uses [Iterative Preorder Traversal](#) to find x in Binary Tree

```
// An iterative method to search an item in Binary Tree
#include <iostream>
#include <stack>
using namespace std;

/* A binary tree node has data, left child and right child */
struct node
{
    int data;
    struct node* left, *right;
};

/* Helper function that allocates a new node with the given data and
NULL left and right pointers.*/
struct node* newNode(int data)
```

```
{  
    struct node* node = new struct node;  
    node->data = data;  
    node->left = node->right = NULL;  
    return(node);  
}  
  
// iterative process to search an element x in a given binary tree  
bool iterativeSearch(node *root, int x)  
{  
    // Base Case  
    if (root == NULL)  
        return false;  
  
    // Create an empty stack and push root to it  
    stack<node *> nodeStack;  
    nodeStack.push(root);  
  
    // Do iterative preorder traversal to search x  
    while (nodeStack.empty() == false)  
    {  
        // See the top item from stack and check if it is same as x  
        struct node *node = nodeStack.top();  
        if (node->data == x)  
            return true;  
        nodeStack.pop();  
  
        // Push right and left children of the popped node to stack  
        if (node->right)  
            nodeStack.push(node->right);  
        if (node->left)  
            nodeStack.push(node->left);  
    }  
  
    return false;  
}  
  
// Driver program  
int main(void)  
{  
    struct node*NewRoot=NULL;  
    struct node *root = newNode(2);  
    root->left = newNode(7);  
    root->right = newNode(5);  
    root->left->right = newNode(6);  
    root->left->right->left=newNode(1);  
    root->left->right->right=newNode(11);  
    root->right->right=newNode(9);
```

```
root->right->right->left=newNode(4);

iterativeSearch(root, 6)? cout << "Found\n": cout << "Not Found\n";
iterativeSearch(root, 12)? cout << "Found\n": cout << "Not Found\n";
return 0;
}
```

Output:

```
Found
Not Found
```

Similarly, [Iterative Inorder](#) and [Iterative Postorder](#) traversals can be used.

## Source

<https://www.geeksforgeeks.org/iterative-search-for-a-key-x-in-binary-tree/>

## Chapter 227

# Iterative Segment Tree (Range Maximum Query with Node Update)

Iterative Segment Tree (Range Maximum Query with Node Update) - GeeksforGeeks

Given an array  $\text{arr}[0 \dots n-1]$ . The task is to perform the following operation:

- Find the maximum of elements from index  $l$  to  $r$  where  $0 \leq l \leq r \leq n-1$ .
- Change value of a specified element of the array to a new value  $x$ . Given  $i$  and  $x$ , change  $A[i]$  to  $x$ ,  $0 \leq i \leq n-1$ .

**Examples:**

Input:  $a[] = \{2, 6, 7, 5, 18, 86, 54, 2\}$

Query1: maximum(2, 7)

Query2: update(3, 90)

Query3: maximum(2, 6)

**Output:**

Maximum in range 2 to 7 is 86.

Maximum in range 2 to 6 is 90.

We have discussed [Recursive segment tree implementation](#). In this post, [iterative implementation](#) is discussed. The iterative version of the segment tree basically uses the fact, that for an index  $i$ , left child  $= 2 * i$  and right child  $= 2 * i + 1$  in the tree. The parent for an index  $i$  in the segment tree array can be found by  $\text{parent} = i / 2$ . Thus we can easily travel up and down through the levels of the tree one by one. At first we compute the maximum in the ranges while constructing the tree starting from the leaf nodes and climbing up through the levels one by one. We use the same concept while processing the queries for finding the maximum in a range. Since there are  $(\log n)$  levels in the worst case, so querying takes  $\log$

n time. For update of a particular index to a given value we start updating the segment tree starting from the leaf nodes and update all those nodes which are affected by the updation of the current node by gradually moving up through the levels at every iteration. Updation also takes log n time because there we have to update all the levels starting from the leaf node where we update the exact value at the exact index given by the user.

Below is the implementation of the above approach.

```
// C++ Program to implement
// iterative segment tree.
#include <bits/stdc++.h>
using namespace std;

void construct_segment_tree(vector<int>& segtree,
                           vector<int>& a, int n)
{
    // assign values to leaves of the segment tree
    for (int i = 0; i < n; i++)
        segtree[n + i] = a[i];

    /* assign values to internal nodes
       to compute maximum in a given range */
    for (int i = n - 1; i >= 1; i--)
        segtree[i] = max(segtree[2 * i],
                         segtree[2 * i + 1]);
}

void update(vector<int>& segtree, int pos, int value,
            int n)
{
    // change the index to leaf node first
    pos += n;

    // update the value at the leaf node
    // at the exact index
    segtree[pos] = value;

    while (pos > 1) {

        // move up one level at a time in the tree
        pos >>= 1;

        // update the values in the nodes in
        // the next higher level
        segtree[pos] = max(segtree[2 * pos],
                           segtree[2 * pos + 1]);
    }
}
```

```
int range_query(vector<int>& segtree, int left, int
                           right,
                           int n)
{
    /* Basically the left and right indices will move
       towards right and left respectively and with
       every each next higher level and compute the
       maximum at each height. */
    // change the index to leaf node first
    left += n;
    right += n;

    // initialize maximum to a very low value
    int ma = INT_MIN;

    while (left < right) {

        // if left index in odd
        if (left & 1) {
            ma = max(ma, segtree[left]);

            // make left index even
            left++;
        }

        // if right index in odd
        if (right & 1) {

            // make right index even
            right--;

            ma = max(ma, segtree[right]);
        }

        // move to the next higher level
        left /= 2;
        right /= 2;
    }
    return ma;
}

// Driver code
int main()
{
    vector<int> a = { 2, 6, 10, 4, 7, 28, 9, 11, 6, 33 };
    int n = a.size();

    /* Construct the segment tree by assigning
```

```
the values to the internal nodes*/
vector<int> segtree(2 * n);
construct_segment_tree(segtree, a, n);

// compute maximum in the range left to right
int left = 1, right = 5;
cout << "Maximum in range " << left << " to "
    << right << " is " << range_query(segtree, left,
                                         right + 1, n)
    << "\n";

// update the value of index 5 to 32
int index = 5, value = 32;

// a[5] = 32;
// Contents of array : {2, 6, 10, 4, 7, 32, 9, 11, 6, 33}
update(segtree, index, value, n);

// compute maximum in the range left to right
left = 2, right = 8;
cout << "Maximum in range " << left << " to "
    << right << " is " << range_query(segtree,
                                         left, right + 1, n)
    << "\n";

return 0;
}
```

**Output:**

```
Maximum in range 1 to 5 is 28
Maximum in range 2 to 8 is 32
```

**Time Complexity:**  $(N * \log N)$   
**Auxiliary Space:**  $O(N)$

**Source**

<https://www.geeksforgeeks.org/iterative-segment-tree-range-maximum-query-with-node-update/>

## Chapter 228

# Iterative Segment Tree (Range Minimum Query)

Iterative Segment Tree (Range Minimum Query) - GeeksforGeeks

We have discussed [recursive segment tree implementation](#). In this post, iterative implementation is discussed.

Let us consider the following problem understand Segment Trees.

We have an array  $\text{arr}[0 \dots n-1]$ . We should be able to

- 1 Find the minimum of elements from index l to r where  $0 \leq l \leq r \leq n-1$
- 2 Change value of a specified element of the array to a new value x. We need to do  $\text{arr}[i] = x$  where  $0 \leq i \leq n-1$ .

Examples:

```
Input : 2, 6, 7, 5, 18, 86, 54, 2
        minimum(2, 7)
        update(3, 4)
        minimum(2, 6)
Output : Minimum in range 2 to 7 is 2.
          Minimum in range 2 to 6 is 4.
```

The iterative version of the segment tree basically uses the fact, that for an index i, left child  $= 2 * i$  and right child  $= 2 * i + 1$  in the tree. The parent for an index i in the segment tree array can be found by parent  $= i / 2$ . Thus we can easily travel up and down through the levels of the tree one by one. At first we compute the minimum in the ranges while constructing the tree starting from the leaf nodes and climbing up through the levels one by one. We use the same concept while processing the queries for finding the minimum in a range. Since there are  $(\log n)$  levels in the worst case, so querying takes  $\log n$  time. For update of a particular index to a given value we start updating the segment tree starting from the leaf nodes and update all those nodes which are affected by the updation of the

current node by gradually moving up through the levels at every iteration. Updation also takes  $\log n$  time because there we have to update all the levels starting from the leaf node where we update the exact value at the exact index given by the user.

```
// CPP Program to implement iterative segment
// tree.
#include <bits/stdc++.h>
#define ll long long

using namespace std;

void construct_segment_tree(vector<int>& segtree,
                           vector<int> &a, int n)
{
    // assign values to leaves of the segment tree
    for (int i = 0; i < n; i++)
        segtree[n + i] = a[i];

    /* assign values to internal nodes
       to compute minimum in a given range */
    for (int i = n - 1; i >= 1; i--)
        segtree[i] = min(segtree[2 * i],
                         segtree[2 * i + 1]);
}

void update(vector<int>& segtree, int pos, int value,
            int n)
{
    // change the index to leaf node first
    pos += n;

    // update the value at the leaf node
    // at the exact index
    segtree[pos] = value;

    while (pos > 1) {

        // move up one level at a time in the tree
        pos >>= 1;

        // update the values in the nodes in
        // the next higher level
        segtree[pos] = min(segtree[2 * pos],
                           segtree[2 * pos + 1]);
    }
}

int range_query(vector<int>& segtree, int left, int
```

```
right, int n)
{
    /* Basically the left and right indices will move
       towards right and left respectively and with
       every each next higher level and compute the
       minimum at each height. */
    // change the index to leaf node first
    left += n;
    right += n;

    // initialize minimum to a very high value
    int mi = (int)1e9;

    while (left < right) {

        // if left index in odd
        if (left & 1) {
            mi = min(mi, segtree[left]);

            // make left index even
            left++;
        }

        // if right index in odd
        if (right & 1) {

            // make right index even
            right--;

            mi = min(mi, segtree[right]);
        }

        // move to the next higher level
        left /= 2;
        right /= 2;
    }
    return mi;
}

// Driver code
int main()
{
    vector<int> a = { 2, 6, 10, 4, 7, 28, 9, 11, 6, 33 };
    int n = a.size();

    /* Construct the segment tree by assigning
       the values to the internal nodes*/
    vector<int> segtree(2 * n);
```

```
construct_segment_tree(segtree, a, n);

// compute minimum in the range left to right
int left = 0, right = 5;
cout << "Minimum in range " << left << " to "
    << right << " is " << range_query(segtree, left,
                                         right + 1, n) << "\n";

// update the value of index 3 to 1
int index = 3, value = 1;

// a[3] = 1;
// Contents of array : {2, 6, 10, 1, 7, 28, 9, 11, 6, 33}
update(segtree, index, value, n); // point update

// compute minimum in the range left to right
left = 2, right = 6;
cout << "Minimum in range " << left << " to "
    << right << " is " << range_query(segtree,
                                         left, right + 1, n) << "\n";

return 0;
}
```

**Output:**

```
Minimum in range 0 to 5 is 2
Minimum in range 2 to 6 is 1
```

Time Complexity( $n \log n$ )  
Auxiliary Space ( $n$ )

**Source**

<https://www.geeksforgeeks.org/iterative-segment-tree-range-minimum-query/>

## Chapter 229

# Iterative diagonal traversal of binary tree

Iterative diagonal traversal of binary tree - GeeksforGeeks

Consider lines of slope -1 passing between nodes. Given a Binary Tree, print all diagonal elements in a binary tree belonging to same line.

Input : Root of below tree

Output :

Diagonal Traversal of binary tree :  
8 10 14  
3 6 7 13  
1 4

We have discussed recursive solution in below post.

[Diagonal Traversal of Binary Tree](#)

In this post, iterative solution is discussed. The idea is to use a queue to store only the left child of **current** node. After printing the data of current node make the current node to its right child, if present.

A delimiter **NULL** is used to mark the starting of next diagonal.

```
/* C++ program to construct string from binary tree*/  
#include <bits/stdc++.h>  
using namespace std;  
  
/* A binary tree node has data, pointer to left  
child and a pointer to right child */
```

```
struct Node {
    int data;
    Node *left, *right;
};

/* Helper function that allocates a new node */
Node* newNode(int data)
{
    Node* node = (Node*)malloc(sizeof(Node));
    node->data = data;
    node->left = node->right = NULL;
    return (node);
}

// Iterative function to print diagonal view
void diagonalPrint(Node* root)
{
    // base case
    if (root == NULL)
        return;

    // inbuilt queue of Treenode
    queue<Node*> q;

    // push root
    q.push(root);

    // push delimiter
    q.push(NULL);

    while (!q.empty()) {
        Node* temp = q.front();
        q.pop();

        // if current is delimiter then insert another
        // for next diagonal and cout nextline
        if (temp == NULL) {

            // if queue is empty return
            if (q.empty())
                return;

            // output newline
            cout << endl;

            // push delimiter again
            q.push(NULL);
        }
    }
}
```

```
else {
    while (temp) {
        cout << temp->data << " ";

        // if left child is present
        // push into queue
        if (temp->left)
            q.push(temp->left);

        // current equals to right child
        temp = temp->right;
    }
}

// Driver Code
int main()
{
    Node* root = newNode(8);
    root->left = newNode(3);
    root->right = newNode(10);
    root->left->left = newNode(1);
    root->left->right = newNode(6);
    root->right->right = newNode(14);
    root->right->right->left = newNode(13);
    root->left->right->left = newNode(4);
    root->left->right->right = newNode(7);
    diagonalPrint(root);
}
```

Output:

```
8 10 14
3 6 7 13
1 4
```

## Source

<https://www.geeksforgeeks.org/iterative-diagonal-traversal-binary-tree/>

## Chapter 230

# Iterative function to check if two trees are identical

Iterative function to check if two trees are identical - GeeksforGeeks

Two trees are identical when they have same data and arrangement of data is also same. To identify if two trees are identical, we need to traverse both trees simultaneously, and while traversing we need to compare data and children of the trees.

Examples:

Input : Roots of below trees  
      10                10  
      / \              /  
     5   6            5  
Output : false

Input : Roots of below trees  
      10                10  
      / \              / \  
     5   6            5   6  
Output : true

We have discussed recursive solution [here](#). In this article iterative solution is discussed. The idea is to use [level order traversal](#). We traverse both trees simultaneously and compare the data whenever we dequeue and item from queue. Below is C++ implementation of the idea.

```
/* Iterative C++ program to check if two */  
#include <bits/stdc++.h>  
using namespace std;
```

```
// A Binary Tree Node
struct Node
{
    int data;
    struct Node *left, *right;
};

// Iterative method to find height of Bianry Tree
bool areIdentical(Node *root1, Node *root2)
{
    // Return true if both trees are empty
    if (!root1 && !root2) return true;

    // Return false if one is empty and other is not
    if (!root1 || !root2) return false;

    // Create an empty queues for simultaneous traversals
    queue<Node *> q1, q2;

    // Enqueue Roots of trees in respective queues
    q1.push(root1);
    q2.push(root2);

    while (!q1.empty() && !q2.empty())
    {
        // Get front nodes and compare them
        Node *n1 = q1.front();
        Node *n2 = q2.front();

        if (n1->data != n2->data)
            return false;

        // Remove front nodes from queues
        q1.pop(), q2.pop();

        /* Enqueue left children of both nodes */
        if (n1->left && n2->left)
        {
            q1.push(n1->left);
            q2.push(n2->left);
        }

        // If one left child is empty and other is not
        else if (n1->left || n2->left)
            return false;

        // Right child code (Similar to left child code)
        if (n1->right && n2->right)
```

```
{  
    q1.push(n1->right);  
    q2.push(n2->right);  
}  
else if (n1->right || n2->right)  
    return false;  
}  
  
return true;  
}  
  
// Utility function to create a new tree node  
Node* newNode(int data)  
{  
    Node *temp = new Node;  
    temp->data = data;  
    temp->left = temp->right = NULL;  
    return temp;  
}  
  
// Driver program to test above functions  
int main()  
{  
    Node *root1 = newNode(1);  
    root1->left = newNode(2);  
    root1->right = newNode(3);  
    root1->left->left = newNode(4);  
    root1->left->right = newNode(5);  
  
    Node *root2 = newNode(1);  
    root2->left = newNode(2);  
    root2->right = newNode(3);  
    root2->left->left = newNode(4);  
    root2->left->right = newNode(5);  
  
    areIdentical(root1, root2)? cout << "Yes"  
                            : cout << "No";  
    return 0;  
}
```

Output:

Yes

Time complexity of above solution is  $O(n + m)$  where m and n are number of nodes in two trees.

**Source**

<https://www.geeksforgeeks.org/iterative-function-check-two-trees-identical/>

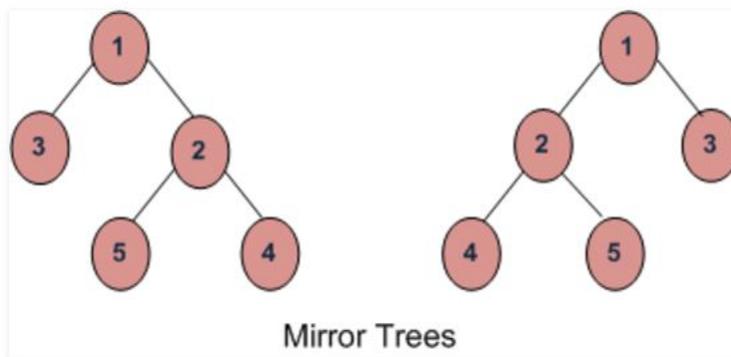
## Chapter 231

# Iterative method to check if two trees are mirror of each other

Iterative method to check if two trees are mirror of each other - GeeksforGeeks

Given two binary trees. The problem is to check whether the two binary trees are mirrors of each other or not.

**Mirror of a Binary Tree:** Mirror of a Binary Tree T is another Binary Tree M(T) with left and right children of all non-leaf nodes interchanged.



Trees in the above figure are mirrors of each other.

We have discussed a [recursive solution to check if two trees are mirror](#). In this post iterative solution is discussed.

**Prerequisite:** Iterative inorder tree traversal using stack

**Approach:** The following steps are:

1. Perform iterative inorder traversal of one tree and iterative reverse inorder traversal of the other tree in parallel.

2. During these two iterative traversals check that the corresponding nodes have the same value or not. If not same then they are not mirrors of each other.
3. If values are same, then check whether at any point in the iterative inorder traversal one of the root becomes null and the other is not null. If this happens then they are not mirrors of each other. This check ensures whether they have the corresponding mirror structures or not.
4. Otherwise, both the trees are mirror of each other.

**Reverse inorder traversal** is the opposite of inorder traversal. In this, the right subtree is traversed first, then root, and then the left subtree.

```
// C++ implementation to check whether the two
// binary trees are mirrors of each other or not
#include <bits/stdc++.h>
using namespace std;

// structure of a node in binary tree
struct Node
{
    int data;
    struct Node *left, *right;
};

// Utility function to create and return
// a new node for a binary tree
struct Node* newNode(int data)
{
    struct Node *temp = new Node();
    temp->data = data;
    temp->left = temp->right = NULL;
    return temp;
}

// function to check whether the two binary trees
// are mirrors of each other or not
string areMirrors(Node *root1, Node *root2)
{
    stack<Node*> st1, st2;
    while (1)
    {
        // iterative inorder traversal of 1st tree and
        // reverse inorder traversal of 2nd tree
        while (root1 && root2)
        {
            // if the corresponding nodes in the two traversal
            // have different data values, then they are not
            // mirrors of each other.
            if (root1->data != root2->data)
                return "No";
            root1 = root1->left;
            root2 = root2->right;
        }
        if (!root1 || !root2)
            break;
        root1 = st1.top();
        st1.pop();
        root2 = st2.top();
        st2.pop();
    }
    if (!root1 && !root2)
        return "Yes";
    else
        return "No";
}
```

```
        return "No";

        st1.push(root1);
        st2.push(root2);
        root1 = root1->left;
        root2 = root2->right;
    }

    // if at any point one root becomes null and
    // the other root is not null, then they are
    // not mirrors. This condition verifies that
    // structures of tree are mirrors of each other.
    if (!(root1 == NULL && root2 == NULL))
        return "No";

    if (!st1.empty() && !st2.empty())
    {
        root1 = st1.top();
        root2 = st2.top();
        st1.pop();
        st2.pop();

        /* we have visited the node and its left subtree.
           Now, it's right subtree's turn */
        root1 = root1->right;

        /* we have visited the node and its right subtree.
           Now, it's left subtree's turn */
        root2 = root2->left;
    }

    // both the trees have been completely traversed
    else
        break;
}

// trees are mirrors of each other
return "Yes";
}

// Driver program to test above
int main()
{
    // 1st binary tree formation
    Node *root1 = newNode(1);          /*      1      */
    root1->left = newNode(3);         /*      / \      */
    root1->right = newNode(2);        /*      3   2      */
    root1->right->left = newNode(5); /*      / \      */
```

```
root1->right->right = newNode(4);      /*      5      4      */
                                              /*           1      */
                                              /*           / \      */
                                              /*          2   3      */
                                              /*          / \      */
                                              /*         4   5      */

// 2nd binary tree formation
Node *root2 = newNode(1);                  /*      1      */
root2->left = newNode(2);                 /*      / \      */
root2->right = newNode(3);                /*      2   3      */
root2->left->left = newNode(4);          /*      / \      */
root2->left->right = newNode(5);         /*      4   5      */

cout << areMirrors(root1, root2);
return 0;
}
```

Output:

Yes

Time Complexity: O(n)

## Source

<https://www.geeksforgeeks.org/iterative-method-check-two-trees-mirror/>

## Chapter 232

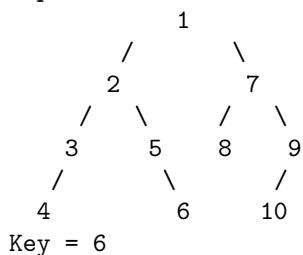
# Iterative method to find ancestors of a given binary tree

Iterative method to find ancestors of a given binary tree - GeeksforGeeks

Given a binary tree, print all the ancestors of a particular key existing in the tree without using recursion.

Here we will be discussing the c++ implementation for the above problem.  
Examples:

Input :



Key = 6

Output : 5 2 1  
Ancestors of 6 are 5, 2 and 1.

The idea is to use [iterative postorder traversal](#) of given binary tree.

```
// C++ program to print all ancestors of a given key
#include <bits/stdc++.h>
using namespace std;

// Structure for a tree node
```

```
struct Node {  
    int data;  
    struct Node* left, *right;  
};  
  
// A utility function to create a new tree node  
struct Node* newNode(int data)  
{  
    struct Node* node = (struct Node*)malloc(sizeof(struct Node));  
    node->data = data;  
    node->left = node->right = NULL;  
    return node;  
}  
  
// Iterative Function to print all ancestors of a  
// given key  
void printAncestors(struct Node* root, int key)  
{  
    if (root == NULL)  
        return;  
  
    // Create a stack to hold ancestors  
    stack<struct Node*> st;  
  
    // Traverse the complete tree in postorder way till  
    // we find the key  
    while (1) {  
  
        // Traverse the left side. While traversing, push  
        // the nodes into the stack so that their right  
        // subtrees can be traversed later  
        while (root && root->data != key) {  
            st.push(root); // push current node  
            root = root->left; // move to next node  
        }  
  
        // If the node whose ancestors are to be printed  
        // is found, then break the while loop.  
        if (root && root->data == key)  
            break;  
  
        // Check if right sub-tree exists for the node at top  
        // If not then pop that node because we don't need  
        // this node any more.  
        if (st.top()->right == NULL) {  
            root = st.top();  
            st.pop();  
        }  
    }  
}
```

```
// If the popped node is right child of top,
// then remove the top as well. Left child of
// the top must have processed before.
while (!st.empty() && st.top()->right == root) {
    root = st.top();
    st.pop();
}
}

// if stack is not empty then simply set the root
// as right child of top and start traversing right
// sub-tree.
root = st.empty() ? NULL : st.top()->right;
}

// If stack is not empty, print contents of stack
// Here assumption is that the key is there in tree
while (!st.empty()) {
    cout << st.top()->data << " ";
    st.pop();
}
}

// Driver program to test above functions
int main()
{
    // Let us construct a binary tree
    struct Node* root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(7);
    root->left->left = newNode(3);
    root->left->right = newNode(5);
    root->right->left = newNode(8);
    root->right->right = newNode(9);
    root->left->left->left = newNode(4);
    root->left->right->right = newNode(6);
    root->right->right->left = newNode(10);

    int key = 6;
    printAncestors(root, key);

    return 0;
}
```

Output:

5 2 1

**Source**

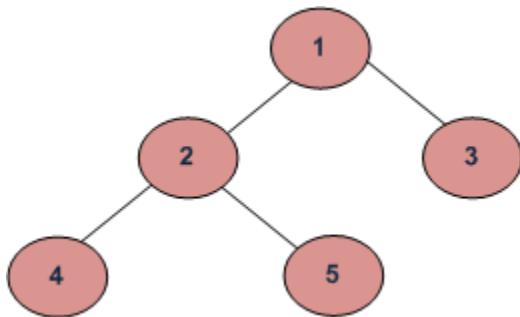
<https://www.geeksforgeeks.org/iterative-method-to-find-ancestors-of-a-given-binary-tree/>

## Chapter 233

# Iterative program to Calculate Size of a tree

Iterative program to Calculate Size of a tree - GeeksforGeeks

Size of a tree is the number of elements present in the tree. Size of the below tree is 5.



Example Tree

### Approach

The idea is to use [Level Order Traversing](#)

- 1) Create an empty queue q
- 2) temp\_node = root /\*start from root\*/
- 3) Loop while temp\_node is not NULL
  - a) Enqueue temp\_node's children (first left then right children) to q
  - b) Increase count with every enqueueing.
  - c) Dequeue a node from q and assign it's value to temp\_node

```
// Java programm to calculate  
// Size of a tree
```

```
import java.util.LinkedList;
import java.util.Queue;

class Node
{
    int data;
    Node left, right;

    public Node(int item)
    {
        data = item;
        left = right = null;
    }
}

class BinaryTree
{
    Node root;

    public int size()
    {
        if (root == null)
            return 0;

        // Using level order Traversal .
        Queue<Node> q = new LinkedList<Node>();
        q.offer(root);

        int count = 1;
        while (!q.isEmpty())
        {
            Node tmp = q.poll();

            // when the queue is empty:
            // the poll() method returns null.
            if (tmp!=null)
            {
                if (tmp.left!=null)
                {
                    // Increment count
                    count++;

                    // Enqueue left child
                    q.offer(tmp.left);
                }
                if (tmp.right!=null)
                {
                    // Increment count
                }
            }
        }
    }
}
```

```
        count++;

        // Enqueue left child
        q.offer(tmp.right);
    }
}

return count;
}

public static void main(String args[])
{
    /* creating a binary tree and entering
       the nodes */
    BinaryTree tree = new BinaryTree();
    tree.root = new Node(1);
    tree.root.left = new Node(2);
    tree.root.right = new Node(3);
    tree.root.left.left = new Node(4);
    tree.root.left.right = new Node(5);

    System.out.println("The size of binary tree" +
                       " is : " + tree.size());
}
}
```

Output:

```
Size of the tree is 5
```

Time Complexity: O(n)

Auxiliary Space : O(level\_max) where level max is maximum number of node in any level.

## Source

<https://www.geeksforgeeks.org/write-program-calculate-size-tree-iterative/>

## Chapter 234

# Iterative program to count leaf nodes in a Binary Tree

Iterative program to count leaf nodes in a Binary Tree - GeeksforGeeks

Given a binary tree, count leaves in the tree without using recursion. A node is a leaf node if both left and right children of it are NULL.

Example Tree

Leaves count for the above tree is 3.

Asked In: GATE-2007

The idea is to use level order traversal. During traversal, if we find a node whose left and right children are NULL, we increment count.

```
// C++ program to count leaf nodes in a Binary Tree
#include <bits/stdc++.h>
using namespace std;

/* A binary tree Node has data, pointer to left
   child and a pointer to right child */
struct Node
{
    int data;
    struct Node* left, *right;
};

/* Function to get the count of leaf Nodes in
   a binary tree*/
unsigned int getLeafCount(struct Node* node)
```

```
{  
    // If tree is empty  
    if (!node)  
        return 0;  
  
    // Initialize empty queue.  
    queue<Node *> q;  
  
    // Do level order traversal starting from root  
    int count = 0; // Initialize count of leaves  
    q.push(node);  
    while (!q.empty())  
    {  
        struct Node *temp = q.front();  
        q.pop();  
  
        if (temp->left != NULL)  
            q.push(temp->left);  
        if (temp->right != NULL)  
            q.push(temp->right);  
        if (temp->left == NULL && temp->right == NULL)  
            count++;  
    }  
    return count;  
}  
  
/* Helper function that allocates a new Node with the  
   given data and NULL left and right pointers. */  
struct Node* newNode(int data)  
{  
    struct Node* node = new Node;  
    node->data = data;  
    node->left = node->right = NULL;  
    return (node);  
}  
  
/* Driver program to test above functions*/  
int main()  
{  
    /*      1  
         / \br/>        2   3  
       / \br/>      4   5  
Let us create Binary Tree shown in  
above example */  
  
    struct Node *root = newNode(1);
```

```
root->left      = newNode(2);
root->right     = newNode(3);
root->left->left  = newNode(4);
root->left->right = newNode(5);

/* get leaf count of the above created tree */
cout << getLeafCount(root);

return 0;
}
```

Output:

3

Time Complexity : O(n)

### Source

<https://www.geeksforgeeks.org/iterative-program-count-leaf-nodes-binary-tree/>

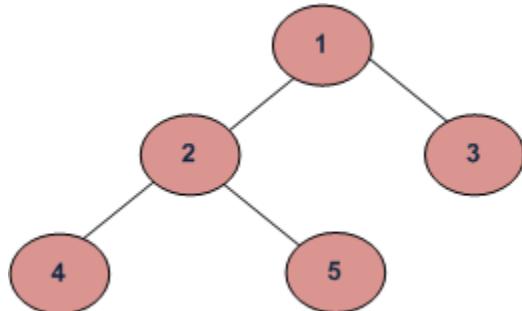
## Chapter 235

# K-th ancestor of a node in Binary Tree

K-th ancestor of a node in Binary Tree - GeeksforGeeks

Given a binary tree in which nodes are numbered from 1 to n. Given a node and a positive integer K. We have to print the K-th ancestor of the given node in the binary tree. If there does not exist any such ancestor then print -1.

For example in the below given binary tree, 2nd ancestor of node 4 and 5 is 1. 3rd ancestor of node 4 will be -1.



The idea to do this is to first traverse the binary tree and store the ancestor of each node in an array of size n. For example, suppose the array is `ancestor[n]`. Then at index i, `ancestor[i]` will store the ancestor of ith node. So, the 2nd ancestor of ith node will be `ancestor[ancestor[i]]` and so on. We will use this idea to calculate the kth ancestor of the given node. We can use [level order traversal](#) to populate this array of ancestors.

Below is the C++ implementation of above idea

```
/* C++ program to calculate Kth ancestor of given node */
#include <iostream>
#include <queue>
using namespace std;
```

```
// A Binary Tree Node
struct Node
{
    int data;
    struct Node *left, *right;
};

// function to generate array of ancestors
void generateArray(Node *root, int ancestors[])
{
    // There will be no ancestor of root node
    ancestors[root->data] = -1;

    // level order traversal to
    // generate 1st ancestor
    queue<Node*> q;
    q.push(root);

    while(!q.empty())
    {
        Node* temp = q.front();
        q.pop();

        if (temp->left)
        {
            ancestors[temp->left->data] = temp->data;
            q.push(temp->left);
        }

        if (temp->right)
        {
            ancestors[temp->right->data] = temp->data;
            q.push(temp->right);
        }
    }
}

// function to calculate Kth ancestor
int kthAncestor(Node *root, int n, int k, int node)
{
    // create array to store 1st ancestors
    int ancestors[n+1] = {0};

    // generate first ancestor array
    generateArray(root,ancestors);

    // variable to track record of number of
```

```
// ancestors visited
int count = 0;

while (node!=-1)
{
    node = ancestors[node];
    count++;

    if(count==k)
        break;
}

// print Kth ancestor
return node;
}

// Utility function to create a new tree node
Node* newNode(int data)
{
    Node *temp = new Node;
    temp->data = data;
    temp->left = temp->right = NULL;
    return temp;
}

// Driver program to test above functions
int main()
{
    // Let us create binary tree shown in above diagram
    Node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);

    int k = 2;
    int node = 5;

    // print kth ancestor of given node
    cout<<kthAncestor(root,5,k,node);
    return 0;
}
```

Output:

**Time Complexity :**  $O(n)$

**Auxiliary Space :**  $O(n)$

### Source

<https://www.geeksforgeeks.org/kth-ancestor-node-binary-tree/>

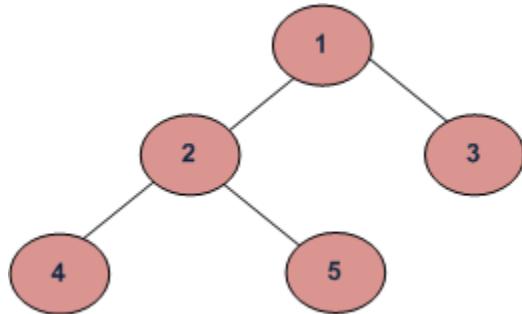
## Chapter 236

# Kth ancestor of a node in binary tree | Set 2

Kth ancestor of a node in binary tree | Set 2 - GeeksforGeeks

Given a binary tree in which nodes are numbered from 1 to n. Given a node and a positive integer K. We have to print the Kth ancestor of the given node in the binary tree. If there does not exist any such ancestor then print -1.

For example in the below given binary tree, 2nd ancestor of node 4 and 5 is 1. 3rd ancestor of node 4 will be -1.



We have discussed a BFS based solution for this problem in our [previous](#) article. If you observe that solution carefully, you will see that the basic approach was to first find the node and then backtrack to the kth parent. The same thing can be done using recursive DFS without using an extra array.

The idea of using DFS is to first find the given node in the tree, and then backtrack k times to reach to kth ancestor, once we have reached to the kth parent, we will simply print the node and return NULL.

Below is the C++ implementation of above idea:

```
/* C++ program to calculate Kth ancestor of given node */
#include <iostream>
```

```
#include <queue>
using namespace std;

// A Binary Tree Node
struct Node
{
    int data;
    struct Node *left, *right;
};

// temporary node to keep track of Node returned
// from previous recursive call during backtrack
Node* temp = NULL;

// recursive function to calculate Kth ancestor
Node* kthAncestorDFS(Node *root, int node , int &k)
{
    // Base case
    if (!root)
        return NULL;

    if (root->data == node ||
        (temp = kthAncestorDFS(root->left,node,k)) ||
        (temp = kthAncestorDFS(root->right,node,k)))
    {
        if (k > 0)
            k--;

        else if (k == 0)
        {
            // print the kth ancestor
            cout<<"Kth ancestor is: "<<root->data;

            // return NULL to stop further backtracking
            return NULL;
        }
    }

    // return current node to previous call
    return root;
}

// Utility function to create a new tree node
Node* newNode(int data)
{
    Node *temp = new Node;
    temp->data = data;
    temp->left = temp->right = NULL;
```

```
    return temp;
}

// Driver program to test above functions
int main()
{
    // Let us create binary tree shown in above diagram
    Node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);

    int k = 2;
    int node = 5;

    // print kth ancestor of given node
    Node* parent = kthAncestorDFS(root, node, k);

    // check if parent is not NULL, it means
    // there is no Kth ancestor of the node
    if (parent)
        cout << "-1";

    return 0;
}
```

Output:

Kth ancestor is: 1

**Time Complexity :**  $O(n)$ , where  $n$  is the number of nodes in the binary tree.

## Source

<https://www.geeksforgeeks.org/kth-ancestor-node-binary-tree-set-2/>

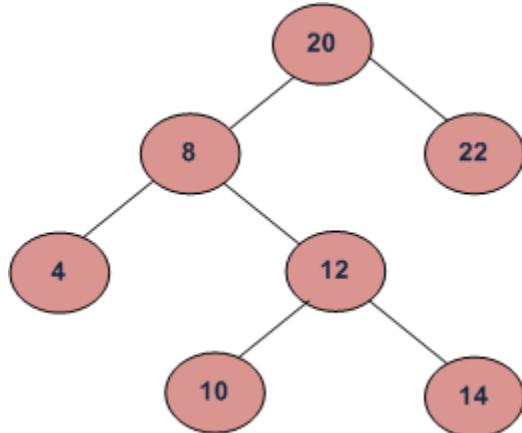
## Chapter 237

# K'th Largest Element in BST when modification to BST is not allowed

K'th Largest Element in BST when modification to BST is not allowed - GeeksforGeeks

Given a Binary Search Tree (BST) and a positive integer k, find the k'th largest element in the Binary Search Tree.

For example, in the following BST, if k = 3, then output should be 14, and if k = 5, then output should be 10.



We have discussed two methods in [this post](#). The method 1 requires O(n) time. The method 2 takes O(h) time where h is height of BST, but requires augmenting the BST (storing count of nodes in left subtree with every node).

Can we find k'th largest element in better than O(n) time and no augmentation?

In this post, a method is discussed that takes O(h + k) time. This method doesn't require any change to BST.

The idea is to do reverse inorder traversal of BST. The reverse inorder traversal traverses all nodes in decreasing order. While doing the traversal, we keep track of count of nodes visited so far. When the count becomes equal to k, we stop the traversal and print the key.

C++

```
// C++ program to find k'th largest element in BST
#include<iostream>
using namespace std;

struct Node
{
    int key;
    Node *left, *right;
};

// A utility function to create a new BST node
Node *newNode(int item)
{
    Node *temp = new Node;
    temp->key = item;
    temp->left = temp->right = NULL;
    return temp;
}

// A function to find k'th largest element in a given tree.
void kthLargestUtil(Node *root, int k, int &c)
{
    // Base cases, the second condition is important to
    // avoid unnecessary recursive calls
    if (root == NULL || c >= k)
        return;

    // Follow reverse inorder traversal so that the
    // largest element is visited first
    kthLargestUtil(root->right, k, c);

    // Increment count of visited nodes
    c++;

    // If c becomes k now, then this is the k'th largest
    if (c == k)
    {
        cout << "K'th largest element is "
            << root->key << endl;
        return;
    }
}
```

```
// Recur for left subtree
kthLargestUtil(root->left, k, c);
}

// Function to find k'th largest element
void kthLargest(Node *root, int k)
{
    // Initialize count of nodes visited as 0
    int c = 0;

    // Note that c is passed by reference
    kthLargestUtil(root, k, c);
}

/* A utility function to insert a new node with given key in BST */
Node* insert(Node* node, int key)
{
    /* If the tree is empty, return a new node */
    if (node == NULL) return newNode(key);

    /* Otherwise, recur down the tree */
    if (key < node->key)
        node->left = insert(node->left, key);
    else if (key > node->key)
        node->right = insert(node->right, key);

    /* return the (unchanged) node pointer */
    return node;
}

// Driver Program to test above functions
int main()
{
    /* Let us create following BST
       50
      /   \
     30   70
    / \   / \
   20 40 60 80 */
    Node *root = NULL;
    root = insert(root, 50);
    insert(root, 30);
    insert(root, 20);
    insert(root, 40);
    insert(root, 70);
    insert(root, 60);
    insert(root, 80);
```

```
int c = 0;
for (int k=1; k<=7; k++)
    kthLargest(root, k);

return 0;
}
```

**Java**

```
// Java code to find k'th largest element in BST

// A binary tree node
class Node {

    int data;
    Node left, right;

    Node(int d)
    {
        data = d;
        left = right = null;
    }
}

class BinarySearchTree {

    // Root of BST
    Node root;

    // Constructor
    BinarySearchTree()
    {
        root = null;
    }

    // function to insert nodes
    public void insert(int data)
    {
        this.root = this.insertRec(this.root, data);
    }

    /* A utility function to insert a new node
    with given key in BST */
    Node insertRec(Node node, int data)
    {
        /* If the tree is empty, return a new node */
        if (node == null) {
            this.root = new Node(data);
```

```
        return this.root;
    }

    if (data == node.data) {
        return node;
    }

    /* Otherwise, recur down the tree */
    if (data < node.data) {
        node.left = this.insertRec(node.left, data);
    } else {
        node.right = this.insertRec(node.right, data);
    }
    return node;
}

// class that stores the value of count
public class count {
    int c = 0;
}

// utility function to find kth largest no in
// a given tree
void kthLargestUtil(Node node, int k, count C)
{
    // Base cases, the second condition is important to
    // avoid unnecessary recursive calls
    if (node == null || C.c >= k)
        return;

    // Follow reverse inorder traversal so that the
    // largest element is visited first
    this.kthLargestUtil(node.right, k, C);

    // Increment count of visited nodes
    C.c++;

    // If c becomes k now, then this is the k'th largest
    if (C.c == k) {
        System.out.println(k + "th largest element is " +
                           node.data);
        return;
    }

    // Recur for left subtree
    this.kthLargestUtil(node.left, k, C);
}
```

```
// Method to find the kth largest no in given BST
void kthLargest(int k)
{
    count c = new count(); // object of class count
    this.kthLargestUtil(this.root, k, c);
}

// Driver function
public static void main(String[] args)
{
    BinarySearchTree tree = new BinarySearchTree();

    /* Let us create following BST
       50
      /   \
     30   70
    / \   / \
   20 40 60 80 */
    tree.insert(50);
    tree.insert(30);
    tree.insert(20);
    tree.insert(40);
    tree.insert(70);
    tree.insert(60);
    tree.insert(80);

    for (int i = 1; i <= 7; i++) {
        tree.kthLargest(i);
    }
}
}

// This code is contributed by Kamal Rawal
```

```
K'th largest element is 80
K'th largest element is 70
K'th largest element is 60
K'th largest element is 50
K'th largest element is 40
K'th largest element is 30
K'th largest element is 20
```

Time complexity: The code first traverses down to the rightmost node which takes  $O(h)$  time, then traverses  $k$  elements in  $O(k)$  time. Therefore overall time complexity is  $O(h + k)$ .

This article is contributed by **Chirag Sharma**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

**Source**

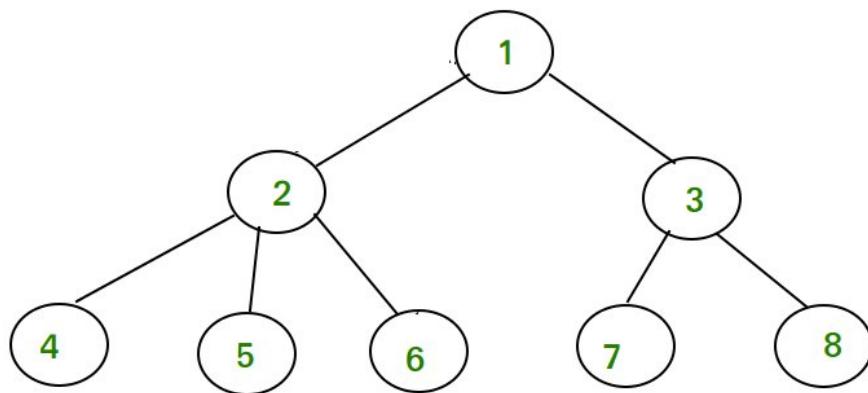
<https://www.geeksforgeeks.org/kth-largest-element-in-bst-when-modification-to-bst-is-not-allowed/>

## Chapter 238

# LCA for general or n-ary trees (Sparse Matrix DP approach < O(nlogn), O(logn)>)

LCA for general or n-ary trees (Sparse Matrix DP approach < O(nlogn), O(logn)>) - GeeksforGeeks

In previous posts, we have discussed how to calculate the Lowest Common Ancestor (LCA) for a binary tree and a binary search tree ([this](#), [this](#) and [this](#)). Now let's look at a method that can calculate LCA for any tree (not only for binary tree). We use Dynamic Programming with Sparse Matrix Approach in our method. This method is very handy and fast when you need to answer multiple queries of LCA for a tree.



$$\begin{aligned} \text{LCA}(4, 6) &= 2 \\ \text{LCA}(5, 7) &= 1 \end{aligned}$$

Pre-requisites : -

- 1) [DFS](#)
- 2) Basic DP knowledge ([This](#) and [this](#))
- 3) [Range Minimum Query \(Square Root Decomposition and Sparse Table\)](#)

### Naive Approach:- $O(n)$

The naive approach for this general tree LCA calculation will be the same as the naive approach for the LCA calculation of Binary Tree (this naive approach is already well described [here](#).

The C++ implementation for the naive approach is given below :-

```
/* Program to find LCA of n1 and n2 using one DFS on
   the Tree */
#include <iostream>
#include <vector>
using namespace std;

// Maximum number of nodes is 100000 and nodes are
// numbered from 1 to 100000
#define MAXN 100001

vector < int > tree[MAXN];
int path[3][MAXN]; // storing root to node path

// storing the path from root to node
void dfs(int cur, int prev, int pathNumber, int ptr,
         int node, bool &flag)
{
    for (int i=0; i<tree[cur].size(); i++)
    {
        if (tree[cur][i] != prev and !flag)
        {
            // pushing current node into the path
            path[pathNumber][ptr] = tree[cur][i];
            if (tree[cur][i] == node)
            {
                // node found
                flag = true;

                // terminating the path
                path[pathNumber][ptr+1] = -1;
                return;
            }
            dfs(tree[cur][i], cur, pathNumber, ptr+1,
                 node, flag);
        }
    }
}
```

```
}  
  
// This Function compares the path from root to 'a' & root  
// to 'b' and returns LCA of a and b. Time Complexity : O(n)  
int LCA(int a, int b)  
{  
    // trivial case  
    if (a == b)  
        return a;  
  
    // setting root to be first element in path  
    path[1][0] = path[2][0] = 1;  
  
    // calculating path from root to a  
    bool flag = false;  
    dfs(1, 0, 1, 1, a, flag);  
  
    // calculating path from root to b  
    flag = false;  
    dfs(1, 0, 2, 1, b, flag);  
  
    // runs till path 1 & path 2 matches  
    int i = 0;  
    while (path[1][i] == path[2][i])  
        i++;  
  
    // returns the last matching node in the paths  
    return path[1][i-1];  
}  
  
void addEdge(int a,int b)  
{  
    tree[a].push_back(b);  
    tree[b].push_back(a);  
}  
  
// Driver code  
int main()  
{  
    int n = 8; // Number of nodes  
    addEdge(1,2);  
    addEdge(1,3);  
    addEdge(2,4);  
    addEdge(2,5);  
    addEdge(2,6);  
    addEdge(3,7);  
    addEdge(3,8);  
}
```

```

cout << "LCA(4, 7) = " << LCA(4,7) << endl;
cout << "LCA(4, 6) = " << LCA(4,6) << endl;
return 0;
}

```

Output:

```

LCA(4, 7) = 1
LCA(4, 6) = 2

```

### Sparse Matrix Approach ( $O(n\log n)$ ) pre-processing, $O(\log n)$ – query

**Pre-computation** :- Here we store the  $2^i$  th parent for every node, where  $0 \leq i < \text{LEVEL}$ , here “LEVEL” is a constant integer that tells the maximum number of  $2^i$  th ancestor possible.

Therefore, we assume the worst case to see what is the value of the constant LEVEL. In our worst case every node in our tree will have at max 1 parent and 1 child or we can say it simply reduces to a linked list.

So, in this case  $\text{LEVEL} = \text{ceil}(\log(\text{number of nodes}))$ .

We also pre-compute the height for each node using one dfs in  $O(n)$  time.

```

int n          // number of nodes
int parent[MAXN][LEVEL] // all initialized to -1

parent[node][0] : contains the  $2^0$ th(first)
parent of all the nodes pre-computed using DFS

// Sparse matrix Approach
for node -> 1 to n :
    for i-> 1 to LEVEL :
        if ( parent[node][i-1] != -1 ) :
            parent[node][i] =
                parent[ parent[node][i-1] ][i-1]

```

Now , as we see the above dynamic programming code runs two nested loop that runs over their complete range respectively.

Hence, it can be easily be inferred that its asymptotic Time Complexity is  $O(\text{number of nodes} * \text{LEVEL}) \sim O(n * \text{LEVEL}) \sim O(n \log n)$ .

#### Return LCA(u,v) :-

- 1) First Step is to bring both the nodes at the same height. As we have already pre-computed the heights for each node. We first calculate the difference in the heights of u and v (let's say  $v \geq u$ ). Now we need the node 'v' to jump  $h$  nodes above. This can be easily done in  $O(\log h)$  time ( where  $h$  is the difference in the heights of u and v) as we have already stored

the  $2^i$  parent for each node. This process is exactly same as calculating  $x \wedge y$  in  $O(\log y)$  time. (See the code for better understanding).

2) Now both u and v nodes are at same height. Therefore now once again we will use  $2^i$  jumping strategy to reach the first Common Parent of u and v.

**Pseudo-code:**

```
For i-> LEVEL to 0 :
    If parent[u][i] != parent[v][i] :
        u = parent[u][i]
        v = parent[v][i]
```

C++ implementation of the above algorithm is given below:

```
// Sparse Matrix DP approach to find LCA of two nodes
#include <bits/stdc++.h>
using namespace std;
#define MAXN 100000
#define level 18

vector <int> tree[MAXN];
int depth[MAXN];
int parent[MAXN][level];

// pre-compute the depth for each node and their
// first parent( $2^0$ th parent)
// time complexity :  $O(n)$ 
void dfs(int cur, int prev)
{
    depth[cur] = depth[prev] + 1;
    parent[cur][0] = prev;
    for (int i=0; i<tree[cur].size(); i++)
    {
        if (tree[cur][i] != prev)
            dfs(tree[cur][i], cur);
    }
}

// Dynamic Programming Sparse Matrix Approach
// populating  $2^i$  parent for each node
// Time complexity :  $O(n\log n)$ 
void precomputeSparseMatrix(int n)
{
    for (int i=1; i<level; i++)
    {
        for (int node = 1; node <= n; node++)
        {
            if (parent[node][i-1] != -1)
```

```
        parent[node][i] =
            parent[parent[node][i-1]][i-1];
    }
}

// Returning the LCA of u and v
// Time complexity : O(log n)
int lca(int u, int v)
{
    if (depth[v] < depth[u])
        swap(u, v);

    int diff = depth[v] - depth[u];

    // Step 1 of the pseudocode
    for (int i=0; i<level; i++)
        if ((diff>>i)&1)
            v = parent[v][i];

    // now depth[u] == depth[v]
    if (u == v)
        return u;

    // Step 2 of the pseudocode
    for (int i=level-1; i>=0; i--)
        if (parent[u][i] != parent[v][i])
    {
        u = parent[u][i];
        v = parent[v][i];
    }

    return parent[u][0];
}

void addEdge(int u,int v)
{
    tree[u].push_back(v);
    tree[v].push_back(u);
}

// driver function
int main()
{
    memset(parent,-1,sizeof(parent));
    int n = 8;
    addEdge(1,2);
    addEdge(1,3);
```

```
addEdge(2,4);
addEdge(2,5);
addEdge(2,6);
addEdge(3,7);
addEdge(3,8);
depth[0] = 0;

// running dfs and precalculating depth
// of each node.
dfs(1,0);

// Precomputing the  $2^i$  th ancestor for every node
precomputeSparseMatrix(n);

// calling the LCA function
cout << "LCA(4, 7) = " << lca(4,7) << endl;
cout << "LCA(4, 6) = " << lca(4,6) << endl;
return 0;
}
```

Output:

```
LCA(4,7) = 1
LCA(4,6) = 2
```

**Time Complexity:** The time complexity for answering a single LCA query will be  $O(\log n)$  but the overall time complexity is dominated by precalculation of the  $2^i$  th ( $0 \leq i \leq \text{level}$ ) ancestors for each node. Hence, the overall asymptotic Time Complexity will be  $O(n * \log n)$  and Space Complexity will be  $O(n \log n)$ , for storing the data about the ancestors of each node.

## Source

<https://www.geeksforgeeks.org/lca-for-general-or-n-ary-trees-sparse-matrix-dp-approach-onlogn-ologn/>

## Chapter 239

# LCA for n-ary Tree | Constant Query O(1)

[LCA for n-ary Tree | Constant Query O\(1\) - GeeksforGeeks](#)

We have seen various methods with different Time Complexities to calculate LCA in n-ary tree:-

[Method 1 : Naive Method \( by calculating root to node path\) | O\(n\) per query](#)

[Method 2 :Using Sqrt Decomposition | O\(sqrt H\)](#)

[Method 3 : Using Sparse Matrix DP approach | O\(logn\)](#)

Lets study another method which has faster query time than all the above methods. So, our aim will be to calculate LCA in **constant time** ~ **O(1)**. Let's see how we can achieve it.

### Method 4 : Using Range Minimum Query

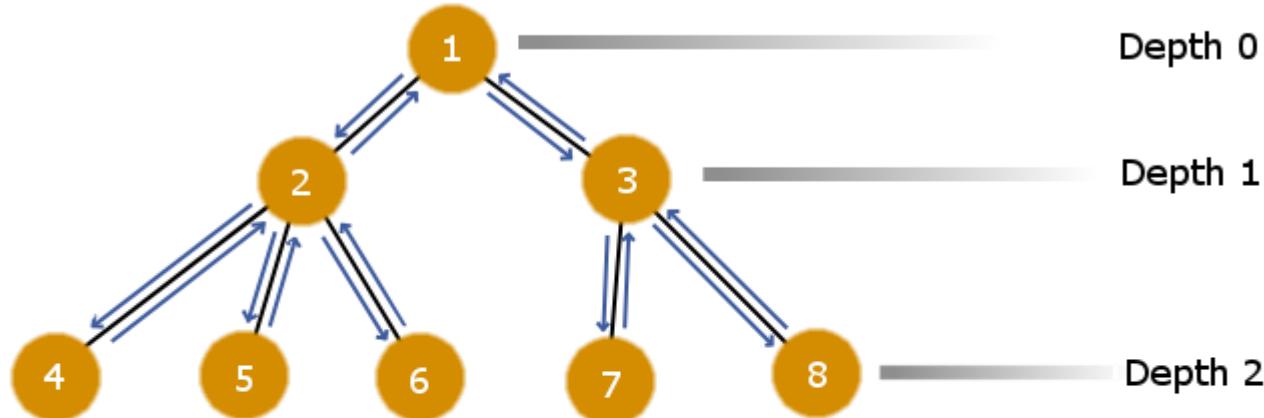
We have discussed [LCA and RMQ for binary tree](#). Here we discuss LCA problem to RMQ problem conversion for n-ary tree.

Pre-requisites:- [LCA in Binary Tree using RMQ](#)  
[RMQ using sparse table](#)

**Key Concept :** In this method, we will be reducing our LCA problem to RMQ(Range Minimum Query) problem over a static array. Once, we do that then we will relate the Range minimum queries to the required LCA queries.

The first step will be to decompose the tree into a flat linear array. To do this we can apply the Euler walk . The Euler walk will give the pre-order traversal of the graph. So we will perform a Euler Walk on the tree and store the nodes in an array as we visit them. This process reduces the tree data-structure to a simple linear array.

Consider the below tree and the euler walk over it :-



*Euler Walk for the above tree*

Euler [] = 

1	2	4	2	5	2	6	2	1	3	7	3	8	3	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Now lets think in general terms : Consider any two nodes on the tree. There will be exactly one path connecting both the nodes and the node that has the smallest depth value in the path will be the LCA of the two given nodes.

Now take any two distinct node say  $u$  and  $v$  in the Euler walk array. Now all the elements in the path from  $u$  to  $v$  will lie in between the index of nodes  $u$  and  $v$  in the Euler walk array. Therefore, we just need to calculate the node with the minimum depth between the index of node  $u$  and node  $v$  in the euler array.

For this we will maintain another array that will contain the depth of all the nodes corresponding to their position in the Euler walk array so that we can Apply our RMQ algorithm on it.

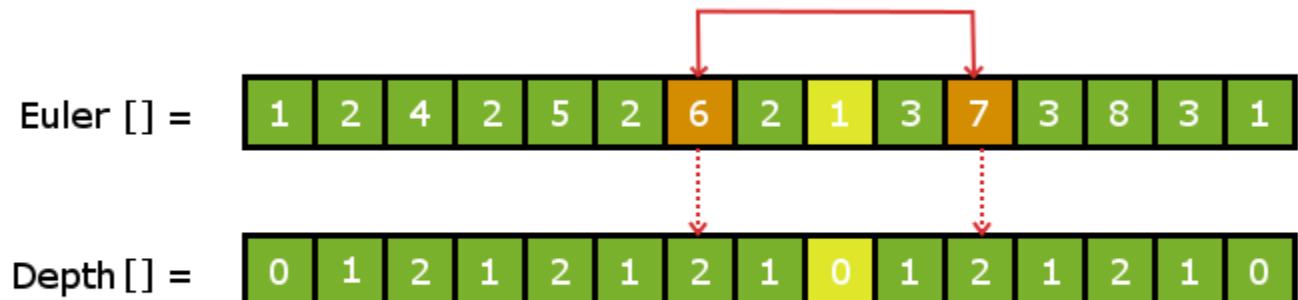
Given below is the euler walk array parallel to its depth track array.

**Euler [] =** 

**Depth [] =** 

Example :- Consider two nodes **node 6** and **node 7** in the euler array. To calculate the LCA of node 6 and node 7 we look for the smallest depth value for all the nodes in between node 6 and node 7 .

Therefore, **node 1** has the smallest *depth value* = 0 and hence, it is the LCA for node 6 and node 7.



$$\text{LCA}(6,7) = 1$$

**Implementation :-**

We will be maintaining three arrays 1)Euler Path  
2)Depth array  
3)First Appearance Index

Euler Path and Depth array are the same as described above

**First Appearance Index FAI[] :** The First Appearance index Array will store the index for the first position of every node in the Euler Path array. FAI[i] = First appearance of ith node in Euler Walk array.

The C++ Implementation for the above method is given below:-

```

// C++ program to demonstrate LCA of n-ary tree
// in constant time.
#include "bits/stdc++.h"
using namespace std;
#define sz 101

vector < int > adj[sz]; // stores the tree
vector < int > euler; // tracks the eulerwalk
vector < int > depthArr; // depth for each node corresponding
                        // to eulerwalk

int FAI[sz]; // stores first appearance index of every node
int level[sz]; // stores depth for all nodes in the tree
int ptr; // pointer to euler walk
int dp[sz][18]; // sparse table
int logn[sz]; // stores log values
int p2[20]; // stores power of 2

void buildSparseTable(int n)
{
    // initializing sparse table
    memset(dp,-1,sizeof(dp));

    // filling base case values
    for (int i=1; i<n; i++)
        dp[i-1][0] = (depthArr[i]>depthArr[i-1])?i-1:i;

    // dp to fill sparse table
    for (int l=1; l<15; l++)
        for (int i=0; i<n; i++)
            if (dp[i][l-1]!=-1 and dp[i+p2[l-1]][l-1]!=-1)
                dp[i][l] =
                    (depthArr[dp[i][l-1]]>depthArr[dp[i+p2[l-1]][l-1]])?
                    dp[i+p2[l-1]][l-1] : dp[i][l-1];
            else
                break;
}

int query(int l,int r)
{
    int d = r-l;
    int dx = logn[d];
    if (l==r) return l;
    if (depthArr[dp[l][dx]] > depthArr[dp[r-p2[dx]][dx]])
        return dp[r-p2[dx]][dx];
    else
        return dp[l][dx];
}

```

```
void preprocess()
{
    // memorizing powers of 2
    p2[0] = 1;
    for (int i=1; i<18; i++)
        p2[i] = p2[i-1]*2;

    // memorizing all log(n) values
    int val = 1,ptr=0;
    for (int i=1; i<sz; i++)
    {
        logn[i] = ptr-1;
        if (val==i)
        {
            val*=2;
            logn[i] = ptr;
            ptr++;
        }
    }
}

/**
 * Euler Walk ( preorder traversal)
 * converting tree to linear depthArray
 * Time Complexity : O(n)
 */
void dfs(int cur,int prev,int dep)
{
    // marking FAI for cur node
    if (FAI[cur]==-1)
        FAI[cur] = ptr;

    level[cur] = dep;

    // pushing root to euler walk
    euler.push_back(cur);

    // incrementing euler walk pointer
    ptr++;

    for (auto x:adj[cur])
    {
        if (x != prev)
        {
            dfs(x,cur,dep+1);

            // pushing cur again in backtrack
        }
    }
}
```

```
// of euler walk
euler.push_back(cur);

// increment euler walk pointer
ptr++;
}

}

// Create Level depthArray corresponding
// to the Euler walk Array
void makeArr()
{
    for (auto x : euler)
        depthArr.push_back(level[x]);
}

int LCA(int u,int v)
{
    // trival case
    if (u==v)
        return u;

    if (FAI[u] > FAI[v])
        swap(u,v);

    // doing RMQ in the required range
    return euler[query(FAI[u], FAI[v])];
}

void addEdge(int u,int v)
{
    adj[u].push_back(v);
    adj[v].push_back(u);
}

int main(int argc, char const *argv[])
{
    // constructing the described tree
    int numberOfNodes = 8;
    addEdge(1,2);
    addEdge(1,3);
    addEdge(2,4);
    addEdge(2,5);
    addEdge(2,6);
    addEdge(3,7);
    addEdge(3,8);
```

```
// performing required precalculations
preprocess();

// doing the Euler walk
ptr = 0;
memset(FAI,-1,sizeof(FAI));
dfs(1,0,0);

// creating depthArray corresponding to euler[]
makeArr();

// building sparse table
buildSparseTable(depthArr.size());

cout << "LCA(6,7) : " << LCA(6,7) << "\n";
cout << "LCA(6,4) : " << LCA(6,4) << "\n";

return 0;
}
```

Output:

```
LCA(6,7) : 1
LCA(6,4) : 2
```

**Note :** We are precalculating all the required power of 2's and also precalculating the all the required log values to ensure constant time complexity per query. Else if we did log calculation for every query operation our Time complexity would have not been constant.

**Time Complexity:** The Conversion process from LCA to RMQ is done by Euler Walk that takes  $O(n)$  time.

Pre-processing for the sparse table in RMQ takes  $O(n\log n)$  time and answering each Query is a Constant time process. Therefore, overall Time Complexity is  $O(n\log n)$  – preprocessing and  **$O(1)$**  for each query.

## Source

<https://www.geeksforgeeks.org/lca-n-ary-tree-constant-query-o1/>

## Chapter 240

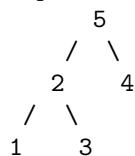
# Largest BST in a Binary Tree | Set 2

Largest BST in a Binary Tree | Set 2 - GeeksforGeeks

Given a Binary Tree, write a function that returns the size of the largest subtree which is also a Binary Search Tree (BST). If the complete Binary Tree is BST, then return the size of whole tree.

Examples:

Input:

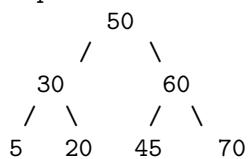


Output: 3

The following subtree is the maximum size BST subtree



Input:



```

      / \
    65   80
Output: 5
The following subtree is the
maximum size BST subtree
      60
     / \
    45   70
   /   \
  65   80

```

We have discussed a two methods in below post.

[Find the largest BST subtree in a given Binary Tree | Set 1](#)

In this post a different  $O(n)$  solution is discussed. This solution is simpler than the solutions discussed above and works in  $O(n)$  time.

The idea is based on method 3 of [check if a binary tree is BST](#) article.

A Tree is BST if following is true for every node x.

1. The largest value in left subtree (of x) is smaller than value of x.
2. The smallest value in right subtree (of x) is greater than value of x.

We traverse tree in bottom up manner. For every traversed node, we return maximum and minimum values in subtree rooted with it. If any node follows above properties and size of

```

// C++ program to find largest BST in a
// Binary Tree.
#include <bits/stdc++.h>
using namespace std;

/* A binary tree node has data,
pointer to left child and a
pointer to right child */
struct Node
{
    int data;
    struct Node* left;
    struct Node* right;
};

/* Helper function that allocates a new
node with the given data and NULL left
and right pointers. */
struct Node* newNode(int data)
{
    struct Node* node = new Node;
    node->data = data;
}

```

```

node->left = node->right = NULL;

return(node);
}

// Information to be returned by every
// node in bottom up traversal.
struct Info
{
    int sz; // Size of subtree
    int max; // Min value in subtree
    int min; // Max value in subtree
    int ans; // Size of largest BST which
    // is subtree of current node
    bool isBST; // If subtree is BST
};

// Returns Information about subtree. The
// Information also includes size of largest
// subtree which is a BST.
Info largestBSTBT(Node* root)
{
    // Base cases : When tree is empty or it has
    // one child.
    if (root == NULL)
        return {0, INT_MIN, INT_MAX, 0, true};
    if (root->left == NULL && root->right == NULL)
        return {1, root->data, root->data, 1, true};

    // Recur for left subtree and right subtrees
    Info l = largestBSTBT(root->left);
    Info r = largestBSTBT(root->right);

    // Create a return variable and initialize its
    // size.
    Info ret;
    ret.sz = (1 + l.sz + r.sz);

    // If whole tree rooted under current root is
    // BST.
    if (l.isBST && r.isBST && l.max < root->data &&
        r.min > root->data)
    {
        ret.min = min(l.min, min(r.min, root->data));
        ret.max = max(r.max, max(l.max, root->data));

        // Update answer for tree rooted under
        // current 'root'
    }
}

```

```
    ret.ans = ret.sz;
    ret.isBST = true;

    return ret;
}

// If whole tree is not BST, return maximum
// of left and right subtrees
ret.ans = max(l.ans, r.ans);
ret.isBST = false;

return ret;
}

/* Driver program to test above functions*/
int main()
{
    /* Let us construct the following Tree
       60
      /   \
     65   70
      /
     50 */

    struct Node *root = newNode(60);
    root->left = newNode(65);
    root->right = newNode(70);
    root->left->left = newNode(50);
    printf(" Size of the largest BST is %d\n",
           largestBSTBT(root).ans);
    return 0;
}

// This code is contributed by Vivek Garg in a
// comment on below set 1.
// www.geeksforgeeks.org/find-the-largest-subtree-in-a-tree-that-is-also-a-bst/
```

Output:

```
Size of largest BST is 2
```

Time Complexity : O(n)

## Source

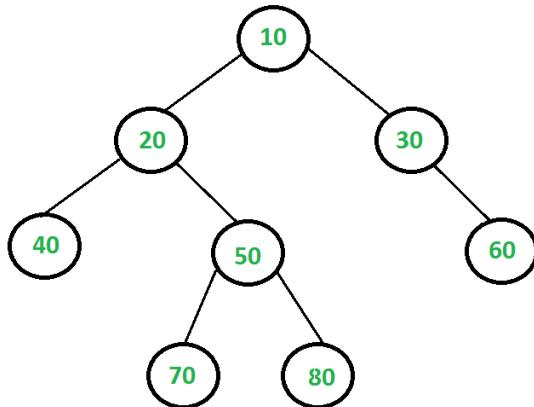
<https://www.geeksforgeeks.org/largest-bst-binary-tree-set-2/>

## Chapter 241

# Largest Independent Set Problem | DP-26

Largest Independent Set Problem | DP-26 - GeeksforGeeks

Given a Binary Tree, find size of the **Largest Independent Set(LIS)** in it. A subset of all tree nodes is an independent set if there is no edge between any two nodes of the subset. For example, consider the following binary tree. The largest independent set(LIS) is {10, 40, 60, 70, 80} and size of the LIS is 5.



A Dynamic Programming solution solves a given problem using solutions of subproblems in bottom up manner. Can the given problem be solved using solutions to subproblems? If yes, then what are the subproblems? Can we find largest independent set size (LISS) for a node X if we know LISS for all descendants of X? If a node is considered as part of LIS, then its children cannot be part of LIS, but its grandchildren can be. Following is optimal substructure property.

### 1) Optimal Substructure:

Let LISS(X) indicates size of largest independent set of a tree with root X.

```
LISS(X) = MAX { (1 + sum of LISS for all grandchildren of X),
                 (sum of LISS for all children of X) }
```

The idea is simple, there are two possibilities for every node X, either X is a member of the set or not a member. If X is a member, then the value of LISS(X) is 1 plus LISS of all grandchildren. If X is not a member, then the value is sum of LISS of all children.

## 2) Overlapping Subproblems

Following is recursive implementation that simply follows the recursive structure mentioned above.

```
// A naive recursive implementation of Largest Independent Set problem
#include <stdio.h>
#include <stdlib.h>

// A utility function to find max of two integers
int max(int x, int y) { return (x > y)? x: y; }

/* A binary tree node has data, pointer to left child and a pointer to
   right child */
struct node
{
    int data;
    struct node *left, *right;
};

// The function returns size of the largest independent set in a given
// binary tree
int LISS(struct node *root)
{
    if (root == NULL)
        return 0;

    // Calculate size excluding the current node
    int size_excl = LISS(root->left) + LISS(root->right);

    // Calculate size including the current node
    int size_incl = 1;
    if (root->left)
        size_incl += LISS(root->left->left) + LISS(root->left->right);
    if (root->right)
        size_incl += LISS(root->right->left) + LISS(root->right->right);

    // Return the maximum of two sizes
    return max(size_incl, size_excl);
}
```

```

// A utility function to create a node
struct node* newNode( int data )
{
    struct node* temp = (struct node *) malloc( sizeof(struct node) );
    temp->data = data;
    temp->left = temp->right = NULL;
    return temp;
}

// Driver program to test above functions
int main()
{
    // Let us construct the tree given in the above diagram
    struct node *root      = newNode(20);
    root->left           = newNode(8);
    root->left->left     = newNode(4);
    root->left->right    = newNode(12);
    root->left->right->left = newNode(10);
    root->left->right->right = newNode(14);
    root->right          = newNode(22);
    root->right->right   = newNode(25);

    printf ("Size of the Largest Independent Set is %d ", LISS(root));

    return 0;
}

```

Output:

```
Size of the Largest Independent Set is 5
```

Time complexity of the above naive recursive approach is exponential. It should be noted that the above function computes the same subproblems again and again. For example, LISS of node with value 50 is evaluated for node with values 10 and 20 as 50 is grandchild of 10 and child of 20.

Since same subproblems are called again, this problem has Overlapping Subproblems property. So LISS problem has both properties (see [this](#) and [this](#)) of a dynamic programming problem. Like other typical [Dynamic Programming\(DP\) problems](#), recomputations of same subproblems can be avoided by storing the solutions to subproblems and solving problems in bottom up manner.

Following is C implementation of Dynamic Programming based solution. In the following solution, an additional field ‘liss’ is added to tree nodes. The initial value of ‘liss’ is set as 0 for all nodes. The recursive function LISS() calculates ‘liss’ for a node only if it is not already set.

C

```

/* Dynamic programming based program for Largest Independent Set problem */
#include <stdio.h>
#include <stdlib.h>

// A utility function to find max of two integers
int max(int x, int y) { return (x > y)? x: y; }

/* A binary tree node has data, pointer to left child and a pointer to
   right child */
struct node
{
    int data;
    int liss;
    struct node *left, *right;
};

// A memoization function returns size of the largest independent set in
// a given binary tree
int LISS(struct node *root)
{
    if (root == NULL)
        return 0;

    if (root->liss)
        return root->liss;

    if (root->left == NULL && root->right == NULL)
        return (root->liss = 1);

    // Calculate size excluding the current node
    int liss_excl = LISS(root->left) + LISS(root->right);

    // Calculate size including the current node
    int liss_incl = 1;
    if (root->left)
        liss_incl += LISS(root->left->left) + LISS(root->left->right);
    if (root->right)
        liss_incl += LISS(root->right->left) + LISS(root->right->right);

    // Maximum of two sizes is LISS, store it for future uses.
    root->liss = max(liss_incl, liss_excl);

    return root->liss;
}

// A utility function to create a node
struct node* newNode(int data)
{

```

```
struct node* temp = (struct node *) malloc( sizeof(struct node) );
temp->data = data;
temp->left = temp->right = NULL;
temp->liss = 0;
return temp;
}

// Driver program to test above functions
int main()
{
    // Let us construct the tree given in the above diagram
    struct node *root      = newNode(20);
    root->left           = newNode(8);
    root->left->left    = newNode(4);
    root->left->right   = newNode(12);
    root->left->right->left = newNode(10);
    root->left->right->right = newNode(14);
    root->right          = newNode(22);
    root->right->right  = newNode(25);

    printf ("Size of the Largest Independent Set is %d ", LISS(root));

    return 0;
}
```

### Java

```
// Java program for calculating LISS
// using dynamic programming

public class LisTree
{
    /* A binary tree node has data, pointer
       to left child and a pointer to right
       child */
    static class node
    {
        int data, liss;
        node left, right;

        public node(int data)
        {
            this.data = data;
            this.liss = 0;
        }
    }

    // A memoization function returns size
```

```

// of the largest independent set in
// a given binary tree
static int liss(node root)
{
    if (root == null)
        return 0;
    if (root.liss != 0)
        return root.liss;
    if (root.left == null && root.right == null)
        return root.liss = 1;

    // Calculate size excluding the
    // current node
    int liss_excl = liss(root.left) + liss(root.right);

    // Calculate size including the
    // current node
    int liss_incl = 1;
    if (root.left != null)
    {
        liss_incl += (liss(root.left.left) + liss(root.left.right));
    }
    if (root.right != null)
    {
        liss_incl += (liss(root.right.left) + liss(root.right.right));
    }

    // Maximum of two sizes is LISS,
    // store it for future uses.
    return root.liss = Math.max(liss_excl, liss_incl);
}

public static void main(String[] args)
{
    // Let us construct the tree given
    // in the above diagram

    node root = new node(20);
    root.left = new node(8);
    root.left.left = new node(4);
    root.left.right = new node(12);
    root.left.right.left = new node(10);
    root.left.right.right = new node(14);
    root.right = new node(22);
    root.right.right = new node(25);
    System.out.println("Size of the Largest Independent Set is " + liss(root));
}
}

```

```
// This code is contributed by Rishabh Mahrsee
```

Output

```
Size of the Largest Independent Set is 5
```

Time Complexity:  $O(n)$  where  $n$  is the number of nodes in given Binary tree.

Following extensions to above solution can be tried as an exercise.

- 1) Extend the above solution for n-ary tree.
- 2) The above solution modifies the given tree structure by adding an additional field ‘liss’ to tree nodes. Extend the solution so that it doesn’t modify the tree structure.
- 3) The above solution only returns size of LIS, it doesn’t print elements of LIS. Extend the solution to print all nodes that are part of LIS.

## Source

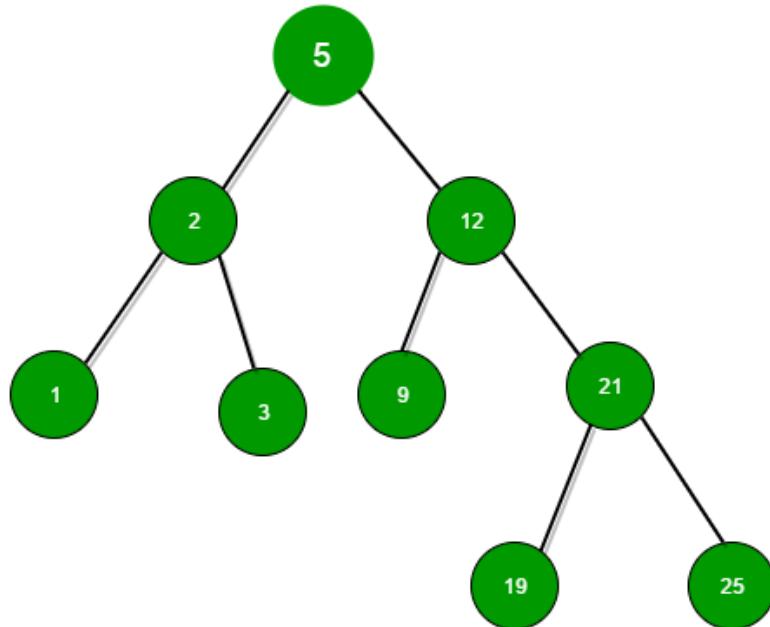
<https://www.geeksforgeeks.org/largest-independent-set-problem-dp-26/>

## Chapter 242

### Largest number in BST which is less than or equal to N

Largest number in BST which is less than or equal to N - GeeksforGeeks

We have a binary search tree and a number N. Our task is to find the greatest number in the binary search tree that is less than or equal to N. Print the value of the element if it exists otherwise print -1.



Examples: For the above given binary search tree-

```
Input : N = 24
Output :result = 21
(searching for 24 will be like-5->12->21)
```

```
Input : N = 4
Output : result = 3
(searching for 4 will be like-5->2->3)
```

We follow recursive approach for solving this problem. We start searching for element from root node. If we reach a leaf and its value is greater than N, element does not exist so return -1. Else if node's value is less than or equal to N and right value is NULL or greater than N, then return the node value as it will be the answer.

Otherwise if node's value is greater than N, then search for the element in the left subtree else search for the element in the right subtree by calling the same function by passing the left or right values accordingly.

```
// C++ code to find the largest value smaller
// than or equal to N
#include <bits/stdc++.h>
using namespace std;

struct Node {
    int key;
    Node *left, *right;
};

// To create new BST Node
Node* newNode(int item)
{
    Node* temp = new Node;
    temp->key = item;
    temp->left = temp->right = NULL;
    return temp;
}

// To insert a new node in BST
Node* insert(Node* node, int key)
{
    // if tree is empty return new node
    if (node == NULL)
        return newNode(key);

    // if key is less then or grater then
    // node value then recur down the tree
    if (key < node->key)
        node->left = insert(node->left, key);
    else if (key > node->key)
        node->right = insert(node->right, key);
}
```

```
// return the (unchanged) node pointer
return node;
}

// function to find max value less then N
int findMaxforN(Node* root, int N)
{
    // Base cases
    if (root == NULL)
        return -1;
    if (root->key == N)
        return N;

    // If root's value is smaller, try in rght
    // subtree
    else if (root->key < N) {
        int k = findMaxforN(root->right, N);
        if (k == -1)
            return root->key;
        else
            return k;
    }

    // If root's key is greater, return value
    // from left subtree.
    else if (root->key > N)
        return findMaxforN(root->left, N);
}

// Driver code
int main()
{
    int N = 4;

    // creating following BST
    /*
        5
       /   \
      2     12
     / \   / \
    1   3   9   21
           /   \
          19   25 */
    Node* root = insert(root, 25);
    insert(root, 2);
    insert(root, 1);
    insert(root, 3);
```

```
insert(root, 12);
insert(root, 9);
insert(root, 21);
insert(root, 19);
insert(root, 25);

printf("%d", findMaxforN(root, N));

return 0;
}
```

Output:

3

Time complexity = O(h) where h is height of BST.

**Reference :**

<https://www.careercup.com/question?id=5765237112307712>

**Improved By :** abhishekta0

## Source

<https://www.geeksforgeeks.org/largest-number-bst-less-equal-n/>

## Chapter 243

# Largest value in each level of Binary Tree

Largest value in each level of Binary Tree - GeeksforGeeks

Given a binary tree, find the largest value in each level.

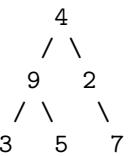
**Examples :**

**Input :**



**Output :** 1 3

**Input :**



**Output :** 4 9 7

**Approach :** The idea is to recursively traverse tree in a pre-order fashion. Root is considered to be at zeroth level. While traversing, keep track of the level of the element and if its current level is not equal to the number of elements present in the list, update the maximum element at that level in the list.

Below is the implementation to find largest value on each level of Binary Tree.

```
// C++ program to find largest
// value on each level of binary tree.
```

```
#include <bits/stdc++.h>
using namespace std;

/* A binary tree node has data,
pointer to left child and a
pointer to right child */
struct Node {
    int val;
    struct Node *left, *right;
};

/* Recursive function to find
the largest value on each level */
void helper(vector<int>& res, Node* root, int d)
{
    if (!root)
        return;

    // Expand list size
    if (d == res.size())
        res.push_back(root->val);

    else
        // to ensure largest value
        // on level is being stored
        res[d] = max(res[d], root->val);

    // Recursively traverse left and
    // right subtrees in order to find
    // out the largest value on each level
    helper(res, root->left, d + 1);
    helper(res, root->right, d + 1);
}

// function to find largest values
vector<int> largestValues(Node* root)
{
    vector<int> res;
    helper(res, root, 0);
    return res;
}

/* Helper function that allocates a
new node with the given data and
NULL left and right pointers. */
Node* newNode(int data)
{
```

```
Node* temp = new Node;
temp->val = data;
temp->left = temp->right = NULL;
return temp;
}

// Driver code
int main()
{
    /* Let us construct a Binary Tree
        4
       / \
      9   2
     / \   \
    3   5   7 */

    Node* root = NULL;
    root = newNode(4);
    root->left = newNode(9);
    root->right = newNode(2);
    root->left->left = newNode(3);
    root->left->right = newNode(5);
    root->right->right = newNode(7);

    vector<int> res = largestValues(root);
    for (int i = 0; i < res.size(); i++)
        cout << res[i] << " ";

    return 0;
}
```

Output:

4 9 7

[Largest value in each level of Binary Tree | Set-2 \(Iterative Approach\)](#)

**Complexity Analysis:**

- Time complexity :  $O(n)$ , where  $n$  is the number of nodes in binary tree.
- Auxiliary Space :  $O(n)$  as in worst case, depth of binary tree will be  $n$ .

**Source**

<https://www.geeksforgeeks.org/largest-value-level-binary-tree/>

## Chapter 244

# Largest value in each level of Binary Tree | Set-2 (Iterative Approach)

Largest value in each level of Binary Tree | Set-2 (Iterative Approach) - GeeksforGeeks

Given a binary tree containing **n** nodes. The problem is to find and print the largest value present in each level.

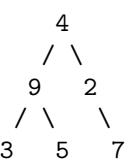
Examples:

Input :



Output : 1 3

Input :



Output : 4 9 7

**Approach:** In the [previous post](#), a recursive method have been discussed. In this post an iterative method has been discussed. The idea is to perform [iterative level order traversal](#) of the binary tree using queue. While traversing keep **max** variable which stores the maximum element of the current level of the tree being processed. When the level is completely traversed, print that **max** value.

```
// C++ implementation to print largest
// value in each level of Binary Tree
#include <bits/stdc++.h>

using namespace std;

// structure of a node of binary tree
struct Node {
    int data;
    Node *left, *right;
};

// function to get a new node
Node* newNode(int data)
{
    // allocate space
    Node* temp = new Node;

    // put in the data
    temp->data = data;
    temp->left = temp->right = NULL;
    return temp;
}

// function to print largest value
// in each level of Binary Tree
void largestValueInEachLevel(Node* root)
{
    // if tree is empty
    if (!root)
        return;

    queue<Node*> q;
    int nc, max;

    // push root to the queue 'q'
    q.push(root);

    while (1) {
        // node count for the current level
        nc = q.size();

        // if true then all the nodes of
        // the tree have been traversed
        if (nc == 0)
            break;

        // maximum element for the current
```

```
// level
max = INT_MIN;

while (nc--) {

    // get the front element from 'q'
    Node* front = q.front();

    // remove front element from 'q'
    q.pop();

    // if true, then update 'max'
    if (max < front->data)
        max = front->data;

    // if left child exists
    if (front->left)
        q.push(front->left);

    // if right child exists
    if (front->right)
        q.push(front->right);
}

// print maximum element of
// current level
cout << max << " ";

}

// Driver program to test above
int main()
{
    /* Construct a Binary Tree
       4
      / \
     9   2
    / \   \
   3   5   7 */

    Node* root = NULL;
    root = newNode(4);
    root->left = newNode(9);
    root->right = newNode(2);
    root->left->left = newNode(3);
    root->left->right = newNode(5);
    root->right->right = newNode(7);
}
```

```
    largestValueInEachLevel(root);  
  
    return 0;  
}
```

Output:

4 9 7

Time Complexity: O(n).

Auxiliary Space: O(n).

### Source

<https://www.geeksforgeeks.org/largest-value-level-binary-tree-set-2-iterative-approach/>

## Chapter 245

# Leaf nodes from Preorder of a Binary Search Tree (Using Recursion)

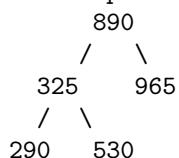
Leaf nodes from Preorder of a Binary Search Tree (Using Recursion) - GeeksforGeeks

Given Preorder traversal of a Binary Search Tree. Then the task is print leaf nodes of the Binary Search Tree from the given preorder.

**Examples :**

```
Input : preorder[] = {890, 325, 290, 530, 965};  
Output : 290 530 965
```

Tree represented is,



```
Input : preorder[] = { 3, 2, 4 };  
Output : 2 4
```

In this post, a simple recursive solution is discussed. The idea is to use two min and max variables and taking i (index in input array), the index for given preorder array, and recursively creating root node and correspondingly checking if left and right are existing or not. This method return boolean variable, and if both left and right are false it simply means that left and right are null hence it must be a leaf node so print it right there and return back true as root at that index existed.

### C++

```
// Recursive C++ program to find leaf
// nodes from given preorder traversal
#include<bits/stdc++.h>
using namespace std;

// Print the leaf node from
// the given preorder of BST.
bool isLeaf(int pre[], int &i, int n,
            int min, int max)
{
    if (i >= n)
        return false;

    if (pre[i] > min && pre[i] < max) {
        i++;

        bool left = isLeaf(pre, i, n, min, pre[i-1]);
        bool right = isLeaf(pre, i, n, pre[i-1], max);

        if (!left && !right)
            cout << pre[i-1] << " ";

        return true;
    }
    return false;
}

void printLeaves(int preorder[], int n)
{
    int i = 0;
    isLeaf(preorder, i, n, INT_MIN, INT_MAX);
}

// Driver code
int main()
{
    int preorder[] = { 890, 325, 290, 530, 965 };
    int n = sizeof(preorder)/sizeof(preorder[0]);
    printLeaves(preorder, n);
    return 0;
}
```

### PHP

```
<?php
```

```
// Recursive PHP program to
// find leaf nodes from given
// preorder traversal

// Print the leaf node from
// the given preorder of BST.

function isLeaf($pre, &$i, $n,
               $min, $max)
{
    if ($i >= $n)
        return false;

    if ($pre[$i] > $min &&
        $pre[$i] < $max)
    {
        $i++;

        $left = isLeaf($pre, $i, $n,
                      $min, $pre[$i - 1]);
        $right = isLeaf($pre, $i, $n,
                      $pre[$i - 1], $max);

        if (!$left && !$right)
            echo $pre[$i - 1] , " ";

        return true;
    }
    return false;
}

function printLeaves($preorder, $n)
{
    $i = 0;
    isLeaf($preorder, $i, $n,
           PHP_INT_MIN, PHP_INT_MAX);
}

// Driver code
$preorder = array (890, 325, 290,
                  530, 965 );
$n = sizeof($preorder);
printLeaves($preorder, $n);

// This code is contributed by ajit
?>
```

**Output :**

290 530 965

**Improved By :** [jit\\_t](#)

### Source

<https://www.geeksforgeeks.org/leaf-nodes-preorder-binary-search-treeusing-recursion/>

## Chapter 246

# Left-Child Right-Sibling Representation of Tree

Left-Child Right-Sibling Representation of Tree - GeeksforGeeks

An n-ary tree in computer science is a collection of nodes normally represented hierarchically in the following fashion.

1. The tree starts at the root node.
2. Each node of the tree holds a list of references to its child nodes.
3. The number of children a node has is less than or equal to n.

A typical representation of n-ary tree uses an array of n references (or pointers) to store children (Note that n is an upper bound on number of children). Can we do better? the idea of Left-Child Right- Sibling representation is to store only two pointers in every node.

### Left-Child Right Sibling Representation

It is a different representation of an n-ary tree where instead of holding a reference to each and every child node, a node holds just two references, first a reference to it's first child, and the other to it's immediate next sibling. This new transformation not only removes the need of advance knowledge of the number of children a node has, but also limits the number of references to a maximum of two, thereby making it so much easier to code. One thing to note is that in the previous representation a link between two nodes denoted a parent-child relationship whereas in this representation a link between two nodes may denote a parent-child relationship or a sibling-sibling relationship.

#### Advantages :

1. This representation saves up memory by limiting the maximum number of references required per node to two.
2. It is easier to code.

#### Disadvantages :

1. Basic operations like searching insertion/deletion tend to take a longer time because in

order to find the appropriate position we would have to traverse through all the siblings of the node to be searched/inserted/deleted (in the worst case).

The image on the left is the normal representation of a 6-ary tree and the one on the right is its corresponding Left-Child Right-Sibling representation.

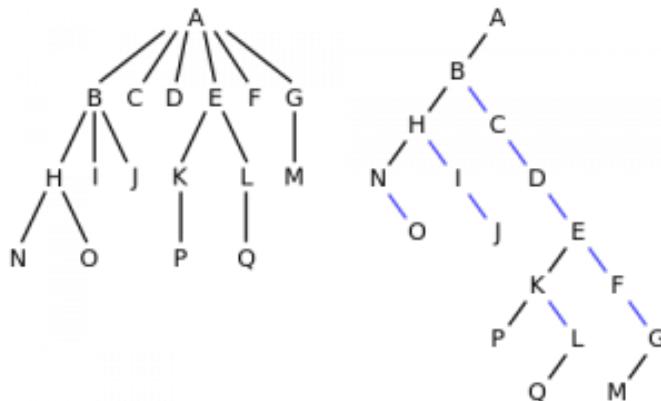


Image source : [https://en.wikipedia.org/wiki/Left-child\\_right-sibling\\_binary\\_tree](https://en.wikipedia.org/wiki/Left-child_right-sibling_binary_tree)

#### An Example Problem :

Now let's see a problem and try to solve it using both the discussed representations for clarity.

Given a family tree. Find the  $k$ th child of some member X in the tree.

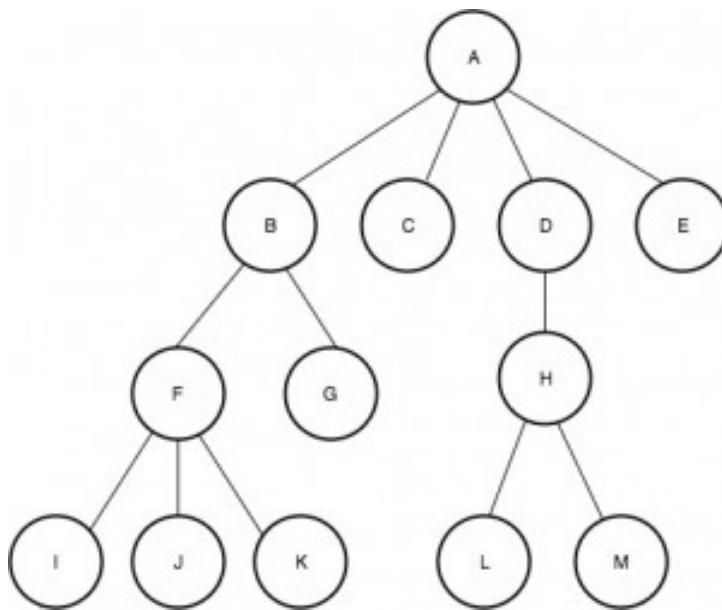
The user inputs two things.

1. A character P (representing the parent whose child is to be found)
2. An integer k (representing the child number)

The problem in itself looks pretty easy. The only issue here is that the maximum number of children a node can have is unspecified which makes it rather tricky to construct the tree.

Example:

Consider the following family tree.



Input : A 2

Output : C

In this case, the user wishes to know A's second child which according to the figure is C.

Input : F 3

Output : K

Similar to the first case, the user wishes to know F's third child which is K.

#### Method 1 (Storing n pointers with every node):

In this method, we assume the maximum number of children a node can have and proceed further. The only (obvious) problem with this method is the upper bound on the number of children. If the value is too low, then the code would fail for certain cases and if the value is too high, then a huge amount of memory is unnecessarily wasted.

If the programmer beforehand knows the structure of the tree, then the upper bound can be set to the maximum number of children a node has in that particular structure. But even in that case, there will be some memory wastage (all nodes may not necessarily have the same number of children, some may even have less. **Example: Leaf nodes have no children** ).

```

// C++ program to find k-th child of a given
// node using typical representation that uses
// an array of pointers.
#include <iostream>
  
```

```
using namespace std;

// Maximum number of children
const int N = 10;

class Node
{
public:
    char val;
    Node * child[N];
    Node(char P)
    {
        val = P;
        for (int i=0; i<MAX; i++)
            child[i] = NULL;
    }
};

// Traverses given n-ary tree to find K-th
// child of P.
void printKthChild(Node *root, char P, int k)
{
    // If P is current root
    if (root->val == P)
    {
        if (root->child[k-1] == NULL)
            cout << "Error : Does not exist\n";
        else
            cout << root->child[k-1]->val << endl;
    }

    // If P lies in a subtree
    for (int i=0; i<N; i++)
        if (root->child[i] != NULL)
            printKthChild(root->child[i], P, k);
}

// Driver code
int main()
{
    Node *root = new Node('A');
    root->child[0] = new Node('B');
    root->child[1] = new Node('C');
    root->child[2] = new Node('D');
    root->child[3] = new Node('E');
    root->child[0]->child[0] = new Node('F');
    root->child[0]->child[1] = new Node('G');
    root->child[2]->child[0] = new Node('H');
```

```

root->child[0]->child[0]->child[0] = new Node('I');
root->child[0]->child[0]->child[1] = new Node('J');
root->child[0]->child[0]->child[2] = new Node('K');
root->child[2]->child[0]->child[0] = new Node('L');
root->child[2]->child[0]->child[1] = new Node('M');

// Print F's 2nd child
char P = 'F';
cout << "F's second child is : ";
printKthChild(root, P, 2);

P = 'A';
cout << "A's seventh child is : ";
printKthChild(root, P, 7);
return 0;
}

```

Output:

```

F's second child is : J
A's seventh child is : Error : Does not exist

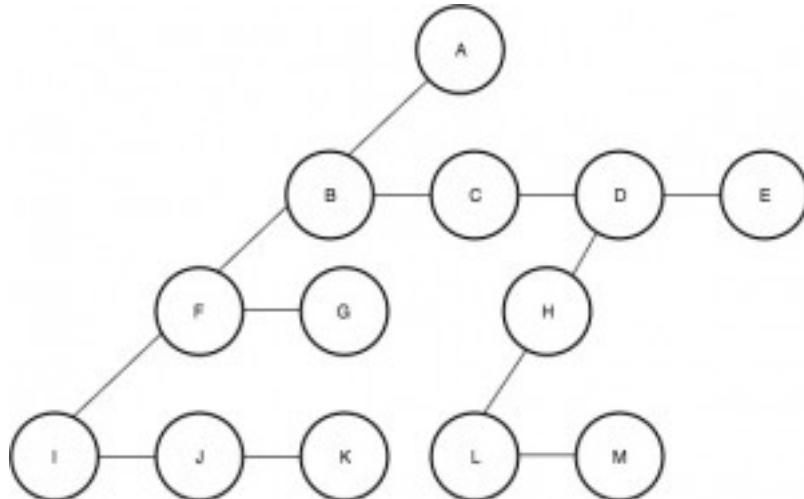
```

In the above tree, had there been a node which had say, 15 children, then this code would have given a Segmentation fault.

### Method 2 : (Left-Child Right-Sibling Representation)

In this method, we change the structure of the family tree. In the standard tree, each parent node is connected to all of its children. Here as discussed above, instead of having each node store pointers to all of its children, a node will store pointer to just one of its child. Apart from this the node will also store a pointer to its immediate right sibling.

The image below is the Left-Child Right-Sibling equivalent of the example used above.



```
// C++ program to find k-th child of a given
// Node using typical representation that uses
// an array of pointers.
#include <iostream>
using namespace std;

// A Node to represent left child right sibling
// representation.
class Node
{
public:
    char val;
    Node *child;
    Node *next;
    Node(char P)
    {
        val = P;
        child = NULL;
        next = NULL;
    }
};

// Traverses given n-ary tree to find K-th
// child of P.
void printKthChild(Node *root, char P, int k)
{
    if (root == NULL)
        return;

    // If P is present at root itself
    if (root->val == P)
    {
        // Traverse children of root starting
        // from left child
        Node *t = root->child;
        int i = 1;
        while (t != NULL && i < k)
        {
            t = t->next;
            i++;
        }
        if (t == NULL)
            cout << "Error : Does not exist\n";
        else
            cout << t->val << " " << endl;
        return;
    }
}
```

```
printKthChild(root->child, P, k);
printKthChild(root->next, P, k);
}

// Driver code
int main()
{
    Node *root = new Node('A');
    root->child = new Node('B');
    root->child->next = new Node('C');
    root->child->next->next = new Node('D');
    root->child->next->next->next = new Node('E');
    root->child->child = new Node('F');
    root->child->child->next = new Node('G');
    root->child->next->next->child = new Node('H');
    root->child->next->next->child->child = new Node('L');
    root->child->next->next->child->child->next = new Node('M');
    root->child->child->child = new Node('I');
    root->child->child->child->next = new Node('J');
    root->child->child->child->next->next = new Node('K');

    // Print F's 2nd child
    char P = 'F';
    cout << "F's second child is : ";
    printKthChild(root, P, 2);

    P = 'A';
    cout << "A's seventh child is : ";
    printKthChild(root, P, 7);
    return 0;
}
```

Output:

```
F's second child is : J
A's seventh child is : Error : Does not exist
```

**Related Article :**

[Creating a tree with Left-Child Right-Sibling Representation](#)

**Source**

<https://www.geeksforgeeks.org/left-child-right-sibling-representation-tree/>

## Chapter 247

# Leftist Tree / Leftist Heap

Leftist Tree / Leftist Heap - GeeksforGeeks

A leftist tree or leftist heap is a priority queue implemented with a variant of a binary heap. Every node has an **s-value (or rank or distance)** which is the distance to the nearest leaf. In contrast to a binary heap (Which is always a [complete binary tree](#)), a leftist tree may be very unbalanced.

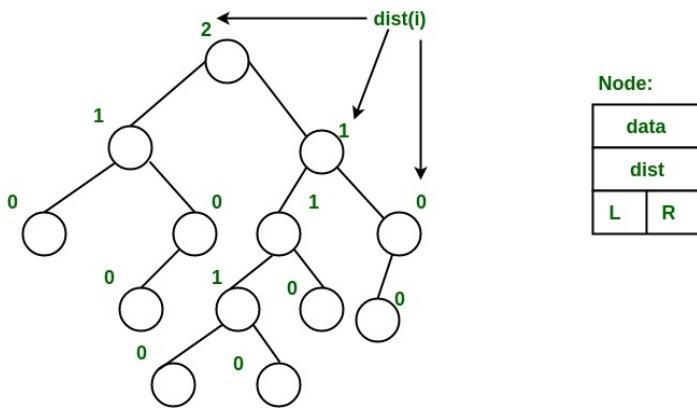
Below are [time complexities](#) of **Leftist Tree / Heap**.

Function	Complexity	Comparison
1) Get Min:	$O(1)$	[same as both Binary and Binomial]
2) Delete Min:	$O(\log n)$	[same as both Binary and Binomial]
3) Insert:	$O(\log n)$	$[O(\log n) \text{ in Binary and } O(1) \text{ in Binomial and } O(\log n) \text{ for worst case}]$
4) Merge:	$O(\log n)$	$[O(\log n) \text{ in Binomial}]$

A leftist tree is a binary tree with properties:

1. **Normal Min Heap Property :**  $\text{key}(i) \geq \text{key}(\text{parent}(i))$
2. **Heavier on left side :**  $\text{dist}(\text{right}(i)) \leq \text{dist}(\text{left}(i))$ . Here,  $\text{dist}(i)$  is the number of edges on the shortest path from node  $i$  to a leaf node in extended binary tree representation (In this representation, a null child is considered as external or leaf node). The shortest path to a descendant external node is through the right child. Every subtree is also a leftist tree and  $\text{dist}(i) = 1 + \text{dist}(\text{right}(i))$ .

**Example:** The below leftist tree is presented with its distance calculated for each node with the procedure mentioned above. The rightmost node has a rank of 0 as the right subtree of this node is null and its parent has a distance of 1 by  $\text{dist}(i) = 1 + \text{dist}(\text{right}(i))$ . The same is followed for each node and their s-value( or rank) is calculated.



From above second property, we can draw two conclusions :

1. The path from root to rightmost leaf is the shortest path from root to a leaf.
2. If the path to rightmost leaf has  $x$  nodes, then leftist heap has atleast  $2^x - 1$  nodes.  
This means the length of path to rightmost leaf is  $O(\log n)$  for a leftist heap with  $n$  nodes.

### Operations :

1. The main operation is merge().
2. deleteMin() (or extractMin()) can be done by removing root and calling merge() for left and right subtrees.
3. insert() can be done be create a leftist tree with single key (key to be inserted) and calling merge() for given tree and tree with single node.

### Idea behind Merging :

Since right subtree is smaller, the idea is to merge right subtree of a tree with other tree.  
Below are abstract steps.

1. Put the root with smaller value as the new root.
2. Hang its left subtree on the left.
3. Recursively merge its right subtree and the other tree.
4. Before returning from recursion:
  - Update dist() of merged root.
  - Swap left and right subtrees just below root, if needed, to keep leftist property of merged result

Source : <http://courses.cs.washington.edu/courses/cse326/08sp/lectures/05-leftist-heaps.pdf>

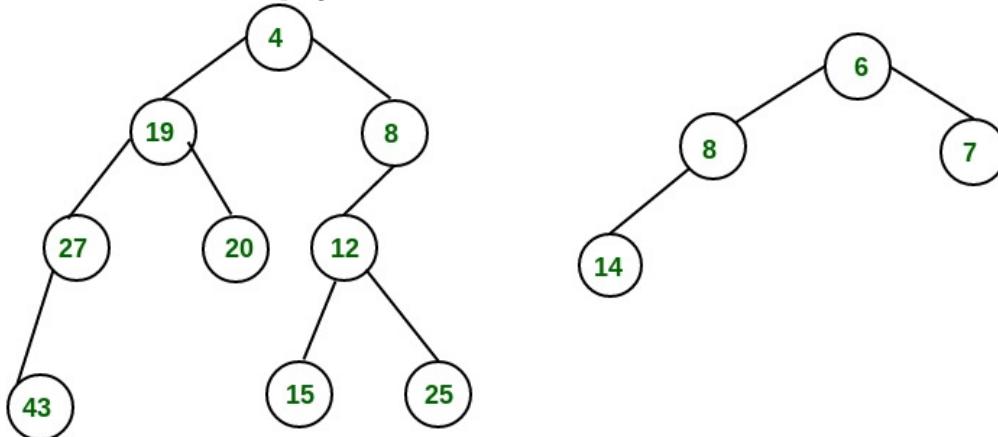
### Detailed Steps for Merge:

1. Compare the roots of two heaps.

2. Push the smaller key into an empty stack, and move to the right child of smaller key.
3. Recursively compare two keys and go on pushing the smaller key onto the stack and move to its right child.
4. Repeat until a null node is reached.
5. Take the last node processed and make it the right child of the node at top of the stack, and convert it to leftist heap if the properties of leftist heap are violated.
6. Recursively go on popping the elements from the stack and making them the right child of new stack top.

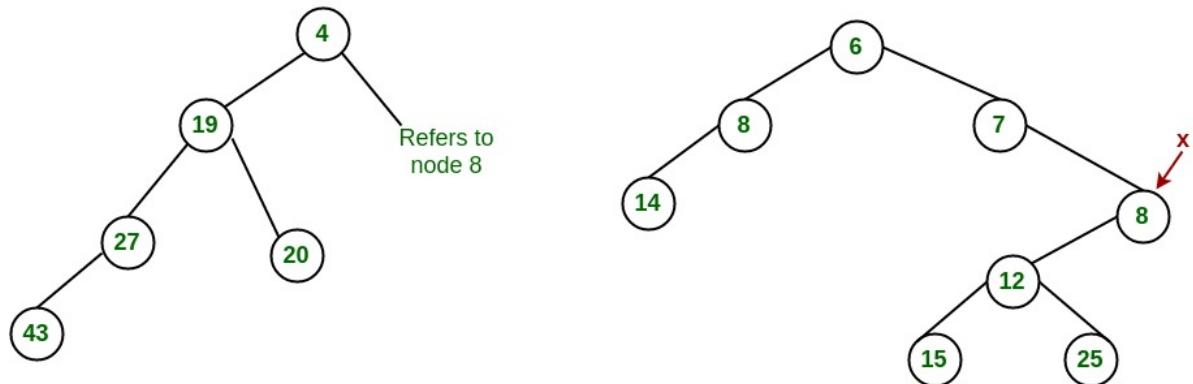
**Example:**

Consider two leftist heaps given below:



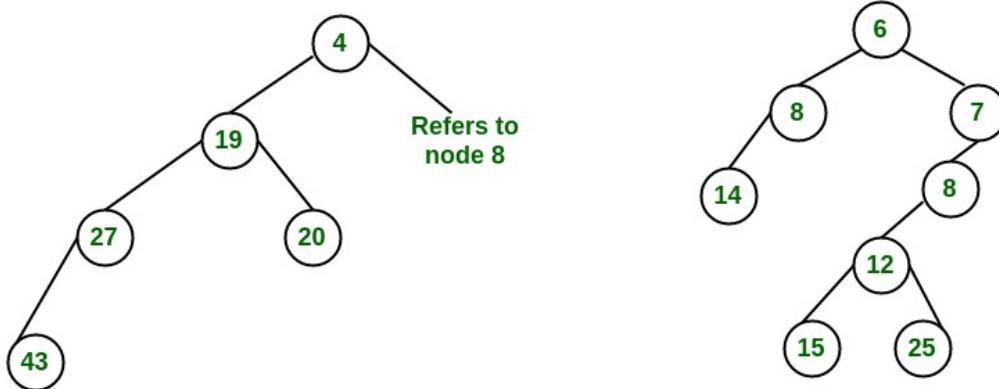
Merge them into a single leftist heap

Compare(4,6)  
 Push 4  
 Compare(8,6)  
 Push 6  
 Compare(8,7)  
 Push 7  
 Compare(8,null)  
 As null is encountered, we make node 8 as right sub-tree of stack top, i.e. 7

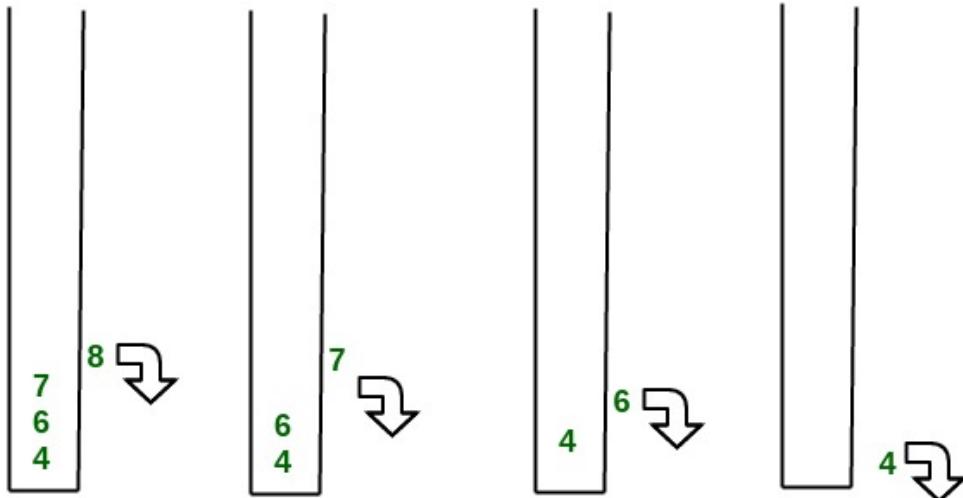


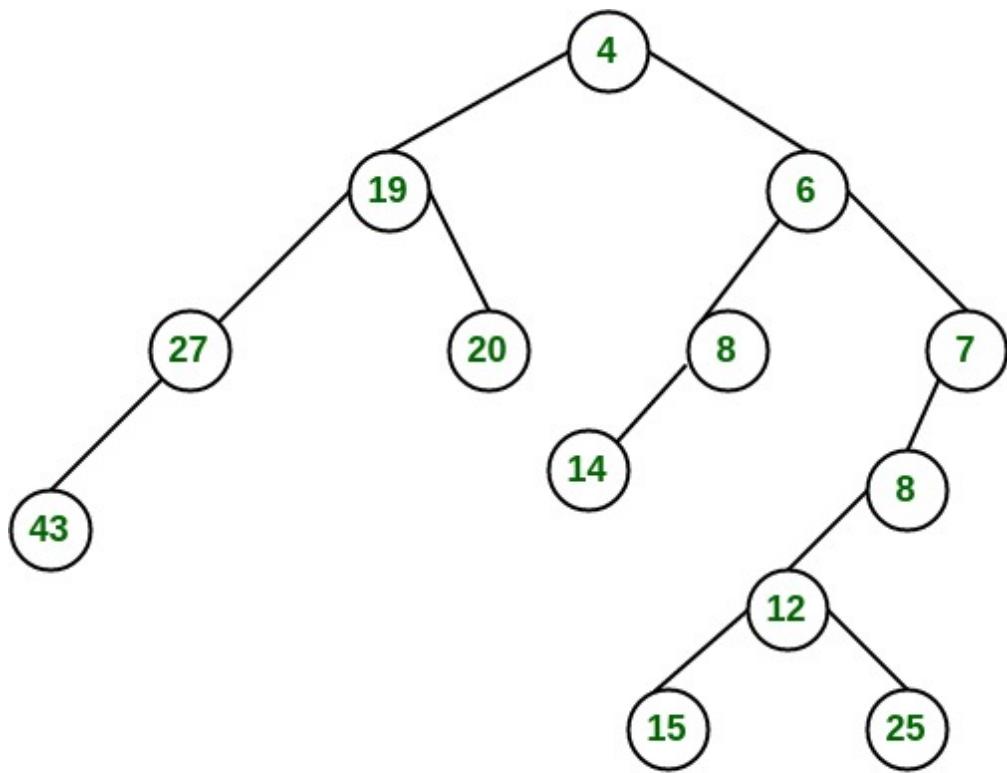
The subtree at node 7 violates the property of leftist heap so we swap it with the left child

and retain the property of leftist heap.



Convert to leftist heap. Repeat the process

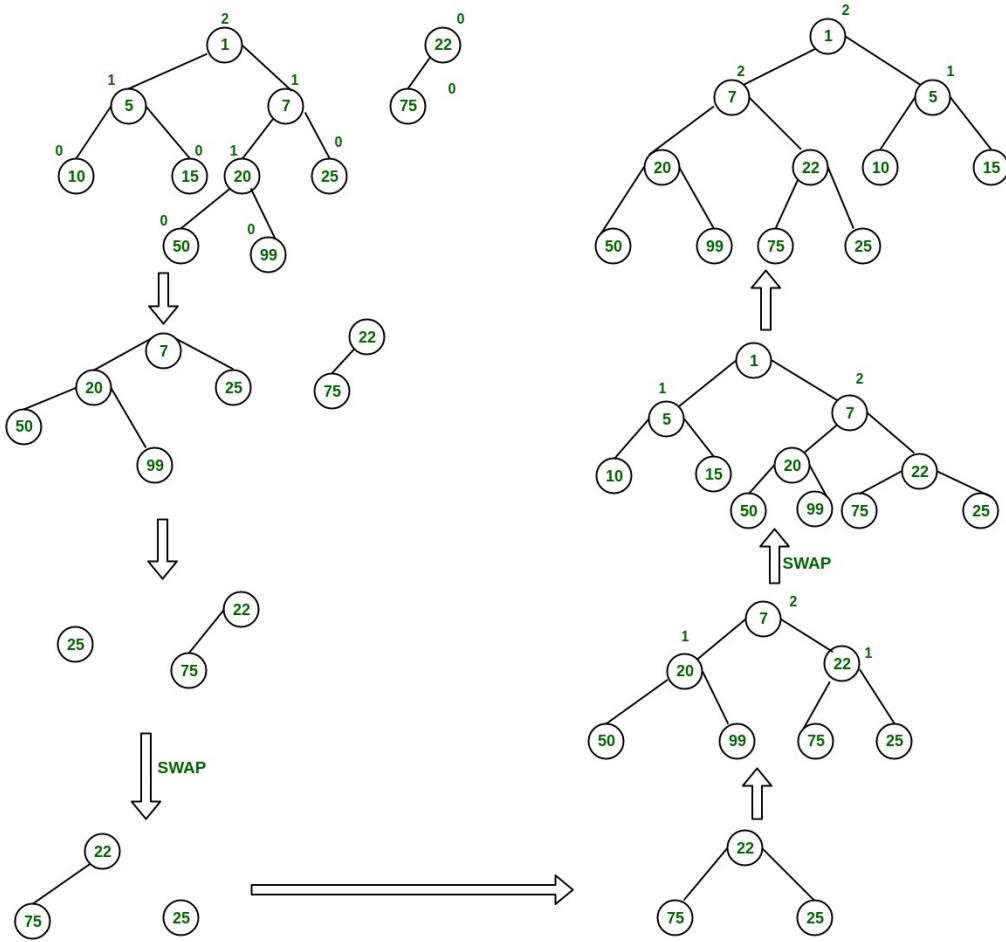




## Final leftist heap

The worst case time complexity of this algorithm is  $O(\log n)$  in the worst case, where  $n$  is the number of nodes in the leftist heap.

Another example of merging two leftist heap:



### Implementation of leftist Tree / leftist Heap:

```

//C++ program for leftist heap / leftist tree
#include <iostream>
#include <cstdlib>
using namespace std;

// Node Class Declaration
class LeftistNode
{
public:
    int element;
    LeftistNode *left;
    LeftistNode *right;
    int dist;
    LeftistNode(int & element, LeftistNode *lt = NULL,
               LeftistNode *rt = NULL, int np = 0)
    {

```

```
        this->element = element;
        right = rt;
        left = lt,
        dist = np;
    }
};

//Class Declaration
class LeftistHeap
{
public:
    LeftistHeap();
    LeftistHeap(LeftistHeap &rhs);
    ~LeftistHeap();
    bool isEmpty();
    bool isFull();
    int &findMin();
    void Insert(int &x);
    void deleteMin();
    void deleteMin(int &minItem);
    void makeEmpty();
    void Merge(LeftistHeap &rhs);
    LeftistHeap & operator =(LeftistHeap &rhs);
private:
    LeftistNode *root;
    LeftistNode *Merge(LeftistNode *h1,
                      LeftistNode *h2);
    LeftistNode *Merge1(LeftistNode *h1,
                      LeftistNode *h2);
    void swapChildren(LeftistNode * t);
    void reclaimMemory(LeftistNode * t);
    LeftistNode *clone(LeftistNode *t);
};

// Construct the leftist heap
LeftistHeap::LeftistHeap()
{
    root = NULL;
}

// Copy constructor.
LeftistHeap::LeftistHeap(LeftistHeap &rhs)
{
    root = NULL;
    *this = rhs;
}

// Destruct the leftist heap
```

```

LeftistHeap::~LeftistHeap()
{
    makeEmpty( );
}

/* Merge rhs into the priority queue.
rhs becomes empty. rhs must be different
from this.*/
void LeftistHeap::Merge(LeftistHeap &rhs)
{
    if (this == &rhs)
        return;
    root = Merge(root, rhs.root);
    rhs.root = NULL;
}

/* Internal method to merge two roots.
Deals with deviant cases and calls recursive Merge1.*/
LeftistNode *LeftistHeap::Merge(LeftistNode * h1,
                               LeftistNode * h2)
{
    if (h1 == NULL)
        return h2;
    if (h2 == NULL)
        return h1;
    if (h1->element < h2->element)
        return Merge1(h1, h2);
    else
        return Merge1(h2, h1);
}

/* Internal method to merge two roots.
Assumes trees are not empty, and h1's root contains
smallest item.*/
LeftistNode *LeftistHeap::Merge1(LeftistNode * h1,
                                LeftistNode * h2)
{
    if (h1->left == NULL)
        h1->left = h2;
    else
    {
        h1->right = Merge(h1->right, h2);
        if (h1->left->dist < h1->right->dist)
            swapChildren(h1);
        h1->dist = h1->right->dist + 1;
    }
    return h1;
}

```

```

// Swaps t's two children.
void LeftistHeap::swapChildren(LeftistNode * t)
{
    LeftistNode *tmp = t->left;
    t->left = t->right;
    t->right = tmp;
}

/* Insert item x into the priority queue, maintaining
   heap order.*/
void LeftistHeap::Insert(int &x)
{
    root = Merge(new LeftistNode(x), root);
}

/* Find the smallest item in the priority queue.
   Return the smallest item, or throw Underflow if empty.*/
int &LeftistHeap::findMin()
{
    return root->element;
}

/* Remove the smallest item from the priority queue.
   Throws Underflow if empty.*/
void LeftistHeap::deleteMin()
{
    LeftistNode *oldRoot = root;
    root = Merge(root->left, root->right);
    delete oldRoot;
}

/* Remove the smallest item from the priority queue.
   Pass back the smallest item, or throw Underflow if empty.*/
void LeftistHeap::deleteMin(int &minItem)
{
    if (isEmpty())
    {
        cout<<"Heap is Empty"<<endl;
        return;
    }
    minItem = findMin();
    deleteMin();
}

/* Test if the priority queue is logically empty.
   Returns true if empty, false otherwise*/
bool LeftistHeap::isEmpty()

```

```

{
    return root == NULL;
}

/* Test if the priority queue is logically full.
   Returns false in this implementation.*/
bool LeftistHeap::isFull()
{
    return false;
}

// Make the priority queue logically empty
void LeftistHeap::makeEmpty()
{
    reclaimMemory(root);
    root = NULL;
}

// Deep copy
LeftistHeap &LeftistHeap::operator =(LeftistHeap & rhs)
{
    if (this != &rhs)
    {
        makeEmpty();
        root = clone(rhs.root);
    }
    return *this;
}

// Internal method to make the tree empty.
void LeftistHeap::reclaimMemory(LeftistNode * t)
{
    if (t != NULL)
    {
        reclaimMemory(t->left);
        reclaimMemory(t->right);
        delete t;
    }
}

// Internal method to clone subtree.
LeftistNode *LeftistHeap::clone(LeftistNode * t)
{
    if (t == NULL)
        return NULL;
    else
        return new LeftistNode(t->element, clone(t->left),
                               clone(t->right), t->dist);
}

```

```
}

//Driver program
int main()
{
    LeftistHeap h;
    LeftistHeap h1;
    LeftistHeap h2;
    int x;
    int arr[] = {1, 5, 7, 10, 15};
    int arr1[] = {22, 75};

    h.Insert(arr[0]);
    h.Insert(arr[1]);
    h.Insert(arr[2]);
    h.Insert(arr[3]);
    h.Insert(arr[4]);
    h1.Insert(arr1[0]);
    h1.Insert(arr1[1]);

    h.deleteMin(x);
    cout << x << endl;

    h1.deleteMin(x);
    cout << x << endl;

    h.Merge(h1);
    h2 = h;

    h2.deleteMin(x);
    cout << x << endl;

    return 0;
}
```

**Output:**

```
1
22
5
```

**References:**

[Wikipedia- Leftist Tree](#)  
[CSC378: Leftist Trees](#)

**Source**

<https://www.geeksforgeeks.org/leftist-tree-leftist-heap/>

## Chapter 248

# Level Ancestor Problem

Level Ancestor Problem - GeeksforGeeks

The [level ancestor problem](#) is the problem of preprocessing a given rooted tree  $T$  into a data structure that can determine the ancestor of a given node at a given depth from the root of the tree. Here **depth** of any node in a tree is the number of edges on the shortest path from the root of the tree to the node.

Given tree is represented as **un-directed connected graph** having  $n$  nodes and  **$n-1$  edges**.

The idea to solve the above query is to use **Jump Pointer Algorithm** and pre-processes the tree in  $O(n \log n)$  time and answer level ancestor queries in  $O(\log n)$  time. In jump pointer, there is a pointer from node  $N$  to  $N$ 's  $j$ -th ancestor, for  $j = 1, 2, 4, 8, \dots$ , and so on. We refer to these pointers as  $\text{Jump}_N[i]$ , where  $\text{Jump}_u[i] = \text{LA}(N, \text{depth}(N) - 2^i)$ .

When the algorithm is asked to process a query, we repeatedly jump up the tree using these jump pointers. The number of jumps will be at most  $\log n$  and therefore queries can be answered in  $O(\log n)$  time.

So we store  **$2^i$ th ancestor** of each node and also find the depth of each node from the root of the tree.

Now our task reduces to find the  $(\text{depth}(N) - 2^i)$ th ancestor of node  $N$ . Let's denote  $X$  as  $(\text{depth}(N) - 2^i)$  and let  $b$  bits of the  $X$  are **set bits** (1) denoted by  $s_1, s_2, s_3, \dots, s_b$ .  
$$X = 2^{(s_1)} + 2^{(s_2)} + \dots + 2^{(s_b)}$$

Now the problem is how to find  $2^j$  ancestors of each node and depth of each node from the root of the tree?

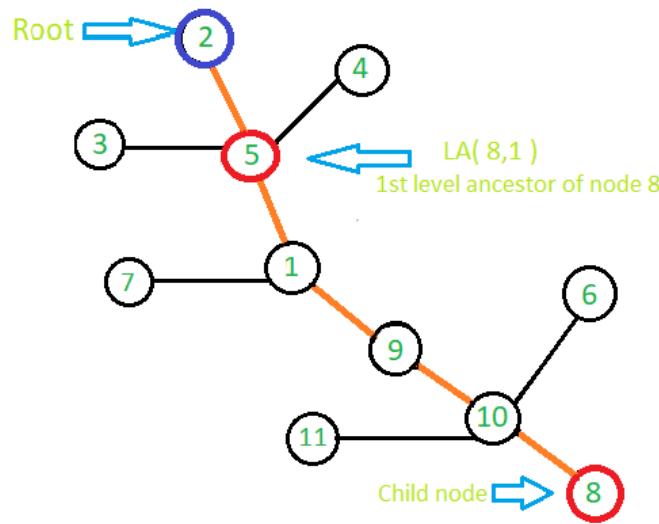
Initially, we know the  $2^0$ th ancestor of each node is its parent. We can recursively compute  $2^j$ -th ancestor. We know  $2^j$ -th ancestor is  $2^{j-1}$ -th ancestor of  $2^{j-1}$ -th ancestor. To calculate the depth of each node we use the ancestor matrix. If we found the root node present in the array of the  $k$ th element at  $j$ th index then the depth of that node is simply  $2^j$  but if root node doesn't present in the array of ancestors of the  $k$ th element than the depth of  $k$ th element is  $2^{(\text{index of last non zero ancestor at } k\text{th row})} + \text{depth of ancestor present at last index of } k\text{th row}$ .

Below is the algorithm to fill the ancestor matrix and depth of each node using dynamic programming. Here, we denote root node as **R** and initially assume the ancestor of root node as **0**. We also initialize depth array with **-1** means the depth of the current node is not set and we need to find its depth. If the depth of the current node is not equal to **-1** means we have already computed its depth.

```
we know the first ancestor of each node so we take j>=1,
For j>=1
```

```
ancstr[k][j] = 2jth ancestor of k
              = 2j-1th ancestor of (2j-1th ancestor of k)
              = ancstr[ancstr[i][j-1][j-1]
                if ancstr[k][j] == R && depth[k] == -1
                  depth[k] = 2j
                else if ancstr[k][j] == -1 && depth[k] == -1
                  depth[k] = 2(j-1) + depth[ ancstr[k][j-1] ]
```

Let's understand this algorithm with below diagram.



In the given figure we need to compute 1st level ancestor of the node with value **8**. First, we make ancestor matrix which stores  $2^{\text{th}}$  ancestor of nodes. Now,  $2^0$  ancestor of node **8** is **10** and similarly  $2^0$  ancestor of node **10** is **9** and for node **9** it is **1** and for node **1** it is **5**. Based on the above algorithm 1st level ancestor of node **8** is  $(\text{depth}(8)-1)$ th ancestor of node **8**. We have pre computed depth of each node and depth of **8** is **5** so we finally need to

find (5-1) = 4<sup>th</sup> ancestor of node 8 which is equal to 2<sup>1</sup>th ancestor of [2<sup>1</sup> ancestor of node 8] and 2<sup>1</sup>th ancestor of node 8 is 2<sup>0</sup>th ancestor of [2<sup>0</sup>th ancestor of node 8]. So, 2<sup>0</sup>th ancestor of [2<sup>0</sup>th ancestor of node 8] is node with value 9 and 2<sup>1</sup>th ancestor of node 9 is node with value 5. Thus in this way we can compute all query in O(logn) time complexity.

```

// CPP program to implement Level Ancestor Algorithm
#include <bits/stdc++.h>
using namespace std;
int R = 0;

// n -> it represent total number of nodes
// len -> it is the maximum length of array to hold
//          ancestor of each node. In worst case,
// the highest value of ancestor a node can have is n-1.
// 2 ^ len <= n-1
// len = O(log2n)
int getLen(int n)
{
    return (int)(log(n) / log(2)) + 1;
}

// ancstr represents 2D matrix to hold ancestor of node.
// Here we pass reference of 2D matrix so that the change
// made occur directly to the original matrix
// depth[] stores depth of each node
// len is same as defined above
// n is total nodes in graph
// R represent root node
void setancestor(vector<vector<int> >& ancstr,
                 vector<int>& depth, int* node, int len, int n)
{
    // depth of root node is set to 0
    depth[R] = 0;

    // if depth of a node is -1 it means its depth
    // is not set otherwise we have computed its depth
    for (int j = 1; j <= len; j++) {
        for (int i = 0; i < n; i++) {
            ancstr[node[i]][j] = ancstr[ancstr[node[i]][j - 1]][j - 1];

            // if ancestor of current node is R its height is
            // previously not set, then its height is 2^j
            if (ancstr[node[i]][j] == R && depth[node[i]] == -1) {

                // set the depth of ith node
                depth[node[i]] = pow(2, j);
            }
        }
    }
}

```

```

        // if ancestor of current node is 0 means it
        // does not have root node at its 2th power
        // on its path so its depth is 2^(index of
        // last non zero ancestor means j-1) + depth
        // of 2^(j-1) th ancestor
        else if (ancstr[node[i]][j] == 0 &&
            node[i] != R && depth[node[i]] == -1) {
            depth[node[i]] = pow(2, j - 1) +
                depth[ancstr[node[i]][j - 1]];
        }
    }
}

// c -> it represent child
// p -> it represent ancestor
// i -> it represent node number
// p=0 means the node is root node
// R represent root node
// here also we pass reference of 2D matrix and depth
// vector so that the change made occur directly to
// the original matrix and original vector
void constructGraph(vector<vector<int> &> ancstr,
                    int* node, vector<int>& depth, int* isNode,
                    int c, int p, int i)
{
    // enter the node in node array
    // it stores all the nodes in the graph
    node[i] = c;

    // to confirm that no child node have 2 ancestors
    if (isNode == 0) {
        isNode = 1;

        // make ancestor of x as y
        ancstr[0] = p;

        // if its first ancestor is root than its depth is 1
        if (R == p) {
            depth = 1;
        }
    }
    return;
}

// this function will delete leaf node
// x is node to be deleted

```

```

void removeNode(vector<vector<int> >& ancstr,
                int* isNode, int len, int x)
{
    if (isNode[x] == 0)
        cout << "node does not present in graph " << endl;
    else {
        isNode[x] = 0;

        // make all ancestor of node x as 0
        for (int j = 0; j <= len; j++) {
            ancstr[x][j] = 0;
        }
    }
    return;
}

// x -> it represent new node to be inserted
// p -> it represent ancestor of new node
void addNode(vector<vector<int> >& ancstr,
             vector<int>& depth, int* isNode, int len,
             int x, int p)
{
    if (isNode[x] == 1) {
        cout << " Node is already present in array " << endl;
        return;
    }
    if (isNode[p] == 0) {
        cout << " ancestor not does not present in an array " << endl;
        return;
    }

    isNode[x] = 1;
    ancstr[x][0] = p;

    // depth of new node is 1 + depth of its ancestor
    depth[x] = depth[p] + 1;
    int j = 0;

    // while we don't reach root node
    while (ancstr[x][j] != 0) {
        ancstr[x][j + 1] = ancstr[ancstr[x][j]][j];
        j++;
    }

    // remaining array will fill with 0 after
    // we find root of tree
    while (j <= len) {
        ancstr[x][j] = 0;
    }
}

```

```

        j++;
    }
    return;
}

// LA function to find Lth level ancestor of node x
void LA(vector<vector<int> >& ancstr, vector<int> depth,
         int* isNode, int x, int L)
{
    int j = 0;
    int temp = x;

    // to check if node is present in graph or not
    if (isNode[x] == 0) {
        cout << "Node is not present in graph " << endl;
        return;
    }

    // we change L as depth of node x -
    int k = depth[x] - L;
    // int q = k;
    // in this loop we decrease the value of k by k/2 and
    // increment j by 1 after each iteration, and check for set bit
    // if we get set bit then we update x with jth ancestor of x
    // as k becomes less than or equal to zero means we
    // reach to kth level ancestor
    while (k > 0) {

        // to check if last bit is 1 or not
        if (k & 1) {
            x = ancstr[x][j];
        }

        // use of shift operator to make k = k/2
        // after every iteration
        k = k >> 1;
        j++;
    }
    cout << L << "th level ancestor of node "
        << temp << " is = " << x << endl;
}

int main()
{
    // n represent number of nodes
    int n = 12;
}

```

```

// initialization of ancestor matrix
// suppose max range of node is up to 1000
// if there are 1000 nodes than also length
// of ancestor matrix will not exceed 10
vector<vector<int> > ancestor(1000, vector<int>(10));

// this vector is used to store depth of each node.
vector<int> depth(1000);

// fill function is used to initialize depth with -1
fill(depth.begin(), depth.end(), -1);

// node array is used to store all nodes
int* node = new int[1000];

// isNode is an array to check whether a
// node is present in graph or not
int* isNode = new int[1000];

// memset function to initialize isNode array with 0
memset(isNode, 0, 1000 * sizeof(int));

// function to calculate len
// len -> it is the maximum length of array to
// hold ancestor of each node.
int len = getLen(n);

// R stores root node
R = 2;

// construction of graph
// here 0 represent that the node is root node
constructGraph(ancestor, node, depth, isNode, 2, 0, 0);
constructGraph(ancestor, node, depth, isNode, 5, 2, 1);
constructGraph(ancestor, node, depth, isNode, 3, 5, 2);
constructGraph(ancestor, node, depth, isNode, 4, 5, 3);
constructGraph(ancestor, node, depth, isNode, 1, 5, 4);
constructGraph(ancestor, node, depth, isNode, 7, 1, 5);
constructGraph(ancestor, node, depth, isNode, 9, 1, 6);
constructGraph(ancestor, node, depth, isNode, 10, 9, 7);
constructGraph(ancestor, node, depth, isNode, 11, 10, 8);
constructGraph(ancestor, node, depth, isNode, 6, 10, 9);
constructGraph(ancestor, node, depth, isNode, 8, 10, 10);

// function to pre compute ancestor matrix
setancestor(ancestor, depth, node, len, n);

```

```
// query to get 1st level ancestor of node 8
LA(ancestor, depth, isNode, 8, 1);

// add node 12 and its ancestor is 8
addNode(ancestor, depth, isNode, len, 12, 8);

// query to get 2nd level ancestor of node 12
LA(ancestor, depth, isNode, 12, 2);

// delete node 12
removeNode(ancestor, isNode, len, 12);

// query to get 5th level ancestor of node
// 12 after deletion of node
LA(ancestor, depth, isNode, 12, 1);

return 0;
}
```

**Output:**

```
1th level ancestor of node 8 is = 5
2th level ancestor of node 12 is = 1
Node is not present in graph
```

**Source**

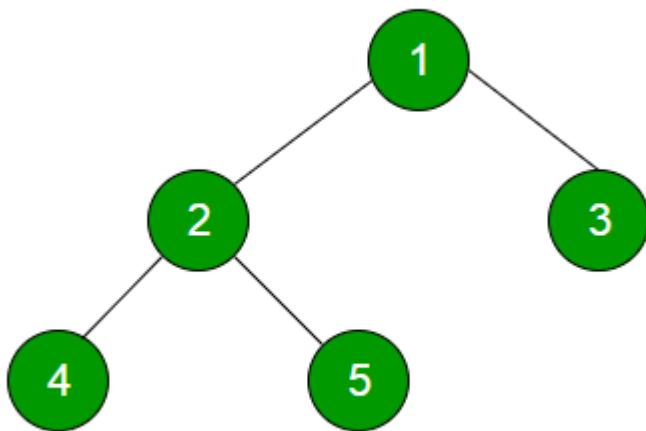
<https://www.geeksforgeeks.org/level-ancestor-problem/>

## Chapter 249

# Level Order Tree Traversal

Level Order Tree Traversal - GeeksforGeeks

Level order traversal of a tree is [breadth first traversal](#) for the tree.



Level order traversal of the above tree is 1 2 3 4 5

### METHOD 1 (Use function to print a given level)

#### Algorithm:

There are basically two functions in this method. One is to print all nodes at a given level (`printGivenLevel`), and other is to print level order traversal of the tree (`printLevelorder`). `printLevelorder` makes use of `printGivenLevel` to print nodes at all levels one by one starting from root.

```
/*Function to print level order traversal of tree*/
printLevelorder(tree)
```

```
for d = 1 to height(tree)
    printGivenLevel(tree, d);

/*Function to print all nodes at a given level*/
printGivenLevel(tree, level)
if tree is NULL then return;
if level is 1, then
    print(tree->data);
else if level greater than 1, then
    printGivenLevel(tree->left, level-1);
    printGivenLevel(tree->right, level-1);
```

**Implementation:**

C

```
// Recursive C program for level order traversal of Binary Tree
#include <stdio.h>
#include <stdlib.h>

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
    int data;
    struct node* left, *right;
};

/* Function prototypes */
void printGivenLevel(struct node* root, int level);
int height(struct node* node);
struct node* newNode(int data);

/* Function to print level order traversal a tree*/
void printLevelOrder(struct node* root)
{
    int h = height(root);
    int i;
    for (i=1; i<=h; i++)
        printGivenLevel(root, i);
}

/* Print nodes at a given level */
void printGivenLevel(struct node* root, int level)
{
    if (root == NULL)
        return;
```

```

if (level == 1)
    printf("%d ", root->data);
else if (level > 1)
{
    printGivenLevel(root->left, level-1);
    printGivenLevel(root->right, level-1);
}
}

/* Compute the "height" of a tree -- the number of
   nodes along the longest path from the root node
   down to the farthest leaf node.*/
int height(struct node* node)
{
    if (node==NULL)
        return 0;
    else
    {
        /* compute the height of each subtree */
        int lheight = height(node->left);
        int rheight = height(node->right);

        /* use the larger one */
        if (lheight > rheight)
            return(lheight+1);
        else return(rheight+1);
    }
}

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* node = (struct node*)
                           malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;

    return(node);
}

/* Driver program to test above functions*/
int main()
{
    struct node *root = newNode(1);
    root->left      = newNode(2);
    root->right     = newNode(3);
}

```

```
root->left->left = newNode(4);
root->left->right = newNode(5);

printf("Level Order traversal of binary tree is \n");
printLevelOrder(root);

return 0;
}
```

**Java**

```
// Recursive Java program for level order traversal of Binary Tree

/* Class containing left and right child of current
   node and key value*/
class Node
{
    int data;
    Node left, right;
    public Node(int item)
    {
        data = item;
        left = right = null;
    }
}

class BinaryTree
{
    // Root of the Binary Tree
    Node root;

    public BinaryTree()
    {
        root = null;
    }

    /* function to print level order traversal of tree*/
    void printLevelOrder()
    {
        int h = height(root);
        int i;
        for (i=1; i<=h; i++)
            printGivenLevel(root, i);
    }

    /* Compute the "height" of a tree -- the number of
       nodes along the longest path from the root node
       down to the farthest leaf node.*/
}
```

```
int height(Node root)
{
    if (root == null)
        return 0;
    else
    {
        /* compute height of each subtree */
        int lheight = height(root.left);
        int rheight = height(root.right);

        /* use the larger one */
        if (lheight > rheight)
            return(lheight+1);
        else return(rheight+1);
    }
}

/* Print nodes at the given level */
void printGivenLevel (Node root ,int level)
{
    if (root == null)
        return;
    if (level == 1)
        System.out.print(root.data + " ");
    else if (level > 1)
    {
        printGivenLevel(root.left, level-1);
        printGivenLevel(root.right, level-1);
    }
}

/* Driver program to test above functions */
public static void main(String args[])
{
    BinaryTree tree = new BinaryTree();
    tree.root= new Node(1);
    tree.root.left= new Node(2);
    tree.root.right= new Node(3);
    tree.root.left.left= new Node(4);
    tree.root.left.right= new Node(5);

    System.out.println("Level order traversal of binary tree is ");
    tree.printLevelOrder();
}
```

Python

```
# Recursive Python program for level order traversal of Binary Tree

# A node structure
class Node:

    # A utility function to create a new node
    def __init__(self, key):
        self.data = key
        self.left = None
        self.right = None

    # Function to print level order traversal of tree
    def printLevelOrder(root):
        h = height(root)
        for i in range(1, h+1):
            printGivenLevel(root, i)

    # Print nodes at a given level
    def printGivenLevel(root , level):
        if root is None:
            return
        if level == 1:
            print "%d" %(root.data),
        elif level > 1 :
            printGivenLevel(root.left , level-1)
            printGivenLevel(root.right , level-1)

    """ Compute the height of a tree--the number of nodes
        along the longest path from the root node down to
        the farthest leaf node
    """
    def height(node):
        if node is None:
            return 0
        else :
            # Compute the height of each subtree
            lheight = height(node.left)
            rheight = height(node.right)

            #Use the larger one
            if lheight > rheight :
                return lheight+1
            else:
                return rheight+1
```

```
# Driver program to test above function
root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)

print "Level order traversal of binary tree is -"
printLevelOrder(root)

#This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

Output:

```
Level order traversal of binary tree is -
1 2 3 4 5
```

Time Complexity:  $O(n^2)$  in worst case. For a skewed tree, printGivenLevel() takes  $O(n)$  time where  $n$  is the number of nodes in the skewed tree. So time complexity of printLevelOrder() is  $O(n) + O(n-1) + O(n-2) + \dots + O(1)$  which is  $O(n^2)$ .

## METHOD 2 (Use Queue)

### Algorithm:

For each node, first the node is visited and then it's child nodes are put in a FIFO queue.

```
printLevelorder(tree)
1) Create an empty queue q
2) temp_node = root /*start from root*/
3) Loop while temp_node is not NULL
   a) print temp_node->data.
   b) Enqueue temp_node's children (first left then right children) to q
   c) Dequeue a node from q and assign it's value to temp_node
```

### Implementation:

Here is a simple implementation of the above algorithm. Queue is implemented using an array with maximum size of 500. We can implement queue as linked list also.

## C

```
// Iterative Queue based C program to do level order traversal
// of Binary Tree
#include <stdio.h>
#include <stdlib.h>
#define MAX_Q_SIZE 500
```

```
/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

/* frunction prototypes */
struct node** createQueue(int *, int *);
void enQueue(struct node **, int *, struct node *);
struct node *deQueue(struct node **, int *);

/* Given a binary tree, print its nodes in level order
   using array for implementing queue */
void printLevelOrder(struct node* root)
{
    int rear, front;
    struct node **queue = createQueue(&front, &rear);
    struct node *temp_node = root;

    while (temp_node)
    {
        printf("%d ", temp_node->data);

        /*Enqueue left child */
        if (temp_node->left)
            enQueue(queue, &rear, temp_node->left);

        /*Enqueue right child */
        if (temp_node->right)
            enQueue(queue, &rear, temp_node->right);

        /*Dequeue node and make it temp_node*/
        temp_node = deQueue(queue, &front);
    }
}

/*UTILITY FUNCTIONS*/
struct node** createQueue(int *front, int *rear)
{
    struct node **queue =
        (struct node **)malloc(sizeof(struct node*)*MAX_Q_SIZE);

    *front = *rear = 0;
    return queue;
```

```
}

void enqueue(struct node **queue, int *rear, struct node *new_node)
{
    queue[*rear] = new_node;
    (*rear)++;
}

struct node *deQueue(struct node **queue, int *front)
{
    (*front)++;
    return queue[*front - 1];
}

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* node = (struct node*)
                           malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;

    return(node);
}

/* Driver program to test above functions*/
int main()
{
    struct node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);

    printf("Level Order traversal of binary tree is \n");
    printLevelOrder(root);

    return 0;
}
```

C++

```
/* C++ program to print level order traversal using STL */
#include <iostream>
#include <queue>
using namespace std;
```

```
// A Binary Tree Node
struct Node
{
    int data;
    struct Node *left, *right;
};

// Iterative method to find height of Binary Tree
void printLevelOrder(Node *root)
{
    // Base Case
    if (root == NULL)  return;

    // Create an empty queue for level order traversal
    queue<Node *> q;

    // Enqueue Root and initialize height
    q.push(root);

    while (q.empty() == false)
    {
        // Print front of queue and remove it from queue
        Node *node = q.front();
        cout << node->data << " ";
        q.pop();

        /* Enqueue left child */
        if (node->left != NULL)
            q.push(node->left);

        /*Enqueue right child */
        if (node->right != NULL)
            q.push(node->right);
    }
}

// Utility function to create a new tree node
Node* newNode(int data)
{
    Node *temp = new Node;
    temp->data = data;
    temp->left = temp->right = NULL;
    return temp;
}

// Driver program to test above functions
int main()
```

```
{  
    // Let us create binary tree shown in above diagram  
    Node *root = newNode(1);  
    root->left = newNode(2);  
    root->right = newNode(3);  
    root->left->left = newNode(4);  
    root->left->right = newNode(5);  
  
    cout << "Level Order traversal of binary tree is \n";  
    printLevelOrder(root);  
    return 0;  
}
```

**Java**

```
// Iterative Queue based Java program to do level order traversal  
// of Binary Tree  
  
/* importing the inbuilt java classes required for the program */  
import java.util.Queue;  
import java.util.LinkedList;  
  
/* Class to represent Tree node */  
class Node {  
    int data;  
    Node left, right;  
  
    public Node(int item) {  
        data = item;  
        left = null;  
        right = null;  
    }  
}  
  
/* Class to print Level Order Traversal */  
class BinaryTree {  
  
    Node root;  
  
    /* Given a binary tree. Print its nodes in level order  
       using array for implementing queue */  
    void printLevelOrder()  
    {  
        Queue<Node> queue = new LinkedList<Node>();  
        queue.add(root);  
        while (!queue.isEmpty())  
        {
```

```

/* poll() removes the present head.
For more information on poll() visit
http://www.tutorialspoint.com/java/util/linkedlist_poll.htm */
Node tempNode = queue.poll();
System.out.print(tempNode.data + " ");

/*Enqueue left child */
if (tempNode.left != null) {
    queue.add(tempNode.left);
}

/*Enqueue right child */
if (tempNode.right != null) {
    queue.add(tempNode.right);
}
}

public static void main(String args[])
{
    /* creating a binary tree and entering
       the nodes */
    BinaryTree tree_level = new BinaryTree();
    tree_level.root = new Node(1);
    tree_level.root.left = new Node(2);
    tree_level.root.right = new Node(3);
    tree_level.root.left.left = new Node(4);
    tree_level.root.left.right = new Node(5);

    System.out.println("Level order traversal of binary tree is - ");
    tree_level.printLevelOrder();
}
}

```

**Python**

```

# Python program to print level order traversal using Queue

# A node structure
class Node:
    # A utility function to create a new node
    def __init__(self ,key):
        self.data = key
        self.left = None
        self.right = None

    # Iterative Method to print the height of binary tree
def printLevelOrder(root):

```

```
# Base Case
if root is None:
    return

# Create an empty queue for level order traversal
queue = []

# Enqueue Root and initialize height
queue.append(root)

while(len(queue) > 0):
    # Print front of queue and remove it from queue
    print queue[0].data,
    node = queue.pop(0)

    #Enqueue left child
    if node.left is not None:
        queue.append(node.left)

    # Enqueue right child
    if node.right is not None:
        queue.append(node.right)

#Driver Program to test above function
root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)

print "Level Order Traversal of binary tree is -"
printLevelOrder(root)
#This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

Output:

```
Level order traversal of binary tree is -
1 2 3 4 5
```

**Time Complexity:** O(n) where n is number of nodes in the binary tree

**References:**

[http://en.wikipedia.org/wiki/Breadth-first\\_traversal](http://en.wikipedia.org/wiki/Breadth-first_traversal)

## Source

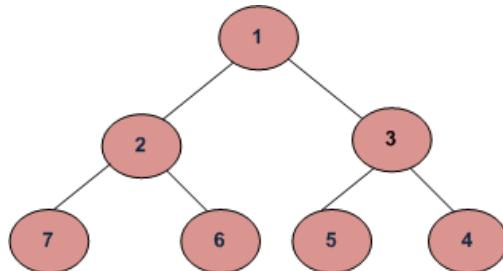
<https://www.geeksforgeeks.org/level-order-tree-traversal/>

## Chapter 250

# Level order traversal in spiral form

Level order traversal in spiral form - GeeksforGeeks

Write a function to print spiral order traversal of a tree. For below tree, function should print 1, 2, 3, 4, 5, 6, 7.



### Method 1 (Recursive)

This problem can be seen as an extension of the [level order traversal](#) post.

To print the nodes in spiral order, nodes at different levels should be printed in alternating order. An additional Boolean variable *ltr* is used to change printing order of levels. If *ltr* is 1 then `printGivenLevel()` prints nodes from left to right else from right to left. Value of *ltr* is flipped in each iteration to change the order.

Function to print level order traversal of tree

```
printSpiral(tree)
    bool ltr = 0;
    for d = 1 to height(tree)
        printGivenLevel(tree, d, ltr);
        ltr ~= ltr /*flip ltr*/
```

Function to print all nodes at a given level

```
printGivenLevel(tree, level, ltr)
if tree is NULL then return;
if level is 1, then
    print(tree->data);
else if level greater than 1, then
    if(ltr)
        printGivenLevel(tree->left, level-1, ltr);
        printGivenLevel(tree->right, level-1, ltr);
    else
        printGivenLevel(tree->right, level-1, ltr);
        printGivenLevel(tree->left, level-1, ltr);
```

Following is C implementation of above algorithm.

## C

```
// C program for recursive level order traversal in spiral form
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

/* Function prototypes */
void printGivenLevel(struct node* root, int level, int ltr);
int height(struct node* node);
struct node* newNode(int data);

/* Function to print spiral traversal of a tree*/
void printSpiral(struct node* root)
{
    int h = height(root);
    int i;

    /*ltr -> Left to Right. If this variable is set,
       then the given level is traversed from left to right. */
    bool ltr = false;
    for(i=1; i<=h; i++)
    {
```

```
printGivenLevel(root, i, ltr);

/*Revert ltr to traverse next level in opposite order*/
ltr = !ltr;
}

/*
 * Print nodes at a given level */
void printGivenLevel(struct node* root, int level, int ltr)
{
    if(root == NULL)
        return;
    if(level == 1)
        printf("%d ", root->data);
    else if (level > 1)
    {
        if(ltr)
        {
            printGivenLevel(root->left, level-1, ltr);
            printGivenLevel(root->right, level-1, ltr);
        }
        else
        {
            printGivenLevel(root->right, level-1, ltr);
            printGivenLevel(root->left, level-1, ltr);
        }
    }
}

/*
 * Compute the "height" of a tree -- the number of
 * nodes along the longest path from the root node
 * down to the farthest leaf node.*/
int height(struct node* node)
{
    if (node==NULL)
        return 0;
    else
    {
        /* compute the height of each subtree */
        int lheight = height(node->left);
        int rheight = height(node->right);

        /* use the larger one */
        if (lheight > rheight)
            return(lheight+1);
        else return(rheight+1);
    }
}
```

```
/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* node = (struct node*)
                         malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;

    return(node);
}

/* Driver program to test above functions*/
int main()
{
    struct node *root = newNode(1);
    root->left      = newNode(2);
    root->right     = newNode(3);
    root->left->left  = newNode(7);
    root->left->right = newNode(6);
    root->right->left = newNode(5);
    root->right->right = newNode(4);
    printf("Spiral Order traversal of binary tree is \n");
    printSpiral(root);

    return 0;
}
```

### Java

```
// Java program for recursive level order traversal in spiral form

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
class Node
{
    int data;
    Node left, right;

    public Node(int d)
    {
        data = d;
        left = right = null;
    }
}
```

```
class BinaryTree
{
    Node root;

    // Function to print the spiral traversal of tree
    void printSpiral(Node node)
    {
        int h = height(node);
        int i;

        /* ltr -> left to right. If this variable is set then the
           given label is transversed from left to right */
        boolean ltr = false;
        for (i = 1; i <= h; i++)
        {
            printGivenLevel(node, i, ltr);

            /*Revert ltr to traverse next level in opposite order*/
            ltr = !ltr;
        }
    }

    /* Compute the "height" of a tree -- the number of
       nodes along the longest path from the root node
       down to the farthest leaf node.*/
    int height(Node node)
    {
        if (node == null)
            return 0;
        else
        {

            /* compute the height of each subtree */
            int lheight = height(node.left);
            int rheight = height(node.right);

            /* use the larger one */
            if (lheight > rheight)
                return (lheight + 1);
            else
                return (rheight + 1);
        }
    }

    /* Print nodes at a given level */
    void printGivenLevel(Node node, int level, boolean ltr)
    {
```

```

        if (node == null)
            return;
        if (level == 1)
            System.out.print(node.data + " ");
        else if (level > 1)
        {
            if (ltr != false)
            {
                printGivenLevel(node.left, level - 1, ltr);
                printGivenLevel(node.right, level - 1, ltr);
            }
            else
            {
                printGivenLevel(node.right, level - 1, ltr);
                printGivenLevel(node.left, level - 1, ltr);
            }
        }
    }
    /* Driver program to test the above functions */
    public static void main(String[] args)
    {
        BinaryTree tree = new BinaryTree();
        tree.root = new Node(1);
        tree.root.left = new Node(2);
        tree.root.right = new Node(3);
        tree.root.left.left = new Node(7);
        tree.root.left.right = new Node(6);
        tree.root.right.left = new Node(5);
        tree.root.right.right = new Node(4);
        System.out.println("Spiral order traversal of Binary Tree is ");
        tree.printSpiral(tree.root);
    }
}

// This code has been contributed by Mayank Jaiswal(mayank_24)
```

Output:

```
Spiral Order traversal of binary tree is
1 2 3 4 5 6 7
```

**Time Complexity:** Worst case time complexity of the above method is  $O(n^2)$ . Worst case occurs in case of skewed trees.

### Method 2 (Iterative)

We can print spiral order traversal in  $O(n)$  time and  $O(n)$  extra space. The idea is to use two stacks. We can use one stack for printing from left to right and other stack for printing from right to left. In every iteration, we have nodes of one level in one of the stacks. We

print the nodes, and push nodes of next level in other stack.

C++

```
// C++ implementation of a O(n) time method for spiral order traversal
#include <iostream>
#include <stack>
using namespace std;

// Binary Tree node
struct node
{
    int data;
    struct node *left, *right;
};

void printSpiral(struct node *root)
{
    if (root == NULL)  return; // NULL check

    // Create two stacks to store alternate levels
    stack<struct node*> s1; // For levels to be printed from right to left
    stack<struct node*> s2; // For levels to be printed from left to right

    // Push first level to first stack 's1'
    s1.push(root);

    // Keep printing while any of the stacks has some nodes
    while (!s1.empty() || !s2.empty())
    {
        // Print nodes of current level from s1 and push nodes of
        // next level to s2
        while (!s1.empty())
        {
            struct node *temp = s1.top();
            s1.pop();
            cout << temp->data << " ";

            // Note that is right is pushed before left
            if (temp->right)
                s2.push(temp->right);
            if (temp->left)
                s2.push(temp->left);
        }

        // Print nodes of current level from s2 and push nodes of
        // next level to s1
        while (!s2.empty())
```

```
{  
    struct node *temp = s2.top();  
    s2.pop();  
    cout << temp->data << " ";  
  
    // Note that is left is pushed before right  
    if (temp->left)  
        s1.push(temp->left);  
    if (temp->right)  
        s1.push(temp->right);  
}  
}  
}  
  
// A utility function to create a new node  
struct node* newNode(int data)  
{  
    struct node* node = new struct node;  
    node->data = data;  
    node->left = NULL;  
    node->right = NULL;  
  
    return(node);  
}  
  
int main()  
{  
    struct node *root = newNode(1);  
    root->left = newNode(2);  
    root->right = newNode(3);  
    root->left->left = newNode(7);  
    root->left->right = newNode(6);  
    root->right->left = newNode(5);  
    root->right->right = newNode(4);  
    cout << "Spiral Order traversal of binary tree is \n";  
    printSpiral(root);  
  
    return 0;  
}
```

### Java

```
// Java implementation of an O(n) approach of level order  
// traversal in spiral form  
  
import java.util.*;  
  
// A Binary Tree node
```

```
class Node
{
    int data;
    Node left, right;

    public Node(int item)
    {
        data = item;
        left = right = null;
    }
}

class BinaryTree
{

    static Node root;

    void printSpiral(Node node)
    {
        if (node == null)
            return; // NULL check

        // Create two stacks to store alternate levels
        Stack<Node> s1 = new Stack<Node>(); // For levels to be printed from right to left
        Stack<Node> s2 = new Stack<Node>(); // For levels to be printed from left to right

        // Push first level to first stack 's1'
        s1.push(node);

        // Keep printing while any of the stacks has some nodes
        while (!s1.empty() || !s2.empty())
        {
            // Print nodes of current level from s1 and push nodes of
            // next level to s2
            while (!s1.empty())
            {
                Node temp = s1.peek();
                s1.pop();
                System.out.print(temp.data + " ");

                // Note that is right is pushed before left
                if (temp.right != null)
                    s2.push(temp.right);

                if (temp.left != null)
                    s2.push(temp.left);
            }
        }
    }
}
```

```
// Print nodes of current level from s2 and push nodes of
// next level to s1
while (!s2.empty())
{
    Node temp = s2.peek();
    s2.pop();
    System.out.print(temp.data + " ");

    // Note that is left is pushed before right
    if (temp.left != null)
        s1.push(temp.left);
    if (temp.right != null)
        s1.push(temp.right);
}
}

public static void main(String[] args)
{
    BinaryTree tree = new BinaryTree();
    tree.root = new Node(1);
    tree.root.left = new Node(2);
    tree.root.right = new Node(3);
    tree.root.left.left = new Node(7);
    tree.root.left.right = new Node(6);
    tree.root.right.left = new Node(5);
    tree.root.right.right = new Node(4);
    System.out.println("Spiral Order traversal of Binary Tree is ");
    tree.printSpiral(root);
}
}

// This code has been contributed by Mayank Jaiswal(mayank_24)
```

Output:

```
Spiral Order traversal of binary tree is
1 2 3 4 5 6 7
```

Please write comments if you find any bug in the above program/algorithm; or if you want to share more information about spiral traversal.

## Source

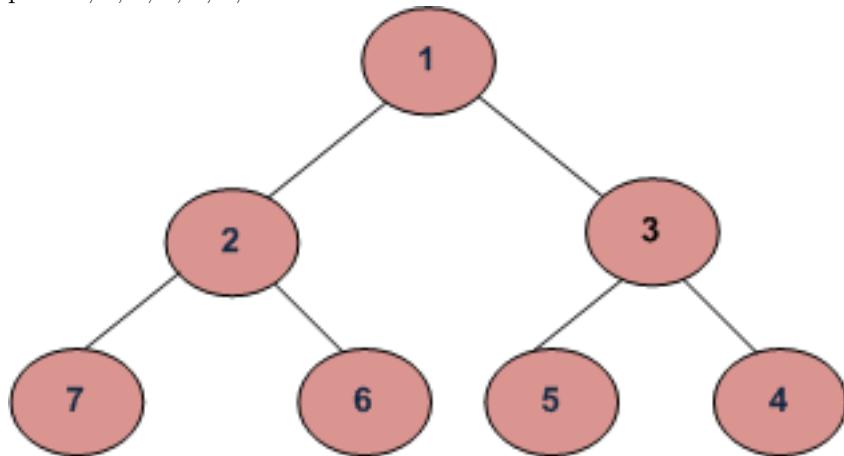
<https://www.geeksforgeeks.org/level-order-traversal-in-spiral-form/>

## Chapter 251

# Level order traversal in spiral form | Using one stack and one queue

Level order traversal in spiral form | Using one stack and one queue - GeeksforGeeks

Write a function to print spiral order traversal of a tree. For below tree, function should print 1, 2, 3, 4, 5, 6, 7.



You are allowed to use only one stack.

We have seen [recursive and iterative solutions using two stacks](#). In this post, a solution with one stack and one queue is discussed. The idea is to keep on entering nodes like normal level order traversal, but during printing, in alternative turns push them onto the stack and print them, and in other traversals, just print them the way they are present in the queue.

Following is the CPP implementation of the idea.

```
// CPP program to print level order traversal
```

```
// in spiral form using one queue and one stack.
#include <bits/stdc++.h>
using namespace std;

struct Node {
    int data;
    Node *left, *right;
};

/* Utility function to create a new tree node */
Node* newNode(int val)
{
    Node* new_node = new Node;
    new_node->data = val;
    new_node->left = new_node->right = NULL;
    return new_node;
}

/* Function to print a tree in spiral form
   using one stack */
void printSpiralUsingOneStack(Node* root)
{
    if (root == NULL)
        return;

    stack<int> s;
    queue<Node*> q;

    bool reverse = true;
    q.push(root);
    while (!q.empty()) {

        int size = q.size();
        while (size) {
            Node* p = q.front();
            q.pop();

            // if reverse is true, push node's
            // data onto the stack, else print it
            if (reverse)
                s.push(p->data);
            else
                cout << p->data << " ";

            if (p->left)
                q.push(p->left);
            if (p->right)
                q.push(p->right);
        }
        reverse = !reverse;
    }
}
```

```
        size--;
    }

    // print nodes from the stack if
    // reverse is true
    if (reverse) {
        while (!s.empty()) {
            cout << s.top() << " ";
            s.pop();
        }
    }

    // the next row has to be printed as
    // it is, hence change the value of
    // reverse
    reverse = !reverse;
}

/*Driver program to test the above functions*/
int main()
{
    Node* root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(7);
    root->left->right = newNode(6);
    root->right->left = newNode(5);
    root->right->right = newNode(4);
    printSpiralUsingOneStack(root);
    return 0;
}
```

**Output:**

1 2 3 4 5 6 7

Time Complexity :  $O(n)$   
Auxiliary Space :  $O(n)$

**Source**

<https://www.geeksforgeeks.org/level-order-traversal-in-spiral-form-using-one-stack-and-one-queue/>

## Chapter 252

# Level order traversal line by line | Set 2 (Using Two Queues)

Level order traversal line by line | Set 2 (Using Two Queues) - GeeksforGeeks

Given a Binary Tree, print the nodes level wise, each level on a new line.

Output:

```
1
2 3
4 5
```

We have discussed one solution in below article.

[Print level order traversal line by line | Set 1](#)

In this post, a different approach using two queues is discussed. We can insert the first level in first queue and print it and while popping from the first queue insert its left and right nodes into the second queue. Now start printing the second queue and before popping insert its left and right child nodes into the first queue. Continue this process till both the queues become empty.

C++

```
// C++ program to do level order traversal line by
// line
#include <bits/stdc++.h>
using namespace std;

struct Node
{
```

```
int data;
Node *left, *right;
};

// Prints level order traversal line by line
// using two queues.
void levelOrder(Node *root)
{
    queue<Node *> q1, q2;

    if (root == NULL)
        return;

    // Pushing first level node into first queue
    q1.push(root);

    // Executing loop till both the queues
    // become empty
    while (!q1.empty() || !q2.empty())
    {
        while (!q1.empty())
        {
            // Pushing left child of current node in
            // first queue into second queue
            if (q1.front()->left != NULL)
                q2.push(q1.front()->left);

            // pushing right child of current node
            // in first queue into second queue
            if (q1.front()->right != NULL)
                q2.push(q1.front()->right);

            cout << q1.front()->data << " ";
            q1.pop();
        }

        cout << "\n";

        while (!q2.empty())
        {
            // pushing left child of current node
            // in second queue into first queue
            if (q2.front()->left != NULL)
                q1.push(q2.front()->left);

            // pushing right child of current
            // node in second queue into first queue
            if (q2.front()->right != NULL)
```

```
        q1.push(q2.front()->right);

        cout << q2.front()->data << " ";
        q2.pop();
    }

    cout << "\n";
}
}

// Utility function to create a new tree node
Node* newNode(int data)
{
    Node *temp = new Node;
    temp->data = data;
    temp->left = NULL;
    temp->right = NULL;
    return temp;
}

// Driver program to test above functions
int main()
{
    Node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->right = newNode(6);

    levelOrder(root);
    return 0;
}
```

### Java

```
//Java program to do level order traversal line by
//line
import java.util.LinkedList;
import java.util.Queue;

public class GFG
{
    static class Node
    {
        int data;
        Node left;
        Node right;
    }
```

```
Node(int data)
{
    this.data = data;
    left = null;
    right = null;
}
}

// Prints level order traversal line by line
// using two queues.
static void levelOrder(Node root)
{
    Queue<Node> q1 = new LinkedList<Node>();
    Queue<Node> q2 = new LinkedList<Node>();

    if (root == null)
        return;

    // Pushing first level node into first queue
    q1.add(root);

    // Executing loop till both the queues
    // become empty
    while (!q1.isEmpty() || !q2.isEmpty())
    {

        while (!q1.isEmpty())
        {

            // Pushing left child of current node in
            // first queue into second queue
            if (q1.peek().left != null)
                q2.add(q1.peek().left);

            // pushing right child of current node
            // in first queue into second queue
            if (q1.peek().right != null)
                q2.add(q1.peek().right);

            System.out.print(q1.peek().data + " ");
            q1.remove();
        }
        System.out.println();

        while (!q2.isEmpty())
        {
```

```
// pushing left child of current node
// in second queue into first queue
if (q2.peek().left != null)
    q1.add(q2.peek().left);

// pushing right child of current
// node in second queue into first queue
if (q2.peek().right != null)
    q1.add(q2.peek().right);

System.out.print(q2.peek().data + " ");
q2.remove();
}
System.out.println();
}

// Driver program to test above functions
public static void main(String[] args)
{
    Node root = new Node(1);
    root.left = new Node(2);
    root.right = new Node(3);
    root.left.left = new Node(4);
    root.left.right = new Node(5);
    root.right.right = new Node(6);

    levelOrder(root);
}
}
// This code is Contributed by Sumit Ghosh
```

### Python

```
"""
Python program to do level order traversal
line by line using dual queue"""

class GFG:

    """Constructor to create a new tree node"""
    def __init__(self,data):
        self.val = data
        self.left = None
        self.right = None

    """Prints level order traversal line by
    line using two queues."""

```

```
def levelOrder(self, node):
    q1 = [] # Queue 1
    q2 = [] # Queue 2
    q1.append(node)

    """Executing loop till both the
    queues become empty"""
    while(len(q1) > 0 or len(q2) > 0):

        """Empty string to concatenate
        the string for q1"""
        concat_str_q1 = ''
        while(len(q1) > 0):

            """Popped node at the first
            pos in queue 1 i.e q1"""
            popped_node = q1.pop(0)
            concat_str_q1 += popped_node.val + ' '

            """Pushing left child of current
            node in first queue into second queue"""
            if popped_node.left:
                q2.append(popped_node.left)

            """Pushing right child of current node
            in first queue into second queue"""
            if popped_node.right:
                q2.append(popped_node.right)
        print( str(concat_str_q1))
        concat_str_q1 = ''

        """Empty string to concatenate the
        string for q1"""
        concat_str_q2 = ''
        while (len(q2) > 0):

            """Popped node at the first pos
            in queue 1 i.e q1"""
            popped_node = q2.pop(0)
            concat_str_q2 += popped_node.val + ' '

            """Pushing left child of current node
            in first queue into first queue"""
            if popped_node.left:
                q1.append(popped_node.left)

            """Pushing right child of current node
            in first queue into first queue"""
            if popped_node.right:
                q1.append(popped_node.right)
```

```
        if poped_node.right:
            q1.append(poped_node.right)
        print(str(concat_str_q2))
        concat_str_q2 = ''

""" Driver program to test above functions"""
node = GFG("1")
node.left = GFG("2")
node.right = GFG("3")
node.left.left = GFG("4")
node.left.right = GFG("5")
node.right.right = GFG("6")
node.levelOrder(node)

# This code is contributed by Vaibhav Kumar 12
```

Output :

```
1
2 3
4 5 6
```

Time Complexity : O(n)

Improved By : [ParulShandilya](#)

## Source

<https://www.geeksforgeeks.org/level-order-traversal-line-line-set-2-using-two-queues/>

## Chapter 253

# Level order traversal line by line | Set 3 (Using One Queue)

Level order traversal line by line | Set 3 (Using One Queue) - GeeksforGeeks

Given a Binary Tree, print the nodes level wise, each level on a new line.

Output:

```
1
2 3
4 5
```

We have discussed two solution in below articles.

[Print level order traversal line by line | Set 1](#)

[Level order traversal line by line | Set 2 \(Using Two Queues\)](#)

In this post, a different approach using one queue is discussed. First insert the root and a null element into the queue. This null element acts as a delimiter. Next pop from the top of the queue and add its left and right nodes to the end of the queue and then print the top of the queue. Continue this process till the queues become empty.

C++

```
/* CPP program to print levels
line by line */
#include <bits/stdc++.h>
using namespace std;

// A Binary Tree Node
struct node
```

```
{  
    struct node *left;  
    int data;  
    struct node *right;  
};  
  
// Function to do level order  
// traversal line by line  
void levelOrder(node *root)  
{  
    if (root == NULL) return;  
  
    // Create an empty queue for  
    // level order traversal  
    queue<node *> q;  
  
    // to store front element of  
    // queue.  
    node *curr;  
  
    // Enqueue Root and NULL node.  
    q.push(root);  
    q.push(NULL);  
  
    while (q.size() > 1)  
    {  
        curr = q.front();  
        q.pop();  
  
        // condition to check  
        // occurrence of next  
        // level.  
        if (curr == NULL)  
        {  
            q.push(NULL);  
            cout << "\n";  
        }  
  
        else {  
  
            // pushing left child of  
            // current node.  
            if(curr->left)  
                q.push(curr->left);  
  
            // pushing right child of  
            // current node.  
            if(curr->right)
```

```
q.push(curr->right);

        cout << curr->data << " ";
    }
}

// Utility function to create a
// new tree node
node* newNode(int data)
{
    node *temp = new node;
    temp->data = data;
    temp->left = NULL;
    temp->right = NULL;
    return temp;
}

// Driver program to test above
// functions
int main()
{

    // Let us create binary tree
    // shown above
    node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->right = newNode(6);

    levelOrder(root);
    return 0;
}

// This code is contributed by
// Nikhil Jindal.
```

### Java

```
// Java program to do level order
// traversal line by line
import java.util.LinkedList;
import java.util.Queue;

public class GFG {
    static class Node {
```

```
int data;
Node left;
Node right;

Node(int data) {
    this.data = data;
    left = null;
    right = null;
}
}

// Prints level order traversal line
// by line using two queues.
static void levelOrder(Node root) {
    if (root == null)
        return;

    Queue<Node> q = new LinkedList<>();

    // Pushing root node into the queue.
    q.add(root);

    // Pushing delimiter into the queue.
    q.add(null);

    // Executing loop till queue becomes
    // empty
    while (!q.isEmpty()) {

        Node curr = q.poll();

        // condition to check the
        // occurrence of next level
        if (curr == null) {
            if (!q.isEmpty()) {
                q.add(null);
                System.out.println();
            }
        } else {
            // Pushing left child current node
            if (curr.left != null)
                q.add(curr.left);

            // Pushing right child current node
            if (curr.right != null)
                q.add(curr.right);

            System.out.print(curr.data + " ");
        }
    }
}
```

```
        }
    }
}

// Driver function
public static void main(String[] args) {

    Node root = new Node(1);
    root.left = new Node(2);
    root.right = new Node(3);
    root.left.left = new Node(4);
    root.left.right = new Node(5);
    root.right.right = new Node(6);

    levelOrder(root);
}
}

// This code is Contributed by Rishabh Jindal
```

Output :

```
1
2 3
4 5 6
```

Time Complexity : O(n)

Improved By : [nik1996](#)

## Source

<https://www.geeksforgeeks.org/level-order-traversal-line-line-set-3-using-one-queue/>

## Chapter 254

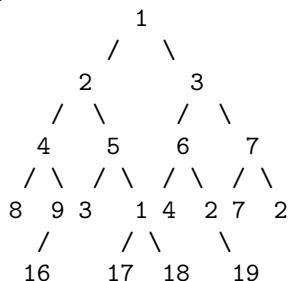
# Level order traversal with direction change after every two levels

Level order traversal with direction change after every two levels - GeeksforGeeks

Given a binary tree, print the level order traversal in such a way that first two levels are printed from left to right, next two levels are printed from right to left, then next two from left to right and so on. So, the problem is to reverse the direction of level order traversal of binary tree after every two levels.

Examples:

Input:



Output:

```
1
2 3
7 6 5 4
2 7 2 4 1 3 9 8
16 17 18 19
```

In the above example, first two levels  
are printed from left to right, next two

levels are printed from right to left,  
and then last level is printed from  
left to right.

**Approach:**

We make use of queue and stack here. Queue is used for performing normal level order traversal. Stack is used for reversing the direction of traversal after every two levels.

While doing normal level order traversal, first two levels nodes are printed at the time when they are popped out from the queue. For the next two levels, we instead of printing the nodes, pushed them onto the stack. When all nodes of current level are popped out, we print the nodes in the stack. In this way, we print the nodes in right to left order by making use of the stack. Now for the next two levels we again do normal level order traversal for printing nodes from left to right. Then for the next two nodes, we make use of the stack for achieving right to left order.

In this way, we will achieve desired modified level order traversal by making use of queue and stack.

```
// CPP program to print Zig-Zag traversal
// in groups of size 2.
#include <iostream>
#include <queue>
#include <stack>
using namespace std;

// A Binary Tree Node
struct Node {
    struct Node* left;
    int data;
    struct Node* right;
};

/* Function to print the level order of
given binary tree. Direction of printing
level order traversal of binary tree changes
after every two levels */
void modifiedLevelOrder(struct Node* node)
{
    // For null root
    if (node == NULL)
        return;

    if (node->left == NULL && node->right == NULL) {
        cout << node->data;
        return;
    }

    // Maintain a queue for normal level order traversal
    queue<Node*> myQueue;
```

```
/* Maintain a stack for printing nodes in reverse
   order after they are popped out from queue.*/
stack<Node*> myStack;

struct Node* temp = NULL;

// sz is used for storing the count of nodes in a level
int sz;

// Used for changing the direction of level order traversal
int ct = 0;

// Used for changing the direction of level order traversal
bool rightToLeft = false;

// Push root node to the queue
myQueue.push(node);

// Run this while loop till queue got empty
while (!myQueue.empty()) {
    ct++;

    sz = myQueue.size();

    // Do a normal level order traversal
    for (int i = 0; i < sz; i++) {
        temp = myQueue.front();
        myQueue.pop();

        /*For printing nodes from left to right,
         simply print the nodes in the order in which
         they are being popped out from the queue.*/
        if (rightToLeft == false)
            cout << temp->data << " ";

        /* For printing nodes from right to left,
         push the nodes to stack instead of printing them.*/
        else
            myStack.push(temp);

        if (temp->left)
            myQueue.push(temp->left);

        if (temp->right)
            myQueue.push(temp->right);
    }
}
```

```
if (rightToLeft == true) {

    // for printing the nodes in order
    // from right to left
    while (!myStack.empty()) {
        temp = myStack.top();
        myStack.pop();

        cout << temp->data << " ";
    }
}

/*Change the direction of printing
nodes after every two levels.*/
if (ct == 2) {
    rightToLeft = !rightToLeft;
    ct = 0;
}

cout << "\n";
}

// Utility function to create a new tree node
Node* newNode(int data)
{
    Node* temp = new Node;
    temp->data = data;
    temp->left = temp->right = NULL;
    return temp;
}

// Driver program to test above functions
int main()
{
    // Let us create binary tree
    Node* root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->left = newNode(6);
    root->right->right = newNode(7);
    root->left->left->left = newNode(8);
    root->left->left->right = newNode(9);
    root->left->right->left = newNode(3);
    root->left->right->right = newNode(1);
    root->right->left->left = newNode(4);
```

```
root->right->left->right = newNode(2);
root->right->right->left = newNode(7);
root->right->right->right = newNode(2);
root->left->right->left->left = newNode(16);
root->left->right->left->right = newNode(17);
root->right->left->right->left = newNode(18);
root->right->right->left->right = newNode(19);

modifiedLevelOrder(root);

return 0;
}
```

**Output:**

```
1
2 3
7 6 5 4
2 7 2 4 1 3 9 8
16 17 18 19
```

**Time Complexity:** Each node is traversed at most twice while doing level order traversal, so time complexity would be O(n).

**Approach 2:**

We make use of queue and stack here, but in a different way. Using macros `#define ChangeDirection(Dir) ((Dir) = 1 - (Dir))`. In following implementation directs the order of push operations in both queue or stack.

In this way, we will achieve desired modified level order traversal by making use of queue and stack.

```
// CPP program to print Zig-Zag traversal
// in groups of size 2.
#include <iostream>
#include <stack>
#include <queue>

using namespace std;

#define LEFT 0
#define RIGHT 1
#define ChangeDirection(Dir) ((Dir) = 1 - (Dir))

// A Binary Tree Node
struct node
{
    int data;
```

```
    struct node *left, *right;
};

// Utility function to create a new tree node
node* newNode(int data)
{
    node* temp = new node;
    temp->data = data;
    temp->left = temp->right = NULL;
    return temp;
}

/* Function to print the level order of
   given binary tree. Direction of printing
   level order traversal of binary tree changes
   after every two levels */
void modifiedLevelOrder(struct node *root)
{
    if (!root)
        return ;

    int dir = LEFT;
    struct node *temp;
    queue <struct node *> Q;
    stack <struct node *> S;

    S.push(root);

    // Run this while loop till queue got empty
    while (!Q.empty() || !S.empty())
    {
        while (!S.empty())
        {
            temp = S.top();
            S.pop();
            cout << temp->data << " ";

            if (dir == LEFT) {
                if (temp->left)
                    Q.push(temp->left);
                if (temp->right)
                    Q.push(temp->right);
            }
            /* For printing nodes from right to left,
               push the nodes to stack instead of printing them.*/
            else {
                if (temp->right)
                    Q.push(temp->right);
            }
        }
    }
}
```

```
        if (temp->left)
            Q.push(temp->left);
    }
}

cout << endl;

// for printing the nodes in order
// from right to left
while (!Q.empty())
{
    temp = Q.front();
    Q.pop();
    cout << temp->data << " ";

    if (dir == LEFT) {
        if (temp->left)
            S.push(temp->left);
        if (temp->right)
            S.push(temp->right);
    } else {
        if (temp->right)
            S.push(temp->right);
        if (temp->left)
            S.push(temp->left);
    }
}
cout << endl;

// Change the direction of traversal.
ChangeDirection(dir);
}

// Driver program to test above functions
int main()
{
    // Let us create binary tree
    node* root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->left = newNode(6);
    root->right->right = newNode(7);
    root->left->left->left = newNode(8);
    root->left->left->right = newNode(9);
    root->left->right->left = newNode(3);
```

```
root->left->right->right = newNode(1);
root->right->left->left = newNode(4);
root->right->left->right = newNode(2);
root->right->right->left = newNode(7);
root->right->right->right = newNode(2);
root->left->right->left->left = newNode(16);
root->left->right->left->right = newNode(17);
root->right->left->right->left = newNode(18);
root->right->right->left->right = newNode(19);

modifiedLevelOrder(root);

return 0;
}
```

**Output:**

```
1
2 3
7 6 5 4
2 7 2 4 1 3 9 8
16 17 18 19
```

**Time Complexity:** every node is also traversed twice. There time complexity is still  $O(n)$ .

**Improved By :** [vhinf2047](#)

**Source**

<https://www.geeksforgeeks.org/level-order-traversal-direction-change-every-two-levels/>

## Chapter 255

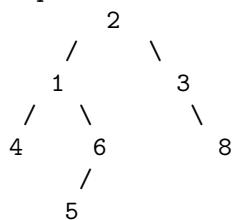
# Level with maximum number of nodes

Level with maximum number of nodes - GeeksforGeeks

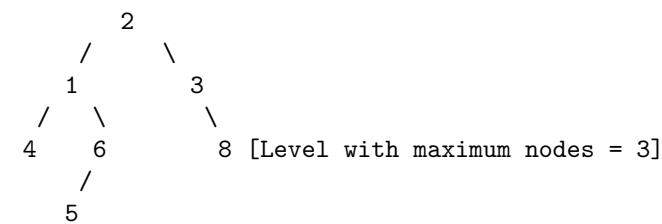
Find the level in a binary tree which has maximum number of nodes. The root is at level 0.

Examples:

Input :



Output : 2



We know that in [level order traversal of binary tree with queue](#), at any time our queue contains all elements of a particular level. So find level with maximum number of nodes in queue.

```
// C++ implementation to find the level
// having maximum number of Nodes
#include <bits/stdc++.h>
using namespace std;

/* A binary tree Node has data, pointer
   to left child and a pointer to right
   child */
struct Node
{
    int data;
    struct Node* left;
    struct Node* right;
};

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct Node* newNode(int data)
{
    struct Node* node = new Node;
    node->data = data;
    node->left = NULL;
    node->right = NULL;
    return(node);
}

// function to find the level
// having maximum number of Nodes
int maxNodeLevel(Node *root)
{
    if (root == NULL)
        return -1;

    queue<Node *> q;
    q.push(root);

    // Current level
    int level = 0;

    // Maximum Nodes at same level
    int max = INT_MIN;

    // Level having maximum Nodes
    int level_no = 0;

    while (1)
    {
        // Count Nodes in a level
```

```
int NodeCount = q.size();

if (NodeCount == 0)
    break;

// If it is maximum till now
// Update level_no to current level
if (NodeCount > max)
{
    max = NodeCount;
    level_no = level;
}

// Pop complete current level
while (NodeCount > 0)
{
    Node *Node = q.front();
    q.pop();
    if (Node->left != NULL)
        q.push(Node->left);
    if (Node->right != NULL)
        q.push(Node->right);
    NodeCount--;
}

// Increment for next level
level++;
}

return level_no;
}

// Driver program to test above
int main()
{
    // binary tree formation
    struct Node *root = newNode(2);      /*      2      */
    root->left     = newNode(1);        /*      / \      */
    root->right    = newNode(3);        /*      1   3      */
    root->left->left  = newNode(4);    /*      / \ \      */
    root->left->right = newNode(6);    /*      4   6   8      */
    root->right->right = newNode(8);   /*          /      */
    root->left->right->left = newNode(5);/*      5      */

    printf("Level having maximum number of Nodes : %d",
           maxNodeLevel(root));
    return 0;
}
```

Output:

```
Level having maximum number of nodes : 2
```

Time Complexity :  $O(n)$

### Source

<https://www.geeksforgeeks.org/level-maximum-number-nodes/>

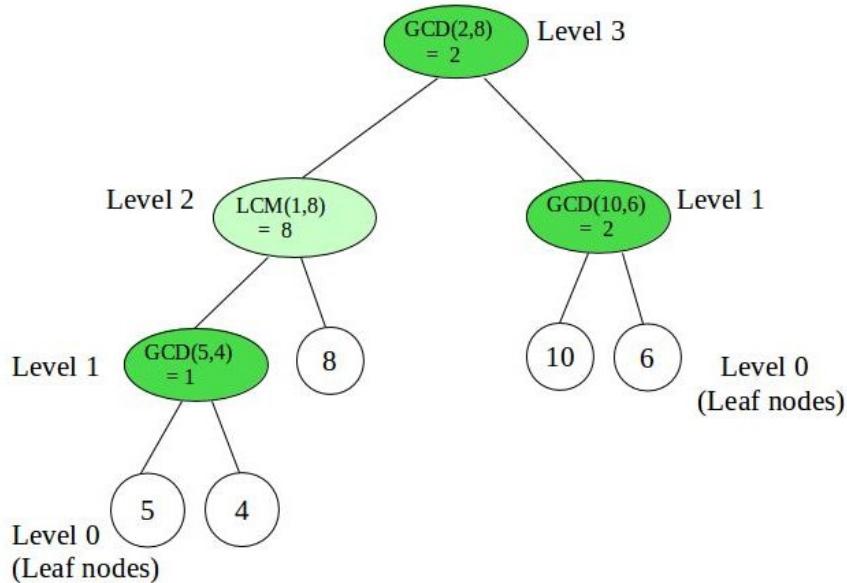
## Chapter 256

# Levelwise Alternating GCD and LCM of nodes in Segment Tree

Levelwise Alternating GCD and LCM of nodes in Segment Tree - GeeksforGeeks

A Levelwise GCD/LCM alternating segment tree is a segment tree, such that at every level the operations GCD and LCM alternate. In other words at Level 1 the left and right sub-trees combine together by the GCD operation i.e Parent node = Left Child **GCD** Right Child and on Level 2 the left and right sub-trees combine together by the LCM operation i.e Parent node = Left Child **LCM** Right Child

Such a type of Segment tree has the following type of structure:



The operations (GCD) and (LCM) indicate which operation was carried out to merge the child nodes

Given N leaf nodes, the task is to build such a segment tree and print the root node.  
Examples:

Input : arr[] = { 5, 4, 8, 10, 6 }  
Output : Value at Root Node = 2  
Explanation : The image given above shows the segment tree corresponding to the given set leaf nodes.

**Prerequisites:** [Segment Trees](#)

In this Segment Tree, we carry two operations:- **GCD and LCM**.

Now, along with the information which is passed recursively for the sub-trees, information regarding the operation to be carried out at that level is also passed since these operations alternate levelwise. It is important to note that a parent node when calls its left and right children the same operation information is passed to both the children as they are on the same level.

Let's represent the two operations i.e GCD and LCM by 0 and 1 respectively. Then, if at Level i GCD operation is performed then at Level (i + 1) LCM operation will be performed. Thus if Level i has 0 as operation then level (i + 1) will have 1 as operation.

Operation at Level (i + 1) = ! (Operation at Level i)  
where,  
Operation at Level i {0, 1}

Careful analysis of the image suggests that if the height of the tree is even then the root node is a result of LCM operation of its left and right children else a result of GCD operation.

```
#include <bits/stdc++.h>
using namespace std;

// Recursive function to return gcd of a and b
int gcd(int a, int b)
{
    // Everything divides 0
    if (a == 0 || b == 0)
        return 0;

    // base case
    if (a == b)
        return a;
```

```

// a is greater
if (a > b)
    return gcd(a-b, b);
return gcd(a, b-a);
}

// A utility function to get the middle index from
// corner indexes.
int getMid(int s, int e) { return s + (e - s) / 2; }

void STconstructUtil(int arr[], int ss, int se, int* st,
                     int si, int op)
{
    // If there is one element in array, store it in
    // current node of segment tree and return
    if (ss == se) {
        st[si] = arr[ss];
        return;
    }

    // If there are more than one elements, then recur
    // for left and right subtrees and store the sum of
    // values in this node
    int mid = getMid(ss, se);

    // Build the left and the right subtrees by using
    // the fact that operation at level (i + 1) = !
    // (operation at level i)
    STconstructUtil(arr, ss, mid, st, si * 2 + 1, !op);
    STconstructUtil(arr, mid + 1, se, st, si * 2 + 2, !op);

    // merge the left and right subtrees by checking
    // the operation to be carried. If operation = 1,
    // then do GCD else LCM
    if (op == 1) {
        // GCD operation
        st[si] = __gcd(st[2 * si + 1], st[2 * si + 2]);
    }
    else {
        // LCM operation
        st[si] = (st[2 * si + 1] * st[2 * si + 2]) /
                  (gcd(st[2 * si + 1], st[2 * si + 2]));
    }
}

/* Function to construct segment tree from given array.
This function allocates memory for segment tree and
calls STconstructUtil() to fill the allocated memory */

```

```
int* STconstruct(int arr[], int n)
{
    // Allocate memory for segment tree

    // Height of segment tree
    int x = (int)(ceil(log2(n)));

    // maximum size of segment tree
    int max_size = 2 * (int)pow(2, x) - 1;

    // allocate memory
    int* st = new int[max_size];

    // operation = 1(GCD) if Height of tree is
    // even else it is 0(LCM) for the root node
    int opAtRoot = (x % 2 == 0 ? 0 : 1);

    // Fill the allocated memory st
    STconstructUtil(arr, 0, n - 1, st, 0, opAtRoot);

    // Return the constructed segment tree
    return st;
}

int main()
{
    int arr[] = { 5, 4, 8, 10, 6 };
    int n = sizeof(arr) / sizeof(arr[0]);

    // Build segment tree
    int* st = STconstruct(arr, n);

    // 0-based indexing in segment tree
    int rootIndex = 0;

    cout << "Value at Root Node = " << st[rootIndex];

    return 0;
}
```

Output:

```
Value at Root Node = 2
```

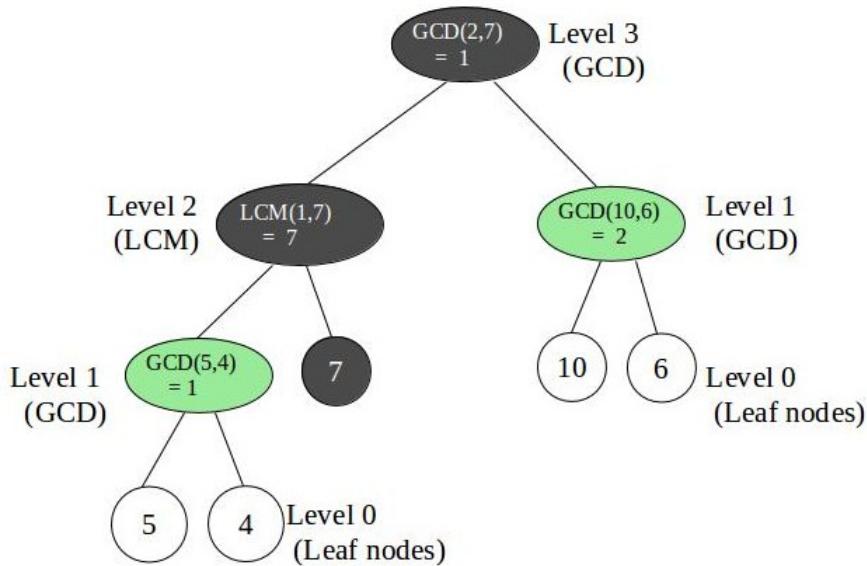
Time complexity for tree construction is  $O(n)$ , as there are total  $2^*n-1$  nodes and value at every node is calculated at once.

Now to perform point updates i.e. update the value with given index and value, can be done by traversing down the tree to the leaf node and performing the update.

While coming back to the root node we build the tree again similar to the build() function by passing the operation to be performed at every level and storing the result of applying that operation on values of its left and right children and storing the result into that node.

Consider the following Segment tree after performing the update,  
 $\text{arr}[2] = 7$

Now the updated segment tree looks like this:



Here nodes in black denote the fact that they are updated.

```

#include <bits/stdc++.h>
using namespace std;

// Recursive function to return gcd of a and b
int gcd(int a, int b)
{
    // Everything divides 0
    if (a == 0 || b == 0)
        return 0;

    // base case
    if (a == b)
        return a;

    // a is greater
    if (a > b)
        return gcd(a-b, b);
    return gcd(a, b-a);
}

```

```

}

// A utility function to get the middle index from
// corner indexes.
int getMid(int s, int e) { return s + (e - s) / 2; }

void STconstructUtil(int arr[], int ss, int se, int* st,
                     int si, int op)
{
    // If there is one element in array, store it in
    // current node of segment tree and return
    if (ss == se) {
        st[si] = arr[ss];
        return;
    }

    // If there are more than one elements, then recur
    // for left and right subtrees and store the sum of
    // values in this node
    int mid = getMid(ss, se);

    // Build the left and the right subtrees by using
    // the fact that operation at level (i + 1) = !
    // (operation at level i)
    STconstructUtil(arr, ss, mid, st, si * 2 + 1, !op);
    STconstructUtil(arr, mid + 1, se, st, si * 2 + 2, !op);

    // merge the left and right subtrees by checking
    // the operation to be carried. If operation = 1,
    // then do GCD else LCM
    if (op == 1) {
        // GCD operation
        st[si] = gcd(st[2 * si + 1], st[2 * si + 2]);
    }
    else {
        // LCM operation
        st[si] = (st[2 * si + 1] * st[2 * si + 2]) /
                  (gcd(st[2 * si + 1], st[2 * si + 2]));
    }
}

void updateUtil(int* st, int ss, int se, int ind, int val,
                int si, int op)
{
    // Base Case: If the input index lies outside
    // this segment
    if (ind < ss || ind > se)
        return;
}

```

```

// If the input index is in range of this node,
// then update the value of the node and its
// children

// leaf node
if (ss == se && ss == ind) {
    st[si] = val;
    return;
}

int mid = getMid(ss, se);

// Update the left and the right subtrees by
// using the fact that operation at level
// (i + 1) = ! (operation at level i)
updateUtil(st, ss, mid, ind, val, 2 * si + 1, !op);
updateUtil(st, mid + 1, se, ind, val, 2 * si + 2, !op);

// merge the left and right subtrees by checking
// the operation to be carried. If operation = 1,
// then do GCD else LCM
if (op == 1) {

    // GCD operation
    st[si] = gcd(st[2 * si + 1], st[2 * si + 2]);
}
else {

    // LCM operation
    st[si] = (st[2 * si + 1] * st[2 * si + 2]) /
        (gcd(st[2 * si + 1], st[2 * si + 2]));
}

void update(int arr[], int* st, int n, int ind, int val)
{
    // Check for erroneous input index
    if (ind < 0 || ind > n - 1) {
        printf("Invalid Input");
        return;
    }

    // Height of segment tree
    int x = (int)(ceil(log2(n)));

    // operation = 1(GCD) if Height of tree is
    // even else it is 0(LCM) for the root node
}

```

```
int opAtRoot = (x % 2 == 0 ? 0 : 1);

arr[ind] = val;

// Update the values of nodes in segment tree
updateUtil(st, 0, n - 1, ind, val, 0, opAtRoot);
}

/* Function to construct segment tree from given array.
This function allocates memory for segment tree and
calls STconstructUtil() to fill the allocated memory */
int* STconstruct(int arr[], int n)
{
    // Allocate memory for segment tree

    // Height of segment tree
    int x = (int)(ceil(log2(n)));

    // maximum size of segment tree
    int max_size = 2 * (int)pow(2, x) - 1;

    // allocate memory
    int* st = new int[max_size];

    // operation = 1(GCD) if Height of tree is
    // even else it is 0(LCM) for the root node
    int opAtRoot = (x % 2 == 0 ? 0 : 1);

    // Fill the allocated memory st
    STconstructUtil(arr, 0, n - 1, st, 0, opAtRoot);

    // Return the constructed segment tree
    return st;
}

int main()
{
    int arr[] = { 5, 4, 8, 10, 6 };
    int n = sizeof(arr) / sizeof(arr[0]);

    // Build segment tree
    int* st = STconstruct(arr, n);

    // 0-based indexing in segment tree
    int rootIndex = 0;

    cout << "Old Value at Root Node = " <<
        st[rootIndex] << endl;
```

```
// perform update arr[2] = 7
update(arr, st, n, 2, 7);

cout << "New Value at Root Node = " <<
        st[rootIndex] << endl;

return 0;
}
```

Output:

```
Old Value at Root Node = 2
New Value at Root Node = 1
```

The time complexity of update is also  $O(\log n)$ . To update a leaf value, one node is processed at every level and number of levels is  $O(\log n)$ .

## Source

<https://www.geeksforgeeks.org/levelwise-alternating-gcd-lcm-nodes-segment-tree/>

## Chapter 257

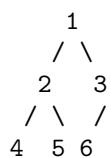
# Linked complete binary tree & its creation

Linked complete binary tree & its creation - GeeksforGeeks

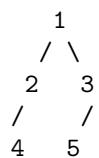
A complete binary tree is a binary tree where each level 'l' except the last has  $2^l$  nodes and the nodes at the last level are all left aligned. Complete binary trees are mainly used in heap based data structures.

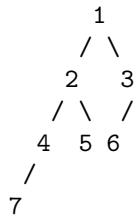
The nodes in the complete binary tree are inserted from left to right in one level at a time. If a level is full, the node is inserted in a new level.

Below are some of the complete binary trees.



Below binary trees are not complete:





Complete binary trees are generally represented using arrays. The array representation is better because it doesn't contain any empty slot. Given parent index i, its left child is given by  $2 * i + 1$  and its right child is given by  $2 * i + 2$ . So no extra space is wasted and space to store left and right pointers is saved. However, it may be an interesting programming question to create a Complete Binary Tree using linked representation. Here Linked mean a non-array representation where left and right pointers(or references) are used to refer left and right children respectively. How to write an insert function that always adds a new node in the last level and at the leftmost available position?

To create a linked complete binary tree, we need to keep track of the nodes in a level order fashion such that the next node to be inserted lies in the leftmost position. A queue data structure can be used to keep track of the inserted nodes.

Following are steps to insert a new node in Complete Binary Tree.

1. If the tree is empty, initialize the root with new node.
2. Else, get the front node of the queue.  
.....If the left child of this front node doesn't exist, set the left child as the new node.  
.....else if the right child of this front node doesn't exist, set the right child as the new node.
3. If the front node has both the left child and right child, Dequeue() it.
4. Enqueue() the new node.

Below is the implementation:

```

// Program for linked implementation of complete binary tree
#include <stdio.h>
#include <stdlib.h>

// For Queue Size
#define SIZE 50

// A tree node
struct node
{
    int data;
    struct node *right,*left;
};

// A queue node
struct Queue
{

```

```
int front, rear;
int size;
struct node* *array;
};

// A utility function to create a new tree node
struct node* newNode(int data)
{
    struct node* temp = (struct node*) malloc(sizeof( struct node ));
    temp->data = data;
    temp->left = temp->right = NULL;
    return temp;
}

// A utility function to create a new Queue
struct Queue* createQueue(int size)
{
    struct Queue* queue = (struct Queue*) malloc(sizeof( struct Queue ));

    queue->front = queue->rear = -1;
    queue->size = size;

    queue->array = (struct node**) malloc(queue->size * sizeof( struct node* ));

    int i;
    for (i = 0; i < size; ++i)
        queue->array[i] = NULL;

    return queue;
}

// Standard Queue Functions
int isEmpty(struct Queue* queue)
{
    return queue->front == -1;
}

int isFull(struct Queue* queue)
{   return queue->rear == queue->size - 1; }

int hasOnlyOneItem(struct Queue* queue)
{   return queue->front == queue->rear;   }

void Enqueue(struct node *root, struct Queue* queue)
{
    if (isFull(queue))
        return;
```

```
queue->array[queue->rear] = root;

if (isEmpty(queue))
    ++queue->front;
}

struct node* Dequeue(struct Queue* queue)
{
    if (isEmpty(queue))
        return NULL;

    struct node* temp = queue->array[queue->front];

    if (hasOnlyOneItem(queue))
        queue->front = queue->rear = -1;
    else
        ++queue->front;

    return temp;
}

struct node* getFront(struct Queue* queue)
{   return queue->array[queue->front]; }

// A utility function to check if a tree node has both left and right children
int hasBothChild(struct node* temp)
{
    return temp && temp->left && temp->right;
}

// Function to insert a new node in complete binary tree
void insert(struct node **root, int data, struct Queue* queue)
{
    // Create a new node for given data
    struct node *temp = newNode(data);

    // If the tree is empty, initialize the root with new node.
    if (!*root)
        *root = temp;

    else
    {
        // get the front node of the queue.
        struct node* front = getFront(queue);

        // If the left child of this front node doesn't exist, set the
        // left child as the new node
        if (!front->left)
```

```
    front->left = temp;

    // If the right child of this front node doesn't exist, set the
    // right child as the new node
    else if (!front->right)
        front->right = temp;

    // If the front node has both the left child and right child,
    // Dequeue() it.
    if (hasBothChild(front))
        Dequeue(queue);
}

// Enqueue() the new node for later insertions
Enqueue(temp, queue);
}

// Standard level order traversal to test above function
void levelOrder(struct node* root)
{
    struct Queue* queue = createQueue(SIZE);

    Enqueue(root, queue);

    while (!isEmpty(queue))
    {
        struct node* temp = Dequeue(queue);

        printf("%d ", temp->data);

        if (temp->left)
            Enqueue(temp->left, queue);

        if (temp->right)
            Enqueue(temp->right, queue);
    }
}

// Driver program to test above functions
int main()
{
    struct node* root = NULL;
    struct Queue* queue = createQueue(SIZE);
    int i;

    for(i = 1; i <= 12; ++i)
        insert(&root, i, queue);
```

```
    levelOrder(root);  
  
    return 0;  
}
```

Output:

```
1 2 3 4 5 6 7 8 9 10 11 12
```

This article is compiled by [Aashish Barnwal](#) and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## Source

<https://www.geeksforgeeks.org/linked-complete-binary-tree-its-creation/>

## Chapter 258

# Locking and Unlocking of Resources arranged in the form of n-ary Tree

Locking and Unlocking of Resources arranged in the form of n-ary Tree - GeeksforGeeks

Given an n-ary tree of resources arranged hierarchically such that height of tree is  $O(\log N)$  where  $N$  is total number of nodes (or resources). A process needs to lock a resource node in order to use it. But a node cannot be locked if any of its descendant or ancestor is locked.

Following operations are required for a given resource node:

- `islock()`- returns true if a given node is locked and false if it is not. A node is locked if `lock()` has successfully executed for the node.
- `Lock()`- locks the given node if possible and updates lock information. Lock is possible only if ancestors and descendants of current node are not locked.
- `unLock()`- unlocks the node and updates information.

How design resource nodes and implement above operations such that following time complexities are achieved.

```
islock()  O(1)
Lock()    O(log N)
unLock()  O(log N)
```

We strongly recommend you to minimize your browser and try this yourself first.

It is given that resources need to be stored in the form of n-ary tree. Now the question is, how to augment the tree to achieve above complexity bounds.

### Method 1 (Simple)

A Simple Solution is to store a boolean variable **isLocked** with every resource node. The boolean variable **isLocked** is true if current node is locked, else false.

Let us see how operations work using this Approach.

- **isLock()**: Check **isLocked** of the given node.
- **Lock()**: If **isLocked** is set, then the node cannot be locked. Else check all descendants and ancestors of the node and if any of them is locked, then also this node cannot be locked. If none of the above conditions is true, then lock this node by setting **isLocked** as true.
- **unLock()**: If **isLocked** of given node is false, nothing to do. Else set **isLocked** as false.

Time Complexities:

```
isLock()  O(1)
Lock()    O(N)
unLock()  O(1)
```

**Lock** is  $O(N)$  because there can be  $O(N)$  descendants.

### Method 2 (Time Complexities according to question)

The idea is to augment tree with following three fields:

1. A boolean field **isLocked** (same as above method).
2. **Parent-Pointer** to access all ancestors in  $O(\log n)$  time.
3. **Count-of-locked-descendants**. Note that a node can be ancestor of many descendants. We can check if any of the descendants is locked using this count.

Let us see how operations work using this Approach.

- **isLock()**: Check **isLocked** of the given node.
- **Lock()**: Traverse all ancestors. If none of the ancestors is locked, **Count-of-locked-descendants** is 0 and **isLocked** is false, set **isLocked** of current node as true. And increment **Count-of-locked-descendants** for all ancestors. Time complexity is  $O(\log N)$  as there can be at most  $O(\log N)$  ancestors.
- **unLock()**: Traverse all ancestors and decrease **Count-of-locked-descendants** for all ancestors. Set **isLocked** of current node as false. Time complexity is  $O(\log N)$

Thanks to Utkarsh Trivedi for suggesting this approach.

This article is contributed by **Abhay Rathi**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

**Source**

<https://www.geeksforgeeks.org/locking-and-unlocking-of-resources-in-the-form-of-n-ary-tree/>

## Chapter 259

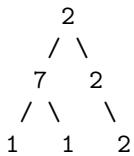
# Longest Path with Same Values in a Binary Tree

Longest Path with Same Values in a Binary Tree - GeeksforGeeks

Given a binary tree, find the length of the longest path where each node in the path has the same value. This path may or may not pass through the root. The length of path between two nodes is represented by the number of edges between them.

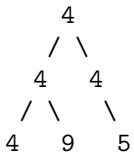
Examples:

Input :



Output : 2

Input :



Output : 3

The idea is to recursively traverse given binary tree. We can think of any path (of nodes with the same values) in up to two directions(left and right) from it's root. Then, for each node, we want to know what is the longest possible length extending in the left and the longest possible length extending in the right directions. The longest length that extends

from the node will be  $1 + \text{length}(\text{node-}>\text{left})$  if  $\text{node-}>\text{left}$  exists, and has the same value as  $\text{node}$ . Similarly for the  $\text{node-}>\text{right}$  case.

While we are computing lengths, each candidate answer will be the sum of the lengths in both directions from that node. We keep updating these answers and return the maximum one.

```
// C++ program to find the length of longest
// path with same values in a binary tree.
#include <bits/stdc++.h>
using namespace std;

/* A binary tree node has data, pointer to
left child and a pointer to right child */
struct Node {
    int val;
    struct Node *left, *right;
};

/* Function to print the longest path
of same values */
int length(Node *node, int *ans) {
    if (!node)
        return 0;

    // Recursive calls to check for subtrees
    int left = length(node->left, ans);
    int right = length(node->right, ans);

    // Variables to store maximum lengths in two directions
    int Leftmax = 0, Rightmax = 0;

    // If curr node and it's left child has same value
    if (node->left && node->left->val == node->val)
        Leftmax += left + 1;

    // If curr node and it's right child has same value
    if (node->right && node->right->val == node->val)
        Rightmax += right + 1;

    *ans = max(*ans, Leftmax + Rightmax);
    return max(Leftmax, Rightmax);
}

/* Driver function to find length of
longest same value path*/
int longestSameValuePath(Node *root) {
    int ans = 0;
    length(root, &ans);
```

```
    return ans;
}

/* Helper function that allocates a
new node with the given data and
NULL left and right pointers. */
Node *newNode(int data) {
    Node *temp = new Node;
    temp->val = data;
    temp->left = temp->right = NULL;
    return temp;
}

// Driver code
int main() {
    /* Let us construct a Binary Tree
        4
       / \
      4   4
     / \   \
    4   9   5 */
    Node *root = NULL;
    root = newNode(4);
    root->left = newNode(4);
    root->right = newNode(4);
    root->left->left = newNode(4);
    root->left->right = newNode(9);
    root->right->right = newNode(5);
    cout << longestSameValuePath(root);
    return 0;
}
```

### Complexity Analysis:

- Time complexity :  $O(n)$ , where  $n$  is the number of nodes in tree as every node is processed once.
- Auxiliary Space :  $O(h)$ , where  $h$  is the height of tree as recursion can go upto depth  $h$ .

### Source

<https://www.geeksforgeeks.org/longest-path-values-binary-tree/>

## Chapter 260

# Longest consecutive sequence in Binary tree

Longest consecutive sequence in Binary tree - GeeksforGeeks

Given a Binary Tree find the length of the longest path which comprises of nodes with consecutive values in increasing order. Every node is considered as a path of length 1.

Examples:

In below diagram binary tree with longest consecutive path(LCP) are shown :

We can solve above problem recursively. At each node we need information of its parent node, if current node has value one more than its parent node then it makes a consecutive path, at each node we will compare node's value with its parent value and update the longest consecutive path accordingly.

For getting the value of parent node, we will pass the (node\_value + 1) as an argument to the recursive method and compare the node value with this argument value, if satisfies, update the current length of consecutive path otherwise reinitialize current path length by 1.

Please see below code for better understanding :

```
// C/C++ program to find longest consecutive
// sequence in binary tree
#include <bits/stdc++.h>
using namespace std;

/* A binary tree node has data, pointer to left
   child and a pointer to right child */
```

```
struct Node
{
    int data;
    Node *left, *right;
};

// A utility function to create a node
Node* newNode(int data)
{
    Node* temp = new Node;
    temp->data = data;
    temp->left = temp->right = NULL;
    return temp;
}

// Utility method to return length of longest
// consecutive sequence of tree
void longestConsecutiveUtil(Node* root, int curLength,
                            int expected, int& res)
{
    if (root == NULL)
        return;

    // if root data has one more than its parent
    // then increase current length
    if (root->data == expected)
        curLength++;
    else
        curLength = 1;

    // update the maximum by current length
    res = max(res, curLength);

    // recursively call left and right subtree with
    // expected value 1 more than root data
    longestConsecutiveUtil(root->left, curLength,
                           root->data + 1, res);
    longestConsecutiveUtil(root->right, curLength,
                           root->data + 1, res);
}

// method returns length of longest consecutive
// sequence rooted at node root
int longestConsecutive(Node* root)
{
    if (root == NULL)
        return 0;
```

```
int res = 0;

// call utility method with current length 0
longestConsecutiveUtil(root, 0, root->data, res);

return res;
}

// Driver code to test above methods
int main()
{
    Node* root = newNode(6);
    root->right = newNode(9);
    root->right->left = newNode(7);
    root->right->right = newNode(10);
    root->right->right->right = newNode(11);

    printf("%d\n", longestConsecutive(root));
    return 0;
}
```

Output:

3

Also discussed on below link:

[Maximum Consecutive Increasing Path Length in Binary Tree](#)

## Source

<https://www.geeksforgeeks.org/longest-consecutive-sequence-binary-tree/>

## Chapter 261

# Longest path in an undirected tree

Longest path in an undirected tree - GeeksforGeeks

Given an undirected tree, we need to find the longest path of this tree where a path is defined as a sequence of nodes.

Example:

```
Input : Below shown Tree using adjacency list
        representation:
Output : 5
In below tree longest path is of length 5
from node 5 to node 7
```

This problem is same as [diameter of n-ary tree](#). We have discussed a simple solution [here](#).

In this post, an efficient solution is discussed. We can find longest path using two **BFSs**. The idea is based on the following fact: If we start BFS from any node  $x$  and find a node with the longest distance from  $x$ , it must be an end point of the longest path. It can be proved using contradiction. So our algorithm reduces to simple two BFSs. First BFS to find an end point of the longest path and second BFS from this end point to find the actual longest path.

As we can see in above diagram, if we start our BFS from node-0, the node at the farthest distance from it will be node-5, now if we start our BFS from node-5 the node at the farthest distance will be node-7, finally, path from node-5 to node-7 will constitute our longest path.

```
// C++ program to find longest path of the tree
#include <bits/stdc++.h>
using namespace std;
```

```
// This class represents a undirected graph using adjacency list
class Graph
{
    int V;           // No. of vertices
    list<int> *adj; // Pointer to an array containing
                     // adjacency lists
public:
    Graph(int V);           // Constructor
    void addEdge(int v, int w); // function to add an edge to graph
    void longestPathLength(); // prints longest path of the tree
    pair<int, int> bfs(int u); // function returns maximum distant
                               // node from u with its distance
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w);    // Add w to v's list.
    adj[w].push_back(v);    // Since the graph is undirected
}

// method returns farthest node and its distance from node u
pair<int, int> Graph::bfs(int u)
{
    // mark all distance with -1
    int dis[V];
    memset(dis, -1, sizeof(dis));

    queue<int> q;
    q.push(u);

    // distance of u from u will be 0
    dis[u] = 0;

    while (!q.empty())
    {
        int t = q.front();      q.pop();

        // loop for all adjacent nodes of node-t
        for (auto it = adj[t].begin(); it != adj[t].end(); it++)
        {
            int v = *it;
```

```
// push node into queue only if
// it is not visited already
if (dis[v] == -1)
{
    q.push(v);

    // make distance of v, one more
    // than distance of t
    dis[v] = dis[t] + 1;
}
}

int maxDis = 0;
int nodeIdx;

// get farthest node distance and its index
for (int i = 0; i < V; i++)
{
    if (dis[i] > maxDis)
    {
        maxDis = dis[i];
        nodeIdx = i;
    }
}
return make_pair(nodeIdx, maxDis);
}

// method prints longest path of given tree
void Graph::longestPathLength()
{
    pair<int, int> t1, t2;

    // first bfs to find one end point of
    // longest path
    t1 = bfs(0);

    // second bfs to find actual longest path
    t2 = bfs(t1.first);

    cout << "Longest path is from " << t1.first << " to "
        << t2.first << " of length " << t2.second;
}

// Driver code to test above methods
int main()
{
```

```
// Create a graph given in the example
Graph g(10);
g.addEdge(0, 1);
g.addEdge(1, 2);
g.addEdge(2, 3);
g.addEdge(2, 9);
g.addEdge(2, 4);
g.addEdge(4, 5);
g.addEdge(1, 6);
g.addEdge(6, 7);
g.addEdge(6, 8);

g.longestPathLength();
return 0;
}
```

Output:

```
Longest path is from 5 to 7 of length 5
```

## Source

<https://www.geeksforgeeks.org/longest-path-undirected-tree/>

## Chapter 262

# Longest word in ternary search tree

Longest word in ternary search tree - GeeksforGeeks

Given a set of words represented in a ternary search tree, find the length of largest word among them.

Examples:

```
Input : {"Prakriti", "Raghav",
         "Rashi", "Sunidhi"}
Output : Length of largest word in
          ternary search tree is: 8
```

```
Input : {"Boats", "Boat", "But", "Best"}
Output : Length of largest word in
          ternary search tree is: 5
```

Prerequisite : [Ternary Search Tree](#)

The idea is to recursively search the max of left subtree, right subtree and equal tree. If the current character is same as the root's character increment with 1.

C

```
// C program to find the length of largest word
// in ternary search tree
#include <stdio.h>
#include <stdlib.h>
#define MAX 50
```

```
// A node of ternary search tree
struct Node
{
    char data;

    // True if this character is last
    // character of one of the words
    unsigned isEndOfString: 1;

    struct Node *left, *eq, *right;
};

// A utility function to create a new
// ternary search tree node
struct Node* newNode(char data)
{
    struct Node* temp =
        (struct Node*) malloc(sizeof( struct Node ));
    temp->data = data;
    temp->isEndOfString = 0;
    temp->left = temp->eq = temp->right = NULL;
    return temp;
}

// Function to insert a new word in a Ternary
// Search Tree
void insert(struct Node** root, char *word)
{
    // Base Case: Tree is empty
    if (!(*root))
        *root = newNode(*word);

    // If current character of word is smaller
    // than root's character, then insert this
    // word in left subtree of root
    if ((*word) < (*root)->data)
        insert(&(*root)->left), word);

    // If current character of word is greater
    // than root's character, then insert this
    // word in right subtree of root
    else if ((*word) > (*root)->data)
        insert(&(*root)->right), word);

    // If current character of word is same as
    // root's character,
    else
    {
```

```
    if (*(word+1))
        insert(&(*root)->eq ), word+1);

    // the last character of the word
    else
        (*root)->isEndOfString = 1;
}
}

// Function to find max of three numbers
int max(int a, int b, int c)
{
    int max;
    if (a >= b && a >= c)
        max = a;
    else if (b >= a && b >= c)
        max = b;
    else
        max = c;
}

// Function to find length of largest word in TST
int maxLengthTST(struct Node *root)
{
    if (root == NULL)
        return 0;
    return max(maxLengthTST(root->left),
               maxLengthTST(root->eq)+1,
               maxLengthTST(root->right));
}

// Driver program to test above functions
int main()
{
    struct Node *root = NULL;
    insert(&root, "Prakriti");
    insert(&root, "Raghav");
    insert(&root, "Rashi");
    insert(&root, "Sunidhi");
    int value = maxLengthTST(root);
    printf("Length of largest word in "
           "ternary search tree is: %d\n", value);

    return 0;
}
```

Java

```
// Java program to find the length of largest word
// in ternary search tree
public class GFG {

    static final int MAX = 50;

    // A node of ternary search tree
    static class Node
    {
        char data;

        // True if this character is last
        // character of one of the words
        int isEndOfString = 1;

        Node left, eq, right;

        // constructor
        Node(char data)
        {
            this.data = data;
            isEndOfString = 0;
            left = null;
            eq = null;
            right = null;
        }
    }

    // Function to insert a new word in a Ternary
    // Search Tree
    static Node insert(Node root, String word, int i)
    {
        // Base Case: Tree is empty
        if (root == null)
            root = new Node(word.charAt(i));

        // If current character of word is smaller
        // than root's character, then insert this
        // word in left subtree of root
        if (word.charAt(i) < root.data)
            root.left = insert(root.left, word, i);

        // If current character of word is greater
        // than root's character, then insert this
        // word in right subtree of root
        else if (word.charAt(i) > root.data)
            root.right = insert(root.right, word, i);
    }
}
```

```
// If current character of word is same as
// root's character,
else
{
    if (i + 1 < word.length())
        root.eq = insert(root.eq, word, i + 1);

    // the last character of the word
    else
        root.isEndOfString = 1;
}
return root;
}

// Function to find max of three numbers
static int max(int a, int b, int c)
{
    int max;
    if (a >= b && a >= c)
        max = a;
    else if (b >= a && b >= c)
        max = b;
    else
        max = c;
    return max;
}

// Function to find length of largest word in TST
static int maxLengthTST(Node root)
{
    if (root == null)
        return 0;
    return max(maxLengthTST(root.left),
               maxLengthTST(root.eq)+1,
               maxLengthTST(root.right));
}

// Driver program to test above functions
public static void main(String args[])
{
    Node root = null;
    root = insert(root, "Prakriti", 0);
    root = insert(root, "Raghav", 0);
    root = insert(root, "Rashi", 0);
    root = insert(root, "Sunidhi", 0);
    int value = maxLengthTST(root);
    System.out.println("Length of largest word in "+
```

```
        "ternary search tree is: "+ value);
    }
}
// This code is contributed by Sumit Ghosh
```

Output:

```
Length of largest word in ternary search tree is: 8
```

## Source

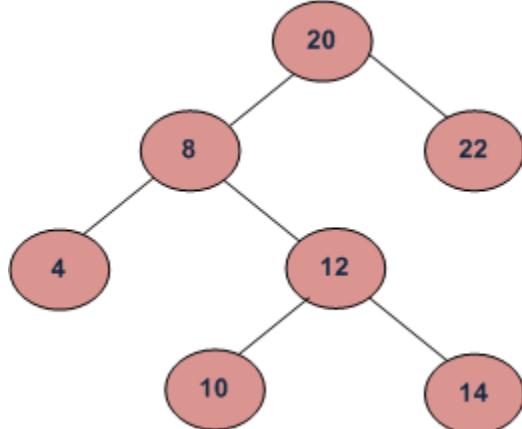
<https://www.geeksforgeeks.org/longest-word-ternary-search-tree/>

## Chapter 263

# Lowest Common Ancestor in Parent Array Representation

Lowest Common Ancestor in Parent Array Representation - GeeksforGeeks

Given a binary tree represented as parent array, find Lowest Common Ancestor between two nodes 'm' and 'n'.



In the above diagram, LCA of 10 and 14 is 12 and LCA of 10 and 12 is 12.

- (1) Make a parent array and store the parent of ith node in it. Parent of root node should be -1.
- (2) Now, access all the nodes from the desired node 'm' till root node and mark them visited.
- (3) Lastly, access all the nodes from the desired node 'n' till first visited node comes.
- (4) This node is the lowest common ancestor

```
// CPP program to find LCA in a tree represented
// as parent array.
#include <bits/stdc++.h>
using namespace std;
```

```
// Maximum value in a node
const int MAX = 1000;

// Function to find the Lowest common ancestor
int findLCA(int n1, int n2, int parent[])
{
    // Create a visited vector and mark
    // all nodes as not visited.
    vector<bool> visited(MAX, false);

    visited[n1] = true;

    // Moving from n1 node till root and
    // mark every accessed node as visited
    while (parent[n1] != -1) {
        visited[n1] = true;

        // Move to the parent of node n1
        n1 = parent[n1];
    }

    visited[n1] = true;

    // For second node finding the first
    // node common
    while (!visited[n2])
        n2 = parent[n2];

    return n2;
}

// Insert function for Binary tree
void insertAdj(int parent[], int i, int j)
{
    parent[i] = j;
}

// Driver Function
int main()
{
    // Maximum capacity of binary tree
    int parent[MAX];

    // Root marked
    parent[20] = -1;
    insertAdj(parent, 8, 20);
    insertAdj(parent, 22, 20);
```

```
insertAdj(parent, 4, 8);
insertAdj(parent, 12, 8);
insertAdj(parent, 10, 12);
insertAdj(parent, 14, 12);

cout << findLCA(10, 14, parent);

return 0;
}
```

**Output:**

12

**Time Complexity** – The time complexity of the above algorithm is  $O(\log n)$  as it requires  $O(\log n)$  time in searching.

**Source**

<https://www.geeksforgeeks.org/lowest-common-ancestor-in-parent-array-representation/>

## Chapter 264

# Lowest Common Ancestor in a Binary Search Tree.

Lowest Common Ancestor in a Binary Search Tree. - GeeksforGeeks

Given values of two values n1 and n2 in a Binary Search Tree, find the **Lowest Common Ancestor** (LCA). You may assume that both the values exist in the tree.

LCA of 10 and 14 is 12  
LCA of 14 and 8 is 8  
LCA of 10 and 22 is 20

**Following is definition of LCA from [Wikipedia](#):**

Let T be a rooted tree. The lowest common ancestor between two nodes n1 and n2 is defined as the lowest node in T that has both n1 and n2 as descendants (where we allow a node to be a descendant of itself).

The LCA of n1 and n2 in T is the shared ancestor of n1 and n2 that is located farthest from the root. Computation of lowest common ancestors may be useful, for instance, as part of a procedure for determining the distance between pairs of nodes in a tree: the distance from n1 to n2 can be computed as the distance from the root to n1, plus the distance from the root to n2, minus twice the distance from the root to their lowest common ancestor. (Source [Wiki](#))

If we are given a BST where every node has **parent pointer**, then LCA can be easily determined by traversing up using parent pointer and printing the first intersecting node.

We can solve this problem using BST properties. We can **recursively traverse** the BST from root. The main idea of the solution is, while traversing from top to bottom, the first node n we encounter with value between n1 and n2, i.e.,  $n_1 < n < n_2$  or same as one of the n1 or n2, is LCA of n1 and n2 (assuming that  $n_1 < n_2$ ). So just recursively traverse the BST in, if node's value is greater than both n1 and n2 then our LCA lies in left side of the

node, if it's is smaller than both n1 and n2, then LCA lies on right side. Otherwise root is LCA (assuming that both n1 and n2 are present in BST)

C

```
// A recursive C program to find LCA of two nodes n1 and n2.
#include <stdio.h>
#include <stdlib.h>

struct node
{
    int data;
    struct node* left, *right;
};

/* Function to find LCA of n1 and n2. The function assumes that both
   n1 and n2 are present in BST */
struct node *lca(struct node* root, int n1, int n2)
{
    if (root == NULL) return NULL;

    // If both n1 and n2 are smaller than root, then LCA lies in left
    if (root->data > n1 && root->data > n2)
        return lca(root->left, n1, n2);

    // If both n1 and n2 are greater than root, then LCA lies in right
    if (root->data < n1 && root->data < n2)
        return lca(root->right, n1, n2);

    return root;
}

/* Helper function that allocates a new node with the given data.*/
struct node* newNode(int data)
{
    struct node* node = (struct node*)malloc(sizeof(struct node));
    node->data = data;
    node->left = node->right = NULL;
    return(node);
}

/* Driver program to test lca() */
int main()
{
    // Let us construct the BST shown in the above figure
    struct node *root = newNode(20);
    root->left = newNode(8);
    root->right = newNode(22);
    root->left->left = newNode(4);
```

```
root->left->right      = newNode(12);
root->left->right->left  = newNode(10);
root->left->right->right = newNode(14);

int n1 = 10, n2 = 14;
struct node *t = lca(root, n1, n2);
printf("LCA of %d and %d is %d \n", n1, n2, t->data);

n1 = 14, n2 = 8;
t = lca(root, n1, n2);
printf("LCA of %d and %d is %d \n", n1, n2, t->data);

n1 = 10, n2 = 22;
t = lca(root, n1, n2);
printf("LCA of %d and %d is %d \n", n1, n2, t->data);

getchar();
return 0;
}
```

**Java**

```
// Recursive Java program to print lca of two nodes

// A binary tree node
class Node
{
    int data;
    Node left, right;

    Node(int item)
    {
        data = item;
        left = right = null;
    }
}

class BinaryTree
{
    Node root;

    /* Function to find LCA of n1 and n2. The function assumes that both
       n1 and n2 are present in BST */
    Node lca(Node node, int n1, int n2)
    {
        if (node == null)
            return null;
```

```
// If both n1 and n2 are smaller than root, then LCA lies in left
if (node.data > n1 && node.data > n2)
    return lca(node.left, n1, n2);

// If both n1 and n2 are greater than root, then LCA lies in right
if (node.data < n1 && node.data < n2)
    return lca(node.right, n1, n2);

return node;
}

/* Driver program to test lca() */
public static void main(String args[])
{
    // Let us construct the BST shown in the above figure
    BinaryTree tree = new BinaryTree();
    tree.root = new Node(20);
    tree.root.left = new Node(8);
    tree.root.right = new Node(22);
    tree.root.left.left = new Node(4);
    tree.root.left.right = new Node(12);
    tree.root.left.right.left = new Node(10);
    tree.root.left.right.right = new Node(14);

    int n1 = 10, n2 = 14;
    Node t = tree.lca(tree.root, n1, n2);
    System.out.println("LCA of " + n1 + " and " + n2 + " is " + t.data);

    n1 = 14;
    n2 = 8;
    t = tree.lca(tree.root, n1, n2);
    System.out.println("LCA of " + n1 + " and " + n2 + " is " + t.data);

    n1 = 10;
    n2 = 22;
    t = tree.lca(tree.root, n1, n2);
    System.out.println("LCA of " + n1 + " and " + n2 + " is " + t.data);

}

// This code has been contributed by Mayank Jaiswal
```

### Python

```
# A recursive python program to find LCA of two nodes
# n1 and n2
```

```
# A Binary tree node
class Node:

    # Constructor to create a new node
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

    # Function to find LCA of n1 and n2. The function assumes
    # that both n1 and n2 are present in BST
    def lca(self, n1, n2):

        # Base Case
        if root is None:
            return None

        # If both n1 and n2 are smaller than root, then LCA
        # lies in left
        if(root.data > n1 and root.data > n2):
            return lca(root.left, n1, n2)

        # If both n1 and n2 are greater than root, then LCA
        # lies in right
        if(root.data < n1 and root.data < n2):
            return lca(root.right, n1, n2)

        return root

# Driver program to test above function

# Let us construct the BST shown in the figure
root = Node(20)
root.left = Node(8)
root.right = Node(22)
root.left.left = Node(4)
root.left.right = Node(12)
root.left.right.left = Node(10)
root.left.right.right = Node(14)

n1 = 10 ; n2 = 14
t = lca(root, n1, n2)
print "LCA of %d and %d is %d" %(n1, n2, t.data)

n1 = 14 ; n2 = 8
t = lca(root, n1, n2)
print "LCA of %d and %d is %d" %(n1, n2, t.data)
```

```
n1 = 10 ; n2 = 22
t = lca(root, n1, n2)
print "LCA of %d and %d is %d" %(n1, n2, t.data)

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

Output:

```
LCA of 10 and 14 is 12
LCA of 14 and 8 is 8
LCA of 10 and 22 is 20
```

Time complexity of above solution is  $O(h)$  where  $h$  is height of tree. Also, the above solution requires  $O(h)$  extra space in function call stack for recursive function calls. We can avoid extra space using **iterative solution**.

```
/* Function to find LCA of n1 and n2. The function assumes that both
   n1 and n2 are present in BST */
struct node *lca(struct node* root, int n1, int n2)
{
    while (root != NULL)
    {
        // If both n1 and n2 are smaller than root, then LCA lies in left
        if (root->data > n1 && root->data > n2)
            root = root->left;

        // If both n1 and n2 are greater than root, then LCA lies in right
        else if (root->data < n1 && root->data < n2)
            root = root->right;

        else break;
    }
    return root;
}
```

See [this](#)for complete program.

You may like to see below articles as well :

[Lowest Common Ancestor in a Binary Tree](#)

[LCA using Parent Pointer](#)

[Find LCA in Binary Tree using RMQ](#)

#### **Exercise**

The above functions assume that n1 and n2 both are in BST. If n1 and n2 are not present, then they may return incorrect result. Extend the above solutions to return NULL if n1 or n2 or both not present in BST.

**Source**

<https://www.geeksforgeeks.org/lowest-common-ancestor-in-a-binary-search-tree/>

## Chapter 265

# Lowest Common Ancestor in a Binary Tree | Set 1

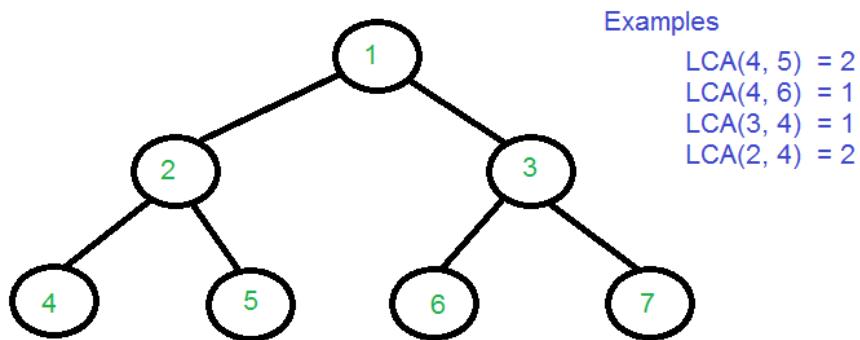
Lowest Common Ancestor in a Binary Tree | Set 1 - GeeksforGeeks

Given a binary tree (not a binary search tree) and two values say n1 and n2, write a program to find the least common ancestor.

*Following is definition of LCA from Wikipedia:*

Let T be a rooted tree. The lowest common ancestor between two nodes n1 and n2 is defined as the lowest node in T that has both n1 and n2 as descendants (where we allow a node to be a descendant of itself).

The LCA of n1 and n2 in T is the shared ancestor of n1 and n2 that is located farthest from the root. Computation of lowest common ancestors may be useful, for instance, as part of a procedure for determining the distance between pairs of nodes in a tree: the distance from n1 to n2 can be computed as the distance from the root to n1, plus the distance from the root to n2, minus twice the distance from the root to their lowest common ancestor. (Source [Wiki](#))



We have discussed an efficient solution to find [LCA in Binary Search Tree](#). In Binary Search Tree, using BST properties, we can find LCA in  $O(h)$  time where h is height of tree. Such

an implementation is not possible in Binary Tree as keys Binary Tree nodes don't follow any order. Following are different approaches to find LCA in Binary Tree.

**Method 1 (By Storing root to n1 and root to n2 paths):**

Following is simple O(n) algorithm to find LCA of n1 and n2.

- 1) Find path from root to n1 and store it in a vector or array.
- 2) Find path from root to n2 and store it in another vector or array.
- 3) Traverse both paths till the values in arrays are same. Return the common element just before the mismatch.

Following is the implementation of above algorithm.

C++

```
// C++ Program for Lowest Common Ancestor in a Binary Tree
// A O(n) solution to find LCA of two given values n1 and n2
#include <iostream>
#include <vector>

using namespace std;

// A Binary Tree node
struct Node
{
    int key;
    struct Node *left, *right;
};

// Utility function creates a new binary tree node with given key
Node * newNode(int k)
{
    Node *temp = new Node;
    temp->key = k;
    temp->left = temp->right = NULL;
    return temp;
}

// Finds the path from root node to given root of the tree, Stores the
// path in a vector path[], returns true if path exists otherwise false
bool findPath(Node *root, vector<int> &path, int k)
{
    // base case
    if (root == NULL) return false;

    // Store this node in path vector. The node will be removed if
    // not in path from root to k
    path.push_back(root->key);

    // See if the k is same as root's key
    if (root->key == k)
        return true;

    // If not present in current root, recur for left and right subtrees
    if (findPath(root->left, path, k))
        return true;
    if (findPath(root->right, path, k))
        return true;

    // If not present in subtree rooted with root, remove root from
    // path and return false
    path.pop_back();
    return false;
}
```

```

if (root->key == k)
    return true;

// Check if k is found in left or right sub-tree
if ( (root->left && findPath(root->left, path, k)) ||
     (root->right && findPath(root->right, path, k)) )
    return true;

// If not present in subtree rooted with root, remove root from
// path[] and return false
path.pop_back();
return false;
}

// Returns LCA if node n1, n2 are present in the given binary tree,
// otherwise return -1
int findLCA(Node *root, int n1, int n2)
{
    // to store paths to n1 and n2 from the root
    vector<int> path1, path2;

    // Find paths from root to n1 and root to n1. If either n1 or n2
    // is not present, return -1
    if ( !findPath(root, path1, n1) || !findPath(root, path2, n2) )
        return -1;

    /* Compare the paths to get the first different value */
    int i;
    for (i = 0; i < path1.size() && i < path2.size() ; i++)
        if (path1[i] != path2[i])
            break;
    return path1[i-1];
}

// Driver program to test above functions
int main()
{
    // Let us create the Binary Tree shown in above diagram.
    Node * root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->left = newNode(6);
    root->right->right = newNode(7);
    cout << "LCA(4, 5) = " << findLCA(root, 4, 5);
    cout << "nLCA(4, 6) = " << findLCA(root, 4, 6);
    cout << "nLCA(3, 4) = " << findLCA(root, 3, 4);
}

```

```
    cout << "nLCA(2, 4) = " << findLCA(root, 2, 4);
    return 0;
}
```

**Java**

```
// Java Program for Lowest Common Ancestor in a Binary Tree
// A O(n) solution to find LCA of two given values n1 and n2
import java.util.ArrayList;
import java.util.List;

// A Binary Tree node
class Node {
    int data;
    Node left, right;

    Node(int value) {
        data = value;
        left = right = null;
    }
}

public class BT_NoParentPtr_Solution1
{

    Node root;
    private List<Integer> path1 = new ArrayList<>();
    private List<Integer> path2 = new ArrayList<>();

    // Finds the path from root node to given root of the tree.
    int findLCA(int n1, int n2) {
        path1.clear();
        path2.clear();
        return findLCAInternal(root, n1, n2);
    }

    private int findLCAInternal(Node root, int n1, int n2) {

        if (!findPath(root, n1, path1) || !findPath(root, n2, path2)) {
            System.out.println((path1.size() > 0) ? "n1 is present" : "n1 is missing");
            System.out.println((path2.size() > 0) ? "n2 is present" : "n2 is missing");
            return -1;
        }

        int i;
        for (i = 0; i < path1.size() && i < path2.size(); i++) {

            // System.out.println(path1.get(i) + " " + path2.get(i));
        }
    }
}
```

```
        if (!path1.get(i).equals(path2.get(i)))
            break;
    }

    return path1.get(i-1);
}

// Finds the path from root node to given root of the tree, Stores the
// path in a vector path[], returns true if path exists otherwise false
private boolean findPath(Node root, int n, List<Integer> path)
{
    // base case
    if (root == null) {
        return false;
    }

    // Store this node . The node will be removed if
    // not in path from root to n.
    path.add(root.data);

    if (root.data == n) {
        return true;
    }

    if (root.left != null && findPath(root.left, n, path)) {
        return true;
    }

    if (root.right != null && findPath(root.right, n, path)) {
        return true;
    }

    // If not present in subtree rooted with root, remove root from
    // path[] and return false
    path.remove(path.size()-1);

    return false;
}

// Driver code
public static void main(String[] args)
{
    BT_NoParentPtr_Solution1 tree = new BT_NoParentPtr_Solution1();
    tree.root = new Node(1);
    tree.root.left = new Node(2);
    tree.root.right = new Node(3);
    tree.root.left.left = new Node(4);
    tree.root.left.right = new Node(5);
```

```
tree.root.right.left = new Node(6);
tree.root.right.right = new Node(7);

System.out.println("LCA(4, 5): " + tree.findLCA(4,5));
System.out.println("LCA(4, 6): " + tree.findLCA(4,6));
System.out.println("LCA(3, 4): " + tree.findLCA(3,4));
System.out.println("LCA(2, 4): " + tree.findLCA(2,4));

}

}

// This code is contributed by Sreenivasulu Rayanki.
```

### Python

```
# Python Program for Lowest Common Ancestor in a Binary Tree
# O(n) solution to find LCS of two given values n1 and n2

# A binary tree node
class Node:
    # Constructor to create a new binary node
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None

    # Finds the path from root node to given root of the tree.
    # Stores the path in a list path[], returns true if path
    # exists otherwise false
    def findPath( root, path, k):

        # Base Case
        if root is None:
            return False

        # Store this node in path vector. The node will be
        # removed if not in path from root to k
        path.append(root.key)

        # See if the k is same as root's key
        if root.key == k :
            return True

        # Check if k is found in left or right sub-tree
        if ((root.left != None and findPath(root.left, path, k)) or
            (root.right!= None and findPath(root.right, path, k))):
            return True

        # If not present in subtree rooted with root, remove
        path.pop()
```

```
# root from path and return False
path.pop()
return False

# Returns LCA if node n1 , n2 are present in the given
# binary tree otherwise return -1
def findLCA(root, n1, n2):

    # To store paths to n1 and n2 from the root
    path1 = []
    path2 = []

    # Find paths from root to n1 and root to n2.
    # If either n1 or n2 is not present , return -1
    if (not findPath(root, path1, n1) or not findPath(root, path2, n2)):
        return -1

    # Compare the paths to get the first different value
    i = 0
    while(i < len(path1) and i < len(path2)):
        if path1[i] != path2[i]:
            break
        i += 1
    return path1[i-1]

# Driver program to test above function
# Let's create the Binary Tree shown in above diagram
root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)
root.right.left = Node(6)
root.right.right = Node(7)

print "LCA(4, 5) = %d" %(findLCA(root, 4, 5))
print "LCA(4, 6) = %d" %(findLCA(root, 4, 6))
print "LCA(3, 4) = %d" %(findLCA(root, 3, 4))
print "LCA(2, 4) = %d" %(findLCA(root, 2, 4))

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

Output:

```
LCA(4, 5) = 2
LCA(4, 6) = 1
```

```
LCA(3, 4) = 1
LCA(2, 4) = 2
```

**Time Complexity:** Time complexity of the above solution is O(n). The tree is traversed twice, and then path arrays are compared.

Thanks to *Ravi Chandra Enaganti* for suggesting the initial solution based on this method.

### Method 2 (Using Single Traversal)

The method 1 finds LCA in O(n) time, but requires three tree traversals plus extra spaces for path arrays. If we assume that the keys n1 and n2 are present in Binary Tree, we can find LCA using single traversal of Binary Tree and without extra storage for path arrays. The idea is to traverse the tree starting from root. If any of the given keys (n1 and n2) matches with root, then root is LCA (assuming that both keys are present). If root doesn't match with any of the keys, we recur for left and right subtree. The node which has one key present in its left subtree and the other key present in right subtree is the LCA. If both keys lie in left subtree, then left subtree has LCA also, otherwise LCA lies in right subtree.

C++

```
/* C++ Program to find LCA of n1 and n2 using one traversal of Binary Tree */
#include <iostream>

using namespace std;

// A Binary Tree Node
struct Node
{
    struct Node *left, *right;
    int key;
};

// Utility function to create a new tree Node
Node* newNode(int key)
{
    Node *temp = new Node;
    temp->key = key;
    temp->left = temp->right = NULL;
    return temp;
}

// This function returns pointer to LCA of two given values n1 and n2.
// This function assumes that n1 and n2 are present in Binary Tree
struct Node *findLCA(struct Node* root, int n1, int n2)
{
    // Base case
    if (root == NULL) return NULL;

    // If either n1 or n2 matches with root's key, report
    // the presence by returning root (Note that if a key is
```

```
// ancestor of other, then the ancestor key becomes LCA
if (root->key == n1 || root->key == n2)
    return root;

// Look for keys in left and right subtrees
Node *left_lca = findLCA(root->left, n1, n2);
Node *right_lca = findLCA(root->right, n1, n2);

// If both of the above calls return Non-NULL, then one key
// is present in once subtree and other is present in other,
// So this node is the LCA
if (left_lca && right_lca) return root;

// Otherwise check if left subtree or right subtree is LCA
return (left_lca != NULL)? left_lca: right_lca;
}

// Driver program to test above functions
int main()
{
    // Let us create binary tree given in the above example
    Node * root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->left = newNode(6);
    root->right->right = newNode(7);
    cout << "LCA(4, 5) = " << findLCA(root, 4, 5)->key;
    cout << "nLCA(4, 6) = " << findLCA(root, 4, 6)->key;
    cout << "nLCA(3, 4) = " << findLCA(root, 3, 4)->key;
    cout << "nLCA(2, 4) = " << findLCA(root, 2, 4)->key;
    return 0;
}
```

### Java

```
//Java implementation to find lowest common ancestor of
// n1 and n2 using one traversal of binary tree

/* Class containing left and right child of current
node and key value*/
class Node
{
    int data;
    Node left, right;

    public Node(int item)
```

```
{  
    data = item;  
    left = right = null;  
}  
}  
  
public class BinaryTree  
{  
    //Root of the Binary Tree  
    Node root;  
  
    Node findLCA(int n1, int n2)  
    {  
        return findLCA(root, n1, n2);  
    }  
  
    // This function returns pointer to LCA of two given  
    // values n1 and n2. This function assumes that n1 and  
    // n2 are present in Binary Tree  
    Node findLCA(Node node, int n1, int n2)  
    {  
        // Base case  
        if (node == null)  
            return null;  
  
        // If either n1 or n2 matches with root's key, report  
        // the presence by returning root (Note that if a key is  
        // ancestor of other, then the ancestor key becomes LCA  
        if (node.data == n1 || node.data == n2)  
            return node;  
  
        // Look for keys in left and right subtrees  
        Node left_lca = findLCA(node.left, n1, n2);  
        Node right_lca = findLCA(node.right, n1, n2);  
  
        // If both of the above calls return Non-NULL, then one key  
        // is present in once subtree and other is present in other,  
        // So this node is the LCA  
        if (left_lca!=null && right_lca!=null)  
            return node;  
  
        // Otherwise check if left subtree or right subtree is LCA  
        return (left_lca != null) ? left_lca : right_lca;  
    }  
  
    /* Driver program to test above functions */  
    public static void main(String args[])  
    {
```

```
BinaryTree tree = new BinaryTree();
tree.root = new Node(1);
tree.root.left = new Node(2);
tree.root.right = new Node(3);
tree.root.left.left = new Node(4);
tree.root.left.right = new Node(5);
tree.root.right.left = new Node(6);
tree.root.right.right = new Node(7);
System.out.println("LCA(4, 5) = " +
                     tree.findLCA(4, 5).data);
System.out.println("LCA(4, 6) = " +
                     tree.findLCA(4, 6).data);
System.out.println("LCA(3, 4) = " +
                     tree.findLCA(3, 4).data);
System.out.println("LCA(2, 4) = " +
                     tree.findLCA(2, 4).data);
}
}
```

### Python

```
# Python program to find LCA of n1 and n2 using one
# traversal of Binary tree

# A binary tree node
class Node:

    # Constructor to create a new tree node
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None

    # This function returns pointer to LCA of two given
    # values n1 and n2
    # This function assumes that n1 and n2 are present in
    # Binary Tree
    def findLCA(root, n1, n2):

        # Base Case
        if root is None:
            return None

        # If either n1 or n2 matches with root's key, report
        # the presence by returning root (Note that if a key is
        # ancestor of other, then the ancestor key becomes LCA
        if root.key == n1 or root.key == n2:
            return root
```

```
# Look for keys in left and right subtrees
left_lca = findLCA(root.left, n1, n2)
right_lca = findLCA(root.right, n1, n2)

# If both of the above calls return Non-NULL, then one key
# is present in once subtree and other is present in other,
# So this node is the LCA
if left_lca and right_lca:
    return root

# Otherwise check if left subtree or right subtree is LCA
return left_lca if left_lca is not None else right_lca

# Driver program to test above function

# Let us create a binary tree given in the above example
root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)
root.right.left = Node(6)
root.right.right = Node(7)
print "LCA(4,5) = ", findLCA(root, 4, 5).key
print "LCA(4,6) = ", findLCA(root, 4, 6).key
print "LCA(3,4) = ", findLCA(root, 3, 4).key
print "LCA(2,4) = ", findLCA(root, 2, 4).key

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

Output:

```
LCA(4, 5) = 2
LCA(4, 6) = 1
LCA(3, 4) = 1
LCA(2, 4) = 2
```

Thanks to *Atul Singh* for suggesting this solution.

**Time Complexity:** Time complexity of the above solution is O(n) as the method does a simple tree traversal in bottom up fashion.

Note that the above method assumes that keys are present in Binary Tree. If one key is present and other is absent, then it returns the present key as LCA (Ideally should have returned NULL).

We can extend this method to handle all cases by passing two boolean variables v1 and v2.

v1 is set as true when n1 is present in tree and v2 is set as true if n2 is present in tree.

C++

```
/* Program to find LCA of n1 and n2 using one traversal of Binary Tree.
   It handles all cases even when n1 or n2 is not there in Binary Tree */
#include <iostream>
using namespace std;

// A Binary Tree Node
struct Node
{
    struct Node *left, *right;
    int key;
};

// Utility function to create a new tree Node
Node* newNode(int key)
{
    Node *temp = new Node;
    temp->key = key;
    temp->left = temp->right = NULL;
    return temp;
}

// This function returns pointer to LCA of two given values n1 and n2.
// v1 is set as true by this function if n1 is found
// v2 is set as true by this function if n2 is found
struct Node *findLCAUtil(struct Node* root, int n1, int n2, bool &v1, bool &v2)
{
    // Base case
    if (root == NULL) return NULL;

    // If either n1 or n2 matches with root's key, report the presence
    // by setting v1 or v2 as true and return root (Note that if a key
    // is ancestor of other, then the ancestor key becomes LCA)
    if (root->key == n1)
    {
        v1 = true;
        return root;
    }
    if (root->key == n2)
    {
        v2 = true;
        return root;
    }

    // Look for keys in left and right subtrees
    struct Node *lca = findLCAUtil(root->left, n1, n2, v1, v2);
    if (lca != NULL) return lca;

    lca = findLCAUtil(root->right, n1, n2, v1, v2);
    return lca;
}
```

```

Node *left_lca = findLCAUtil(root->left, n1, n2, v1, v2);
Node *right_lca = findLCAUtil(root->right, n1, n2, v1, v2);

// If both of the above calls return Non-NULL, then one key
// is present in once subtree and other is present in other,
// So this node is the LCA
if (left_lca && right_lca) return root;

// Otherwise check if left subtree or right subtree is LCA
return (left_lca != NULL)? left_lca: right_lca;
}

// Returns true if key k is present in tree rooted with root
bool find(Node *root, int k)
{
    // Base Case
    if (root == NULL)
        return false;

    // If key is present at root, or in left subtree or right subtree,
    // return true;
    if (root->key == k || find(root->left, k) || find(root->right, k))
        return true;

    // Else return false
    return false;
}

// This function returns LCA of n1 and n2 only if both n1 and n2 are present
// in tree, otherwise returns NULL;
Node *findLCA(Node *root, int n1, int n2)
{
    // Initialize n1 and n2 as not visited
    bool v1 = false, v2 = false;

    // Find lca of n1 and n2 using the technique discussed above
    Node *lca = findLCAUtil(root, n1, n2, v1, v2);

    // Return LCA only if both n1 and n2 are present in tree
    if (v1 && v2 || v1 && find(lca, n2) || v2 && find(lca, n1))
        return lca;

    // Else return NULL
    return NULL;
}

// Driver program to test above functions
int main()

```

```
{  
    // Let us create binary tree given in the above example  
    Node * root = newNode(1);  
    root->left = newNode(2);  
    root->right = newNode(3);  
    root->left->left = newNode(4);  
    root->left->right = newNode(5);  
    root->right->left = newNode(6);  
    root->right->right = newNode(7);  
    Node *lca = findLCA(root, 4, 5);  
    if (lca != NULL)  
        cout << "LCA(4, 5) = " << lca->key;  
    else  
        cout << "Keys are not present ";  
  
    lca = findLCA(root, 4, 10);  
    if (lca != NULL)  
        cout << "nLCA(4, 10) = " << lca->key;  
    else  
        cout << "nKeys are not present ";  
  
    return 0;  
}
```

### Java

```
// Java implementation to find lowest common ancestor of  
// n1 and n2 using one traversal of binary tree  
// It also handles cases even when n1 and n2 are not there in Tree  
  
/* Class containing left and right child of current node and key */  
class Node  
{  
    int data;  
    Node left, right;  
  
    public Node(int item)  
    {  
        data = item;  
        left = right = null;  
    }  
}  
  
public class BinaryTree  
{  
    // Root of the Binary Tree  
    Node root;  
    static boolean v1 = false, v2 = false;
```

```

// This function returns pointer to LCA of two given
// values n1 and n2.
// v1 is set as true by this function if n1 is found
// v2 is set as true by this function if n2 is found
Node findLCAUtil(Node node, int n1, int n2)
{
    // Base case
    if (node == null)
        return null;

    //Store result in temp, in case of key match so that we can search for other key also.
    Node temp=null;

    // If either n1 or n2 matches with root's key, report the presence
    // by setting v1 or v2 as true and return root (Note that if a key
    // is ancestor of other, then the ancestor key becomes LCA)
    if (node.data == n1)
    {
        v1 = true;
        temp = node;
    }
    if (node.data == n2)
    {
        v2 = true;
        temp = node;
    }

    // Look for keys in left and right subtrees
    Node left_lca = findLCAUtil(node.left, n1, n2);
    Node right_lca = findLCAUtil(node.right, n1, n2);

    if (temp != null)
        return temp;

    // If both of the above calls return Non-NULL, then one key
    // is present in once subtree and other is present in other,
    // So this node is the LCA
    if (left_lca != null && right_lca != null)
        return node;

    // Otherwise check if left subtree or right subtree is LCA
    return (left_lca != null) ? left_lca : right_lca;
}

// Finds lca of n1 and n2 under the subtree rooted with 'node'
Node findLCA(int n1, int n2)
{

```

```
// Initialize n1 and n2 as not visited
v1 = false;
v2 = false;

// Find lca of n1 and n2 using the technique discussed above
Node lca = findLCAUtil(root, n1, n2);

// Return LCA only if both n1 and n2 are present in tree
if (v1 && v2)
    return lca;

// Else return NULL
return null;
}

/* Driver program to test above functions */
public static void main(String args[])
{
    BinaryTree tree = new BinaryTree();
    tree.root = new Node(1);
    tree.root.left = new Node(2);
    tree.root.right = new Node(3);
    tree.root.left.left = new Node(4);
    tree.root.left.right = new Node(5);
    tree.root.right.left = new Node(6);
    tree.root.right.right = new Node(7);

    Node lca = tree.findLCA(4, 5);
    if (lca != null)
        System.out.println("LCA(4, 5) = " + lca.data);
    else
        System.out.println("Keys are not present");

    lca = tree.findLCA(4, 10);
    if (lca != null)
        System.out.println("LCA(4, 10) = " + lca.data);
    else
        System.out.println("Keys are not present");
}
}
```

### Python

```
""" Program to find LCA of n1 and n2 using one traversal of
Binary tree
It handles all cases even when n1 or n2 is not there in tree
"""
```

```

# A binary tree node
class Node:

    # Constructor to create a new node
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None

    # This function return pointer to LCA of two given values
    # n1 and n2
    # v1 is set as true by this function if n1 is found
    # v2 is set as true by this function if n2 is found
    def findLCAUtil(root, n1, n2, v):

        # Base Case
        if root is None:
            return None

        # IF either n1 or n2 matches ith root's key, report
        # the presence by setting v1 or v2 as true and return
        # root (Note that if a key is ancestor of other, then
        # the ancestor key becomes LCA)
        if root.key == n1 :
            v[0] = True
            return root

        if root.key == n2:
            v[1] = True
            return root

        # Look for keys in left and right subtree
        left_lca = findLCAUtil(root.left, n1, n2, v)
        right_lca = findLCAUtil(root.right, n1, n2, v)

        # If both of the above calls return Non-NULL, then one key
        # is present in once subtree and other is present in other,
        # So this node is the LCA
        if left_lca and right_lca:
            return root

        # Otherwise check if left subtree or right subtree is LCA
        return left_lca if left_lca is not None else right_lca

def find(root, k):

    # Base Case

```

```
if root is None:  
    return False  
  
# If key is present at root, or if left subtree or right  
# subtree , return true  
if (root.key == k or find(root.left, k) or  
    find(root.right, k)):  
    return True  
  
# Else return false  
return False  
  
# This function returns LCA of n1 and n2 onlue if both  
# n1 and n2 are present in tree, otherwise returns None  
def findLCA(root, n1, n2):  
  
    # Initialize n1 and n2 as not visited  
    v = [False, False]  
  
    # Find lac of n1 and n2 using the technique discussed above  
    lca = findLCAUtil(root, n1, n2, v)  
  
    # Returns LCA only if both n1 and n2 are present in tree  
    if (v[0] and v[1] or v[0] and find(lca, n2) or v[1] and  
        find(lca, n1)):  
        return lca  
  
    # Else return None  
    return None  
  
# Driver program to test above function  
root = Node(1)  
root.left = Node(2)  
root.right = Node(3)  
root.left.left = Node(4)  
root.left.right = Node(5)  
root.right.left = Node(6)  
root.right.right = Node(7)  
  
lca = findLCA(root, 4, 5)  
  
if lca is not None:  
    print "LCA(4, 5) = ", lca.key  
else :  
    print "Keys are not present"  
  
lca = findLCA(root, 4, 10)  
if lca is not None:
```

```
print "LCA(4,10) = ", lca.key
else:
    print "Keys are not present"

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

Output:

```
LCA(4, 5) = 2
Keys are not present
```

Thanks to Dhruv for suggesting this extended solution.

You may like to see below articles as well :

[LCA using Parent Pointer](#)

[Lowest Common Ancestor in a Binary Search Tree.](#)

[Find LCA in Binary Tree using RMQ](#)

**Improved By :** [Tushar Garg 6](#)

## Source

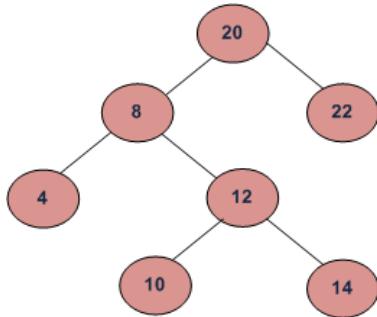
<https://www.geeksforgeeks.org/lowest-common-ancestor-binary-tree-set-1/>

## Chapter 266

# Lowest Common Ancestor in a Binary Tree | Set 2 (Using Parent Pointer)

Lowest Common Ancestor in a Binary Tree | Set 2 (Using Parent Pointer) - GeeksforGeeks

Given values of two nodes in a Binary Tree, find the **Lowest Common Ancestor** (LCA). It may be assumed that both nodes exist in the tree.



For example, consider the Binary Tree in diagram, LCA of 10 and 14 is 12 and LCA of 8 and 14 is 8.

Let T be a rooted tree. The lowest common ancestor between two nodes n1 and n2 is defined as the lowest node in T that has both n1 and n2 as descendants (where we allow a node to be a descendant of itself). Source : [Wikipedia](#).

We have discussed [different approaches to find LCA in set 1](#). Finding LCA becomes easy when parent pointer is given as we can easily find all ancestors of a node using parent pointer.

Below are steps to find LCA.

1. Create an empty hash table. </li>
2. Insert n1 and all of its ancestors in hash table.
3. Check if n2 or any of its ancestors exist in hash table, if yes return the first existing ancestor.

## Source

<https://www.geeksforgeeks.org/lowest-common-ancestor-in-a-binary-tree-set-2-using-parent-pointer/>

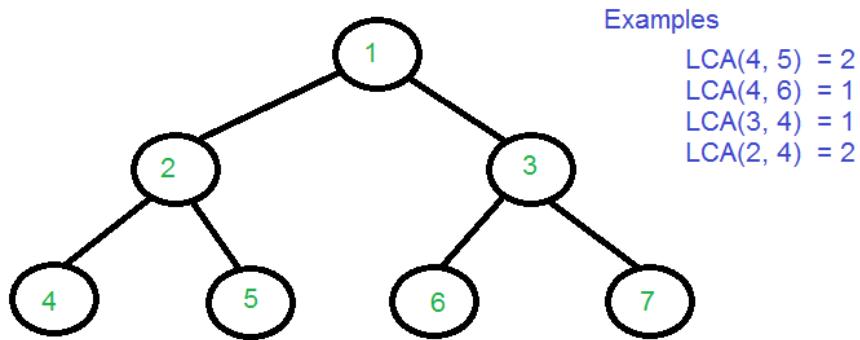
## Chapter 267

# Lowest Common Ancestor in a Binary Tree | Set 3 (Using RMQ)

Lowest Common Ancestor in a Binary Tree | Set 3 (Using RMQ) - GeeksforGeeks

Given a rooted tree, and two nodes which is in the tree, find the Lowest common ancestor of both the nodes. The LCA for two nodes u and v is defined as the farthest node from root that is ancestor to both u and v.

Prerequisites : [LCA | SET 1](#)



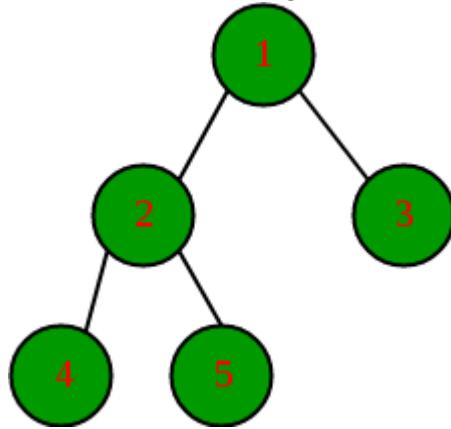
Example for above figure :

Input : 4 5  
Output : 2

Input : 4 7  
Output : 1

**Converting LCA to RMQ(Range Minimum Query):**

Take an array named  $E[]$ , which stores the order of dfs traversal i.e. the order in which the nodes are covered during the dfs traversal. For example,



The tree given above has dfs traversal in the order: 1-2-4-2-5-2-1-3.

Take another array  $L[]$ , in which  $L[i]$  is the level of node  $E[i]$ .

And the array  $H[]$ , which stores the index of first occurrence of  $i$ th node in the array  $E[]$ .

So, for the above tree,

$$\begin{aligned} E[] &= \{1, 2, 4, 2, 5, 2, 1, 3\} \\ L[] &= \{1, 2, 3, 2, 3, 2, 1, 2\} \\ H[] &= \{0, 1, 7, 2, 4\} \end{aligned}$$

Note that the arrays  $E$  and  $L$  are with zero-based indexing but the array  $H$  has one-based indexing.

Now, to find the  $\text{LCA}(4, 3)$ , first use the array  $H$  and find the indices at which 4 and 3 are found in  $E$  i.e.  $H[4]$  and  $H[3]$ . So, the indices comes out to be 2 and 7. Now, look at the subarray  $L[2 : 7]$ , and find the minimum in this subarray which is 1 (at the 6th index), and the corresponding element in the array  $E$  i.e.  $E[6]$  is the  $\text{LCA}(4, 3)$ .

To understand why this works, take  $\text{LCA}(4, 3)$  again. The path by which one can reach node 3 from node 4 is the subarray  $E[2 : 7]$ . And, if there is a node with lowest level in this path, then it can simply claimed to be the  $\text{LCA}(4, 3)$ .

Now, the problem is to find the minimum in the subarray  $E[H[u] \dots H[v]]$  (assuming that  $H[u] \geq H[v]$ ). And, that could be done using segment tree or sparse table. Below is the code using segment tree.

C++

```

// CPP code to find LCA of given
// two nodes in a tree
#include <algorithm>
#include <iostream>
#include <vector>
  
```

```
#define sz(x) x.size()
#define pb push_back
#define left 2 * i + 1
#define right 2 * i + 2
using namespace std;

const int maxn = 100005;

// the graph
vector<vector<int>> g(maxn);

// level of each node
int level[maxn];

vector<int> e;
vector<int> l;
int h[maxn];

// the segment tree
int st[5 * maxn];

// adding edges to the graph(tree)
void add_edge(int u, int v) {
    g[u].pb(v);
    g[v].pb(u);
}

// assigning level to nodes
void leveling(int src) {
    for (int i = 0; i < sz(g[src]); i++) {
        int des = g[src][i];
        if (!level[des]) {
            level[des] = level[src] + 1;
            leveling(des);
        }
    }
}

bool visited[maxn];

// storing the dfs traversal
// in the array e
void dfs(int src) {
    e.pb(src);
    visited[src] = 1;
    for (int i = 0; i < sz(g[src]); i++) {
        int des = g[src][i];
        if (!visited[des]) {
```

```

        dfs(des);
        e.pb(src);
    }
}

// making the array l
void setting_l(int n) {
    for (int i = 0; i < sz(e); i++)
        l.pb(level[e[i]]);
}

// making the array h
void setting_h(int n) {
    for (int i = 0; i <= n; i++)
        h[i] = -1;
    for (int i = 0; i < sz(e); i++) {
        // if is already stored
        if (h[e[i]] == -1)
            h[e[i]] = i;
    }
}

// Range minimum query to return the index
// of minimum in the subarray L[qs:qe]
int RMQ(int ss, int se, int qs, int qe, int i) {
    if (ss > se)
        return -1;

    // out of range
    if (se < qs || qe < ss)
        return -1;

    // in the range
    if (qs <= ss && se <= qe)
        return st[i];

    int mid = (ss + se) >> 1;
    int st = RMQ(ss, mid, qs, qe, left);
    int en = RMQ(mid + 1, se, qs, qe, right);

    if (st != -1 && en != -1) {
        if (l[st] < l=en])
            return st;
        return en;
    } else if (st != -1)
        return st;
    else if (en != -1)

```

```

        return en;
    }

    // constructs the segment tree
    void SegmentTreeConstruction(int ss, int se, int i) {
        if (ss > se)
            return;
        if (ss == se) // leaf
        {
            st[i] = ss;
            return;
        }
        int mid = (ss + se) >> 1;

        SegmentTreeConstruction(ss, mid, left);
        SegmentTreeConstruction(mid + 1, se, right);

        if (l[st[left]] < l[st[right]])
            st[i] = st[left];
        else
            st[i] = st[right];
    }

    // Function to get LCA
    int LCA(int x, int y) {
        if (h[x] > h[y])
            swap(x, y);
        return e[RMQ(0, sz(l) - 1, h[x], h[y], 0)];
    }

    // Driver code
    int main() {
        ios::sync_with_stdio(0);

        // n=number of nodes in the tree
        // q=number of queries to answer
        int n = 15, q = 5;

        // making the tree
        /*
                    1
                   / | \
                  2   3   4
                     |
                     5       6
                     / | \
                    8   7   9 (right of 5)
                     / | \   | \
                    10  11  12  13  14
        */
    }
}

```

```
10 11 12 13 14
|
15

*/
add_edge(1, 2);
add_edge(1, 3);
add_edge(1, 4);
add_edge(3, 5);
add_edge(4, 6);
add_edge(5, 7);
add_edge(5, 8);
add_edge(5, 9);
add_edge(7, 10);
add_edge(7, 11);
add_edge(7, 12);
add_edge(9, 13);
add_edge(9, 14);
add_edge(12, 15);

level[1] = 1;
leveling(1);

dfs(1);

setting_l(n);

setting_h(n);

SegmentTreeConstruction(0, sz(l) - 1, 0);

cout << LCA(10, 15) << endl;
cout << LCA(11, 14) << endl;

return 0;
}
```

**Output:**

```
7
5
```

**Time Complexity :**

The arrays defined are stored in  $O(n)$ . The segment tree construction also takes  $O(n)$  time. The LCA function calls the function RMQ which takes  $O(\log n)$  per query (as it uses the segment tree). So overall time complexity is  $O(n + q * \log n)$ .

**Source**

<https://www.geeksforgeeks.org/lowest-common-ancestor-in-a-binary-tree-set-3-using-rmq/>

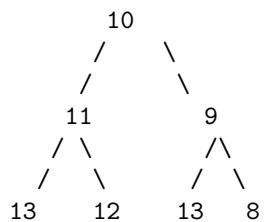
## Chapter 268

# Maximum Consecutive Increasing Path Length in Binary Tree

Maximum Consecutive Increasing Path Length in Binary Tree - GeeksforGeeks

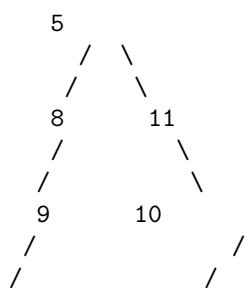
Given a Binary Tree find the length of the longest path which comprises of nodes with consecutive values in increasing order. Every node is considered as a path of length 1.

Examples:



Maximum Consecutive Path Length is 3 (10, 11, 12)

Note: 10, 9 ,8 is NOT considered since  
the nodes should be in increasing order.



```
6           15
Maximum Consecutive Path Length is 2 (8, 9).
```

Every node in the Binary Tree can either become part of the path which is starting from one of its parent node or a new path can start from this node itself. The key is to recursively find the path length for the left and right sub tree and then return the maximum. Some cases need to be considered while traversing the tree which are discussed below.

*prev* : stores the value of the parent node. Initialize *prev* with one less than value of root node so that the path starting at root can be of length at least 1.  
*len* : Stores the path length which ends at the parent of currently visited node.

**Case 1:** Value of Current Node is *prev* +1

In this case increase the path length by 1, and then recursively find the path length for the left and the right sub tree then return the maximum between two lengths.

**Case 2:** Value of Current Node is NOT *prev*+1

A new path can start from this node, so recursively find the path length for the left and the right sub tree. The path which ends at the parent node of current node might be greater than the path which starts from this node. So take the maximum of the path which starts from this node and which ends at previous node.

Below is C++ implementation of above idea.

C++

```
// C++ Program to find Maximum Consecutive
// Path Length in a Binary Tree
#include <iostream>
using namespace std;

// To represent a node of a Binary Tree
struct Node
{
    Node *left, *right;
    int val;
};

// Create a new Node and return its address
Node *newNode(int val)
{
    Node *temp = new Node();
    temp->val = val;
    temp->left = temp->right = NULL;
    return temp;
}

// Returns the maximum consecutive Path Length
int maxPathLenUtil(Node *root, int prev_val, int prev_len)
{
```

```
if (!root)
    return prev_len;

// Get the value of Current Node
// The value of the current node will be
// prev Node for its left and right children
int cur_val = root->val;

// If current node has to be a part of the
// consecutive path then it should be 1 greater
// than the value of the previous node
if (cur_val == prev_val+1)
{
    // a) Find the length of the Left Path
    // b) Find the length of the Right Path
    // Return the maximum of Left path and Right path
    return max(maxPathLenUtil(root->left, cur_val, prev_len+1),
               maxPathLenUtil(root->right, cur_val, prev_len+1));
}

// Find length of the maximum path under subtree rooted with this
// node (The path may or may not include this node)
int newPathLen = max(maxPathLenUtil(root->left, cur_val, 1),
                     maxPathLenUtil(root->right, cur_val, 1));

// Take the maximum previous path and path under subtree rooted
// with this node.
return max(prev_len, newPathLen);
}

// A wrapper over maxPathLenUtil().
int maxConsecutivePathLength(Node *root)
{
    // Return 0 if root is NULL
    if (root == NULL)
        return 0;

    // Else compute Maximum Consecutive Increasing Path
    // Length using maxPathLenUtil.
    return maxPathLenUtil(root, root->val-1, 0);
}

//Driver program to test above function
int main()
{
    Node *root = newNode(10);
    root->left = newNode(11);
```

```
root->right = newNode(9);
root->left->left = newNode(13);
root->left->right = newNode(12);
root->right->left = newNode(13);
root->right->right = newNode(8);

cout << "Maximum Consecutive Increasing Path Length is "
     << maxConsecutivePathLength(root);

return 0;
}
```

### Python

```
# Python program to find Maximum consecutive
# path length in binary tree

# A binary tree node
class Node:

    # Constructor to create a new node
    def __init__(self, val):
        self.val = val
        self.left = None
        self.right = None

    # Returns the maximum consecutive path length
    def maxPathLenUtil(self, prev_val, prev_len):
        if self is None:
            return prev_len

        # Get the value of current node
        # The value of the current node will be
        # prev node for its left and right children
        curr_val = self.val

        # If current node has to be a part of the
        # consecutive path then it should be 1 greater
        # than the value of the previous node
        if curr_val == prev_val + 1 :

            # a) Find the length of the left path
            # b) Find the length of the right path
            # Return the maximum of left path and right path
            return max(maxPathLenUtil(self.left, curr_val, prev_len+1),
                      maxPathLenUtil(self.right, curr_val, prev_len+1))

        # Find the length of the maximum path under subtree
```

```
# rooted with this node
newPathLen = max(maxPathLenUtil(root.left, curr_val, 1),
                  maxPathLenUtil(root.right, curr_val, 1))

# Take the maximum previous path and path under subtree
# rooted with this node
return max(prev_len , newPathLen)

# A Wrapper over maxPathLenUtil()
def maxConsecutivePathLength(root):

    # Return 0 if root is None
    if root is None:
        return 0

    # Else compute maximum consecutive increasing path
    # length using maxPathLenUtil
    return maxPathLenUtil(root, root.val - 1 , 0)

# Driver program to test above function
root = Node(10)
root.left = Node(11)
root.right = Node(9)
root.left.left = Node(13)
root.left.right = Node(12)
root.right.left = Node(13)
root.right.right = Node(8)

print "Maximum Consecutive Increasing Path Length is",
print maxConsecutivePathLength(root)

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

Output:

```
Maximum Consecutive Increasing Path Length is 3
```

## Source

<https://www.geeksforgeeks.org/maximum-consecutive-increasing-path-length-in-binary-tree/>

## Chapter 269

# Maximum Path Sum in a Binary Tree

Maximum Path Sum in a Binary Tree - GeeksforGeeks

Given a binary tree, find the maximum path sum. The path may start and end at any node in the tree.

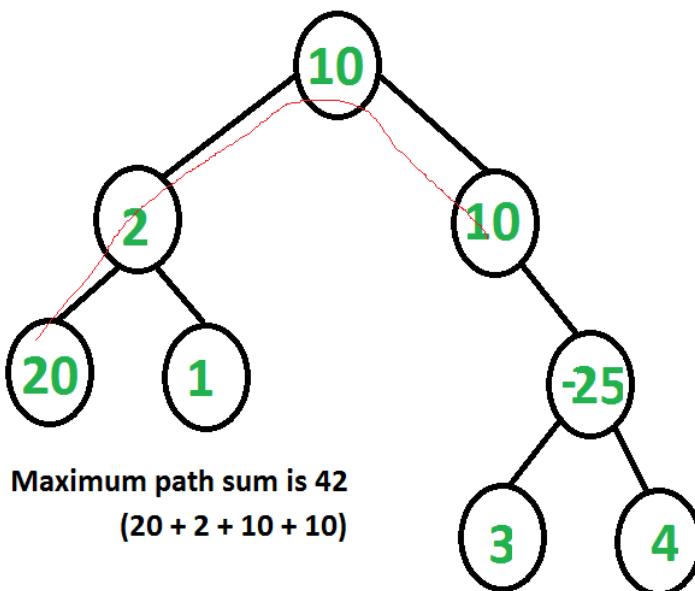
Example:

Input: Root of below tree



Output: 6

See below diagram for another example.  
1+2+3



For each node there can be four ways that the max path goes through the node:

1. Node only
2. Max path through Left Child + Node
3. Max path through Right Child + Node
4. Max path through Left Child + Node + Max path through Right Child

The idea is to keep trace of four paths and pick up the max one in the end. An important thing to note is, root of every subtree need to return maximum path sum such that at most one child of root is involved. This is needed for parent function call. In below code, this sum is stored in ‘max\_single’ and returned by the recursive function.

C++

```
// C/C++ program to find maximum path sum in Binary Tree
#include<bits/stdc++.h>
using namespace std;

// A binary tree node
struct Node
{
    int data;
    struct Node* left, *right;
};

// A utility function to allocate a new node
struct Node* newNode(int data)
{
    struct Node* newNode = new Node;
    newNode->data = data;
```

```
newNode->left = newNode->right = NULL;
return (newNode);
}

// This function returns overall maximum path sum in 'res'
// And returns max path sum going through root.
int findMaxUtil(Node* root, int &res)
{
    //Base Case
    if (root == NULL)
        return 0;

    // l and r store maximum path sum going through left and
    // right child of root respectively
    int l = findMaxUtil(root->left,res);
    int r = findMaxUtil(root->right,res);

    // Max path for parent call of root. This path must
    // include at-most one child of root
    int max_single = max(max(l, r) + root->data, root->data);

    // Max Top represents the sum when the Node under
    // consideration is the root of the maxsum path and no
    // ancestors of root are there in max sum path
    int max_top = max(max_single, l + r + root->data);

    res = max(res, max_top); // Store the Maximum Result.

    return max_single;
}

// Returns maximum path sum in tree with given root
int findMaxSum(Node *root)
{
    // Initialize result
    int res = INT_MIN;

    // Compute and return result
    findMaxUtil(root, res);
    return res;
}

// Driver program
int main(void)
{
    struct Node *root = newNode(10);
    root->left      = newNode(2);
    root->right     = newNode(10);
```

```
root->left->left = newNode(20);
root->left->right = newNode(1);
root->right->right = newNode(-25);
root->right->right->left = newNode(3);
root->right->right->right = newNode(4);
cout << "Max path sum is " << findMaxSum(root);
return 0;
}
```

**Java**

```
// Java program to find maximum path sum in Binary Tree

/* Class containing left and right child of current
node and key value*/
class Node {

    int data;
    Node left, right;

    public Node(int item) {
        data = item;
        left = right = null;
    }
}

// An object of Res is passed around so that the
// same value can be used by multiple recursive calls.
class Res {
    public int val;
}

class BinaryTree {

    // Root of the Binary Tree
    Node root;

    // This function returns overall maximum path sum in 'res'
    // And returns max path sum going through root.
    int findMaxUtil(Node node, Res res)
    {

        // Base Case
        if (node == null)
            return 0;

        // l and r store maximum path sum going through left and
        // right child of root respectively
```

```
int l = findMaxUtil(node.left, res);
int r = findMaxUtil(node.right, res);

// Max path for parent call of root. This path must
// include at-most one child of root
int max_single = Math.max(Math.max(l, r) + node.data,
                          node.data);

// Max Top represents the sum when the Node under
// consideration is the root of the maxsum path and no
// ancestors of root are there in max sum path
int max_top = Math.max(max_single, l + r + node.data);

// Store the Maximum Result.
res.val = Math.max(res.val, max_top);

return max_single;
}

int findMaxSum() {
    return findMaxSum(root);
}

// Returns maximum path sum in tree with given root
int findMaxSum(Node node) {

    // Initialize result
    // int res2 = Integer.MIN_VALUE;
    Res res = new Res();
    res.val = Integer.MIN_VALUE;

    // Compute and return result
    findMaxUtil(node, res);
    return res.val;
}

/* Driver program to test above functions */
public static void main(String args[]) {
    BinaryTree tree = new BinaryTree();
    tree.root = new Node(10);
    tree.root.left = new Node(2);
    tree.root.right = new Node(10);
    tree.root.left.left = new Node(20);
    tree.root.left.right = new Node(1);
    tree.root.right.right = new Node(-25);
    tree.root.right.right.left = new Node(3);
    tree.root.right.right.right = new Node(4);
```

```
        System.out.println("maximum path sum is : " +
                           tree.findMaxSum());
    }
}
```

### Python

```
# Python program to find maximum path sum in Binary Tree

# A Binary Tree Node
class Node:

    # Constructor to create a new node
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

    # This function returns overall maximum path sum in 'res'
    # And returns max path sum going through root
    def findMaxUtil(root):

        # Base Case
        if root is None:
            return 0

        # l and r store maximum path sum going through left
        # and right child of root respectively
        l = findMaxUtil(root.left)
        r = findMaxUtil(root.right)

        # Max path for parent call of root. This path
        # must include at most one child of root
        max_single = max(max(l, r) + root.data, root.data)

        # Max top represents the sum when the node under
        # consideration is the root of the maxSum path and
        # no ancestor of root are there in max sum path
        max_top = max(max_single, l+r+root.data)

        # Static variable to store the changes
        # Store the maximum result
        findMaxUtil.res = max(findMaxUtil.res, max_top)

        return max_single

    # Return maximum path sum in tree with given root
    def findMaxSum(root):
```

```
# Initialize result
findMaxUtil.res = float("-inf")

# Compute and return result
findMaxUtil(root)
return findMaxUtil.res

# Driver program
root = Node(10)
root.left = Node(2)
root.right = Node(10);
root.left.left = Node(20);
root.left.right = Node(1);
root.right.right = Node(-25);
root.right.right.left = Node(3);
root.right.right.right = Node(4);
print "Max path sum is " ,findMaxSum(root);

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

Output:

```
Max path sum is 42
```

Time Complexity: O(n) where n is number of nodes in Binary Tree.

This article is contributed by **Anmol Varshney** (FB Profile: <https://www.facebook.com/anmolvarshney695>). Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## Source

<https://www.geeksforgeeks.org/find-maximum-path-sum-in-a-binary-tree/>

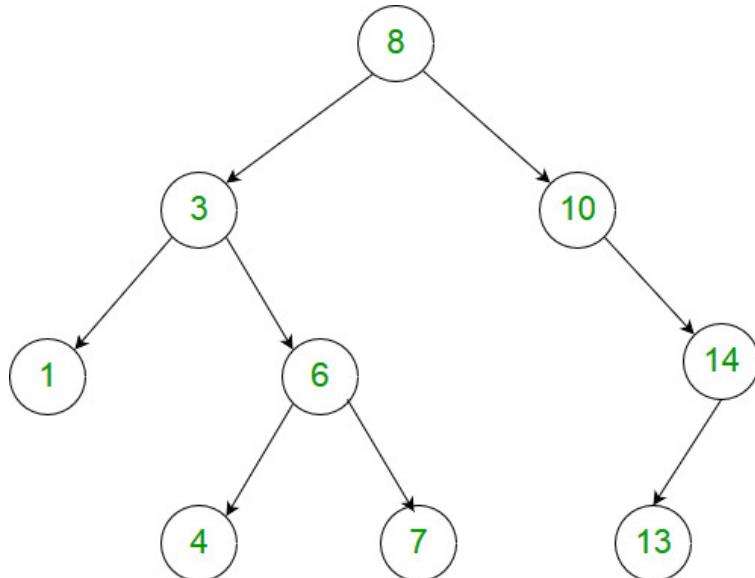
## Chapter 270

# Maximum difference between node and its ancestor in Binary Tree

Maximum difference between node and its ancestor in Binary Tree - GeeksforGeeks

Given a binary tree, we need to find maximum value we can get by subtracting value of node B from value of node A, where A and B are two nodes of the binary tree and A is an ancestor of B. Expected time complexity is O(n).

For example, consider below binary tree



We can have various ancestor-node difference, some of which are given below :  
 $8 - 3 = 5$

```
3 - 7 = -4
8 - 1 = 7
10 - 13 = -3
. . .
```

But among all those differences maximum value is 7 obtained by subtracting 1 from 8, which we need to return as result.

As we are given a binary tree, there is no relationship between node values so we need to traverse whole binary tree to get max difference and we can obtain the result in one traversal only by following below steps :

If we are at leaf node then just return its value because it can't be ancestor of any node. Then at each internal node we will try to get minimum value from left subtree and right subtree and calculate the difference between node value and this minimum value and according to that we will update the result.

As we are calculating minimum value while retuning in recurrence we will check all optimal possibilities (checking node value with minimum subtree value only) of differences and hence calculate the result in one traversal only.

Below is C++ implementation of above idea.

### C++

```
// C++ program to find maximum difference between node
// and its ancestor
#include <bits/stdc++.h>
using namespace std;

/* A binary tree node has key, pointer to left
   child and a pointer to right child */
struct Node
{
    int key;
    struct Node* left, *right;
};

/* To create a newNode of tree and return pointer */
struct Node* newNode(int key)
{
    Node* temp = new Node;
    temp->key = key;
    temp->left = temp->right = NULL;
    return (temp);
}

/* Recursive function to calculate maximum ancestor-node
   difference in binary tree. It updates value at 'res'
   to store the result. The returned value of this function
   is minimum value in subtree rooted with 't' */
```

```
int maxDiffUtil(Node* t, int *res)
{
    /* Returning Maximum int value if node is not
       there (one child case) */
    if (t == NULL)
        return INT_MAX;

    /* If leaf node then just return node's value */
    if (t->left == NULL && t->right == NULL)
        return t->key;

    /* Recursively calling left and right subtree
       for minimum value */
    int val = min(maxDiffUtil(t->left, res),
                  maxDiffUtil(t->right, res));

    /* Updating res if (node value - minimum value
       from subtree) is bigger than res */
    *res = max(*res, t->key - val);

    /* Returning minimum value got so far */
    return min(val, t->key);
}

/* This function mainly calls maxDiffUtil() */
int maxDiff(Node *root)
{
    // Initialising result with minimum int value
    int res = INT_MIN;

    maxDiffUtil(root, &res);

    return res;
}

/* Helper function to print inorder traversal of
   binary tree */
void inorder(Node* root)
{
    if (root)
    {
        inorder(root->left);
        printf("%d ", root->key);
        inorder(root->right);
    }
}

// Driver program to test above functions
```

```
int main()
{
    // Making above given diagram's binary tree
    Node* root;
    root = newNode(8);
    root->left = newNode(3);

    root->left->left = newNode(1);
    root->left->right = newNode(6);
    root->left->right->left = newNode(4);
    root->left->right->right = newNode(7);

    root->right = newNode(10);
    root->right->right = newNode(14);
    root->right->right->left = newNode(13);

    printf("Maximum difference between a node and"
          " its ancestor is : %d\n", maxDiff(root));
}
```

**Java**

```
/* Java program to find maximum difference between node
   and its ancestor */

// A binary tree node has key, pointer to left
// and right child
class Node
{
    int key;
    Node left, right;

    public Node(int key)
    {
        this.key = key;
        left = right = null;
    }
}

/* Class Res created to implement pass by reference
   of 'res' variable */
class Res
{
    int r = Integer.MIN_VALUE;
}

public class BinaryTree
{
```

```
Node root;

/* Recursive function to calculate maximum ancestor-node
   difference in binary tree. It updates value at 'res'
   to store the result. The returned value of this function
   is minimum value in subtree rooted with 't' */
int maxDiffUtil(Node t, Res res)
{
    /* Returning Maximum int value if node is not
       there (one child case) */
    if (t == null)
        return Integer.MAX_VALUE;

    /* If leaf node then just return node's value */
    if (t.left == null && t.right == null)
        return t.key;

    /* Recursively calling left and right subtree
       for minimum value */
    int val = Math.min(maxDiffUtil(t.left, res),
                       maxDiffUtil(t.right, res));

    /* Updating res if (node value - minimum value
       from subtree) is bigger than res */
    res.r = Math.max(res.r, t.key - val);

    /* Returning minimum value got so far */
    return Math.min(val, t.key);
}

/* This function mainly calls maxDiffUtil() */
int maxDiff(Node root)
{
    // Initialising result with minimum int value
    Res res = new Res();
    maxDiffUtil(root, res);

    return res.r;
}

/* Helper function to print inorder traversal of
   binary tree */
void inorder(Node root)
{
    if (root != null)
    {
        inorder(root.left);
        System.out.print(root.key + " ");
    }
}
```

```
        inorder(root.right);
    }
}

// Driver program to test the above functions
public static void main(String[] args)
{
    BinaryTree tree = new BinaryTree();

    // Making above given diagram's binary tree
    tree.root = new Node(8);
    tree.root.left = new Node(3);
    tree.root.left.left = new Node(1);
    tree.root.left.right = new Node(6);
    tree.root.left.right.left = new Node(4);
    tree.root.left.right.right = new Node(7);

    tree.root.right = new Node(10);
    tree.root.right.right = new Node(14);
    tree.root.right.right.left = new Node(13);

    System.out.println("Maximum difference between a node and"
                       + " its ancestor is : " + tree.maxDiff(tree.root));
}
}

// This code has been contributed by Mayank Jaiswal(mayank_24)
```

Output :

```
Maximum difference between a node and its ancestor is : 7
```

This article is contributed by Utkarsh Trivedi. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## Source

<https://www.geeksforgeeks.org/maximum-difference-between-node-and-its-ancestor-in-binary-tree/>

## Chapter 271

# Maximum edge removal from tree to make even forest

Maximum edge removal from tree to make even forest - GeeksforGeeks

Given an undirected tree which has even number of vertices, we need to remove the maximum number of edges from this tree such that each connected component of the resultant forest has an even number of vertices.

Examples:

In above shown tree, we can remove at max 2 edges 0-2 and 0-4 shown in red such that each connected component will have even number of vertices.

As we need connected components that have even number of vertices so when we get one component we can remove the edge that connects it to the remaining tree and we will be left with a tree with even number of vertices which will be the same problem but of smaller size, we have to repeat this algorithm until the remaining tree cannot be decomposed further in the above manner.

We will traverse the tree using [DFS](#) which will return the number of vertices in the component of which the current node is the root. If a node gets an even number of vertices from one of its children then the edge from that node to its child will be removed and result will be increased by one and if the returned number is odd then we will add it to the number of vertices that the component will have if the current node is the root of it.

- 1) Do DFS from any starting node as tree is connected.

- 2) Initialize count of nodes in subtree rooted under current node as 0.
- 3) Do following recursively for every subtree of current node.
  - a) If size of current subtree is even, increment result by 1 as we can disconnect the subtree.
  - b) Else add count of nodes in current subtree to current count.

Please see below code for better understanding,

```
/* Program to get maximum number of edges which
   can be removed such that each connected component
   of this tree will have an even number of vertices */
#include <bits/stdc++.h>
using namespace std;

// Utility method to do DFS of the graph and count edge
// deletion for even forest
int dfs(vector<int> g[], int u, bool visit[], int& res)
{
    visit[u] = true;
    int currComponentNode = 0;

    // iterate over all neighbor of node u
    for (int i = 0; i < g[u].size(); i++)
    {
        int v = g[u][i];
        if (!visit[v])
        {
            // Count the number of nodes in a subtree
            int subtreeNodeCount = dfs(g, v, visit, res);

            // if returned node count is even, disconnect
            // the subtree and increase result by one.
            if (subtreeNodeCount % 2 == 0)
                res++;

            // else add subtree nodes in current component
            else
                currComponentNode += subtreeNodeCount;
        }
    }

    // number of nodes in current component and one for

```

```
// current node
return (currComponentNode + 1);
}

/* method returns max edge that we can remove, after which
   each connected component will have even number of
   vertices */
int maxEdgeRemovalToMakeForestEven(vector<int> g[], int N)
{
    // Create a visited array for DFS and make all nodes
    // unvisited in starting
    bool visit[N + 1];
    for (int i = 0; i <= N; i++)
        visit[i] = false;

    int res = 0; // Passed as reference

    // calling the dfs from node-0
    dfs(g, 0, visit, res);

    return res;
}

// Utility function to add an undirected edge (u,v)
void addEdge(vector<int> g[], int u, int v)
{
    g[u].push_back(v);
    g[v].push_back(u);
}

// Driver code to test above methods
int main()
{
    int edges[] [2] = {{0, 2}, {0, 1}, {0, 4},
                      {2, 3}, {4, 5}, {5, 6},
                      {5, 7}};
    int N = sizeof(edges)/sizeof(edges[0]);
    vector<int> g[N + 1];
    for (int i = 0; i < N; i++)
        addEdge(g, edges[i][0], edges[i][1]);

    cout << maxEdgeRemovalToMakeForestEven(g, N);
    return 0;
}
```

Output:

2

Time Complexity :  $O(n)$  where  $n$  is number of nodes in tree.

### Source

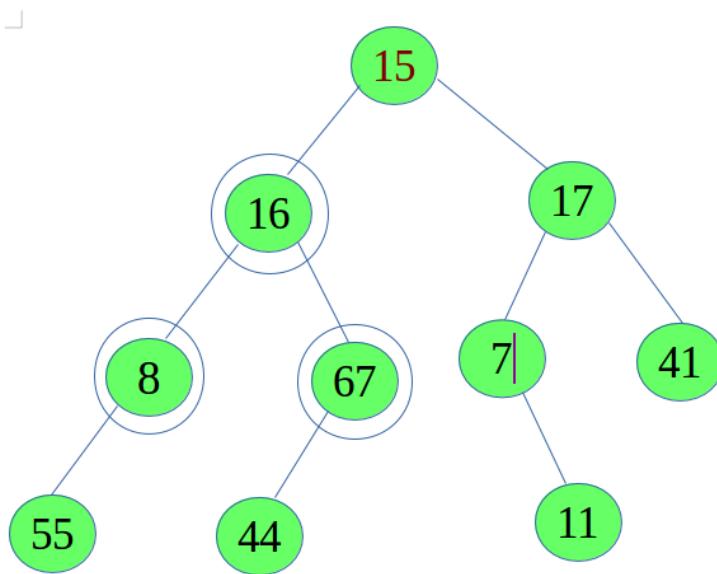
<https://www.geeksforgeeks.org/maximum-edge-removal-tree-make-even-forest/>

## Chapter 272

# Maximum parent children sum in Binary tree

Maximum parent children sum in Binary tree - GeeksforGeeks

Given a Binary Tree, find the maximum sum in a binary tree by adding the parent with its children. Exactly three Node needs to be added. If the tree does not have a node with both of its children as not NULL, return 0.



if we add all the node which are represented by a big circle then  
we get maximum sum in binary tree

We simply traverse the tree and find the Node that has the maximum sum. We need to take care of the leaves.

C++

```
// CPP program to find maximum sum of a node
// and its children
#include <iostream>
using namespace std;

struct Node {
    int data;
    struct Node *left, *right;
};

// insertion of Node in Tree
struct Node* newNode(int n)
{
    struct Node* root = new Node();
    root->data = n;
    root->left = root->right = NULL;
    return root;
}

int maxSum(struct Node* root)
{
    if (root == NULL)
        return 0;

    int res = maxSum(root->left);

    // if left and right link are null then
    // add all the three Node
    if (root->left != NULL && root->right != NULL) {
        int sum = root->data + root->left->data + root->right->data;
        res = max(res, sum);
    }

    return max(res, maxSum(root->right));
}

int main()
{
    struct Node* root = newNode(15);
    root->left = newNode(16);
    root->left->left = newNode(8);
    root->left->left->left = newNode(55);
    root->left->right = newNode(67);
```

```
root->left->right->left = newNode(44);
root->right = newNode(17);
root->right->left = newNode(7);
root->right->left->right = newNode(11);
root->right->right = newNode(41);
cout << maxSum(root);
return 0;
}
```

**Java**

```
// Java program to find
// maximum sum of a node
// and its children
import java.util.*;

// insertion of Node in Tree
class Node
{
    int data;
    Node left, right;

    Node(int key)
    {
        data = key;
        left = right = null;
    }
}
class GFG
{
    public static int maxSum(Node root)
    {
        if (root == null)
            return 0;

        int res = maxSum(root.left);

        // if left and right link are null
        // then add all the three Node
        if (root.left != null &&
            root.right != null)
        {
            int sum = root.data +
                      root.left.data +
                      root.right.data;
            res = Math.max(res, sum);
        }
    }
}
```

```
return Math.max(res, maxSum(root.right));
}

// Driver code
public static void main (String[] args)
{
    Node root = new Node(15);
    root.left = new Node(16);
    root.left.right = new Node(67);
    root.left.right.left = new Node(44);
    root.left.left = new Node(8);
    root.left.left.left = new Node(55);
    root.right = new Node(17);
    root.right.right = new Node(41);
    root.right.left = new Node(7);
    root.right.left.right = new Node(11);
    System.out.print(maxSum(root));
}
}

// This code is contributed
// by akash1295
```

**Output:**

91

Improved By : [akash1295](#)

**Source**

<https://www.geeksforgeeks.org/maximum-parent-children-sum-in-binary-tree/>

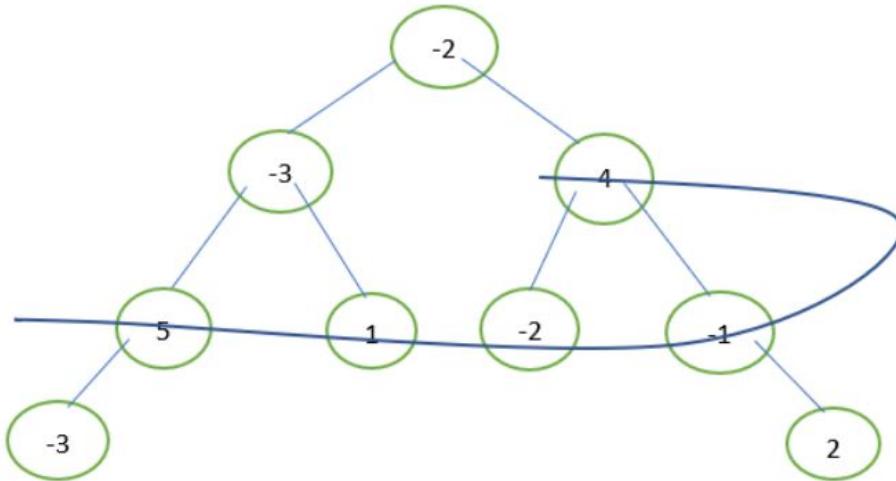
## Chapter 273

# Maximum spiral sum in Binary Tree

Maximum spiral sum in Binary Tree - GeeksforGeeks

Given a binary tree containing **n** nodes. The problem is to find the maximum sum obtained when the tree is spirally traversed. In spiral traversal one by one all levels are being traversed with the root level traversed from right to left, then next level from left to right, then further next level from right to left and so on.

Example:



$$\text{Maximum spiral sum} = 4 + (-1) + (-2) + 1 + 5 = 7$$

**Approach:** Obtain the [level order traversal in spiral form](#) of the given binary tree with the help of two stacks and store it in an array. Find the [maximum sum sub-array](#) of the array so obtained.

```
// C++ implementation to find maximum spiral sum
#include <bits/stdc++.h>

using namespace std;

// structure of a node of binary tree
struct Node {
    int data;
    Node *left, *right;
};

// A utility function to create a new node
Node* newNode(int data)
{
    // allocate space
    Node* node = new Node;

    // put in the data
    node->data = data;
    node->left = node->right = NULL;

    return node;
}

// function to find the maximum sum contiguous subarray.
// implements kadane's algorithm
int maxSum(vector<int> arr, int n)
{
    // to store the maximum value that is ending
    // up to the current index
    int max_ending_here = INT_MIN;

    // to store the maximum value encountered so far
    int max_so_far = INT_MIN;

    // traverse the array elements
    for (int i = 0; i < n; i++) {

        // if max_ending_here < 0, then it could
        // not possibly contribute to the maximum
        // sum further
        if (max_ending_here < 0)
            max_ending_here = arr[i];

        // else add the value arr[i] to max_ending_here
        else
            max_ending_here += arr[i];
    }
}
```

```

        // update max_so_far
        max_so_far = max(max_so_far, max_ending_here);
    }

    // required maximum sum contiguous subarray value
    return max_so_far;
}

// function to find maximum spiral sum
int maxSpiralSum(Node* root)
{
    // if tree is empty
    if (root == NULL)
        return 0;

    // Create two stacks to store alternate levels
    stack<Node*> s1; // For levels from right to left
    stack<Node*> s2; // For levels from left to right

    // vector to store spiral order traversal
    // of the binary tree
    vector<int> arr;

    // Push first level to first stack 's1'
    s1.push(root);

    // traversing tree in spiral form until
    // there are elements in any one of the
    // stacks
    while (!s1.empty() || !s2.empty()) {

        // traverse current level from s1 and
        // push nodes of next level to s2
        while (!s1.empty()) {
            Node* temp = s1.top();
            s1.pop();

            // push temp-data to 'arr'
            arr.push_back(temp->data);

            // Note that right is pushed before left
            if (temp->right)
                s2.push(temp->right);
            if (temp->left)
                s2.push(temp->left);
        }

        // traverse current level from s2 and
    }
}

```

```

// push nodes of next level to s1
while (!s2.empty()) {
    Node* temp = s2.top();
    s2.pop();

    // push temp-data to 'arr'
    arr.push_back(temp->data);

    // Note that left is pushed before right
    if (temp->left)
        s1.push(temp->left);
    if (temp->right)
        s1.push(temp->right);
}
}

// required maximum spiral sum
return maxSum(arr, arr.size());
}

// Driver program to test above
int main()
{
    Node* root = newNode(-2);
    root->left = newNode(-3);
    root->right = newNode(4);
    root->left->left = newNode(5);
    root->left->right = newNode(1);
    root->right->left = newNode(-2);
    root->right->right = newNode(-1);
    root->left->left->left = newNode(-3);
    root->right->right->right = newNode(2);

    cout << "Maximum Spiral Sum = "
        << maxSpiralSum(root);

    return 0;
}

```

Output:

Maximum Spiral Sum = 7

Time Complexity: O(n).  
 Auxiliary Space: O(n).

**Source**

<https://www.geeksforgeeks.org/maximum-spiral-sum-in-binary-tree/>

## Chapter 274

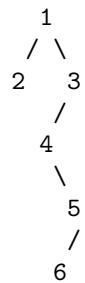
# Maximum sum from a tree with adjacent levels not allowed

Maximum sum from a tree with adjacent levels not allowed - GeeksforGeeks

Given a binary tree with positive integer values. Find the maximum sum of nodes such that we cannot pick two levels for computing sum

Examples:

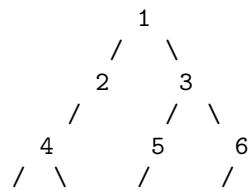
Input : Tree



Output :11

Explanation: Total items we can take: {1, 4, 6}  
or {2, 3, 5}. Max sum = 11.

Input : Tree



```
    17   18   19    30
   /     /   \
  11    12   13
Output :89
Explanation: Total items we can take: {2, 3, 17, 18,
19, 30} or {1, 4, 5, 6, 11, 12, 13}.
Max sum from first set = 89.
```

**Explanation:** We know that we need to get item values from alternate tree levels. This means that if we pick from level 1, the next pick would be from level 3, then level 5 and so on. Similarly, if we start from level 2, next pick will be from level 4, then level 6 and so on. So, we actually need to recursively sum all the grandchildren of a particular element as those are guaranteed to be at the alternate level.

We know for any node of tree, there are 4 grandchildren of it.

```
grandchild1 = root.left.left;
grandchild2 = root.left.right;
grandchild3 = root.right.left;
grandchild4 = root.right.right;
```

We can recursively call the `getSum()` method in the below program to find the sum of these children and their grandchildren. At the end, we just need to **return maximum sum obtained by starting at level 1 and starting at level 2**.

```
// Java code for max sum with adjacent levels
// not allowed
import java.util.*;

public class Main {

    // Tree node class for Binary Tree
    // representation
    static class Node {
        int data;
        Node left, right;
        Node(int item)
        {
            data = item;
            left = right = null;
        }
    }

    // Recursive function to find the maximum
    // sum returned for a root node and its
    // grandchildren
    public static int getSumAlternate(Node root)
```

```
{  
    if (root == null)  
        return 0;  
  
    int sum = root.data;  
    if (root.left != null) {  
        sum += getSum(root.left.left);  
        sum += getSum(root.left.right);  
    }  
  
    if (root.right != null) {  
        sum += getSum(root.right.left);  
        sum += getSum(root.right.right);  
    }  
    return sum;  
}  
  
// Returns maximum sum with adjacent  
// levels not allowed. This function  
// mainly uses getSumAlternate()  
public static int getSum(Node root)  
{  
    if (root == null)  
        return 0;  
  
    // We compute sum of alternate levels  
    // starting first level and from second  
    // level.  
    // And return maximum of two values.  
    return Math.max(getSumAlternate(root),  
                    (getSumAlternate(root.left) +  
                     getSumAlternate(root.right)));  
}  
  
// Driver function  
public static void main(String[] args)  
{  
    Node root = new Node(1);  
    root.left = new Node(2);  
    root.right = new Node(3);  
    root.right.left = new Node(4);  
    root.right.left.right = new Node(5);  
    root.right.left.right.left = new Node(6);  
    System.out.println(getSum(root));  
}  
}
```

Output:

11

Time Complexity :  $O(n)$

Exercise: Try printing the same solution for a n-ary Tree rather than a binary tree. The trick lies in the representation of the tree.

### **Source**

<https://www.geeksforgeeks.org/maximum-sum-tree-adjacent-levels-not-allowed/>

## Chapter 275

# Maximum sum of nodes in Binary tree such that no two are adjacent

Maximum sum of nodes in Binary tree such that no two are adjacent - GeeksforGeeks

Given a binary tree with a value associated with each node, we need to choose a subset of these nodes such that sum of chosen nodes is maximum under a constraint that no two chosen node in subset should be directly connected that is, if we have taken a node in our sum then we can't take its any children in consideration and vice versa.

Examples:

In above binary tree chosen nodes are encircled and are not directly connected and their sum is maximum possible.

### Method 1

We can solve this problem by considering the fact that both node and its children can't be in sum at same time, so when we take a node into our sum we will call recursively for its grandchildren or when we don't take this node we will call for all its children nodes and finally we will choose maximum from both of these results.

It can be seen easily that above approach can lead to solving same subproblem many times, for example in above diagram node 1 calls node 4 and 5 when its value is chosen and node 3 also calls them when its value is not chosen so these nodes are processed more than once. We can stop solving these nodes more than once by memoizing the result at all nodes.

In below code a map is used for memoizing the result which stores result of complete subtree rooted at a node in the map, so that if it is called again, the value is not calculated again

instead stored value from map is returned directly.  
Please see below code for better understanding.

```
// C++ program to find maximum sum from a subset of
// nodes of binary tree
#include <bits/stdc++.h>
using namespace std;

/* A binary tree node structure */
struct node
{
    int data;
    struct node *left, *right;
};

/* Utility function to create a new Binary Tree node */
struct node* newNode(int data)
{
    struct node *temp = new struct node;
    temp->data = data;
    temp->left = temp->right = NULL;
    return temp;
}

// Declaration of methods
int sumOfGrandChildren(node* node);
int getMaxSum(node* node);
int getMaxSumUtil(node* node, map<struct node*, int>& mp);

// method returns maximum sum possible from subtrees rooted
// at grandChildrens of node 'node'
int sumOfGrandChildren(node* node, map<struct node*, int>& mp)
{
    int sum = 0;

    // call for children of left child only if it is not NULL
    if (node->left)
        sum += getMaxSumUtil(node->left->left, mp) +
               getMaxSumUtil(node->left->right, mp);

    // call for children of right child only if it is not NULL
    if (node->right)
        sum += getMaxSumUtil(node->right->left, mp) +
               getMaxSumUtil(node->right->right, mp);

    return sum;
}
```

```
// Utility method to return maximum sum rooted at node 'node'
int getMaxSumUtil(node* node, map<struct node*, int>& mp)
{
    if (node == NULL)
        return 0;

    // If node is already processed then return calculated
    // value from map
    if (mp.find(node) != mp.end())
        return mp[node];

    // take current node value and call for all grand children
    int incl = node->data + sumOfGrandChildren(node, mp);

    // don't take current node value and call for all children
    int excl = getMaxSumUtil(node->left, mp) +
               getMaxSumUtil(node->right, mp);

    // choose maximum from both above calls and store that in map
    mp[node] = max(incl, excl);

    return mp[node];
}

// Returns maximum sum from subset of nodes
// of binary tree under given constraints
int getMaxSum(node* node)
{
    if (node == NULL)
        return 0;
    map<struct node*, int> mp;
    return getMaxSumUtil(node, mp);
}

// Driver code to test above methods
int main()
{
    node* root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->right->left = newNode(4);
    root->right->right = newNode(5);
    root->left->left = newNode(1);

    cout << getMaxSum(root) << endl;
    return 0;
}
```

Output:

11

### Method 2 (Using pair in STL)

Return a pair for each node in the binary tree such that first of the pair indicates maximum sum when the data of node is included and second indicates maximum sum when the data of a particular node is not included.

```
// C++ program to find maximum sum in Binary Tree
// such that no two nodes are adjacent.
#include<iostream>
using namespace std;

class Node
{
public:
    int data;
    Node* left, *right;
    Node(int data)
    {
        this->data = data;
        left = NULL;
        right = NULL;
    }
};

pair<int, int> maxSumHelper(Node *root)
{
    if (root==NULL)
    {
        pair<int, int> sum(0, 0);
        return sum;
    }
    pair<int, int> sum1 = maxSumHelper(root->left);
    pair<int, int> sum2 = maxSumHelper(root->right);
    pair<int, int> sum;

    // This node is included (Left and right children
    // are not included)
    sum.first = sum1.second + sum2.second + root->data;

    // This node is excluded (Either left or right
    // child is included)
    sum.second = max(sum1.first, sum2.first);
}
```

```
sum.second = max(sum1.first, sum1.second) +
             max(sum2.first, sum2.second);

    return sum;
}

int maxSum(Node *root)
{
    pair<int, int> res = maxSumHelper(root);
    return max(res.first, res.second);
}

// Driver code
int main()
{
    Node *root= new Node(10);
    root->left= new Node(1);
    root->left->left= new Node(2);
    root->left->left->left= new Node(1);
    root->left->left->right= new Node(3);
    root->left->right->left= new Node(4);
    root->left->right->right= new Node(5);
    cout << maxSum(root);
    return 0;
}
```

Output:

21

Time complexity O(n)

Thanks to Surbhi Rastogi for suggesting this method.

## Source

<https://www.geeksforgeeks.org/maximum-sum-nodes-binary-tree-no-two-adjacent/>

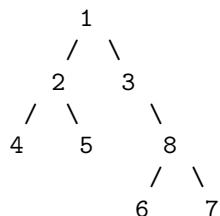
## Chapter 276

# Maximum width of a binary tree

Maximum width of a binary tree - GeeksforGeeks

Given a binary tree, write a function to get the maximum width of the given tree. Width of a tree is maximum of widths of all levels.

Let us consider the below example tree.



For the above tree,  
width of level 1 is 1,  
width of level 2 is 2,  
width of level 3 is 3  
width of level 4 is 2.

So the maximum width of the tree is 3.

### Method 1 (Using Level Order Traversal)

This method mainly involves two functions. One is to count nodes at a given level (getWidth), and other is to get the maximum width of the tree(getMaxWidth). getMaxWidth() makes use of getWidth() to get the width of all levels starting from root.

```
/*Function to print level order traversal of tree*/
getMaxWidth(tree)
maxWdth = 0
for i = 1 to height(tree)
    width = getWidth(tree, i);
    if(width > maxWdth)
        maxWdth = width
return maxWidth

/*Function to get width of a given level */
getWidth(tree, level)
if tree is NULL then return 0;
if level is 1, then return 1;
else if level greater than 1, then
    return getWidth(tree->left, level-1) +
    getWidth(tree->right, level-1);
```

## C

```
#include <stdio.h>
#include <stdlib.h>

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

/*Function prototypes/
int getWidth(struct node* root, int level);
int height(struct node* node);
struct node* newNode(int data);

/* Function to get the maximum width of a binary tree*/
int getMaxWidth(struct node* root)
{
    int maxWidth = 0;
    int width;
    int h = height(root);
    int i;

    /* Get width of each level and compare
       the width with maximum width so far */
```

```
for(i=1; i<=h; i++)
{
    width = getWidth(root, i);
    if(width > maxWidth)
        maxWidth = width;
}

return maxWidth;
}

/* Get width of a given level */
int getWidth(struct node* root, int level)
{
    if(root == NULL)
        return 0;

    if(level == 1)
        return 1;

    else if (level > 1)
        return getWidth(root->left, level-1) +
               getWidth(root->right, level-1);
}

/* UTILITY FUNCTIONS */
/* Compute the "height" of a tree -- the number of
   nodes along the longest path from the root node
   down to the farthest leaf node.*/
int height(struct node* node)
{
    if (node==NULL)
        return 0;
    else
    {
        /* compute the height of each subtree */
        int lHeight = height(node->left);
        int rHeight = height(node->right);
        /* use the larger one */

        return (lHeight > rHeight)? (lHeight+1): (rHeight+1);
    }
}

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newNode(int data)
{
```

```
struct node* newNode(int data)
{
    struct node* node = (struct node*)
                           malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;
    return(node);
}

/* Driver program to test above functions*/
int main()
{
    struct node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->right = newNode(8);
    root->right->right->left = newNode(6);
    root->right->right->right = newNode(7);

    /*
     Constructed binary tree is:
      1
     / \
    2   3
   / \   \
  4   5   8
      /   \
     6   7
    */
    printf("Maximum width is %d \n", getMaxWidth(root));
    getchar();
    return 0;
}
```

### Java

```
// Java program to calculate width of binary tree

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
class Node
{
    int data;
    Node left, right;

    Node(int item)
    {
        data = item;
```

```
        left = right = null;
    }
}

class BinaryTree
{
    Node root;

    /* Function to get the maximum width of a binary tree*/
    int getMaxWidth(Node node)
    {
        int maxWidth = 0;
        int width;
        int h = height(node);
        int i;

        /* Get width of each level and compare
           the width with maximum width so far */
        for (i = 1; i <= h; i++)
        {
            width = getWidth(node, i);
            if (width > maxWidth)
                maxWidth = width;
        }

        return maxWidth;
    }

    /* Get width of a given level */
    int getWidth(Node node, int level)
    {
        if (node == null)
            return 0;

        if (level == 1)
            return 1;
        else if (level > 1)
            return getWidth(node.left, level - 1)
                  + getWidth(node.right, level - 1);
        return 0;
    }

    /* UTILITY FUNCTIONS */

    /* Compute the "height" of a tree -- the number of
       nodes along the longest path from the root node
       down to the farthest leaf node.*/
    int height(Node node)
```

```
{  
    if (node == null)  
        return 0;  
    else  
    {  
        /* compute the height of each subtree */  
        int lHeight = height(node.left);  
        int rHeight = height(node.right);  
  
        /* use the larger one */  
        return (lHeight > rHeight) ? (lHeight + 1) : (rHeight + 1);  
    }  
}  
  
/* Driver program to test above functions */  
public static void main(String args[])  
{  
    BinaryTree tree = new BinaryTree();  
  
    /*  
     * Constructed binary tree is:  
     *  
     *      1  
     *     / \br/>     *    2   3  
     *   / \   \  
     *  4   5   8  
     *     / \ \  
     *    6   7  
     */  
    tree.root = new Node(1);  
    tree.root.left = new Node(2);  
    tree.root.right = new Node(3);  
    tree.root.left.left = new Node(4);  
    tree.root.left.right = new Node(5);  
    tree.root.right.right = new Node(8);  
    tree.root.right.right.left = new Node(6);  
    tree.root.right.right.right = new Node(7);  
  
    System.out.println("Maximum width is " + tree.getMaxWidth(tree.root));  
}  
}  
  
// This code has been contributed by Mayank Jaiswal
```

### Python

```
# Python program to find the maximum width of binary tree using Level Order Traversal.
```

```
# A binary tree node
class Node:

    # Constructor to create a new node
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

# Function to get the maximum width of a binary tree
def getMaxWidth(root):
    maxWidth = 0
    h = height(root)
    # Get width of each level and compare the width with maximum width so far
    for i in range(1,h+1):
        width = getWidth(root, i)
        if (width > maxWidth):
            maxWidth = width
    return maxWidth

# Get width of a given level
def getWidth(root,level):
    if root is None:
        return 0
    if level == 1:
        return 1
    elif level > 1:
        return (getWidth(root.left,level-1) + getWidth(root.right,level-1))

# UTILITY FUNCTIONS
# Compute the "height" of a tree -- the number of
# nodes along the longest path from the root node
# down to the farthest leaf node.
def height(node):
    if node is None:
        return 0
    else:

        # compute the height of each subtree
        lHeight = height(node.left)
        rHeight = height(node.right)

        # use the larger one
        return (lHeight+1) if (lHeight > rHeight) else (rHeight+1)

# Driver program to test above function
root = Node(1)
root.left = Node(2)
```

```

root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)
root.right.right = Node(8)
root.right.right.left = Node(6)
root.right.right.right = Node(7)

"""
Constructed binary tree is:
      1
     / \
    2   3
   / \   \
  4   5   8
     / \
    6   7
"""

print "Maximum width is %d" %(getWidth(root))

# This code is contributed by Naveen Aili

```

Time Complexity:  $O(n^2)$  in the worst case.

We can use Queue based level order traversal to optimize the time complexity of this method. The Queue based level order traversal will take  $O(n)$  time in worst case. Thanks to [Nitish](#), [DivyaCand](#) [tech.login.id2](#) for suggesting this optimization. See their comments for implementation using queue based traversal.

### Method 2 (Using Level Order Traversal with Queue)

In this method we store all the child nodes at the current level in the queue and then count the total number of nodes after the level order traversal for a particular level is completed. Since the queue now contains all the nodes of the next level, we can easily find out the total number of nodes in the next level by finding the size of queue. We then follow the same procedure for the successive levels. We store and update the maximum number of nodes found at each level.

C++

```

// A queue based C++ program to find maximum width
// of a Binary Tree
#include<bits/stdc++.h>
using namespace std ;

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct Node
{
    int data ;

```

```
struct Node * left ;
struct Node * right ;
};

// Function to find the maximum width of the tree
// using level order traversal
int maxWidth(struct Node * root)
{
    // Base case
    if (root == NULL)
        return 0;

    // Initialize result
    int result = 0;

    // Do Level order traversal keeping track of number
    // of nodes at every level.
    queue<Node*> q;
    q.push(root);
    while (!q.empty())
    {
        // Get the size of queue when the level order
        // traversal for one level finishes
        int count = q.size() ;

        // Update the maximum node count value
        result = max(count, result);

        // Iterate for all the nodes in the queue currently
        while (count--)
        {
            // Dequeue an node from queue
            Node *temp = q.front();
            q.pop();

            // Enqueue left and right children of
            // dequeued node
            if (temp->left != NULL)
                q.push(temp->left);
            if (temp->right != NULL)
                q.push(temp->right);
        }
    }

    return result;
}

/* Helper function that allocates a new node with the
```

```
given data and NULL left and right pointers. */
struct Node * newNode(int data)
{
    struct Node * node = new Node;
    node->data = data;
    node->left = node->right = NULL;
    return (node);
}

int main()
{
    struct Node *root = newNode(1);
    root->left      = newNode(2);
    root->right     = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->right = newNode(8);
    root->right->right->left = newNode(6);
    root->right->right->right = newNode(7);

    /* Constructed Binary tree is:
        1
       /   \
      2     3
     / \   \
    4   5   8
       /   \
      6   7   */

    cout << "Maximum width is "
        << maxWidth(root) << endl;
    return 0;
}

// This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

### Java

```
// Java program to calculate maximum width
// of a binary tree using queue
import java.util.LinkedList;
import java.util.Queue;

public class maxwidthusingqueue
{
    /* A binary tree node has data, pointer to
       left child and a pointer to right child */
    static class node
    {
```

```
int data;
node left, right;

public node(int data)
{
    this.data = data;
}
}

// Function to find the maximum width of
// the tree using level order traversal
static int maxwidth(node root)
{
    // Base case
    if (root == null)
        return 0;

    // Initialize result
    int maxwidth = 0;

    // Do Level order traversal keeping
    // track of number of nodes at every level
    Queue<node> q = new LinkedList<>();
    q.add(root);
    while (!q.isEmpty())
    {
        // Get the size of queue when the level order
        // traversal for one level finishes
        int count = q.size();

        // Update the maximum node count value
        maxwidth = Math.max(maxwidth, count);

        // Iterate for all the nodes in
        // the queue currently
        while (count-- > 0)
        {
            // Dequeue an node from queue
            node temp = q.remove();

            // Enqueue left and right children
            // of dequeued node
            if (temp.left != null)
            {
                q.add(temp.left);
            }
            if (temp.right != null)
            {
```

```
        q.add(temp.right);
    }
}
return maxwidth;
}

public static void main(String[] args)
{
    node root = new node(1);
    root.left = new node(2);
    root.right = new node(3);
    root.left.left = new node(4);
    root.left.right = new node(5);
    root.right.right = new node(8);
    root.right.right.left = new node(6);
    root.right.right.right = new node(7);

        /* Constructed Binary tree is:
           1
          / \
         2   3
        / \   \
       4   5   8
          / \
         6   7   */
System.out.println("Maximum width = " + maxwidth(root));
}
}

// This code is contributed by Rishabh Mahrsee
```

### Python

```
# Python program to find the maximum width of binary
# tree using Level Order Traversal with queue.

# A binary tree node
class Node:

    # Constructor to create a new node
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

    # Function to get the maximum width of a binary tree
```

```

def getMaxWidth(root):
    # base case
    if root is None:
        return 0
    q = []
    maxWidth = 0

    q.insert(0,root)

    while (q != []):
        # Get the size of queue when the level order
        # traversal for one level finishes
        count = len(q)

        # Update the maximum node count value
        maxWidth = max(count,maxWidth)

        while (count is not 0):
            count = count-1
            temp = q[-1]
            q.pop() ;
            if temp.left is not None:
                q.insert(0,temp.left)

            if temp.right is not None:
                q.insert(0,temp.right)

    return maxWidth

# Driver program to test above function
root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)
root.right.right = Node(8)
root.right.right.left = Node(6)
root.right.right.right = Node(7)

"""
Constructed bimary tree is:
      1
     / \
    2   3
   / \   \
  4   5   8
     / \
    6   7

```

```
"""
print "Maximum width is %d" %(getMaxWidth(root))

# This code is contributed by Naveen Aili
```

### Method 3 (Using Preorder Traversal)

In this method we create a temporary array count[] of size equal to the height of tree. We initialize all values in count as 0. We traverse the tree using preorder traversal and fill the entries in count so that the count array contains count of nodes at each level in Binary Tree.

C

```
#include <stdio.h>
#include <stdlib.h>

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

// A utility function to get height of a binary tree
int height(struct node* node);

// A utility function to allocate a new node with given data
struct node* newNode(int data);

// A utility function that returns maximum value in arr[] of size n
int getMax(int arr[], int n);

// A function that fills count array with count of nodes at every
// level of given binary tree
void getMaxWidthRecur(struct node *root, int count[], int level);

/* Function to get the maximum width of a binary tree*/
int getMaxWidth(struct node* root)
{
    int width;
    int h = height(root);

    // Create an array that will store count of nodes at each level
    int *count = (int *)calloc(sizeof(int), h);
```

```

int level = 0;

// Fill the count array using preorder traversal
getMaxWidthRecur(root, count, level);

// Return the maximum value from count array
return getMax(count, h);
}

// A function that fills count array with count of nodes at every
// level of given binary tree
void getMaxWidthRecur(struct node *root, int count[], int level)
{
    if(root)
    {
        count[level]++;
        getMaxWidthRecur(root->left, count, level+1);
        getMaxWidthRecur(root->right, count, level+1);
    }
}

/* UTILITY FUNCTIONS */
/* Compute the "height" of a tree -- the number of
   nodes along the longest path from the root node
   down to the farthest leaf node.*/
int height(struct node* node)
{
    if (node==NULL)
        return 0;
    else
    {
        /* compute the height of each subtree */
        int lHeight = height(node->left);
        int rHeight = height(node->right);
        /* use the larger one */

        return (lHeight > rHeight)? (lHeight+1): (rHeight+1);
    }
}

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* node = (struct node*)
                           malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
}

```

```
node->right = NULL;
return(node);
}

// Return the maximum value from count array
int getMax(int arr[], int n)
{
    int max = arr[0];
    int i;
    for (i = 0; i < n; i++)
    {
        if (arr[i] > max)
            max = arr[i];
    }
    return max;
}

/* Driver program to test above functions*/
int main()
{
    struct node *root = newNode(1);
    root->left      = newNode(2);
    root->right     = newNode(3);
    root->left->left  = newNode(4);
    root->left->right = newNode(5);
    root->right->right = newNode(8);
    root->right->right->left  = newNode(6);
    root->right->right->right = newNode(7);

    /*
     Constructed binary tree is:
          1
         / \
        2   3
       / \   \
      4   5   8
         / \
        6   7
    */
    printf("Maximum width is %d \n", getMaxWidth(root));
    getchar();
    return 0;
}
```

### Java

```
// Java program to calculate width of binary tree
```

```
/* A binary tree node has data, pointer to left child
   and a pointer to right child */
class Node
{
    int data;
    Node left, right;

    Node(int item)
    {
        data = item;
        left = right = null;
    }
}

class BinaryTree
{
    Node root;

    /* Function to get the maximum width of a binary tree*/
    int getMaxWidth(Node node)
    {
        int width;
        int h = height(node);

        // Create an array that will store count of nodes at each level
        int count[] = new int[10];

        int level = 0;

        // Fill the count array using preorder traversal
        getMaxWidthRecur(node, count, level);

        // Return the maximum value from count array
        return getMax(count, h);
    }

    // A function that fills count array with count of nodes at every
    // level of given binary tree
    void getMaxWidthRecur(Node node, int count[], int level)
    {
        if (node != null)
        {
            count[level]++;
            getMaxWidthRecur(node.left, count, level + 1);
            getMaxWidthRecur(node.right, count, level + 1);
        }
    }
}
```

```

/* UTILITY FUNCTIONS */

/* Compute the "height" of a tree -- the number of
   nodes along the longest path from the root node
   down to the farthest leaf node.*/
int height(Node node)
{
    if (node == null)
        return 0;
    else
    {
        /* compute the height of each subtree */
        int lHeight = height(node.left);
        int rHeight = height(node.right);

        /* use the larger one */
        return (lHeight > rHeight) ? (lHeight + 1) : (rHeight + 1);
    }
}

// Return the maximum value from count array
int getMax(int arr[], int n)
{
    int max = arr[0];
    int i;
    for (i = 0; i < n; i++)
    {
        if (arr[i] > max)
            max = arr[i];
    }
    return max;
}

/* Driver program to test above functions */
public static void main(String args[])
{
    BinaryTree tree = new BinaryTree();

    /*
     * Constructed binary tree is:
     *      1
     *     / \
     *    2   3
     *   / \   \
     *  4   5   8
     *   / \
     *  6   7 */
    tree.root = new Node(1);
}

```

```
tree.root.left = new Node(2);
tree.root.right = new Node(3);
tree.root.left.left = new Node(4);
tree.root.left.right = new Node(5);
tree.root.right.right = new Node(8);
tree.root.right.right.left = new Node(6);
tree.root.right.right.right = new Node(7);

System.out.println("Maximum width is " +
                    tree.getMaxWidth(tree.root));
}

}

// This code has been contributed by Mayank Jaiswal
```

### Python

```
# Python program to find the maximum width of binary tree using Preorder Traversal.

# A binary tree node
class Node:

    # Constructor to create a new node
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

    # Function to get the maximum width of a binary tree
def getMaxWidth(root):
    h = height(root)
    # Create an array that will store count of nodes at each level
    count = [0] * h

    level = 0
    # Fill the count array using preorder traversal
    getMaxWidthRecur(root, count, level)

    # Return the maximum value from count array
    return getMax(count,h)

# A function that fills count array with count of nodes at every
# level of given binary tree
def getMaxWidthRecur(root, count, level):
    if root is not None:
        count[level] += 1
        getMaxWidthRecur(root.left, count, level+1)
        getMaxWidthRecur(root.right, count, level+1)
```

```
# UTILITY FUNCTIONS
# Compute the "height" of a tree -- the number of
# nodes along the longest path from the root node
# down to the farthest leaf node.
def height(node):
    if node is None:
        return 0
    else:
        # compute the height of each subtree
        lHeight = height(node.left)
        rHeight = height(node.right)
        # use the larger one
        return (lHeight+1) if (lHeight > rHeight) else (rHeight+1)

# Return the maximum value from count array
def getMax(count, n):
    max = count[0]
    for i in range (1,n):
        if (count[i] > max):
            max = count[i]
    return max

# Driver program to test above function
root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)
root.right.right = Node(8)
root.right.right.left = Node(6)
root.right.right.right = Node(7)

"""
Constructed bimary tree is:
      1
     / \
    2   3
   / \   \
  4   5   8
     / \
    6   7

print "Maximum width is %d" %(getMaxWidth(root))

# This code is contributed by Naveen Aili
```

Thanks to [Rajaand jagdish](#)for suggesting this method.

Time Complexity:  $O(n)$

**Improved By :** [Ayush Jain 7398](#)

## Source

<https://www.geeksforgeeks.org/maximum-width-of-a-binary-tree/>

## Chapter 277

# Merge Sort Tree for Range Order Statistics

Merge Sort Tree for Range Order Statistics - GeeksforGeeks

Given an array of n numbers, the task is to answer the following queries:

```
kthSmallest(start, end, k) : Find the Kth smallest
                                number in the range from array
                                index 'start' to 'end'.
```

Examples:

```
Input : arr[] = {3, 2, 5, 1, 8, 9|
    Query 1: start = 2, end = 5, k = 2
    Query 2: start = 1, end = 6, k = 4
Output : 2
        5
```

Explanation:

[2, 5, 1, 8] represents the range from 2 to 5 and 2 is the 2nd smallest number in the range[3, 2, 5, 1, 8, 9] represents the range from 1 to 6 and 5 is the 4th smallest number in the range

The key idea is to build a [Segment Tree](#) with a vector at every node and the vector contains all the elements of the sub-range in a sorted order. And if we observe this segment tree structure this is somewhat similar to the tree formed during the [merge sort algorithm](#)(that is why it is called merge sort tree)

We use same implementation as discussed in [Merge Sort Tree \(Smaller or equal elements in given row range\)](#)

Firstly, we maintain a vector of pairs where each pair {value, index} is such that first element of pair represents the element of the input array and the second element of the pair represents the index at which it occurs.

Now we sort this vector of pairs on the basis of the first element of each pair.

After this we build a Merge Sort Tree where each node has a vector of indices in the sorted range.

When we have to answer a query we find if the  $K^{\text{th}}$  smallest number lies in the left sub-tree or in the right sub-tree. The idea is to use two binary searches and find the number of elements in the left sub-tree such that the indices lie within the given query range.

Let the number of such indices be  $M$ .

If  $M \geq K$ , it means we will be able to find the  $K^{\text{th}}$  smallest Number in the left sub-tree thus we call on the left sub-tree.

Else the  $K^{\text{th}}$  smallest number lies in the right sub-tree but this time we don't have to look for the  $K^{\text{th}}$  smallest number as we already have first  $M$  smallest numbers of the range in the left sub-tree thus we should look for the remaining part ie the  $(K-M)^{\text{th}}$  number in the right sub-tree.

This is the Index of  $K^{\text{th}}$  smallest number the value at this index is the required number.

```
// CPP program to implement k-th order statistics
#include <bits/stdc++.h>
using namespace std;

const int MAX = 1000;

// Constructs a segment tree and stores tree[]
void buildTree(int treeIndex, int l, int r,
               vector<pair<int, int>> a, vector<int> tree[])
{
    /* l => start of range,
       r => ending of a range
       treeIndex => index in the Segment Tree/Merge
       Sort Tree */
    /* leaf node */
    if (l == r) {
        tree[treeIndex].push_back(a[l].second);
        return;
    }

    int mid = (l + r) / 2;
```

```

/* building left subtree */
buildTree(2 * treeIndex, l, mid, a, tree);

/* building left subtree */
buildTree(2 * treeIndex + 1, mid + 1, r, a, tree);

/* merging left and right child in sorted order */
merge(tree[2 * treeIndex].begin(),
      tree[2 * treeIndex].end(),
      tree[2 * treeIndex + 1].begin(),
      tree[2 * treeIndex + 1].end(),
      back_inserter(tree[treeIndex]));
}

// Returns the Kth smallest number in query range
int queryRec(int segmentStart, int segmentEnd,
             int queryStart, int queryEnd, int treeIndex,
             int K, vector<int> tree[])
{
    /*
        segmentStart => start of a Segment,
        segmentEnd   => ending of a Segment,
        queryStart   => start of a query range,
        queryEnd     => ending of a query range,
        treeIndex     => index in the Segment
                           Tree/Merge Sort Tree,
        K            => kth smallest number to find */
}

if (segmentStart == segmentEnd)
    return tree[treeIndex][0];

int mid = (segmentStart + segmentEnd) / 2;

// finds the last index in the segment
// which is <= queryEnd
int last_in_query_range =
    (upper_bound(tree[2 * treeIndex].begin(),
                tree[2 * treeIndex].end(),
                queryEnd)
     - tree[2 * treeIndex].begin());

// finds the first index in the segment
// which is >= queryStart
int first_in_query_range =
    (lower_bound(tree[2 * treeIndex].begin(),
                tree[2 * treeIndex].end(),
                queryStart)
     - tree[2 * treeIndex].begin());

```

```
int M = last_in_query_range - first_in_query_range;

if (M >= K) {

    // Kth smallest is in left subtree,
    // so recursively call left subtree for Kth
    // smallest number
    return queryRec(segmentStart, mid, queryStart,
                    queryEnd, 2 * treeIndex, K, tree);
}

else {

    // Kth smallest is in right subtree,
    // so recursively call right subtree for the
    // (K-M)th smallest number
    return queryRec(mid + 1, segmentEnd, queryStart,
                    queryEnd, 2 * treeIndex + 1, K - M, tree);
}

// A wrapper over query()
int query(int queryStart, int queryEnd, int K, int n,
          vector<pair<int, int> > a, vector<int> tree[])
{

    return queryRec(0, n - 1, queryStart - 1, queryEnd - 1,
                   1, K, tree);
}

// Driver code
int main()
{
    int arr[] = { 3, 2, 5, 1, 8, 9 };
    int n = sizeof(arr)/sizeof(arr[0]);

    // vector of pairs of form {element, index}
    vector<pair<int, int> > v;
    for (int i = 0; i < n; i++) {
        v.push_back(make_pair(arr[i], i));
    }

    // sort the vector
    sort(v.begin(), v.end());

    // Construct segment tree in tree[]
    vector<int> tree[MAX];
```

```
buildTree(1, 0, n - 1, v, tree);

// Answer queries
// kSmallestIndex hold the index of the kth smallest number
int kSmallestIndex = query(2, 5, 2, n, v, tree);
cout << arr[kSmallestIndex] << endl;

kSmallestIndex = query(1, 6, 4, n, v, tree);
cout << arr[kSmallestIndex] << endl;

return 0;
}
```

**Output:**

```
2
5
```

Thus, we can get the  $K^{th}$  smallest number query in range L to R, in  $O(n(\log n)^2)$  by building the merge sort tree on indices.

**Source**

<https://www.geeksforgeeks.org/merge-sort-tree-for-range-order-statistics/>

## Chapter 278

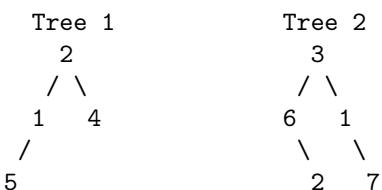
# Merge Two Binary Trees by doing Node Sum (Recursive and Iterative)

Merge Two Binary Trees by doing Node Sum (Recursive and Iterative) - GeeksforGeeks

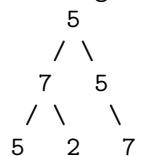
Given two binary trees. We need to merge them into a new binary tree. The merge rule is that if two nodes overlap, then sum node values up as the new value of the merged node. Otherwise, the non-null node will be used as the node of new tree.

Example:

**Input:**



**Output: Merged tree:**



Note: The merging process must start from the root nodes of both trees.

**Recursive Algorithm:**

1. Traverse the tree in Preorder fashion
2. Check if both the tree nodes are NULL
  - (a) If not, then update the value
3. Recur for left subtrees
4. Recur for right subtrees
5. Return root of updated Tree

C++

```
// C++ program to Merge Two Binary Trees
#include <bits/stdc++.h>
using namespace std;

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct Node
{
    int data;
    struct Node *left, *right;
};

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
Node *newNode(int data)
{
    Node *new_node = new Node;
    new_node->data = data;
    new_node->left = new_node->right = NULL;
    return new_node;
}

/* Given a binary tree, print its nodes in inorder*/
void inorder(Node * node)
{
    if (!node)
        return;

    /* first recur on left child */
    inorder(node->left);

    /* then print the data of node */
    printf("%d ", node->data);

    /* now recur on right child */
    inorder(node->right);
}
```

```
/* Function to merge given two binary trees*/
Node *MergeTrees(Node * t1, Node * t2)
{
    if (!t1)
        return t2;
    if (!t2)
        return t1;
    t1->data += t2->data;
    t1->left = MergeTrees(t1->left, t2->left);
    t1->right = MergeTrees(t1->right, t2->right);
    return t1;
}

// Driver code
int main()
{
    /* Let us construct the first Binary Tree
       1
      /   \
     2     3
    / \   /
   4   5   6
    */
    Node *root1 = newNode(1);
    root1->left = newNode(2);
    root1->right = newNode(3);
    root1->left->left = newNode(4);
    root1->left->right = newNode(5);
    root1->right->right = newNode(6);

    /* Let us construct the second Binary Tree
       4
      /   \
     1     7
    /   /   \
   3   2   6   */
    Node *root2 = newNode(4);
    root2->left = newNode(1);
    root2->right = newNode(7);
    root2->left->left = newNode(3);
    root2->right->left = newNode(2);
    root2->right->right = newNode(6);

    Node *root3 = MergeTrees(root1, root2);
    printf("The Merged Binary Tree is:\n");
    inorder(root3);
    return 0;
}
```

}

**Java**

```
// Java program to Merge Two Binary Trees

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
class Node
{
    int data;
    Node left, right;

    public Node(int data, Node left, Node right) {
        this.data = data;
        this.left = left;
        this.right = right;
    }

    /* Helper method that allocates a new node with the
       given data and NULL left and right pointers. */
    static Node newNode(int data)
    {
        return new Node(data, null, null);
    }

    /* Given a binary tree, print its nodes in inorder*/
    static void inorder(Node node)
    {
        if (node == null)
            return;

        /* first recur on left child */
        inorder(node.left);

        /* then print the data of node */
        System.out.printf("%d ", node.data);

        /* now recur on right child */
        inorder(node.right);
    }

    /* Method to merge given two binary trees*/
    static Node MergeTrees(Node t1, Node t2)
    {
        if (t1 == null)
            return t2;
        if (t2 == null)
```

```
        return t1;
    t1.data += t2.data;
    t1.left = MergeTrees(t1.left, t2.left);
    t1.right = MergeTrees(t1.right, t2.right);
    return t1;
}

// Driver method
public static void main(String[] args)
{
    /* Let us construct the first Binary Tree
        1
       /   \
      2     3
     / \     \
    4   5     6
    */
    Node root1 = newNode(1);
    root1.left = newNode(2);
    root1.right = newNode(3);
    root1.left.left = newNode(4);
    root1.left.right = newNode(5);
    root1.right.right = newNode(6);

    /* Let us construct the second Binary Tree
        4
       /   \
      1     7
     /   /   \
    3   2   6   */
    Node root2 = newNode(4);
    root2.left = newNode(1);
    root2.right = newNode(7);
    root2.left.left = newNode(3);
    root2.right.left = newNode(2);
    root2.right.right = newNode(6);

    Node root3 = MergeTrees(root1, root2);
    System.out.printf("The Merged Binary Tree is:\n");
    inorder(root3);
}

// This code is contributed by Gaurav Miglani
```

Output:

The Merged Binary Tree is:

7 3 5 5 2 10 12

### Complexity Analysis:

- Time complexity :  $O(n)$   
A total of  $n$  nodes need to be traversed. Here,  $n$  represents the minimum number of nodes from the two given trees.
- Auxiliary Space :  $O(n)$   
The depth of the recursion tree can go upto  $n$  in case of a skewed tree. In average case, depth will be  $O(\log n)$ .

### Iterative Algorithm:

1. Create a stack
2. Push the root nodes of both the trees onto the stack.
3. While the stack is not empty, perform following steps :
  - (a) Pop a node pair from the top of the stack
  - (b) For every node pair removed, add the values corresponding to the two nodes and update the value of the corresponding node in the first tree
  - (c) If the left child of the first tree exists, push the left child(pair) of both the trees onto the stack.
  - (d) If the left child of the first tree doesn't exist, append the left child of the second tree to the current node of the first tree
  - (e) Do same for right child pair as well.
  - (f) If both the current nodes are NULL, continue with popping the next nodes from the stack.
4. Return root of updated Tree

```
// C++ program to Merge Two Binary Trees
#include <bits/stdc++.h>
using namespace std;

/* A binary tree node has data, pointer to left child
and a pointer to right child */
struct Node
{
    int data;
    struct Node *left, *right;
};

// Structure to store node pair onto stack
struct snode
{
    Node *l, *r;
};
```

```
/* Helper function that allocates a new node with the
given data and NULL left and right pointers. */
Node *newNode(int data)
{
    Node *new_node = new Node;
    new_node->data = data;
    new_node->left = new_node->right = NULL;
    return new_node;
}

/* Given a binary tree, print its nodes in inorder*/
void inorder(Node * node)
{
    if (! node)
        return;

    /* first recur on left child */
    inorder(node->left);

    /* then print the data of node */
    printf("%d ", node->data);

    /* now recur on right child */
    inorder(node->right);
}

/* Function to merge given two binary trees*/

Node* MergeTrees(Node* t1, Node* t2)
{
    if (! t1)
        return t2;
    if (! t2)
        return t1;
    stack<snode> s;
    snode temp;
    temp.l = t1;
    temp.r = t2;
    s.push(temp);
    snode n;
    while (! s.empty())
    {
        n = s.top();
        s.pop();
        if (n.l == NULL || n.r == NULL)
            continue;
        n.l->data += n.r->data;
    }
}
```

```
if (n.l->left == NULL)
    n.l->left = n.r->left;
else
{
    snode t;
    t.l = n.l->left;
    t.r = n.r->left;
    s.push(t);
}
if (n.l->right == NULL)
    n.l->right = n.r->right;
else
{
    snode t;
    t.l = n.l->right;
    t.r = n.r->right;
    s.push(t);
}
}
return t1;
}

// Driver code
int main()
{
/* Let us construct the first Binary Tree
      1
     /   \
    2     3
   / \   \
  4   5   6
*/
Node *root1 = newNode(1);
root1->left = newNode(2);
root1->right = newNode(3);
root1->left->left = newNode(4);
root1->left->right = newNode(5);
root1->right->right = newNode(6);

/* Let us construct the second Binary Tree
      4
     /   \
    1     7
   /   /   \
  3   2   6   */
Node *root2 = newNode(4);
root2->left = newNode(1);
```

```
root2->right = newNode(7);
root2->left->left = newNode(3);
root2->right->left = newNode(2);
root2->right->right = newNode(6);

Node *root3 = MergeTrees(root1, root2);
printf("The Merged Binary Tree is:\n");
inorder(root3);
return 0;
}
```

Output:

```
The Merged Binary Tree is:
7 3 5 5 2 10 12
```

#### **Complexity Analysis:**

- Time complexity :  $O(n)$   
A total of  $n$  nodes need to be traversed. Here,  $n$  represents the minimum number of nodes from the two given trees.
- Auxiliary Space :  $O(n)$   
The depth of the stack can go upto  $n$  in case of a skewed tree.

#### **Source**

<https://www.geeksforgeeks.org/merge-two-binary-trees-node-sum/>

## Chapter 279

# Merge two BSTs with limited extra space

Merge two BSTs with limited extra space - GeeksforGeeks

Given two Binary Search Trees(BST), print the elements of both BSTs in sorted form. The expected time complexity is  $O(m+n)$  where m is the number of nodes in first tree and n is the number of nodes in second tree. Maximum allowed auxiliary space is  $O(\text{height of the first tree} + \text{height of the second tree})$ .

Examples:

```
First BST
      3
     /   \
    1     5
Second BST
      4
     /   \
    2     6
Output: 1 2 3 4 5 6
```

```
First BST
      8
     / \
    2   10
     /
    1
Second BST
      5
     /
    3
```

```
3
/
0
Output: 0 1 2 3 5 8 10
```

Source: Google interview question

A similar question has been discussed earlier. Let us first discuss already discussed methods of the [previous post](#) which was for Balanced BSTs. The method 1 can be applied here also, but the time complexity will be  $O(n^2)$  in worst case. The method 2 can also be applied here, but the extra space required will be  $O(n)$  which violates the constraint given in this question. Method 3 can be applied here but the step 3 of method 3 can't be done in  $O(n)$  for an unbalanced BST.

Thanks to [Kumar](#) for suggesting the following solution.

The idea is to use [iterative inorder traversal](#). We use two auxiliary stacks for two BSTs. Since we need to print the elements in sorted form, whenever we get a smaller element from any of the trees, we print it. If the element is greater, then we push it back to stack for the next iteration.

```
#include<stdio.h>
#include<stdlib.h>

// Structure of a BST Node
struct node
{
    int data;
    struct node *left;
    struct node *right;
};

//..... START OF STACK RELATED STUFF.....
// A stack node
struct snode
{
    struct node *t;
    struct snode *next;
};

// Function to add an elemnt k to stack
void push(struct snode **s, struct node *k)
{
    struct snode *tmp = (struct snode *) malloc(sizeof(struct snode));

    //perform memory check here
    tmp->t = k;
    tmp->next = *s;
    (*s) = tmp;
```

```
}  
  
// Function to pop an element t from stack  
struct node *pop(struct snode **s)  
{  
    struct node *t;  
    struct snode *st;  
    st=*s;  
    (*s) = (*s)->next;  
    t = st->t;  
    free(st);  
    return t;  
}  
  
// Fucntion to check whether the stack is empty or not  
int isEmpty(struct snode *s)  
{  
    if (s == NULL )  
        return 1;  
  
    return 0;  
}  
//..... END OF STACK RELATED STUFF.....  
  
/* Utility function to create a new Binary Tree node */  
struct node* newNode (int data)  
{  
    struct node *temp = new struct node;  
    temp->data = data;  
    temp->left = NULL;  
    temp->right = NULL;  
    return temp;  
}  
  
/* A utility function to print Inoder traversal of a Binary Tree */  
void inorder(struct node *root)  
{  
    if (root != NULL)  
    {  
        inorder(root->left);  
        printf("%d ", root->data);  
        inorder(root->right);  
    }  
}  
  
// The function to print data of two BSTs in sorted order  
void merge(struct node *root1, struct node *root2)
```

```
{  
    // s1 is stack to hold nodes of first BST  
    struct snode *s1 = NULL;  
  
    // Current node of first BST  
    struct node *current1 = root1;  
  
    // s2 is stack to hold nodes of second BST  
    struct snode *s2 = NULL;  
  
    // Current node of second BST  
    struct node *current2 = root2;  
  
    // If first BST is empty, then output is inorder  
    // traversal of second BST  
    if (root1 == NULL)  
    {  
        inorder(root2);  
        return;  
    }  
    // If second BST is empty, then output is inorder  
    // traversal of first BST  
    if (root2 == NULL)  
    {  
        inorder(root1);  
        return ;  
    }  
  
    // Run the loop while there are nodes not yet printed.  
    // The nodes may be in stack(explored, but not printed)  
    // or may be not yet explored  
    while (current1 != NULL || !isEmpty(s1) ||  
          current2 != NULL || !isEmpty(s2))  
    {  
        // Following steps follow iterative Inorder Traversal  
        if (current1 != NULL || current2 != NULL )  
        {  
            // Reach the leftmost node of both BSTs and push ancestors of  
            // leftmost nodes to stack s1 and s2 respectively  
            if (current1 != NULL)  
            {  
                push(&s1, current1);  
                current1 = current1->left;  
            }  
            if (current2 != NULL)  
            {  
                push(&s2, current2);  
                current2 = current2->left;  
            }  
        }  
    }  
}
```

```
}

}

else
{
    // If we reach a NULL node and either of the stacks is empty,
    // then one tree is exhausted, print the other tree
    if (isEmpty(s1))
    {
        while (!isEmpty(s2))
        {
            current2 = pop (&s2);
            current2->left = NULL;
            inorder(current2);
        }
        return ;
    }
    if (isEmpty(s2))
    {
        while (!isEmpty(s1))
        {
            current1 = pop (&s1);
            current1->left = NULL;
            inorder(current1);
        }
        return ;
    }

    // Pop an element from both stacks and compare the
    // popped elements
    current1 = pop(&s1);
    current2 = pop(&s2);

    // If element of first tree is smaller, then print it
    // and push the right subtree. If the element is larger,
    // then we push it back to the corresponding stack.
    if (current1->data < current2->data)
    {
        printf("%d ", current1->data);
        current1 = current1->right;
        push(&s2, current2);
        current2 = NULL;
    }
    else
    {
        printf("%d ", current2->data);
        current2 = current2->right;
        push(&s1, current1);
    }
}
```

```
        current1 = NULL;
    }
}
}

/* Driver program to test above functions */
int main()
{
    struct node *root1 = NULL, *root2 = NULL;

    /* Let us create the following tree as first tree
       3
      / \
     1   5
    */
    root1 = newNode(3);
    root1->left = newNode(1);
    root1->right = newNode(5);

    /* Let us create the following tree as second tree
       4
      / \
     2   6
    */
    root2 = newNode(4);
    root2->left = newNode(2);
    root2->right = newNode(6);

    // Print sorted nodes of both trees
    merge(root1, root2);

    return 0;
}
```

Time Complexity: O(m+n)

Auxiliary Space: O(height of the first tree + height of the second tree)

## Source

<https://www.geeksforgeeks.org/merge-two-bsts-with-limited-extra-space/>

## Chapter 280

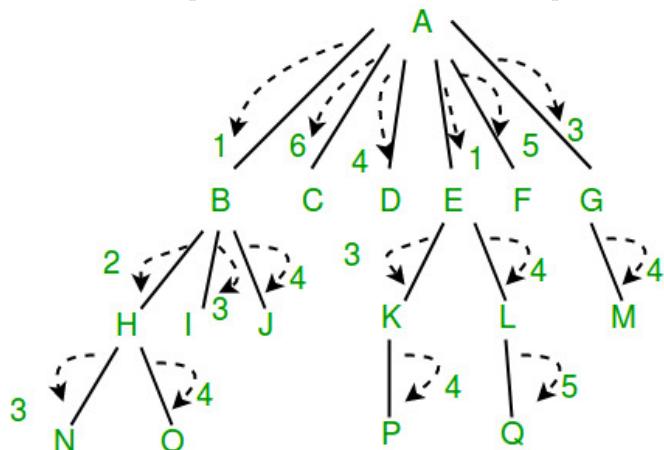
# Minimum no. of iterations to pass information to all nodes in the tree

Minimum no. of iterations to pass information to all nodes in the tree - GeeksforGeeks

Given a very large n-ary tree. Where the root node has some information which it wants to pass to all of its children down to the leaves with the constraint that it can only pass the information to one of its children at a time (take it as one iteration).

Now in the next iteration the child node can transfer that information to only one of its children and at the same time instance the child's parent i.e. root can pass the info to one of its remaining children. Continuing in this way we have to find the minimum no of iterations required to pass the information to all nodes in the tree.

Minimum no of iterations for tree below is 6. The root A first passes information to B. In next iteration, A passes information to E and B passes information to H and so on.



We strongly recommend to minimize the browser and try this yourself first.

This can be done using Post Order Traversal. The idea is to consider height and children count on each and every node.

If a child node  $i$  takes  $c_i$  iterations to pass info below its subtree, then its parent will take  $(c_i + 1)$  iterations to pass info to subtree rooted at that child  $i$ .

If parent has more children, it will pass info to them in subsequent iterations. Let's say children of a parent takes  $c_1, c_2, c_3, c_4, \dots, c_n$  iterations to pass info in their own subtree, Now parent has to pass info to these  $n$  children one by one in  $n$  iterations. If parent picks child  $i$  in  $i$ th iteration, then parent will take  $(i + c_i)$  iterations to pass info to child  $i$  and all it's subtree.

In any iteration, when parent passes info to a child  $i+1$ , children (1 to  $i$ ) which got info from parent already in previous iterations, will pass info to further down in subsequent iterations, if any child (1 to  $i$ ) has its own child further down.

To pass info to whole tree in minimum iterations, it needs to be made sure that bandwidth is utilized as efficiently as possible (i.e. maximum passable no of nodes should pass info further down in any iteration)

The best possible scenario would be that in  $n$ th iteration,  $n$  different nodes pass info to their child.

**Nodes with height = 0:** (Trivial case) Leaf node has no children (no information passing needed), so no of iterations would be ZERO.

**Nodes with height = 1:** Here node has to pass info to all the children one by one (all children are leaf node, so no more information passing further down). Since all children are leaf, node can pass info to any child in any order (pick any child who didn't receive the info yet). One iteration needed for each child and so no of iterations would be no of children. So node with height 1 with  $n$  children will take  $n$  iterations.

Take a counter initialized with ZERO, loop through all children and keep incrementing counter.

**Nodes with height > 1:** Let's assume that there are  $n$  children (1 to  $n$ ) of a node and minimum no iterations for all  $n$  children are  $c_1, c_2, \dots, c_n$ .

To make sure maximum no of nodes participate in info passing in any iteration, parent should 1st pass info to that child who will take maximum iteration to pass info further down in subsequent iterations. i.e. in any iteration, parent should choose the child who takes maximum iteration later on. It can be thought of as a greedy approach where parent choose that child 1st, who needs maximum no of iterations so that all subsequent iterations can be utilized efficiently.

If parent goes in any other fashion, then in the end, there could be some nodes which are done quite early, sitting idle and so bandwidth is not utilized efficiently in further iterations. If there are two children  $i$  and  $j$  with minimum iterations  $c_i$  and  $c_j$  where  $c_i > c_j$ , then If parent picks child  $j$  1st then no of iterations needed by parent to pass info to both children and their subtree would be:  $\max(1 + c_j, 2 + c_i) = 2 + c_i$

If parent picks child  $i$  1st then no of iterations needed by parent to pass info to both children and their subtree would be:  $\max(1 + c_i, 2 + c_j) = 1 + c_i$  (So picking  $c_i$  gives better result than picking  $c_j$ )

This tells that parent should always choose child  $i$  with max  $c_i$  value in any iteration.

SO here greedy approach is:

sort all  $c_i$  values decreasing order,

let's say after sorting, values are  $c_1 > c_2 > c_3 > \dots > c_n$

take a counter  $c$ , set  $c = 1 + c_1$  (for child with maximum no of iterations)

for all children i from 2 to n,  $c = c + 1 + ci$   
then total no of iterations needed by parent is  $\max(n, c)$

Let  $\minItr(A)$  be the minimum iteration needed to pass info from node A to it's all the sub-tree. Let  $\text{child}(A)$  be the count of all children for node A. So recursive relation would be:

1. Get  $\minItr(B)$  of all children (B) of a node (A)
2. Sort all  $\minItr(B)$  in descending order
3. Get  $\minItr$  of A based on all  $\minItr(B)$   
 $\minItr(A) = \text{child}(A)$   
For children B from i = 0 to  $\text{child}(A)$   
 $\minItr(A) = \max(\minItr(A), \minItr(B) + i + 1)$

Base cases would be:

If node is leaf,  $\minItr = 0$   
If node's height is 1,  $\minItr = \text{children count}$

Following is C++ implementation of above idea.

```
// C++ program to find minimum number of iterations to pass
// information from root to all nodes in an n-ary tree
#include<iostream>
#include<list>
#include<cmath>
#include <stdlib.h>
using namespace std;

// A class to represent n-ary tree (Note that the implementation
// is similar to graph for simplicity of implementation
class NAryTree
{
    int N;      // No. of nodes in Tree

    // Pointer to an array containing list of children
    list<int> *adj;

    // A function used by getMinIter(), it basically does postorder
    void getMinIterUtil(int v, int minItr[]);

public:
    NAryTree(int N);    // Constructor

    // function to add a child w to v
    void addChild(int v, int w);

    // The main function to find minimum iterations
    int getMinIter();
```

```

        static int compare(const void * a, const void * b);
    };

NAryTree::NAryTree(int N)
{
    this->N = N;
    adj = new list<int>[N];
}

// To add a child w to v
void NAryTree::addChild(int v, int w)
{
    adj[v].push_back(w); // Add w to v's list.
}

/* A recursive function to used by getMinIter(). This function
// mainly does postorder traversal and get minimum iteration of all children
// of node u, sort them in decreasing order and then get minimum iteration
// of node u

1. Get minItr(B) of all children (B) of a node (A)
2. Sort all minItr(B) in descending order
3. Get minItr of A based on all minItr(B)
    minItr(A) = child(A) --> child(A) is children count of node A
    For children B from i = 0 to child(A)
        minItr(A) = max ( minItr(A), minItr(B) + i + 1)

Base cases would be:
If node is leaf, minItr = 0
If node's height is 1, minItr = children count
*/

```

```

void NAryTree::getMinIterUtil(int u, int minItr[])
{
    minItr[u] = adj[u].size();
    int *minItrTemp = new int[minItr[u]];
    int k = 0, tmp = 0;
    // Recur for all the vertices adjacent to this vertex
    list<int>::iterator i;
    for (i = adj[u].begin(); i != adj[u].end(); ++i)
    {
        getMinIterUtil(*i, minItr);
        minItrTemp[k++] = minItr[*i];
    }
    qsort(minItrTemp, minItr[u], sizeof (int), compare);
    for (k = 0; k < adj[u].size(); k++)
    {

```

```
        tmp = minItrTemp[k] + k + 1;
        minItr[u] = max(minItr[u], tmp);
    }
    delete[] minItrTemp;
}

// The function to do PostOrder traversal. It uses
// recursive getMinIterUtil()
int NARYTree::getMinIter()
{
    // Set minimum iteration all the vertices as zero
    int *minItr = new int[N];
    int res = -1;
    for (int i = 0; i < N; i++)
        minItr[i] = 0;

    // Start Post Order Traversal from Root
    getMinIterUtil(0, minItr);
    res = minItr[0];
    delete[] minItr;
    return res;
}

int NARYTree::compare(const void * a, const void * b)
{
    return ( *(int*)b - *(int*)a );
}

// Driver function to test above functions
int main()
{
    // TestCase 1
    NARYTree tree1(17);
    tree1.addChild(0, 1);
    tree1.addChild(0, 2);
    tree1.addChild(0, 3);
    tree1.addChild(0, 4);
    tree1.addChild(0, 5);
    tree1.addChild(0, 6);

    tree1.addChild(1, 7);
    tree1.addChild(1, 8);
    tree1.addChild(1, 9);

    tree1.addChild(4, 10);
    tree1.addChild(4, 11);

    tree1.addChild(6, 12);
```

```
tree1.addChild(7, 13);
tree1.addChild(7, 14);
tree1.addChild(10, 15);
tree1.addChild(11, 16);

cout << "TestCase 1 - Minimum Iteration: "
    << tree1.getMinIter() << endl;

// TestCase 2
NAryTree tree2(3);
tree2.addChild(0, 1);
tree2.addChild(0, 2);
cout << "TestCase 2 - Minimum Iteration: "
    << tree2.getMinIter() << endl;

// TestCase 3
NAryTree tree3(1);
cout << "TestCase 3 - Minimum Iteration: "
    << tree3.getMinIter() << endl;

// TestCase 4
NAryTree tree4(6);
tree4.addChild(0, 1);
tree4.addChild(1, 2);
tree4.addChild(2, 3);
tree4.addChild(3, 4);
tree4.addChild(4, 5);
cout << "TestCase 4 - Minimum Iteration: "
    << tree4.getMinIter() << endl;

// TestCase 5
NAryTree tree5(6);
tree5.addChild(0, 1);
tree5.addChild(0, 2);
tree5.addChild(2, 3);
tree5.addChild(2, 4);
tree5.addChild(2, 5);
cout << "TestCase 5 - Minimum Iteration: "
    << tree5.getMinIter() << endl;

// TestCase 6
NAryTree tree6(6);
tree6.addChild(0, 1);
tree6.addChild(0, 2);
tree6.addChild(2, 3);
tree6.addChild(2, 4);
tree6.addChild(3, 5);
```

```
cout << "TestCase 6 - Minimum Iteration: "
    << tree6.getMinIter() << endl;

// TestCase 7
NAryTree tree7(14);
tree7.addChild(0, 1);
tree7.addChild(0, 2);
tree7.addChild(0, 3);
tree7.addChild(1, 4);
tree7.addChild(2, 5);
tree7.addChild(2, 6);
tree7.addChild(4, 7);
tree7.addChild(5, 8);
tree7.addChild(5, 9);
tree7.addChild(7, 10);
tree7.addChild(8, 11);
tree7.addChild(8, 12);
tree7.addChild(10, 13);
cout << "TestCase 7 - Minimum Iteration: "
    << tree7.getMinIter() << endl;

// TestCase 8
NAryTree tree8(14);
tree8.addChild(0, 1);
tree8.addChild(0, 2);
tree8.addChild(0, 3);
tree8.addChild(0, 4);
tree8.addChild(0, 5);
tree8.addChild(1, 6);
tree8.addChild(2, 7);
tree8.addChild(3, 8);
tree8.addChild(4, 9);
tree8.addChild(6, 10);
tree8.addChild(7, 11);
tree8.addChild(8, 12);
tree8.addChild(9, 13);
cout << "TestCase 8 - Minimum Iteration: "
    << tree8.getMinIter() << endl;

// TestCase 9
NAryTree tree9(25);
tree9.addChild(0, 1);
tree9.addChild(0, 2);
tree9.addChild(0, 3);
tree9.addChild(0, 4);
tree9.addChild(0, 5);
tree9.addChild(0, 6);
```

```
tree9.addChild(1, 7);
tree9.addChild(2, 8);
tree9.addChild(3, 9);
tree9.addChild(4, 10);
tree9.addChild(5, 11);
tree9.addChild(6, 12);

tree9.addChild(7, 13);
tree9.addChild(8, 14);
tree9.addChild(9, 15);
tree9.addChild(10, 16);
tree9.addChild(11, 17);
tree9.addChild(12, 18);

tree9.addChild(13, 19);
tree9.addChild(14, 20);
tree9.addChild(15, 21);
tree9.addChild(16, 22);
tree9.addChild(17, 23);
tree9.addChild(19, 24);

cout << "TestCase 9 - Minimum Iteration: "
<< tree9.getMinIter() << endl;

return 0;
}
```

Output:

```
TestCase 1 - Minimum Iteration: 6
TestCase 2 - Minimum Iteration: 2
TestCase 3 - Minimum Iteration: 0
TestCase 4 - Minimum Iteration: 5
TestCase 5 - Minimum Iteration: 4
TestCase 6 - Minimum Iteration: 3
TestCase 7 - Minimum Iteration: 6
TestCase 8 - Minimum Iteration: 6
TestCase 9 - Minimum Iteration: 8
```

This article is contributed by **Anurag Singh**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## Source

<https://www.geeksforgeeks.org/minimum-iterations-pass-information-nodes-tree/>

## Chapter 281

# Minimum swap required to convert binary tree to binary search tree

Minimum swap required to convert binary tree to binary search tree - GeeksforGeeks

Given the array representation of Complete Binary Tree i.e, if index i is the parent, index  $2*i + 1$  is the left child and index  $2*i + 2$  is the right child. The task is to find the minimum number of swap required to convert it into Binary Search Tree.

Examples:

Input : arr[] = { 5, 6, 7, 8, 9, 10, 11 }

Output : 3

Binary tree of the given array:

Swap 1: Swap node 8 with node 5.

Swap 2: Swap node 9 with node 10.

Swap 3: Swap node 10 with node 7.

So, minimum 3 swaps are required.

Input : arr[] = { 1, 2, 3 }

Output : 1

Binary tree of the given array:

After swapping node 1 with node 2.

So, only 1 swap required.

The idea is to use the fact that inorder traversal of Binary Search Tree is in increasing order of their value.

So, find the inorder traversal of the Binary Tree and store it in the array and try to sort the array. The minimum number of swap required to get the array sorted will be the answer. Please refer below post to find minimum number of swaps required to get the array sorted.

[Minimum number of swaps required to sort an array](#)

**Time Complexity:**  $O(n \log n)$ .

**Exercise:** Can we extend this to normal binary tree, i.e., a binary tree represented using left and right pointers, and not necessarily complete?

## Source

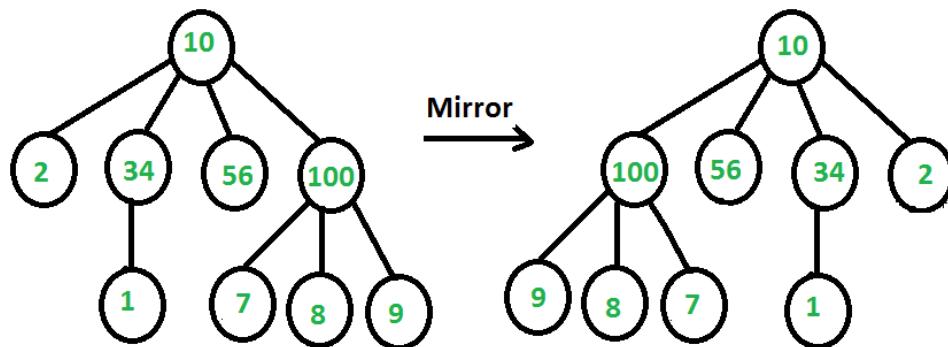
<https://www.geeksforgeeks.org/minimum-swap-required-convert-binary-tree-binary-search-tree/>

## Chapter 282

# Mirror of n-ary Tree

Mirror of n-ary Tree - GeeksforGeeks

Given a Tree where every node contains variable number of children, convert the tree to its mirror. Below diagram shows an example.



We strongly recommend you to minimize your browser and try this yourself first.

Node of tree is represented as a key and a variable sized array of children pointers. The idea is similar to mirror of Binary Tree. For every node, we first recur for all of its children and then reverse array of children pointers. We can also do these steps in other way, i.e., reverse array of children pointers first and then recur for children.

Below is C++ implementation of above idea.

C++

```
// C++ program to mirror an n-ary tree
#include <bits/stdc++.h>
using namespace std;
```

```
// Represents a node of an n-ary tree
struct Node
{
    int key;
    vector<Node *> child;
};

// Function to convert a tree to its mirror
void mirrorTree(Node * root)
{
    // Base case: Nothing to do if root is NULL
    if (root==NULL)
        return;

    // Number of children of root
    int n = root->child.size();

    // If number of child is less than 2 i.e.
    // 0 or 1 we do not need to do anything
    if (n < 2)
        return;

    // Calling mirror function for each child
    for (int i=0; i<n; i++)
        mirrorTree(root->child[i]);

    // Reverse vector (variable sized array) of child
    // pointers
    reverse(root->child.begin(), root->child.end());
}

// Utility function to create a new tree node
Node *newNode(int key)
{
    Node *temp = new Node;
    temp->key = key;
    return temp;
}

// Prints the n-ary tree level wise
void printNodeLevelWise(Node * root)
{
    if (root==NULL)
        return;

    // Create a queue and enqueue root to it
    queue<Node *>q;
```

```

q.push(root);

// Do level order traversal. Two loops are used
// to make sure that different levels are printed
// in different lines
while (!q.empty())
{
    int n = q.size();
    while (n>0)
    {
        // Dequeue an item from queue and print it
        Node * p = q.front();
        q.pop();
        cout << p->key << " ";

        // Enqueue all children of the dequeued item
        for (int i=0; i<p->child.size(); i++)
            q.push(p->child[i]);
        n--;
    }

    cout << endl; // Separator between levels
}
}

// Driver program
int main()
{
    /* Let us create below tree
     *          10
     *          /   \
     *         2   34   56   100
     *           |       /   |
     *           1       7   8   9
     */
    Node *root = newNode(10);
    (root->child).push_back(newNode(2));
    (root->child).push_back(newNode(34));
    (root->child).push_back(newNode(56));
    (root->child).push_back(newNode(100));
    (root->child[2]->child).push_back(newNode(1));
    (root->child[3]->child).push_back(newNode(7));
    (root->child[3]->child).push_back(newNode(8));
    (root->child[3]->child).push_back(newNode(9));

    cout << "Level order traversal Before Mirroring\n";
    printNodeLevelWise(root);
}

```

```
    mirrorTree(root);

    cout << "\nLevel order traversal After Mirroring\n";
    printNodeLevelWise(root);

    return 0;
}
```

### Python

```
# Python program to mirror an n-ary tree

# Represents a node of an n-ary tree
class Node :

    # Utility function to create a new tree node
    def __init__(self ,key):
        self.key = key
        self.child = []

    # Function to convert a tree to its mirror
def mirrorTree(root):

    # Base Case : nothing to do if root is None
    if root is None:
        return

    # Number of children of root
    n = len(root.child)

    # If number of child is less than 2 i.e.
    # 0 or 1 we don't need to do anything
    if n <2 :
        return

    # Calling mirror function for each child
    for i in range(n):
        mirrorTree(root.child[i]);

    # Reverse variable sized array of child pointers
    root.child.reverse()

    # Prints the n-ary tree level wise

def printNodeLevelWise(root):
    if root is None:
```

```

    return

# create a queue and enqueue root to it
queue = []
queue.append(root)

# Do level order traversal. Two loops are used
# to make sure that different levels are printed
# in different lines
while(len(queue) >0):

    n = len(queue)
    while(n > 0) :

        # Dequeue an item from queue and print it
        p = queue[0]
        queue.pop(0)
        print p.key,

        # Enqueue all children of the dequeued item
        for index, value in enumerate(p.child):
            queue.append(value)

        n -= 1
    print "" # Seperator between levels

# Driver Program

""" Let us create below tree
*          10
*          /   /
*          2   34   56   100
*                  |   /   |
*                  1   7   8   9
"""
root = Node(10)
root.child.append(Node(2))
root.child.append(Node(34))
root.child.append(Node(56))
root.child.append(Node(100))
root.child[2].child.append(Node(1))
root.child[3].child.append(Node(7))
root.child[3].child.append(Node(8))
root.child[3].child.append(Node(9))

print "Level order traversal Before Mirroring"

```

```
printNodeLevelWise(root)  
  
mirrorTree(root)  
  
print "\nLevel Order traversal After Mirroring"  
printNodeLevelWise(root)
```

Output:

```
Level order traversal Before Mirroring  
10  
2 34 56 100  
1 7 8 9  
  
Level order traversal After Mirroring  
10  
100 56 34 2  
9 8 7 1
```

Thanks to **Nitin Agrawal** for providing initial implementation. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## Source

<https://www.geeksforgeeks.org/mirror-of-n-ary-tree/>

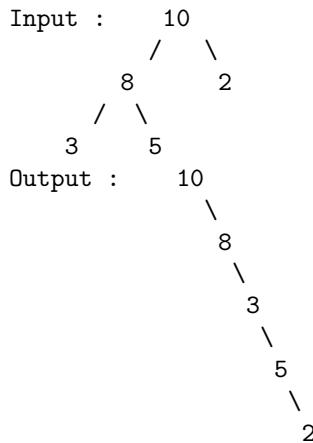
## Chapter 283

# Modify a binary tree to get preorder traversal using right pointers only

Modify a binary tree to get preorder traversal using right pointers only - GeeksforGeeks

Given a binary tree. Modify it in such a way that after modification you can have a preorder traversal of it using only the right pointers. During modification, you can use right as well as left pointers.

Examples:



Explanation : The preorder traversal of given binary tree is 10 8 3 5 2.

### Method 1 (Recursive)

One needs to make the right pointer of root point to the left subtree.

If the node has just left child, then just moving the child to right will complete the processing for that node.

If there is a right child too, then it should be made **right child of the right-most of the original left subtree**.

The above function used in the code process a node and then returns the rightmost node of the transformed subtree.

C++

```
// C code to modify binary tree for
// traversal using only right pointer
#include <iostream>
#include <stack>
#include <stdio.h>
#include <stdlib.h>

using namespace std;

// A binary tree node has data,
// left child and right child
struct Node {
    int data;
    struct Node* left;
    struct Node* right;
};

// function that allocates a new node
// with the given data and NULL left
// and right pointers.
struct Node* newNode(int data)
{
    struct Node* node = new struct Node;
    node->data = data;
    node->left = NULL;
    node->right = NULL;
    return (node);
}

// Function to modify tree
struct Node* modifytree(struct Node* root)
{
    struct Node* right = root->right;
    struct Node* rightMost = root;

    // if the left tree exists
    if (root->left) {

        // get the right-most of the
        // original left subtree
```

```
rightMost = modifytree(root->left);

// set root right to left subtree
root->right = root->left;
root->left = NULL;
}

// if the right subtree does
// not exists we are done!
if (!right)
    return rightMost;

// set right pointer of right-most
// of the original left subtree
rightMost->right = right;

// modify the rightsubtree
rightMost = modifytree(right);
return rightMost;
}

// printing using right pointer only
void printpre(struct Node* root)
{
    while (root != NULL) {
        cout << root->data << " ";
        root = root->right;
    }
}

// Driver program to test above functions
int main()
{
    /* Constructed binary tree is
       10
      / \
     8   2
    / \
   3   5 */
    struct Node* root = newNode(10);
    root->left = newNode(8);
    root->right = newNode(2);
    root->left->left = newNode(3);
    root->left->right = newNode(5);

    modifytree(root);
    printpre(root);
}
```

```
    return 0;
}
```

**Output:**

```
10 8 3 5 2
```

**Method 2 (Iterative)**

This can be easily done using iterative preorder traversal. See here. [Iterative preorder traversal](#)

The idea is to maintain a variable prev which maintains the previous node of the preorder traversal. Every-time a new node is encountered, the node set its right to previous one and prev is made equal to the current node. In the end we will have a sort of linked list whose first element is root then left child then right, so on and so forth.

C++

```
// C code to modify binary tree for
// traversal using only right pointer
#include <iostream>
#include <stack>
#include <stdio.h>
#include <stdlib.h>

using namespace std;

// A binary tree node has data,
// left child and right child
struct Node {
    int data;
    struct Node* left;
    struct Node* right;
};

// Helper function that allocates a new
// node with the given data and NULL
// left and right pointers.
struct Node* newNode(int data)
{
    struct Node* node = new struct Node;
    node->data = data;
    node->left = NULL;
    node->right = NULL;
    return (node);
}

// An iterative process to set the right
```

```
// pointer of Binary tree
void modifytree(struct Node* root)
{
    // Base Case
    if (root == NULL)
        return;

    // Create an empty stack and push root to it
    stack<Node*> nodeStack;
    nodeStack.push(root);

    /* Pop all items one by one.
       Do following for every popped item
       a) print it
       b) push its right child
       c) push its left child
    Note that right child is pushed first
    so that left is processed first */
    struct Node* pre = NULL;
    while (nodeStack.empty() == false) {

        // Pop the top item from stack
        struct Node* node = nodeStack.top();

        nodeStack.pop();

        // Push right and left children of
        // the popped node to stack
        if (node->right)
            nodeStack.push(node->right);
        if (node->left)
            nodeStack.push(node->left);

        // check if some previous node exists
        if (pre != NULL) {

            // set the right pointer of
            // previous node to current
            pre->right = node;
        }

        // set previous node as current node
        pre = node;
    }

    // printing using right pointer only
    void printpre(struct Node* root)
```

```
{  
    while (root != NULL) {  
        cout << root->data << " ";  
        root = root->right;  
    }  
}  
  
// Driver code  
int main()  
{  
    /* Constructed binary tree is  
        10  
       /   \  
      8     2  
     / \  
    3   5  
 */  
    struct Node* root = newNode(10);  
    root->left = newNode(8);  
    root->right = newNode(2);  
    root->left->left = newNode(3);  
    root->left->right = newNode(5);  
  
    modifytree(root);  
    printpre(root);  
  
    return 0;  
}
```

**Output:**

10 8 3 5 2

**Improved By :** 02DCE

**Source**

<https://www.geeksforgeeks.org/modify-binary-tree-get-preorder-traversal-using-right-pointers/>

## Chapter 284

# Morris traversal for Preorder

Morris traversal for Preorder - GeeksforGeeks

Using Morris Traversal, we can traverse the tree without using stack and recursion. The algorithm for Preorder is almost similar to [Morris traversal for Inorder](#).

- 1...**If** left child is null, print the current node data. Move to right child.
- ....**Else**, Make the right child of the inorder predecessor point to the current node. Two cases arise:
  - .....**a)** The right child of the inorder predecessor already points to the current node. Set right child to NULL. Move to right child of current node.
  - .....**b)** The right child is NULL. Set it to current node. Print current node's data and move to left child of current node.
- 2...Iterate until current node is not NULL.

Following is the implementation of the above algorithm.

C

```
// C program for Morris Preorder traversal
#include <stdio.h>
#include <stdlib.h>

struct node
{
    int data;
    struct node *left, *right;
};

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* temp = (struct node*) malloc(sizeof(struct node));
    temp->data = data;
```

```
temp->left = temp->right = NULL;
return temp;
}

// Preorder traversal without recursion and without stack
void morrisTraversalPreorder(struct node* root)
{
    while (root)
    {
        // If left child is null, print the current node data. Move to
        // right child.
        if (root->left == NULL)
        {
            printf( "%d ", root->data );
            root = root->right;
        }
        else
        {
            // Find inorder predecessor
            struct node* current = root->left;
            while (current->right && current->right != root)
                current = current->right;

            // If the right child of inorder predecessor already points to
            // this node
            if (current->right == root)
            {
                current->right = NULL;
                root = root->right;
            }

            // If right child doesn't point to this node, then print this
            // node and make right child point to this node
            else
            {
                printf("%d ", root->data);
                current->right = root;
                root = root->left;
            }
        }
    }
}

// Function for standard preorder traversal
void preorder(struct node* root)
{
    if (root)
    {
```

```
    printf( "%d ", root->data);
    preorder(root->left);
    preorder(root->right);
}
}

/* Driver program to test above functions*/
int main()
{
    struct node* root = NULL;

    root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);

    root->left->left = newNode(4);
    root->left->right = newNode(5);

    root->right->left = newNode(6);
    root->right->right = newNode(7);

    root->left->left->left = newNode(8);
    root->left->left->right = newNode(9);

    root->left->right->left = newNode(10);
    root->left->right->right = newNode(11);

    morrisTraversalPreorder(root);

    printf("\n");
    preorder(root);

    return 0;
}
```

**Java**

```
// Java program to implement Morris preorder traversal

// A binary tree node
class Node {

    int data;
    Node left, right;

    Node(int item) {
        data = item;
        left = right = null;
    }
}
```

```
        }
    }

class BinaryTree {

    Node root;

    void morrisTraversalPreorder()
    {
        morrisTraversalPreorder(root);
    }

    // Preorder traversal without recursion and without stack
    void morrisTraversalPreorder(Node node) {
        while (node != null) {

            // If left child is null, print the current node data. Move to
            // right child.
            if (node.left == null) {
                System.out.print(node.data + " ");
                node = node.right;
            } else {

                // Find inorder predecessor
                Node current = node.left;
                while (current.right != null && current.right != node) {
                    current = current.right;
                }

                // If the right child of inorder predecessor already points to
                // this node
                if (current.right == node) {
                    current.right = null;
                    node = node.right;
                }

                // If right child doesn't point to this node, then print this
                // node and make right child point to this node
                else {
                    System.out.print(node.data + " ");
                    current.right = node;
                    node = node.left;
                }
            }
        }
    }

    void preorder()
```

```
{  
    preorder(root);  
}  
  
// Function for Standard preorder traversal  
void preorder(Node node) {  
    if (node != null) {  
        System.out.print(node.data + " ");  
        preorder(node.left);  
        preorder(node.right);  
    }  
}  
  
// Driver programs to test above functions  
public static void main(String args[]) {  
    BinaryTree tree = new BinaryTree();  
    tree.root = new Node(1);  
    tree.root.left = new Node(2);  
    tree.root.right = new Node(3);  
    tree.root.left.left = new Node(4);  
    tree.root.left.right = new Node(5);  
    tree.root.right.left = new Node(6);  
    tree.root.right.right = new Node(7);  
    tree.root.left.left.left = new Node(8);  
    tree.root.left.left.right = new Node(9);  
    tree.root.left.right.left = new Node(10);  
    tree.root.left.right.right = new Node(11);  
    tree.morrisTraversalPreorder();  
    System.out.println("");  
    tree.preorder();  
}  
}  
  
// this code has been contributed by Mayank Jaiswal
```

### Python3

```
# Python program for Morris Preorder traversal  
  
# A binary tree Node  
class Node:  
    def __init__(self, data):  
        self.data = data  
        self.left = None  
        self.right = None  
  
# Preorder traversal without
```

```
# recursion and without stack
def MorrisTraversal(root):
    curr = root

    while curr:
        # If left child is null, print the
        # current node data. And, update
        # the current pointer to right child.
        if curr.left is None:
            print(curr.data, end= " ")
            curr = curr.right

        else:
            # Find the inorder predecessor
            prev = curr.left

            while prev.right is not None and prev.right is not curr:
                prev = prev.right

            # If the right child of inorder
            # predecessor already points to
            # the current node, update the
            # current with it's right child
            if prev.right is curr:
                prev.right = None
                curr = curr.right

            # else If right child doesn't point
            # to the current node, then print this
            # node's data and update the right child
            # pointer with the current node and update
            # the current with it's left child
            else:
                print (curr.data, end=" ")
                prev.right = curr
                curr = curr.left

# Function for standard preorder traversal
def preorfer(root):
    if root :
        print(root.data, end = " ")
        preorfer(root.left)
        preorfer(root.right)

# Driver program to test
root = Node(1)
root.left = Node(2)
```

```
root.right = Node(3)

root.left.left = Node(4)
root.left.right = Node(5)

root.right.left= Node(6)
root.right.right = Node(7)

root.left.left.left = Node(8)
root.left.left.right = Node(9)

root.left.right.left = Node(10)
root.left.right.right = Node(11)

MorrisTraversal(root)
print("\n")
preorfer(root)

# This code is contributed by 'Aartee'
```

Output:

```
1 2 4 8 9 5 10 11 3 6 7
1 2 4 8 9 5 10 11 3 6 7
```

#### **Limitations:**

Morris traversal modifies the tree during the process. It establishes the right links while moving down the tree and resets the right links while moving up the tree. So the algorithm cannot be applied if write operations are not allowed.

This article is compiled by [Aashish Barnwal](#) and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

#### **Source**

<https://www.geeksforgeeks.org/morris-traversal-for-preorder/>

## Chapter 285

# Next Larger element in n-ary tree

Next Larger element in n-ary tree - GeeksforGeeks

Given a generic tree and a integer x. Find and return the node with next larger element in the tree i.e. find a node just greater than x. Return NULL if no node is present with value greater than x.

For example, in the given tree

x = 10, just greater node value is 12

The idea is maintain a node pointer **res**, which will contain the final resultant node. Traverse the tree and check if root data is greater than x. If so, then compare the root data with res data. If root data is greater than n and less than res data update res.

```
// CPP program to find next larger element
// in an n-ary tree.
#include <bits/stdc++.h>
using namespace std;

// Structure of a node of an n-ary tree
struct Node {
    int key;
    vector<Node*> child;
};

// Utility function to create a new tree node
Node* newNode(int key)
{
    Node* temp = new Node;
    temp->key = key;
```

```
        return temp;
    }

void nextLargerElementUtil(Node* root, int x, Node** res)
{
    if (root == NULL)
        return;

    // if root is less than res but greater than
    // x update res
    if (root->key > x)
        if (!(*res) || (*res)->key > root->key)
            *res = root;

    // Number of children of root
    int numChildren = root->child.size();

    // Recur calling for every child
    for (int i = 0; i < numChildren; i++)
        nextLargerElementUtil(root->child[i], x, res);

    return;
}

// Function to find next Greater element of x in tree
Node* nextLargerElement(Node* root, int x)
{
    // resultant node
    Node* res = NULL;

    // calling helper function
    nextLargerElementUtil(root, x, &res);

    return res;
}

// Driver program
int main()
{
    /* Let us create below tree
     *          5
     *         /   |   \
     *        1   2   3
     *       /   / \   \
     *      15  4   5   6
     */
    Node* root = newNode(5);
```

```
(root->child).push_back(newNode(1));
(root->child).push_back(newNode(2));
(root->child).push_back(newNode(3));
(root->child[0]->child).push_back(newNode(15));
(root->child[1]->child).push_back(newNode(4));
(root->child[1]->child).push_back(newNode(5));
(root->child[2]->child).push_back(newNode(6));

int x = 5;

cout << "Next larger element of " << x << " is ";
cout << nextLargerElement(root, x)->key << endl;

return 0;
}
```

Output:

Next larger element of 5 is 6

## Source

<https://www.geeksforgeeks.org/next-larger-element-n-ary-tree/>

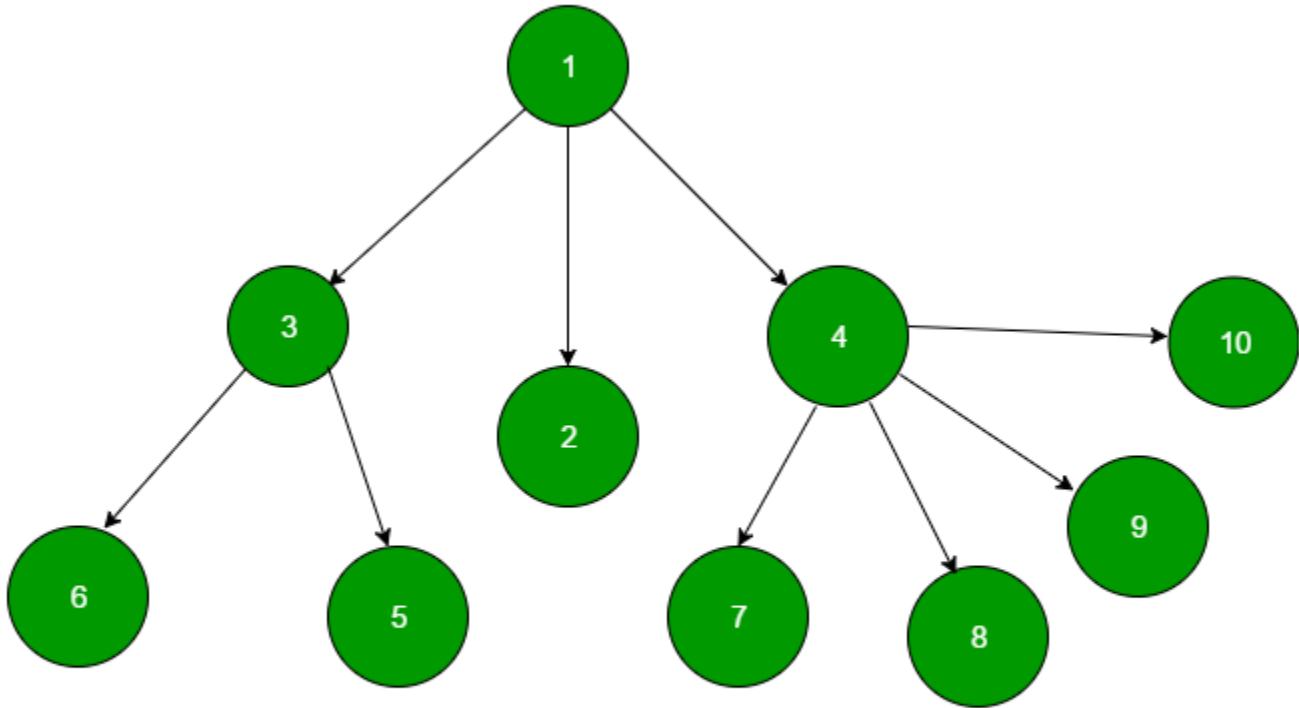
## Chapter 286

# Node having maximum sum of immediate children and itself in n-ary tree

Node having maximum sum of immediate children and itself in n-ary tree - GeeksforGeeks

Given an **N-Ary tree**, find and return the node for which sum of data of all children and the node itself is maximum. In the sum, data of node itself and data of its immediate children is to be taken.

For example in the given tree,



maxSum node = 4 with a maximum sum of 34

maxSum Node = 4 with maximum sum of 28

The idea is we will maintain a integer variable **maxsum** which contains the maximum sum yet, and a **resnode** node pointer which points to the node with maximum sum.

Traverse the tree and maintain the sum of root and data of all its immediate children in **cursum**

integer variable and update the **maxsum** variable accordingly.

```
// CPP program to find the node whose children
// and node sum is maximum.
#include <bits/stdc++.h>
using namespace std;

// Structure of a node of an n-ary tree
struct Node {
    int key;
    vector<Node*> child;
};

// Function to find the node with maximum sum of
// immediate children and itself
Node* maxSumNode(Node* root) {
    if (root == NULL)
        return NULL;

    // Initialize maxsum and resnode
    int maxsum = root->key;
    Node* resnode = root;

    // Traverse the tree
    for (int i = 0; i < root->child.size(); i++) {
        Node* child = root->child[i];
        int cursum = child->key;

        // Recursively find the maxsum node for each child
        Node* child_resnode = maxSumNode(child);
        int child_maxsum = child_resnode->key;

        // Update maxsum and resnode if current child's maxsum is greater
        if (cursum + child_maxsum > maxsum) {
            maxsum = cursum + child_maxsum;
            resnode = child_resnode;
        }
    }

    return resnode;
}
```

```
// Utility function to create a new tree node
Node* newNode(int key)
{
    Node* temp = new Node;
    temp->key = key;
    return temp;
}

// Helper function to find the node
void maxSumUtil(Node* root, Node** resNode,
                 int* maxsum)
{
    // Base Case
    if (root == NULL)
        return;

    // curr contains the sum of the root and
    // its children
    int currsum = root->key;

    // total no of children
    int count = root->child.size();

    // for every child call recursively
    for (int i = 0; i < count; i++) {
        currsum += root->child[i]->key;
        maxSumUtil(root->child[i], resNode, maxsum);
    }

    // if curr is greater than sum, update it
    if (currsum > *maxsum) {

        // resultant node
        *resNode = root;
        *maxsum = currsum;
    }
    return;
}

// Function to find the node having max sum of
// children and node
int maxSum(Node* root)
{
    // resultant node with max sum of children
    // and node
    Node* resNode;

    // sum of node and its children
```

```
int maxsum = 0;

maxSumUtil(root, &resNode, &maxsum);

// return the key of resultant node
return resNode->key;
}

// Driver program
int main()
{
    /* Let us create below tree
     *          1
     *         /   |   \
     *        2     3   4
     *       / \   / |   \ \
     *      5   6  7  8  9  10
     */
    Node* root = newNode(1);
    (root->child).push_back(newNode(2));
    (root->child).push_back(newNode(3));
    (root->child).push_back(newNode(4));
    (root->child[0]->child).push_back(newNode(5));
    (root->child[0]->child).push_back(newNode(6));
    (root->child[2]->child).push_back(newNode(5));
    (root->child[2]->child).push_back(newNode(6));
    (root->child[2]->child).push_back(newNode(6));

    cout << maxSum(root) << endl;

    return 0;
}
```

Output:

4

## Source

<https://www.geeksforgeeks.org/node-maximum-sum-immediate-children-n-ary-tree/>

## Chapter 287

# Non-recursive program to delete an entire binary tree

Non-recursive program to delete an entire binary tree - GeeksforGeeks

We have discussed recursive implementation to delete an entire binary tree [here](#).

**We strongly recommend you to minimize your browser and try this yourself first.**

Now how to delete an entire tree without using recursion. This could easily be done with the help of [Level Order Tree Traversal](#). The idea is for each dequeued node from the queue, delete it after queuing its left and right nodes (if any). The solution will work as we are traverse all the nodes of the tree level by level from top to bottom, and before deleting the parent node, we are storing its children into queue that will be deleted later.

C++

```
/* Non-Recursive Program to delete an entire binary tree. */
#include <bits/stdc++.h>
using namespace std;

// A Binary Tree Node
struct Node
{
    int data;
    struct Node *left, *right;
};

/* Non-recursive function to delete an entire binary tree. */
void _deleteTree(Node *root)
{
    // Base Case
    if (root == NULL)
```

```
    return;

    // Create an empty queue for level order traversal
    queue<Node *> q;

    // Do level order traversal starting from root
    q.push(root);
    while (!q.empty())
    {
        Node *node = q.front();
        q.pop();

        if (node->left != NULL)
            q.push(node->left);
        if (node->right != NULL)
            q.push(node->right);

        free(node);
    }
}

/* Deletes a tree and sets the root as NULL */
void deleteTree(Node** node_ref)
{
    _deleteTree(*node_ref);
    *node_ref = NULL;
}

// Utility function to create a new tree Node
Node* newNode(int data)
{
    Node *temp = new Node;
    temp->data = data;
    temp->left = temp->right = NULL;

    return temp;
}

// Driver program to test above functions
int main()
{
    // create a binary tree
    Node *root = newNode(15);
    root->left = newNode(10);
    root->right = newNode(20);
    root->left->left = newNode(8);
    root->left->right = newNode(12);
    root->right->left = newNode(16);
```

```
root->right->right = newNode(25);

//delete entire binary tree
deleteTree(&root);

return 0;
}
```

**Java**

```
/* Non-recursive program to delete the entire binary tree */
import java.util.*;

// A binary tree node
class Node
{
    int data;
    Node left, right;

    public Node(int data)
    {
        this.data = data;
        left = right = null;
    }
}

class BinaryTree
{
    Node root;

    /* Non-recursive function to delete an entire binary tree. */
    void _deleteTree()
    {
        // Base Case
        if (root == null)
            return;

        // Create an empty queue for level order traversal
        Queue<Node> q = new LinkedList<Node>();

        // Do level order traversal starting from root
        q.add(root);
        while (!q.isEmpty())
        {
            Node node = q.peek();
            q.poll();

            if (node.left != null)
```

```
        q.add(node.left);
    if (node.right != null)
        q.add(node.right);
    }
}

/* Deletes a tree and sets the root as NULL */
void deleteTree()
{
    _deleteTree();
    root = null;
}

// Driver program to test above functions
public static void main(String[] args)
{
    // create a binary tree
    BinaryTree tree = new BinaryTree();
    tree.root = new Node(15);
    tree.root.left = new Node(10);
    tree.root.right = new Node(20);
    tree.root.left.left = new Node(8);
    tree.root.left.right = new Node(12);
    tree.root.right.left = new Node(16);
    tree.root.right.right = new Node(25);

    // delete entire binary tree
    tree.deleteTree();
}
}

// This code has been contributed by Mayank Jaiswal(mayank_24)
```

### Python

```
# Python program to delete an entire binary tree
# using non-recursive approach

# A binary tree node
class Node:

    # A constructor to create a new node
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

    # Non-recursive function to delete an entrie binary tree
```

```
def _deleteTree(root):

    # Base Case
    if root is None:
        return

    # Create a empty queue for level order traversal
    q = []

    # Do level order traversal starting from root
    q.append(root)
    while(len(q)>0):
        node = q.pop(0)

        if node.left is not None:
            q.append(node.left)

        if node.right is not None:
            q.append(node.right)

        node = None
    return node

# Deletes a tree and sets the root as None
def deleteTree(node_ref):
    node_ref = _deleteTree(node_ref)
    return node_ref

# Driver program to test above function

# Create a binary tree
root = Node(15)
root.left = Node(10)
root.right = Node(20)
root.left.left = Node(8)
root.left.right = Node(12)
root.right.left = Node(16)
root.right.right = Node(25)

# delete entire binary tree
root = deleteTree(root)

# This program is contributed by Nikhil Kumar Singh(nickzuck_007)
```

**Improved By :** [subodh sawrav](#)

**Source**

<https://www.geeksforgeeks.org/non-recursive-program-to-delete-an-entire-binary-tree/>

## Chapter 288

# Number of Binary Trees for given Preorder Sequence length

Number of Binary Trees for given Preorder Sequence length - GeeksforGeeks

Count the number of Binary Tree possible for a given Preorder Sequence length n.

**Examples:**

Input : n = 1  
Output : 1

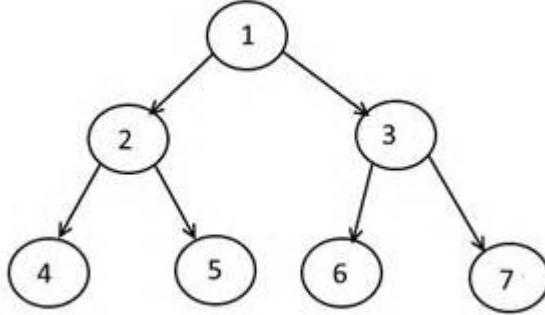
Input : n = 2  
Output : 2

Input : n = 3  
Output : 5

**Background :**

In [Preorder traversal](#), we process the root node first, then traverse the left child node and then right child node.

For example preorder traversal of below tree is 1 2 4 5 3 6 7



#### Finding number of trees with given Preorder:

Number of Binary Tree possible if such a traversal length (let's say n) is given.

**Let's take an Example :** Given Preorder Sequence  $\rightarrow 2\ 4\ 6\ 8\ 10$  (length 5).

- Assume there is only 1 node (that is 2 in this case), So only 1 Binary tree is Possible
- Now, assume there are 2 nodes (namely 2 and 4), So only 2 Binary Tree are Possible:
- Now, when there are 3 nodes (namely 2, 4 and 6), So Possible Binary tree are 5
- Consider 4 nodes (that are 2, 4, 6 and 8), So Possible Binary Tree are 14.

Let's say BT(1) denotes number of Binary tree for 1 node. (We assume BT(0)=1)

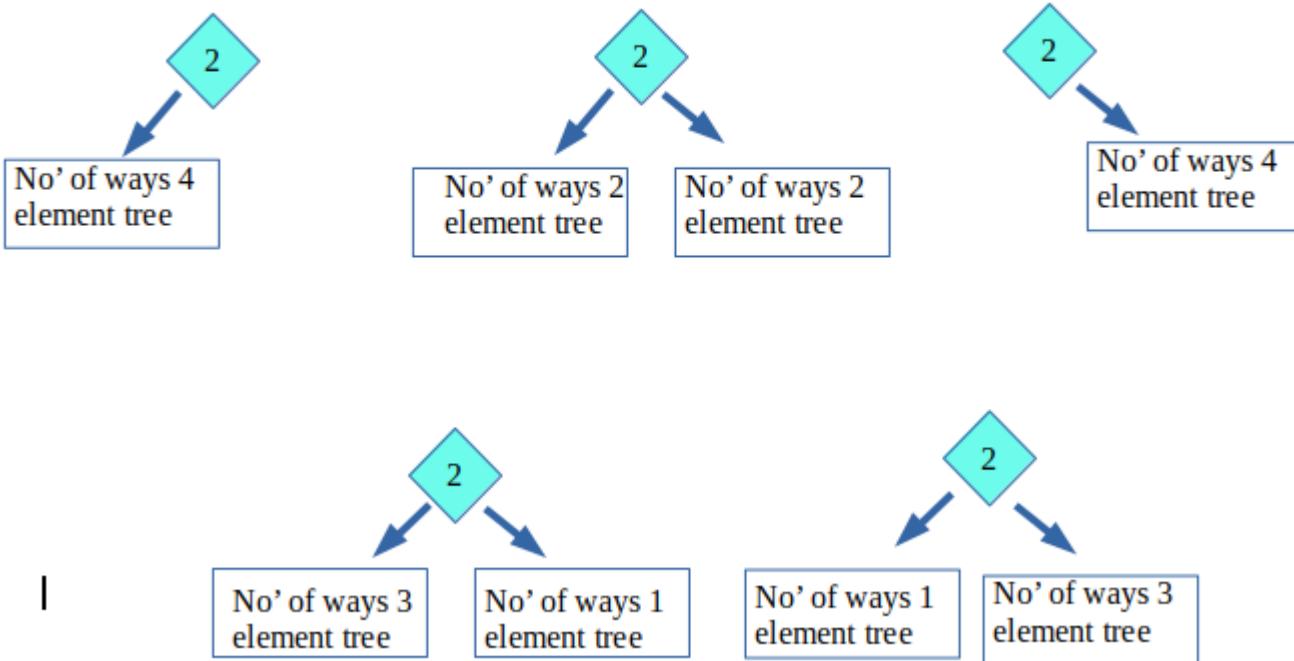
$$\mathbf{BT(4)} = \mathbf{BT(0)} * \mathbf{BT(3)} + \mathbf{BT(1)} * \mathbf{BT(2)} + \mathbf{BT(2)} * \mathbf{BT(1)} + \mathbf{BT(3)} * \mathbf{BT(0)}$$

$$\mathbf{BT(4)} = 1 * 5 + 1 * 2 + 2 * 1 + 5 * 1 = 14$$

- Similarly, considering all the 5 nodes (2, 4, 6, 8 and 10). Possible number of Binary Tree are:

$$\mathbf{BT(5)} = \mathbf{BT(0)} * \mathbf{BT(4)} + \mathbf{BT(1)} * \mathbf{BT(3)} + \mathbf{BT(2)} * \mathbf{BT(2)} + \mathbf{BT(3)} * \mathbf{BT(1)} + \mathbf{BT(4)} * \mathbf{BT(0)}$$

$$\mathbf{BT(5)} = 1 * 14 + 1 * 5 + 2 * 2 + 5 * 1 + 14 * 1 = 42$$



Hence, Total binary Tree for Pre-order sequence of length 5 is 42.

We use [Dynamic programming](#) to calculate the possible number of Binary Tree. We take one node at a time and calculate the possible Trees using previously calculated Trees.

C++

```

// C++ Program to count possible binary trees
// using dynamic programming
#include <bits/stdc++.h>
using namespace std;

int countTrees(int n)
{
    // Array to store number of Binary tree
    // for every count of nodes
    int BT[n + 1];
    memset(BT, 0, sizeof(BT));

    BT[0] = BT[1] = 1;

    // Start finding from 2 nodes, since
    // already know for 1 node.

```

```
for (int i = 2; i <= n; ++i)
    for (int j = 0; j < i; j++)
        BT[i] += BT[j] * BT[i - j - 1];

    return BT[n];
}

// Driver code
int main()
{
    int n = 5;
    cout << "Total Possible Binary Tree are : "
         << countTrees(n) << endl;
    return 0;
}
```

**Java**

```
// Java Program to count
// possible binary trees
// using dynamic programming
import java.io.*;

class GFG
{
static int countTrees(int n)
{
    // Array to store number
    // of Binary tree for
    // every count of nodes
    int BT[] = new int[n + 1];
    for(int i = 0; i <= n; i++)
        BT[i] = 0;
    BT[0] = BT[1] = 1;

    // Start finding from 2
    // nodes, since already
    // know for 1 node.
    for (int i = 2; i <= n; ++i)
        for (int j = 0; j < i; j++)
            BT[i] += BT[j] *
                      BT[i - j - 1];

    return BT[n];
}

// Driver code
public static void main (String[] args)
```

```
{  
int n = 5;  
System.out.println("Total Possible " +  
    "Binary Tree are : " +  
    countTrees(n));  
}  
}  
  
// This code is contributed by anuj_67.
```

**Output:**

Total Possible Binary Tree are : 42

**Alternative :**

This can also be done using [Catalan number](#)  $C_n = (2n)!/(n+1)!*n!$

For  $n = 0, 1, 2, 3, \dots$  values of Catalan numbers are 1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, .... So are [numbers of Binary Search Trees](#).

**Improved By :** [vt\\_m](#)

**Source**

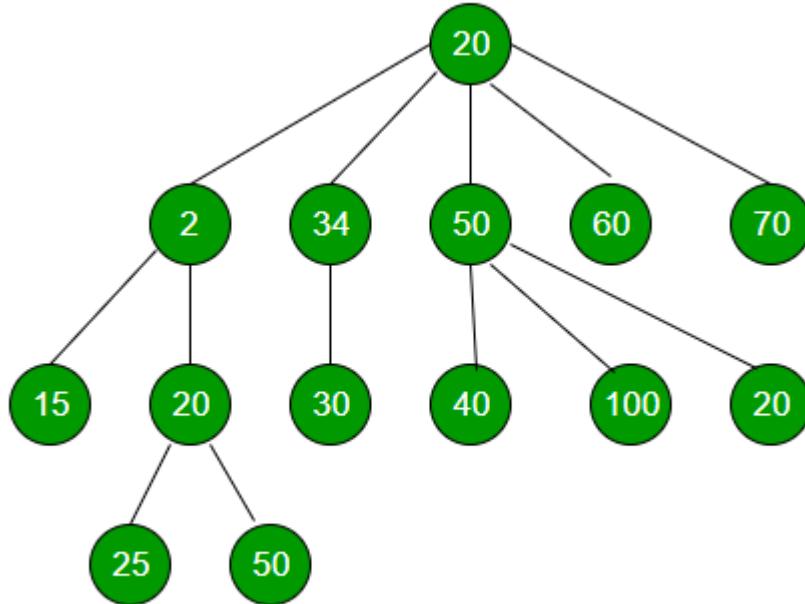
<https://www.geeksforgeeks.org/number-of-binary-trees-for-given-preorder-sequence-length/>

## Chapter 289

### Number of children of given node in n-ary Tree

Number of children of given node in n-ary Tree - GeeksforGeeks

Given a node x, find the number of children of x(if it exists) in the given n-ary tree.



Example :

Input : x = 50

Output : 3

Explanation : 50 has 3 children having values 40, 100 and 20.

**Approach :**

- Initialize the number of children as 0.
- For every node in the n-ary tree, check if its value is equal to x or not. If yes, then return the number of children.
- If the value of x is not equal to the current node then, push all the children of current node in the queue.
- Keep Repeating the above step until the queue becomes empty.

Below is the implementation of the above idea :

```
// C++ program to find number
// of children of given node
#include <bits/stdc++.h>
using namespace std;

// Represents a node of an n-ary tree
class Node {

public:
    int key;
    vector<Node*> child;

    Node(int data)
    {
        key = data;
    }
};

// Function to calculate number
// of children of given node
int numberOfChildren(Node* root, int x)
{
    // initialize the numChildren as 0
    int numChildren = 0;

    if (root == NULL)
        return 0;

    // Creating a queue and pushing the root
    queue<Node*> q;
    q.push(root);

    while (!q.empty()) {
```

```
int n = q.size();

// If this node has children
while (n > 0) {

    // Dequeue an item from queue and
    // check if it is equal to x
    // If YES, then return number of children
    Node* p = q.front();
    q.pop();
    if (p->key == x) {
        numChildren = numChildren + p->child.size();
        return numChildren;
    }

    // Enqueue all children of the dequeued item
    for (int i = 0; i < p->child.size(); i++)
        q.push(p->child[i]);
    n--;
}
return numChildren;
}

// Driver program
int main()
{
    // Creating a generic tree
    Node* root = new Node(20);
    (root->child).push_back(new Node(2));
    (root->child).push_back(new Node(34));
    (root->child).push_back(new Node(50));
    (root->child).push_back(new Node(60));
    (root->child).push_back(new Node(70));
    (root->child[0]->child).push_back(new Node(15));
    (root->child[0]->child).push_back(new Node(20));
    (root->child[1]->child).push_back(new Node(30));
    (root->child[2]->child).push_back(new Node(40));
    (root->child[2]->child).push_back(new Node(100));
    (root->child[2]->child).push_back(new Node(20));
    (root->child[0]->child[1]->child).push_back(new Node(25));
    (root->child[0]->child[1]->child).push_back(new Node(50));

    // Node whose number of
    // children is to be calculated
    int x = 50;

    // Function calling
```

```
cout << numberOfChildren(root, x) << endl;  
return 0;  
}
```

**Output:**

3

**Time Complexity :**  $O(N)$ , where N is the number of nodes in tree.

**Auxiliary Space :**  $O(N)$ , where N is the number of nodes in tree.

**Source**

<https://www.geeksforgeeks.org/number-children-given-node-n-ary-tree/>

## Chapter 290

# Number of full binary trees such that each node is product of its children

Number of full binary trees such that each node is product of its children - GeeksforGeeks

Given an array of **n** integers, each integer is greater than 1. The task is to find the number of [Full binary tree](#) from the given integers, such that each non-node leaf node value is the product of its children value. Given that, each integer can be used multiple times in a full binary tree.

Examples:

```
Input : arr[] = { 2, 3, 4, 6 }.
Output : 7
There can be 7 full binary tree with the given product property.

// Four trees with single nodes
2 3 4 6

// Three trees with three nodes
        4
      / \
    2   2

        6
      / \
    2   3

        6
      / \
    / \
```

3 2

We find maximum value in given array and create an array to store presence of elements in this array. The idea is, for all multiples of each integer less than the maximum value of the array, try to make full binary tree if the multiple is present in the array.

Observe that for any full binary tree with given property, the smaller values will always be at the last level. So, try to find the number of such full binary tree from the minimum value of the array to maximum value of the array.

Algorithm to solve the problem:

1. Initialize possible number of such full binary tree for each element equal to 1. Since single node also contribute to the answer.
2. For each element of the array, arr[i], from minimum value to maximum value of array.
  - .....a) For each multiple of arr[i], find if multiple is present or not.
  - .....b) If yes, then the number of such possible full binary tree for multiple of arr[i], say m, is equal to the product of the number of such possible full binary tree of arr[i] and number of such possible full binary tree of arr[i]/m.

C++

```
// C++ program to find number of full binary tree
// such that each node is product of its children.
#include<bits/stdc++.h>
using namespace std;

// Return the number of all possible full binary
// tree with given product propert.
int numoffbt(int arr[], int n)
{
    // Finding the minimum and maximum values in
    // given array.
    int maxvalue = INT_MIN, minvalue = INT_MAX;
    for (int i = 0; i < n; i++)
    {
        maxvalue = max(maxvalue, arr[i]);
        minvalue = min(minvalue, arr[i]);
    }

    int mark[maxvalue + 2];
    int value[maxvalue + 2];
    memset(mark, 0, sizeof(mark));
    memset(value, 0, sizeof(value));

    // Marking the presence of each array element
    // and initialising the number of possible
    // full binary tree for each integer equal
    // to 1 because single node will also
    // contribute as a full binary tree.
```

```
for (int i = 0; i < n; i++)
{
    mark[arr[i]] = 1;
    value[arr[i]] = 1;
}

// From minimum value to maximum value of array
// finding the number of all possible Full
// Binary Trees.
int ans = 0;
for (int i = minvalue; i <= maxvalue; i++)
{
    // Find if value present in the array
    if (mark[i])
    {
        // For each multiple of i, less than
        // equal to maximum value of array
        for (int j = i + i;
             j <= maxvalue && j/i <= i; j += i)
        {
            // If multiple is not present in the
            // array then continue.
            if (!mark[j])
                continue;

            // Finding the number of possible Full
            // binary trees for multiple j by
            // multiplying number of possible Full
            // binary tree from the number i and
            // number of possible Full binary tree
            // from i/j.
            value[j] = value[j] + (value[i] * value[j/i]);

            // Condition for possibility when left
            // child became right child and vice versa.
            if (i != j/i)
                value[j] = value[j] + (value[i] * value[j/i]);
        }
    }

    ans += value[i];
}

return ans;
}

// Driven Program
int main()
```

```
{  
    int arr[] = { 2, 3, 4, 6 };  
    int n = sizeof(arr)/sizeof(arr[0]);  
  
    cout << numoffbt(arr, n) << endl;  
    return 0;  
}
```

**Java**

```
// Java program to find number of full  
// binary tree such that each node is  
// product of its children.  
import java.util.Arrays;  
  
class GFG {  
  
    // Return the number of all possible  
    // full binary tree with given product  
    // propert.  
    static int numoffbt(int arr[], int n)  
    {  
  
        // Finding the minimum and maximum  
        // values in given array.  
        int maxvalue = -2147483647;  
        int minvalue = 2147483647;  
        for (int i = 0; i < n; i++)  
        {  
            maxvalue = Math.max(maxvalue, arr[i]);  
            minvalue = Math.min(minvalue, arr[i]);  
        }  
  
        int mark[] = new int[maxvalue + 2];  
        int value[] = new int[maxvalue + 2];  
        Arrays.fill(mark, 0);  
        Arrays.fill(value, 0);  
  
        // Marking the presence of each array element  
        // and initialising the number of possible  
        // full binary tree for each integer equal  
        // to 1 because single node will also  
        // contribute as a full binary tree.  
        for (int i = 0; i < n; i++)  
        {  
            mark[arr[i]] = 1;  
            value[arr[i]] = 1;  
        }
```

```
// From minimum value to maximum value of array
// finding the number of all possible Full
// Binary Trees.
int ans = 0;
for (int i = minvalue; i <= maxvalue; i++)
{
    // Find if value present in the array
    if (mark[i] != 0)
    {
        // For each multiple of i, less than
        // equal to maximum value of array
        for (int j = i + i;
            j <= maxvalue && j/i <= i; j += i)
        {
            // If multiple is not present in
            // the array then continue.
            if (mark[j] == 0)
                continue;

            // Finding the number of possible
            // Full binary trees for multiple
            // j by multiplying number of
            // possible Full binary tree from
            // the number i and number of
            // possible Full binary tree from i/j.
            value[j] = value[j] + (value[i]
                * value[j/i]);

            // Condition for possibility when
            // left child became right child
            // and vice versa.
            if (i != j / i)
                value[j] = value[j] + (value[i]
                    * value[j/i]);
        }
    }
    ans += value[i];
}

return ans;
}

//driver code
public static void main (String[] args)
{
```

```
    int arr[] = { 2, 3, 4, 6 };
    int n = arr.length;

    System.out.print(numoffbt(arr, n));
}
}

//This code is contributed by Anant Agarwal.
```

### Python3

```
# Python3 program to find number of
# full binary tree such that each node
# is product of its children.

# Return the number of all possible full
# binary tree with given product propert.
def numoffbt(arr, n):

    # Finding the minimum and maximum
    # values in given array.
    maxvalue = -2147483647
    minvalue = 2147483647
    for i in range(n):

        maxvalue = max(maxvalue, arr[i])
        minvalue = min(minvalue, arr[i])

    mark = [0 for i in range(maxvalue + 2)]
    value = [0 for i in range(maxvalue + 2)]

    # Marking the presence of each array element
    # and initialising the number of possible
    # full binary tree for each integer equal
    # to 1 because single node will also
    # contribute as a full binary tree.
    for i in range(n):

        mark[arr[i]] = 1
        value[arr[i]] = 1

    # From minimum value to maximum value
    # of array finding the number of all
    # possible Full Binary Trees.
    ans = 0
```

```
for i in range(minvalue, maxvalue + 1):

    # Find if value present in the array
    if (mark[i] != 0):

        # For each multiple of i, less than
        # equal to maximum value of array
        j = i + i
        while(j <= maxvalue and j // i <= i):

            # If multiple is not present in the
            # array then continue.
            if (mark[j] == 0):
                continue

            # Finding the number of possible Full
            # binary trees for multiple j by
            # multiplying number of possible Full
            # binary tree from the number i and
            # number of possible Full binary tree
            # from i/j.
            value[j] = value[j] + (value[i] * value[j // i])

            # Condition for possibility when left
            # child became right child and vice versa.
            if (i != j // i):
                value[j] = value[j] + (value[i] * value[j // i])
            j += i

ans += value[i]

return ans

# Driver Code
arr = [ 2, 3, 4, 6 ]
n = len(arr)

print(numoffbt(arr, n))

# This code is contributed by Anant Agarwal.
```

C#

```
// C# program to find number of
// full binary tree such that each
// node is product of its children.
using System;
```

```
class GFG
{
    // Return the number of all possible full binary
    // tree with given product propert.
    static int numoffbt(int []arr, int n)
    {
        // Finding the minimum and maximum values in
        // given array.
        int maxvalue = -2147483647, minvalue = 2147483647;
        for (int i = 0; i < n; i++)
        {
            maxvalue = Math.Max(maxvalue, arr[i]);
            minvalue = Math.Min(minvalue, arr[i]);
        }

        int []mark=new int[maxvalue + 2];
        int []value=new int[maxvalue + 2];
        for(int i = 0;i < maxvalue + 2; i++)
        {
            mark[i]=0;
            value[i]=0;
        }

        // Marking the presence of each array element
        // and initialising the number of possible
        // full binary tree for each integer equal
        // to 1 because single node will also
        // contribute as a full binary tree.
        for (int i = 0; i < n; i++)
        {
            mark[arr[i]] = 1;
            value[arr[i]] = 1;
        }

        // From minimum value to maximum value of array
        // finding the number of all possible Full
        // Binary Trees.
        int ans = 0;
        for (int i = minvalue; i <= maxvalue; i++)
        {
            // Find if value present in the array
            if (mark[i] != 0)
            {
                // For each multiple of i, less than
                // equal to maximum value of array
                for (int j = i + i;
                     j <= maxvalue && j/i <= i; j += i)

```

```
{  
    // If multiple is not present in the  
    // array then continue.  
    if (mark[j] == 0)  
        continue;  
  
    // Finding the number of possible Full  
    // binary trees for multiple j by  
    // multiplying number of possible Full  
    // binary tree from the number i and  
    // number of possible Full binary tree  
    // from i/j.  
    value[j] = value[j] + (value[i] * value[j/i]);  
  
    // Condition for possibility when left  
    // child became right child and vice versa.  
    if (i != j/i)  
        value[j] = value[j] + (value[i] * value[j/i]);  
}  
}  
  
ans += value[i];  
}  
  
return ans;  
}  
  
// Driver code  
public static void Main()  
{  
    int []arr = { 2, 3, 4, 6 };  
    int n = arr.Length;  
  
    Console.WriteLine(numoffbt(arr, n));  
}  
}  
  
// This code is contributed by Anant Agarwal.
```

Output:

7

## Source

<https://www.geeksforgeeks.org/number-full-binary-trees-node-product-children/>

## Chapter 291

# Number of nodes greater than a given value in n-ary tree

Number of nodes greater than a given value in n-ary tree - GeeksforGeeks

Given a **n-ary** tree and a number **x**, find and return the number of nodes which are greater than **x**.

Example:

In the given tree, **x** = 7

Number of nodes greater than **x** are 4.

### Approach :

The idea is maintain a count variable initialize to 0. Traverse the tree and compare root data with **x**. If root data is greater than **x**, increment the count variable and recursively call for all its children.

Below is the implementation of idea.

```
// CPP program to find number of nodes
// greater than x
#include <bits/stdc++.h>
using namespace std;

// Structure of a node of n-ary tree
struct Node {
    int key;
    vector<Node*> child;
```

```
};

// Utility function to create
// a new tree node
Node* newNode(int key)
{
    Node* temp = new Node;
    temp->key = key;
    return temp;
}

// Function to find number of nodes
// greater than x
int nodesGreaterThanX(Node* root, int x)
{
    if (root == NULL)
        return 0;

    int count = 0;

    // if current root is greater
    // than x increment count
    if (root->key > x)
        count++;

    // Number of children of root
    int numChildren = root->child.size();

    // recursively calling for every child
    for (int i = 0; i < numChildren; i++) {
        Node* child = root->child[i];
        count += nodesGreaterThanX(child, x);
    }

    // return the count
    return count;
}

// Driver program
int main()
{
    /* Let us create below tree
     *      5
     *      / \ \
     *      1  2  3
     *      /   / \ \
     *     15  4   5   6
     */
}
```

```
Node* root = newNode(5);
(root->child).push_back(newNode(1));
(root->child).push_back(newNode(2));
(root->child).push_back(newNode(3));
(root->child[0]->child).push_back(newNode(15));
(root->child[1]->child).push_back(newNode(4));
(root->child[1]->child).push_back(newNode(5));
(root->child[2]->child).push_back(newNode(6));

int x = 5;

cout << "Number of nodes greater than "
    << x << " are ";
cout << nodesGreaterThanX(root, x)
    << endl;

return 0;
}
```

Output:

```
Number of nodes greater than 5 are 2
```

## Source

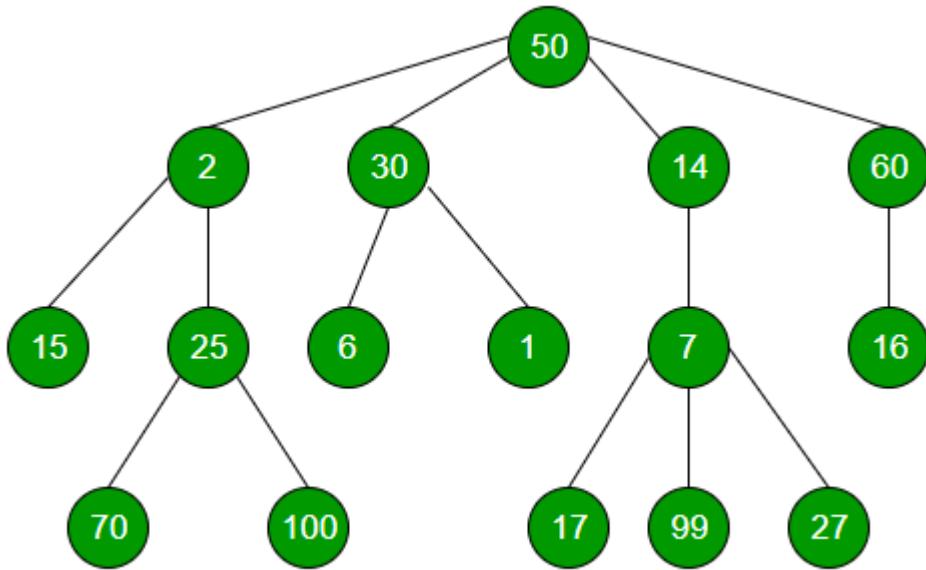
<https://www.geeksforgeeks.org/number-nodes-greater-given-value-n-ary-tree/>

## Chapter 292

### Number of siblings of a given Node in n-ary Tree

Number of siblings of a given Node in n-ary Tree - GeeksforGeeks

Given an N-ary tree, find the number of siblings of given node x. Assume that x exists in the given n-ary tree.



Example :

Input : 30  
Output : 3

**Approach :** For every node in the given n-ary tree, push the children of the current node in the queue. While adding the children of current node in queue, check if any children is equal to the given value x or not. If yes, then return the number of siblings of x.

**Below is the implementation of the above idea :**

```
// C++ program to find number
// of siblings of a given node
#include <bits/stdc++.h>
using namespace std;

// Represents a node of an n-ary tree
class Node
{
public:
    int key;
    vector<Node*> child;

    Node(int data)
    {
        key = data;
    }
};

// Function to calculate number
// of siblings of a given node
int numberOfWorkingSiblings(Node* root, int x)
{
    if (root == NULL)
        return 0;

    // Creating a queue and
    // pushing the root
    queue<Node*> q;
    q.push(root);

    while (!q.empty())
    {
        int n = q.size();

        // If this node has children
        while (n > 0) {

            // Dequeue an item from queue and
            // check if it is equal to x If YES,
            // then return number of children
            Node* p = q.front();
            q.pop();
            if (p->key == x)
                return n;
        }
    }
}
```

```
// Enqueue all children of
// the dequeued item
for (int i = 0; i < p->child.size(); i++)
{
    // If the value of children
    // is equal to x, then return
    // the number of siblings
    if (p->child[i]->key == x)
        return p->child.size() - 1;

    q.push(p->child[i]);
}
n--;
}
}

// Driver program
int main()
{
    // Creating a generic tree as shown in above figure
    Node* root = new Node(50);
    (root->child).push_back(new Node(2));
    (root->child).push_back(new Node(30));
    (root->child).push_back(new Node(14));
    (root->child).push_back(new Node(60));
    (root->child[0]->child).push_back(new Node(15));
    (root->child[0]->child).push_back(new Node(25));
    (root->child[0]->child[1]->child).push_back(new Node(70));
    (root->child[0]->child[1]->child).push_back(new Node(100));
    (root->child[1]->child).push_back(new Node(6));
    (root->child[1]->child).push_back(new Node(1));
    (root->child[2]->child).push_back(new Node(7));
    (root->child[2]->child[0]->child).push_back(new Node(17));
    (root->child[2]->child[0]->child).push_back(new Node(99));
    (root->child[2]->child[0]->child).push_back(new Node(27));
    (root->child[3]->child).push_back(new Node(16));

    // Node whose number of
    // siblings is to be calculated
    int x = 100;

    // Function calling
    cout << numberofSiblings(root, x) << endl;
}

return 0;
}
```

**Output:**

1

**Time Complexity :**  $O(N^2)$ , where N is the number of nodes in tree.

**Auxiliary Space :**  $O(N)$ , where N is the number of nodes in tree.

**Source**

<https://www.geeksforgeeks.org/number-siblings-given-node-n-ary-tree/>

## Chapter 293

# Number of subtrees having odd count of even numbers

Number of subtrees having odd count of even numbers - GeeksforGeeks

Given a binary tree, find the number of subtrees having odd count of even numbers.

Examples:

Input :  
              2  
              /      \  
          1      3  
         /    \    /    \  
        4    10    8    5  
          /      \  
         6

Output : 6  
The subtrees are 4, 6,    1,        8,    3  
              /    \  
         4    10      8    5  
          /      \  
         6

              2  
              /      \  
          1      3  
         /    \    /    \  
        4    10    8    5  
          /      \  
         6

The idea is to recursively traverse the tree. For every node, recursively count even numbers

in left and right subtrees. If root is also even, then add it also to count. If count becomes odd, increment result.

```
count = 0 // Initialize result

int countSubtrees(root)
{
    if (root == NULL)
        return 0;

    // count even numbers in left subtree
    int c = countSubtrees(root-> left);

    // Add count of even numbers in right subtree
    c += countSubtrees(root-> right);

    // check if root data is an even number
    if (root-> data % 2 == 0)
        c += 1;

    // If total count of even numbers
    // for the subtree is odd
    if (c % 2 != 0)
        count++;

    // return total count of even
    // numbers of the subtree
    return c;
}

// C implementation to find number of
// subtrees having odd count of even numbers
#include <stdio.h>
#include <stdlib.h>

/* A binary tree Node */
struct Node
{
    int data;
    struct Node* left, *right;
};

/* Helper function that allocates a new Node with the
   given data and NULL left and right pointers. */
struct Node* newNode(int data)
{
    struct Node* node = new Node;
```

```
node->data = data;
node->left = node->right = NULL;
return(node);
}

// Returns count of subtrees having odd count of
// even numbers
int countRec(struct Node* root, int *pcount)
{
    // base condition
    if (root == NULL)
        return 0;

    // count even nodes in left subtree
    int c = countRec(root->left, pcount);

    // Add even nodes in right subtree
    c += countRec(root->right, pcount);

    // Check if root data is an even number
    if (root->data % 2 == 0)
        c += 1;

    // if total count of even numbers
    // for the subtree is odd
    if (c % 2 != 0)
        (*pcount)++;

    // Total count of even nodes of the subtree
    return c;
}

// A wrapper over countRec()
int countSubtrees(Node *root)
{
    int count = 0;
    int *pcount = &count;
    countRec(root, pcount);
    return count;
}

// Driver program to test above
int main()
{
    // binary tree formation
    struct Node *root = newNode(2); /*      2      */
    root->left = newNode(1); /*      / \      */
    root->right = newNode(3); /*      1   3      */
}
```

```
root->left->left = newNode(4); /* / \ / \ */
root->left->right = newNode(10); /* 4   10   8   5 */
root->right->left = newNode(8); /*       /           */
root->right->right = newNode(5); /*           6           */
root->left->right->left = newNode(6);

printf("Count = %d", countSubtrees(root));
return 0;
}
```

Output:

Count = 6

## Source

<https://www.geeksforgeeks.org/number-subtrees-odd-count-even-numbers/>

## Chapter 294

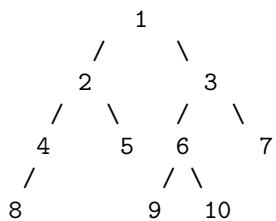
# Number of turns to reach from one node to other in binary tree

Number of turns to reach from one node to other in binary tree - GeeksforGeeks

Given a binary tree and two nodes. The task is to count the number of turns needed to reach from one node to another node of the Binary tree.

Examples:

Input: Below Binary Tree and two nodes  
5 & 6



Output: Number of Turns needed to reach  
from 5 to 6: 3

Input: For above tree if two nodes are 1 & 4

Output: Straight line : 0 turn

Idea based on the [Lowest Common Ancestor in a Binary Tree](#)

We have to follow the step.

1...Find the LCA of given two node

2...Given node present either on the left side or right side or equal to LCA.

.... According to above condition over program falls under two Case.

Case 1:

If none of the nodes is equal to LCA, we get these nodes either on the left side or right side.  
We call two functions for each node.

....a) if (CountTurn(LCA->right, first,  
                      false, &Count)  
         || CountTurn(LCA->left, first,  
                      true, &Count)) ;  
....b) Same for second node.  
....Here Count is used to store number of turns need to reached the target node.

Case 2:

If one of the nodes is equal to LCA\_Node.  
Then we count only number of turns needs to reached the second node.  
If LCA == (Either first or second)

....a) if (countTurn(LCA->right, second/first,  
                      false, &Count)  
         || countTurn(LCA->left, second/first,  
                      true, &Count)) ;

### 3... Working of CountTurn Function

// we pass turn **true** if we move  
// left subtree and **false** if we  
// move right subTree

```
CountTurn(LCA, Target_node, count, Turn)

// if found the key value in tree
if (root->key == key)
    return true;
case 1:
    If Turn is true that means we are
        in left_subtree
    If we going left_subtree then there
        is no need to increment count
    else
        Increment count and set turn as false
case 2:
    if Turn is false that means we are in
        right_subtree
    if we going right_subtree then there is
        no need to increment count else
        increment count and set turn as true.
```

```
// if key is not found.  
return false;
```

Below c++ implementation of above idea.

### C++

```
// C++ Program to count number of turns  
// in a Binary Tree.  
#include <iostream>  
using namespace std;  
  
// A Binary Tree Node  
struct Node {  
    struct Node* left, *right;  
    int key;  
};  
  
// Utility function to create a new  
// tree Node  
Node* newNode(int key)  
{  
    Node* temp = new Node;  
    temp->key = key;  
    temp->left = temp->right = NULL;  
    return temp;  
}  
  
// Utility function to find the LCA of  
// two given values n1 and n2.  
struct Node* findLCA(struct Node* root,  
                     int n1, int n2)  
{  
    // Base case  
    if (root == NULL)  
        return NULL;  
  
    // If either n1 or n2 matches with  
    // root's key, report the presence by  
    // returning root (Note that if a key  
    // is ancestor of other, then the  
    // ancestor key becomes LCA  
    if (root->key == n1 || root->key == n2)  
        return root;  
  
    // Look for keys in left and right subtrees
```

```
Node* left_lca = findLCA(root->left, n1, n2);
Node* right_lca = findLCA(root->right, n1, n2);

// If both of the above calls return
// Non-NULL, then one key is present
// in once subtree and other is present
// in other, So this node is the LCA
if (left_lca && right_lca)
    return root;

// Otherwise check if left subtree or right
// subtree is LCA
return (left_lca != NULL) ? left_lca :
                           right_lca;
}

// function count number of turn need to reach
// given node from it's LCA we have two way to
bool CountTurn(Node* root, int key, bool turn,
                int* count)
{
    if (root == NULL)
        return false;

    // if found the key value in tree
    if (root->key == key)
        return true;

    // Case 1:
    if (turn == true) {
        if (CountTurn(root->left, key, turn, count))
            return true;
        if (CountTurn(root->right, key, !turn, count)) {
            *count += 1;
            return true;
        }
    }
    else // Case 2:
    {
        if (CountTurn(root->right, key, turn, count))
            return true;
        if (CountTurn(root->left, key, !turn, count)) {
            *count += 1;
            return true;
        }
    }
    return false;
}
```

```
// Function to find nodes common to given two nodes
int NumberOfTurn(struct Node* root, int first,
                  int second)
{
    struct Node* LCA = findLCA(root, first, second);

    // there is no path between these two node
    if (LCA == NULL)
        return -1;
    int Count = 0;

    // case 1:
    if (LCA->key != first && LCA->key != second) {

        // count number of turns needs to reached
        // the second node from LCA
        if (CountTurn(LCA->right, second, false,
                      &Count)
            || CountTurn(LCA->left, second, true,
                          &Count))
            ;
        // count number of turns needs to reached
        // the first node from LCA
        if (CountTurn(LCA->left, first, true,
                      &Count)
            || CountTurn(LCA->right, first, false,
                          &Count))
            ;
        return Count + 1;
    }

    // case 2:
    if (LCA->key == first) {

        // count number of turns needs to reached
        // the second node from LCA
        CountTurn(LCA->right, second, false, &Count);
        CountTurn(LCA->left, second, true, &Count);
        return Count;
    } else {

        // count number of turns needs to reached
        // the first node from LCA
        CountTurn(LCA->right, first, false, &Count);
        CountTurn(LCA->left, first, true, &Count);
        return Count;
    }
}
```

```
        }
    }

// Driver program to test above functions
int main()
{
    // Let us create binary tree given in the above
    // example
    Node* root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->left = newNode(6);
    root->right->right = newNode(7);
    root->left->left->left = newNode(8);
    root->right->left->left = newNode(9);
    root->right->left->right = newNode(10);

    int turn = 0;
    if ((turn = NumberOTurn(root, 5, 10)))
        cout << turn << endl;
    else
        cout << "Not Possible" << endl;

    return 0;
}
```

### Java

```
//A java Program to count number of turns
//in a Binary Tree.
public class Turns_to_reach_another_node {

    // making Count global such that it can get
    // modified by different methods
    static int Count;

    // A Binary Tree Node
    static class Node {
        Node left, right;
        int key;

        // Constructor
        Node(int key) {
            this.key = key;
            left = null;
            right = null;
        }
    }
}
```

```
    }

}

// Utility function to find the LCA of
// two given values n1 and n2.
static Node findLCA(Node root, int n1, int n2) {
    // Base case
    if (root == null)
        return null;

    // If either n1 or n2 matches with
    // root's key, report the presence by
    // returning root (Note that if a key
    // is ancestor of other, then the
    // ancestor key becomes LCA
    if (root.key == n1 || root.key == n2)
        return root;

    // Look for keys in left and right subtrees
    Node left_lca = findLCA(root.left, n1, n2);
    Node right_lca = findLCA(root.right, n1, n2);

    // If both of the above calls return
    // Non-NULL, then one key is present
    // in once subtree and other is present
    // in other, So this node is the LCA
    if (left_lca != null && right_lca != null)
        return root;

    // Otherwise check if left subtree or right
    // subtree is LCA
    return (left_lca != null) ? left_lca : right_lca;
}

// function count number of turn need to reach
// given node from it's LCA we have two way to
static boolean CountTurn(Node root, int key, boolean turn) {
    if (root == null)
        return false;

    // if found the key value in tree
    if (root.key == key)
        return true;

    // Case 1:
    if (turn == true) {
        if (CountTurn(root.left, key, turn))
            return true;
```

```
if (CountTurn(root.right, key, !turn)) {
    Count += 1;
    return true;
}
} else // Case 2:
{
    if (CountTurn(root.right, key, turn))
        return true;
    if (CountTurn(root.left, key, !turn)) {
        Count += 1;
        return true;
    }
}
return false;
}

// Function to find nodes common to given two nodes
static int NumberOfTurn(Node root, int first, int second) {
    Node LCA = findLCA(root, first, second);

    // there is no path between these two node
    if (LCA == null)
        return -1;
    Count = 0;

    // case 1:
    if (LCA.key != first && LCA.key != second) {

        // count number of turns needs to reached
        // the second node from LCA
        if (CountTurn(LCA.right, second, false)
            || CountTurn(LCA.left, second, true))
            ;

        // count number of turns needs to reached
        // the first node from LCA
        if (CountTurn(LCA.left, first, true)
            || CountTurn(LCA.right, first, false))
            ;
    }
    return Count + 1;
}

// case 2:
if (LCA.key == first) {

    // count number of turns needs to reached
    // the second node from LCA
    CountTurn(LCA.right, second, false);
}
```

```
CountTurn(LCA.left, second, true);
    return Count;
} else {

    // count number of turns needs to reached
    // the first node from LCA1
    CountTurn(LCA.right, first, false);
    CountTurn(LCA.left, first, true);
    return Count;
}
}

// Driver program to test above functions
public static void main(String[] args) {
    // Let us create binary tree given in the above
    // example
    Node root = new Node(1);
    root.left = new Node(2);
    root.right = new Node(3);
    root.left.left = new Node(4);
    root.left.right = new Node(5);
    root.right.left = new Node(6);
    root.right.right = new Node(7);
    root.left.left.left = new Node(8);
    root.right.left.left = new Node(9);
    root.right.left.right = new Node(10);

    int turn = 0;
    if ((turn = NumberOfTurn(root, 5, 10)) != 0)
        System.out.println(turn);
    else
        System.out.println("Not Possible");
}

// This code is contributed by Sumit Ghosh
```

Output:

4

Time Complexity : O(n)

## Source

<https://www.geeksforgeeks.org/number-turns-reach-one-node-binary-tree/>

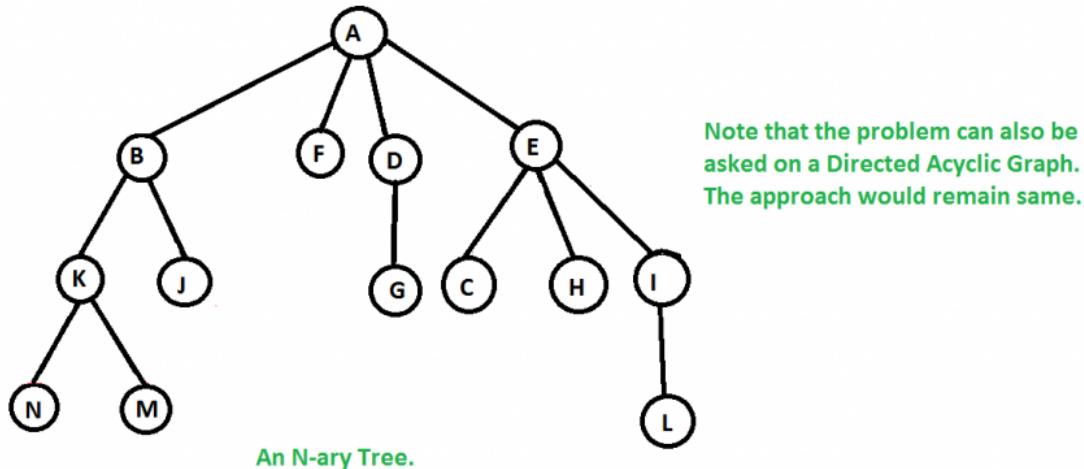
## Chapter 295

# Number of ways to traverse an N-ary tree

Number of ways to traverse an N-ary tree - GeeksforGeeks

Given an n-ary tree, count number of ways to traverse an n-ary (or a Directed Acyclic Graph) tree starting from the root vertex.

Suppose we have a given N-ary tree as shown below.



Now we have to find the number of ways of traversing the whole tree starting from the root vertex. There can be many such ways. Some of them are listed below.

- 1) N->M->K->J->B->F->D->E->C->H->I->L->A (kind-of depth first traversal).
- 2) A->B->F->D->E->K->J->G->C->H->I->N->M->L (level order traversal)
- 3) .....

4) .....

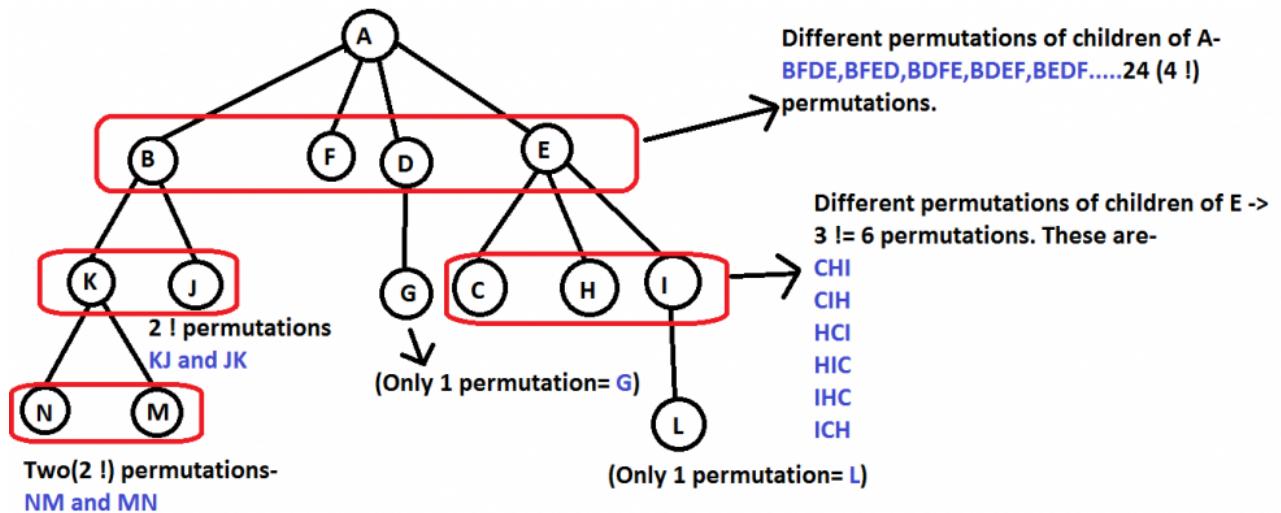
.

.

and so on....

We strongly recommend you to minimize your browser and try this yourself first.

The count of all ways to traverse is the product of factorials of the number of children of each node. Refer to the below figure for clear understanding-



Here,

'A' has four children, so  $4!$  permutations possible  
'B' has two children, so  $2!$  permutations possible  
'F' has no children, so  $0!$  permutations possible

.....

And so on

Hence all such ways are-  $4! * 2! * 0! * 1! * 3! * 2! * 0! * 0! * 0! * 0! * 1! * 0! * 0!$   
 $= 576$  way

That's a huge number of ways and among them only few proves to be useful, like- inorder, level-order, preorder, postorder (arranged according to the popularity of these traversals)

```
// C++ program to find the number of ways to traverse a
// n-ary tree starting from the root node
#include <bits/stdc++.h>
using namespace std;

// Structure of a node of an n-ary tree
struct Node
```

```
{  
    char key;  
    vector<Node *> child;  
};  
  
// Utility function to create a new tree node  
Node *newNode(int key)  
{  
    Node *temp = new Node;  
    temp->key = key;  
    return temp;  
}  
  
// Utility Function to find factorial of given number  
int factorial(int n)  
{  
    if (n == 0)  
        return 1;  
    return n*factorial(n-1);  
}  
  
// Function to calculate the number of ways of traversing  
// the n-ary starting from root.  
// This function is just a modified breadth-first search.  
// We can use a depth-first search too.  
int calculateWays(Node * root)  
{  
    int ways = 1; // Initialize result  
  
    // If the tree is empty there is no way of traversing  
    // the tree.  
    if (root == NULL)  
        return 0;  
  
    // Create a queue and enqueue root to it.  
    queue<Node *>q;  
    q.push(root);  
  
    // Level order traversal.  
    while (!q.empty())  
    {  
        // Dequeue an item from queue and print it  
        Node * p = q.front();  
        q.pop();  
  
        // The number of ways is the product of  
        // factorials of number of children of each node.  
        ways = ways*(factorial(p->child.size()));  
    }  
}
```

```

        // Enqueue all children of the dequeued item
        for (int i=0; i<p->child.size(); i++)
            q.push(p->child[i]);
    }

    return(ways);
}

// Driver program
int main()
{
    /* Let us create below tree
     *      A
     *      / \   \
     *     B   F   D   E
     *     / \           |   /|\
     *    K   J           G   C H I
     *    / \           \
     *   N   M           L
    */

    Node *root = newNode('A');
    (root->child).push_back(newNode('B'));
    (root->child).push_back(newNode('F'));
    (root->child).push_back(newNode('D'));
    (root->child).push_back(newNode('E'));
    (root->child[0]->child).push_back(newNode('K'));
    (root->child[0]->child).push_back(newNode('J'));
    (root->child[2]->child).push_back(newNode('G'));
    (root->child[3]->child).push_back(newNode('C'));
    (root->child[3]->child).push_back(newNode('H'));
    (root->child[3]->child).push_back(newNode('I'));
    (root->child[0]->child[0]->child).push_back(newNode('N'));
    (root->child[0]->child[0]->child).push_back(newNode('M'));
    (root->child[3]->child[2]->child).push_back(newNode('L'));

    cout << calculateWays(root); ;

    return 0;
}

```

Output :

576

**Time Complexity:** We visit each node once during the level order traversal and take  $O(n)$  time to compute factorial for every node. Total time taken is  $O(Nn)$  where  $N =$

number of nodes in the n-ary tree. We can optimize the solution to work in  $O(N)$  time by pre-computing factorials of all numbers from 1 to  $n$ .

**Auxiliary Space :** Since we are only using a queue and a structure for every node, so overall space complexity is also  $O(N)$ .

**Common Pitfalls:** Since, products of factorials can tend to grow very huge, so it may overflow. It is preferable to use data types like- unsigned long long int in C/C++, as the number of ways can never be a negative number. In Java and Python there are BigInteger to take care of overflows.

## Source

<https://www.geeksforgeeks.org/number-of-ways-to-traverse-an-n-ary-tree/>

## Chapter 296

# Overview of Data Structures | Set 2 (Binary Tree, BST, Heap and Hash)

Overview of Data Structures | Set 2 (Binary Tree, BST, Heap and Hash) - GeeksforGeeks

We have discussed [Overview of Array](#), [Linked List](#), [Queue and Stack](#). In this article following Data Structures are discussed.

- 5. [Binary Tree](#)
- 6. [Binary Search Tree](#)
- 7. [Binary Heap](#)
- 9. [Hashing](#)

### Binary Tree

Unlike Arrays, Linked Lists, Stack and queues, which are linear data structures, trees are hierarchical data structures.

A binary tree is a tree data structure in which each node has at most two children, which are referred to as the left child and the right child. It is implemented mainly using Links.

**Binary Tree Representation:** A tree is represented by a pointer to the topmost node in tree. If the tree is empty, then value of root is NULL. A Binary Tree node contains following parts.

- 1. Data
- 2. Pointer to left child
- 3. Pointer to right child

A Binary Tree can be traversed in two ways:

Depth First Traversal: Inorder (Left-Root-Right), Preorder (Root-Left-Right) and Postorder (Left-Right-Root)

Breadth First Traversal: Level Order Traversal

### Binary Tree Properties:

The maximum number of nodes at level 'l' =  $2^l - 1$ .

Maximum number of nodes =  $2^h - 1$ .

Here h is height of a tree. Height is considered as is maximum number of nodes on root to leaf path

Minimum possible height =  $\lceil \log_2(n+1) \rceil$

In Binary tree, number of leaf nodes is always one more than nodes with two children.

Time Complexity of Tree Traversal:  $O(n)$

**Examples :** One reason to use binary tree or tree in general is for the things that form a hierarchy. They are useful in File structures where each file is located in a particular directory and there is a specific hierarchy associated with files and directories. Another example where Trees are useful is storing hierarchical objects like JavaScript Document Object Model considers HTML page as a tree with nesting of tags as parent child relations.

### Binary Search Tree

In Binary Search Tree is a Binary Tree with following additional properties:

1. The left subtree of a node contains only nodes with keys less than the node's key.
2. The right subtree of a node contains only nodes with keys greater than the node's key.
3. The left and right subtree each must also be a binary search tree.

Time Complexities:

Search :  $O(h)$

Insertion :  $O(h)$

Deletion :  $O(h)$

Extra Space :  $O(n)$  for pointers

h: Height of BST

n: Number of nodes in BST

If Binary Search Tree is Height Balanced,  
then  $h = O(\log n)$

Self-Balancing BSTs such as AVL Tree, Red-Black Tree and Splay Tree make sure that height of BST remains  $O(\log n)$

BST provide moderate access/search (quicker than Linked List and slower than arrays).  
BST provide moderate insertion/deletion (quicker than Arrays and slower than Linked Lists).

**Examples :** Its main use is in search application where data is constantly entering/leaving and data needs to be printed in sorted order. For example in implementation in E-commerce

websites where a new product is added or product goes out of stock and all products are listed in sorted order.

### **Binary Heap**

A Binary Heap is a Binary Tree with following properties.

- 1) It's a complete tree (All levels are completely filled except possibly the last level and the last level has all keys as left as possible). This property of Binary Heap makes them suitable to be stored in an array.
- 2) A Binary Heap is either Min Heap or Max Heap. In a Min Binary Heap, the key at root must be minimum among all keys present in Binary Heap. The same property must be recursively true for all nodes in Binary Tree. Max Binary Heap is similar to Min Heap. It is mainly implemented using array.

Get Minimum in Min Heap:  $O(1)$  [Or Get Max in Max Heap]

Extract Minimum Min Heap:  $O(\log n)$  [Or Extract Max in Max Heap]

Decrease Key in Min Heap:  $O(\log n)$  [Or Extract Max in Max Heap]

Insert:  $O(\log n)$

Delete:  $O(\log n)$

**Example :** Used in implementing efficient priority-queues, which in turn are used for scheduling processes in operating systems. Priority Queues are also used in Dijkstra's and Prim's graph algorithms.

The Heap data structure can be used to efficiently find the k smallest (or largest) elements in an array, merging k sorted arrays, median of a stream, etc.

Heap is a special data structure and it cannot be used for searching of a particular element.

**HashingHash Function:** A function that converts a given big input key to a small practical integer value. The mapped integer value is used as an index in hash table. A good hash function should have following properties

- 1) Efficiently computable.
- 2) Should uniformly distribute the keys (Each table position equally likely for each key)

**Hash Table:** An array that stores pointers to records corresponding to a given phone number. An entry in hash table is NIL if no existing phone number has hash function value equal to the index for the entry.

**Collision Handling:** Since a hash function gets us a small number for a key which is a big integer or string, there is possibility that two keys result in same value. The situation where a newly inserted key maps to an already occupied slot in hash table is called collision and must be handled using some collision handling technique. Following are the ways to handle collisions:

**Chaining:** The idea is to make each cell of hash table point to a linked list of records that have same hash function value. Chaining is simple, but requires additional memory outside the table.

**Open Addressing:** In open addressing, all elements are stored in the hash table itself. Each table entry contains either a record or NIL. When searching for an element, we one by one examine table slots until the desired element is found or it is clear that the element is not in the table.

Space :	$O(n)$
Search :	$O(1)$ [Average] $O(n)$ [Worst case]
Insertion :	$O(1)$ [Average] $O(n)$ [Worst Case]
Deletion :	$O(1)$ [Average] $O(n)$ [Worst Case]

Hashing seems better than BST for all the operations. But in hashing, elements are unordered and in BST elements are stored in an ordered manner. Also BST is easy to implement but hash functions can sometimes be very complex to generate. In BST, we can also efficiently find floor and ceil of values.

**Example :** Hashing can be used to remove duplicates from a set of elements. Can also be used find frequency of all items. For example, in web browsers, we can check visited urls using hashing. In firewalls, we can use hashing to detect spam. We need to hash IP addresses. Hashing can be used in any situation where want search() insert() and delete() in  $O(1)$  time.

This article is contributed by **Abhiraj Smit**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

**Improved By :** [Rahul1421](#)

## Source

<https://www.geeksforgeeks.org/overview-of-data-structures-set-2-binary-tree-bst-heap-and-hash/>

## Chapter 297

# Pairs involved in Balanced Parentheses

Pairs involved in Balanced Parentheses - GeeksforGeeks

Given a string of brackets, task is to find the number of pairs of brackets involved in a balanced sequence in a given range.

**Examples :**

```
Input : ((())()
Range : 1 5
Range : 3 8
Output : 2
         2
Explanation : In range 1 to 5 ((()),
there are the two pairs. In range 3 to 8 (),
(), there are the two pairs.
```

```
Input : )()())
Range : 1 2
Range : 4 7
Output : 0
         1
Explanation : In range 1 to 2 )( there
is no any pair. In range 4 to 7 (()),
there is the only pair
```

**Prerequisite :** Segment Trees

**Approach :**

Here, in segment tree, for each node, keep some simple elements, like integers or sets or

vectors or etc.

For each node keep three integers :

1. t = Answer for the interval.
2. o = The number of opening brackets '(' remaining after deleting the brackets those who belong to the correct bracket sequence in this interval with length t.
3. c = The number of closing brackets ')' remaining after deleting the brackets those who belong to the correct bracket sequence in this interval with length t.

Now, having these variables, queries can be answered easily using segment tree.

**Below is the implementation of above approach :**

```
// CPP code to find the number of pairs
// involved in balanced parentheses
#include <bits/stdc++.h>
using namespace std;

// Our struct node
struct node {

    // three variables required
    int t, o, c;
};

// Declare array of nodes of very big
// size which acts as segment tree here.
struct node tree_arr[5 * 1000];

// To build a segment tree we pass 1 as
// 'id' 0 as 'l' and 1 as 'n'.
// Here, we consider query's interval as [x, y)
void build(int id, int l, int r, string s)
{
    /* this base condition is common to
       build any segment tree*/
    // This is the base
    // Only one element left
    if (r - l < 2) {

        // If that element is open bracket
        if (s[l] == '(')
            tree_arr[id].o = 1;

        // If that element is open bracket
        else
            tree_arr[id].c = 1;
        return;
    }

    // Next three lines are common
```

```

// for any segment tree.
int mid = (l + r) / 2;

// for left tree
build(2 * id, l, mid, s);

// for right tree
build(2 * id + 1, mid, r, s);

// Here we take minimum of left tree
// opening brackets and right tree
// closing brackets
int tmp = min(tree_arr[2 * id].o,
              tree_arr[2 * id + 1].c);

// we add that to our answer.
tree_arr[id].t = tree_arr[2 * id].t +
                 tree_arr[2 * id + 1].t + tmp;

// Remove the answer from opening brackets
tree_arr[id].o = tree_arr[2 * id].o +
                 tree_arr[2 * id + 1].o - tmp;

// Remove the answer from opening brackets
tree_arr[id].c = tree_arr[2 * id].c +
                 tree_arr[2 * id + 1].c - tmp;
}

// This will return the answer for each query.
// Here we consider query's interval as [x, y)
node segment(int x, int y, int id,
             int l, int r, string s)
{
    // If the given interval is out of range
    if (l >= y || x >= r) {
        struct node tem;
        tem.t = 0;
        tem.o = 0;
        tem.c = 0;
        return tem;
    }

    // If the given interval completely lies
    if (x <= l && r <= y)
        return tree_arr[id];

    // Next three lines are common for
    // any segment tree.

```

```
int mid = (l + r) / 2;

// For left tree
struct node a =
    segment(x, y, 2 * id, l, mid, s);

// For right tree
struct node b =
    segment(x, y, 2 * id + 1, mid, r, s);

// Same as made in build function
int temp;
temp = min(a.o, b.c);
struct node vis;
vis.t = a.t + b.t + temp;
vis.o = a.o + b.o - temp;
vis.c = a.c + b.c - temp;

return vis;
}

// Driver code
int main()
{
    string s = "((())())";
    int n = s.size();

    // range for query
    int a = 3, b = 8;

    build(1, 0, n, s);

    // Here we consider query's interval as [a, b)
    // We subtract 1 from 'a' because indexes start
    // from 0.
    struct node p = segment(a-1, b, 1, 0, n, s);
    cout << p.t << endl;

    return 0;
}
```

**Output:**

**Source**

<https://www.geeksforgeeks.org/pairs-involved-balanced-parentheses/>

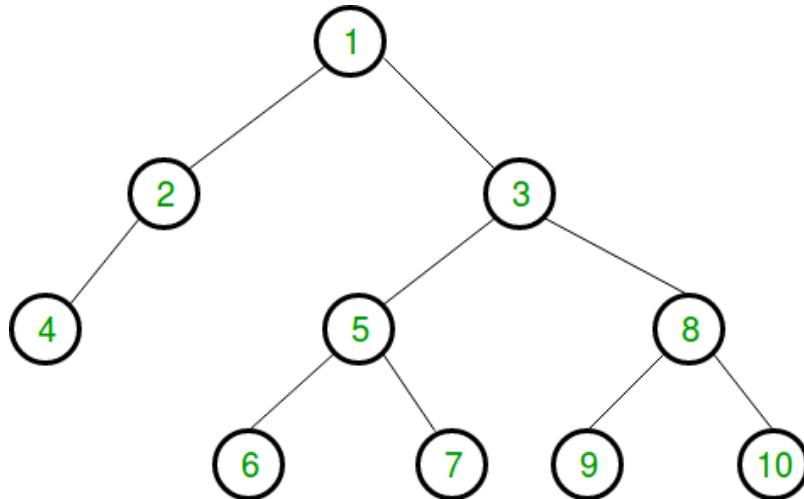
## Chapter 298

# Pairwise Swap leaf nodes in a binary tree

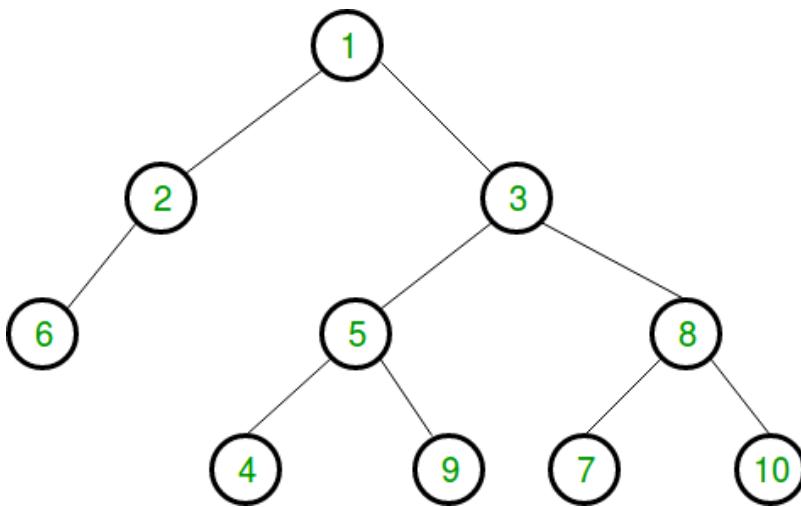
Pairwise Swap leaf nodes in a binary tree - GeeksforGeeks

Given a binary tree, we need to write a program to swap leaf nodes in the given binary tree pairwise starting from left to right as shown below.

Tree before swapping:



Tree after swapping:



The sequence of leaf nodes in original binary tree from left to right is (4, 6, 7, 9, 10). Now if we try to form pairs from this sequence, we will have two pairs as (4, 6), (7, 9). The last node (10) is unable to form pair with any node and thus left unswapped.

The idea to solve this problem is to first [traverse the leaf nodes of the binary tree from left to right](#).

While traversing the leaf nodes, we maintain two pointers to keep track of first and second leaf nodes in a pair and a variable *count* to keep track of count of leaf nodes traversed. Now, if we observe carefully then we see that while traversing if the count of leaf nodes traversed is even, it means that we can form a pair of leaf nodes. To keep track of this pair we take two pointers *firstPtr* and *secondPtr* as mentioned above. Every time we encounter a leaf node we initialize *secondPtr* with this leaf node. Now if the *count* is odd, we initialize *firstPtr* with *secondPtr* otherwise we simply swap these two nodes.

Below is the C++ implementation of above idea:

```
/* C++ program to pairwise swap
leaf nodes from left to right */
#include <bits/stdc++.h>
using namespace std;

// A Binary Tree Node
struct Node
{
    int data;
    struct Node *left, *right;
};

// function to swap two Node
void Swap(Node **a, Node **b)
{
    Node * temp = *a;
    *a = *b;
    *b = temp;
}
```

```
*b = temp;
}

// two pointers to keep track of
// first and second nodes in a pair
Node **firstPtr;
Node **secondPtr;

// function to pairwise swap leaf
// nodes from left to right
void pairwiseSwap(Node **root, int &count)
{
    // if node is null, return
    if (!(*root))
        return;

    // if node is leaf node, increment count
    if(!(*root)->left&&!(*root)->right)
    {
        // initialize second pointer
        // by current node
        secondPtr = root;

        // increment count
        count++;

        // if count is even, swap first
        // and second pointers
        if (count%2 == 0)
            Swap(firstPtr, secondPtr);

        else

            // if count is odd, initialize
            // first pointer by second pointer
            firstPtr = secondPtr;
    }

    // if left child exists, check for leaf
    // recursively
    if ((*root)->left)
        pairwiseSwap(&(*root)->left, count);

    // if right child exists, check for leaf
    // recursively
    if ((*root)->right)
        pairwiseSwap(&(*root)->right, count);
}
```

```
}

// Utility function to create a new tree node
Node* newNode(int data)
{
    Node *temp = new Node;
    temp->data = data;
    temp->left = temp->right = NULL;
    return temp;
}

// function to print inorder traversal
// of binary tree
void printInorder(Node* node)
{
    if (node == NULL)
        return;

    /* first recur on left child */
    printInorder(node->left);

    /* then print the data of node */
    printf("%d ", node->data);

    /* now recur on right child */
    printInorder(node->right);
}

// Driver program to test above functions
int main()
{
    // Let us create binary tree shown in
    // above diagram
    Node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->right->left = newNode(5);
    root->right->right = newNode(8);
    root->right->left->left = newNode(6);
    root->right->left->right = newNode(7);
    root->right->right->left = newNode(9);
    root->right->right->right = newNode(10);

    // print inorder traversal before swapping
    cout << "Inorder traversal before swap:\n";
    printInorder(root);
    cout << "\n";
}
```

```
// variable to keep track
// of leafs traversed
int c = 0;

// Pairwise swap of leaf nodes
pairwiseSwap(&root, c);

// print inorder traversal after swapping
cout << "Inorder traversal after swap:\n";
printInorder(root);
cout << "\n";

return 0;
}
```

Output:

```
Inorder traversal before swap:
4 2 1 6 5 7 3 9 8 10
Inorder traversal after swap:
6 2 1 4 5 9 3 7 8 10
```

## Source

<https://www.geeksforgeeks.org/pairwise-swap-leaf-nodes-binary-tree/>

## Chapter 299

# Palindromic Tree | Introduction & Implementation

Palindromic Tree | Introduction & Implementation - GeeksforGeeks

We encounter various problems like Maximum length palindrome in a string, number of palindromic substrings and many more interesting problems on palindromic substrings . Mostly of these palindromic substring problems have some DP  $O(n^2)$  solution (n is length of the given string) or then we have a complex algorithm like [Manacher's algorithm](#) which solves the Palindromic problems in linear time.

In this article, we will study an interesting Data Structure, which will solve all the above similar problems in much more simpler way. This data structure is invented by [Mikhail Rubinchik](#).

Features of Palindromic Tree : Online query and updation  
Easy to implement  
Very Fast

### Structure of Palindromic Tree

Palindromic Tree's actual structure is **close to directed graph**. It is actually a merged structure of two Trees which share some common nodes(see the figure below for better understanding). Tree nodes store palindromic substrings of given string by storing their indices.

This tree consists of two types of edges :

- 1) Insertion edge (weighted edge)
- 2) Maximum Palindromic Suffix (un-weighted)

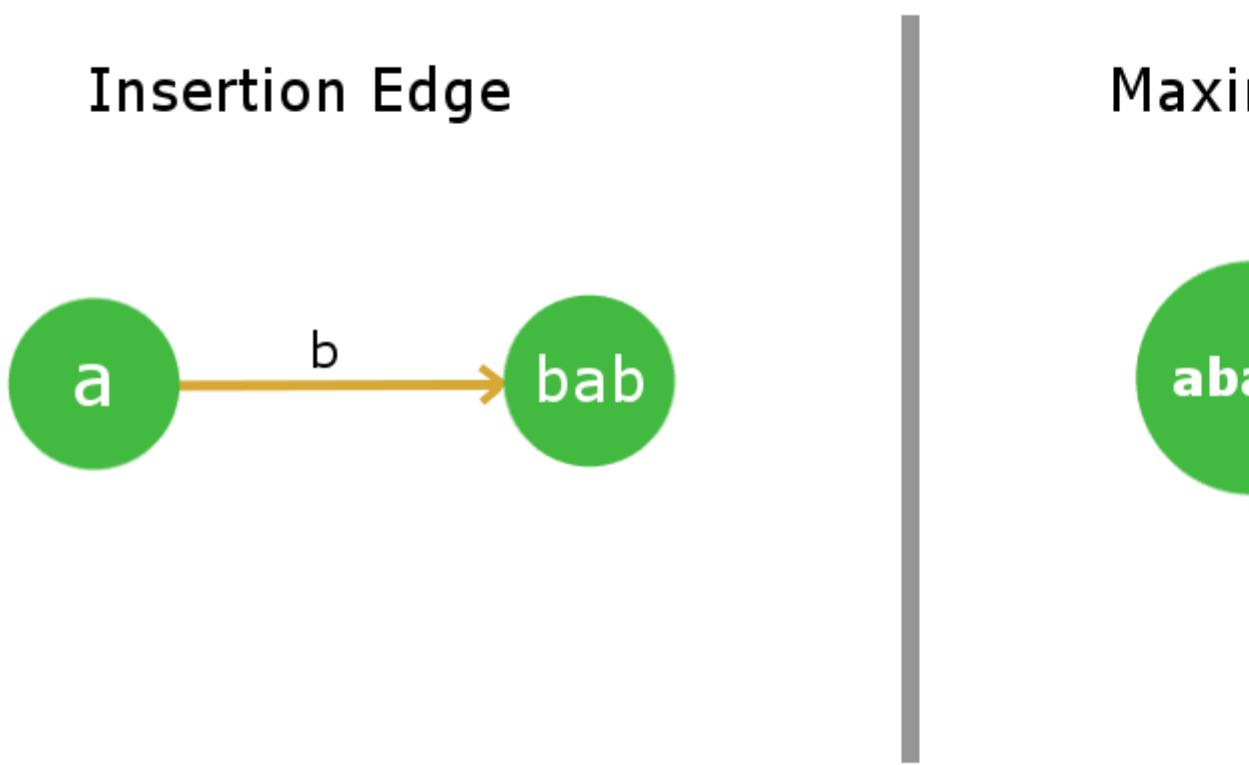
#### Insertion Edge :

Insertion edge from a node **u** to **v** with some weight **x** means that the node v is formed by inserting x at the front and end of the string at u. As u is already a palindrome, hence the resulting string at node v will also be a palindrome.

$x$  will be a single character for every edge. Therefore, a node can have max 26 insertion edges (considering lower letter string). We will use orange color for this edge in our pictorial representation.

#### Maximum Palindromic Suffix Edge:

As the name itself indicates that for a node this edge will point to its Maximum Palindromic Suffix String node. We will not be considering the complete string itself as the Maximum Palindromic Suffix as this will make no sense(self loops). For simplicity purpose, we will call it as Suffix edge(by which we mean maximum suffix except the complete string). It is quite obvious that every node will have only 1 Suffix Edge as we will not store duplicate strings in the tree. We will use Blue dashed edges for its Pictorial representation.



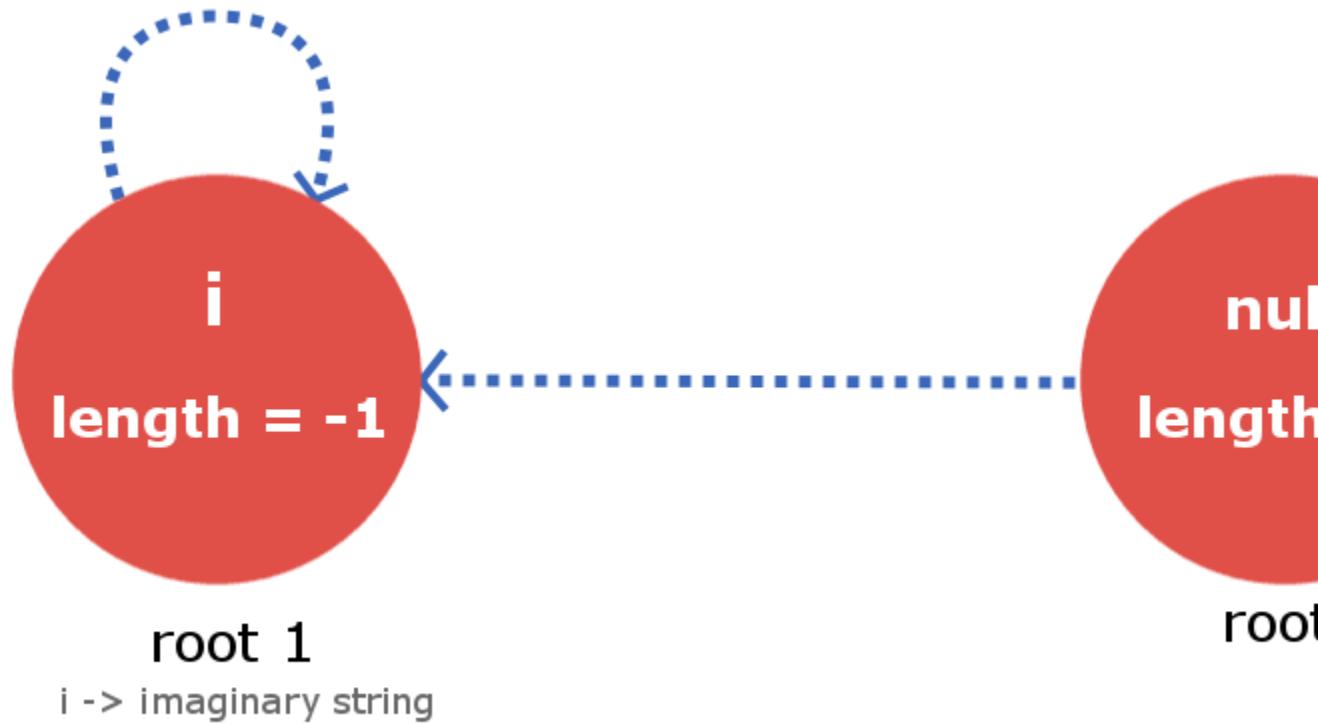
#### Root Nodes and their convention:

This tree/graph data structure will contain **2 root dummy nodes**. More, precisely consider it as roots of two separate trees, which are linked together.

*Root-1* will be a dummy node which will describe a string for  $length = -1$  ( you can easily infer from the implementation point of view that why we used so ). *Root-2* will be a node which will describe a null string of  $length = 0$ .

*Root-1* has a suffix edge connected to itself(self-loop) as for any imaginary string of length  $-1$  , its Maximum palindromic suffix will also be imaginary, so this is justified. Now *Root-2*

will also have its suffix edge connected to Root-1 as for a null string (length 0) there is no real palindromic suffix string of length less than 0.



### Building the Palindromic Tree

To build a Palindromic Tree, we will simply insert the characters in our string one by one till we reach its end and when we are done inserting we will be with our palindromic tree which will contain all the distinct palindromic substrings of the given strings. All we need to ensure is that, at every insertion of a new character, our palindromic tree maintains the above discussed feature. Let's see how we can accomplish it.

Let's say we are given a string  $s$  with length  $l$  and we have inserted the string up till index  $k$  ( $k < l-1$ ). Now, we need to insert the  $(k+1)$ th character. Insertion of  $(k+1)$ th character means insertion of a node that is longest palindrome ending at index  $(k+1)$ . So, the longest palindromic string will be of form (' $s[k+1]$ ' + "X" + ' $s[k+1]$ ') and X will itself be a palindrome. Now the fact is that the string X lies at index  $< k+1$  and is palindrome. So, it will already exist in our palindromic tree as we have maintained the very basic property of it saying that it will contain all the distinct palindromic substrings.

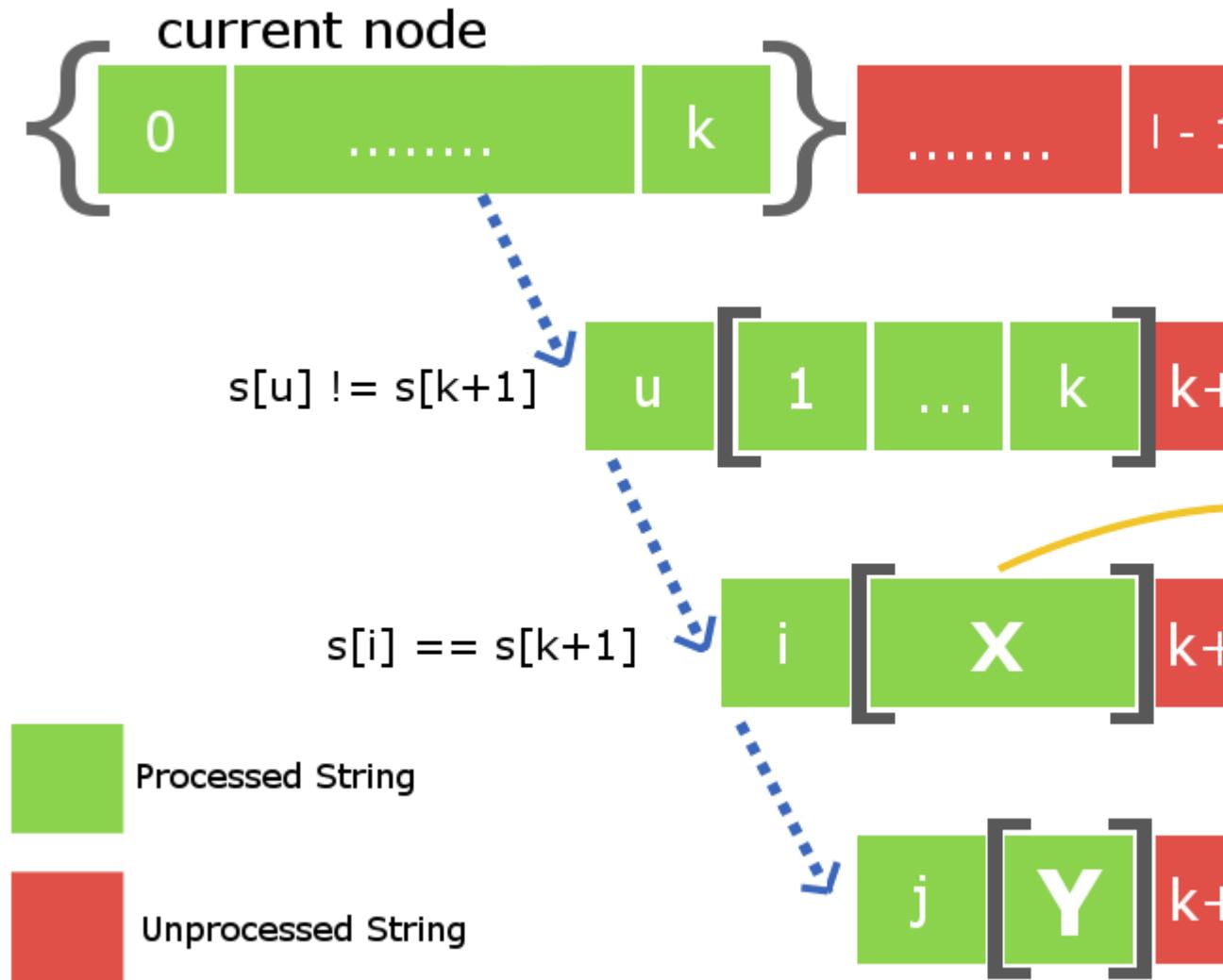
So, to insert the character  $s[k+1]$ , we only need to find the String X in our tree and direct the insertion edge from X with weight  $s[k+1]$  to a new node, which contains  $s[k+1]+X+s[k+1]$ . The main job now is to find the string X in efficient time. As we know that we are storing the suffix link for all the nodes. Therefore to track the node with

string X we just need to move down the suffix link for the current node i.e the node which contains insertion of  $s[k]$ . See the below image for better understanding.

The current node in the below figure tells that it is the largest palindrome that ends at index k after processing all the indices from 0 to k. The blue dotted path is the link of suffix edges from current node to other processed nodes in the tree. String X will exist in one of these nodes that lie on this chain of suffix link. All we need is to find it by iterating over it down the chain.

To find the required node that contains the string X we will place the  $k+1$  th character at the end of every node that lies in suffix link chain and check if first character of the corresponding suffix link string is equal to the  $k+1$  th character.

Once, we find the X string we will direct an insertion edge with weight  $s[k+1]$  and link it to the new node that contains largest palindrome ending at index  $k+1$ . The array elements between the brackets as described in the figure below are the nodes that are stored in the tree.



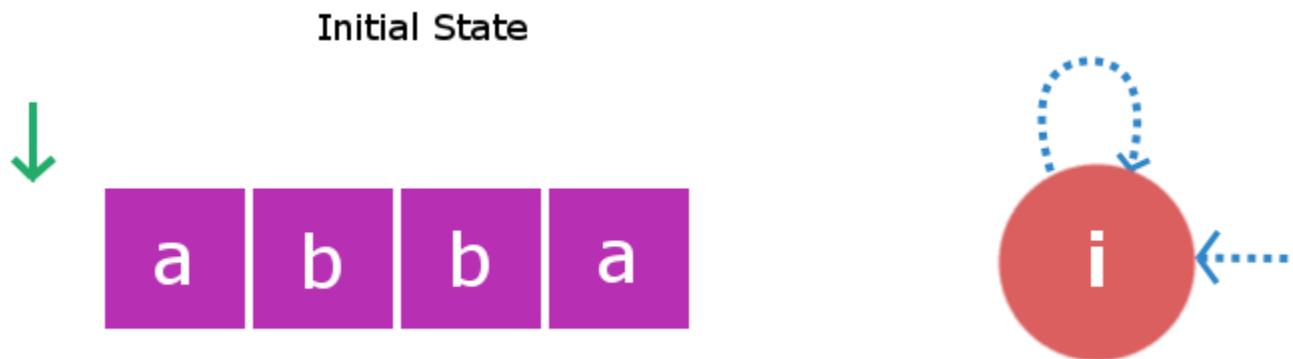
There is one more thing left that is to be done. As we have created a new node at this  $s[k+1]$  insertion, therefore we will also have to connect it with its suffix link child. Once, again to do so we will use the above down the suffix link iteration from node X to find some new string Y such that  $s[k+1] + Y + s[k+1]$  is a largest palindromic suffix for the newly created node. Once, we find it we will then connect the suffix link of our newly created node with the node Y.

**Note :** There are two possibilities when we find the string X. First possibility is that string  $s[k+1]Xs[k+1]$  do not exist in the tree and second possibility is if it already exists in the tree. In first case we will proceed the same way but in second case we will not

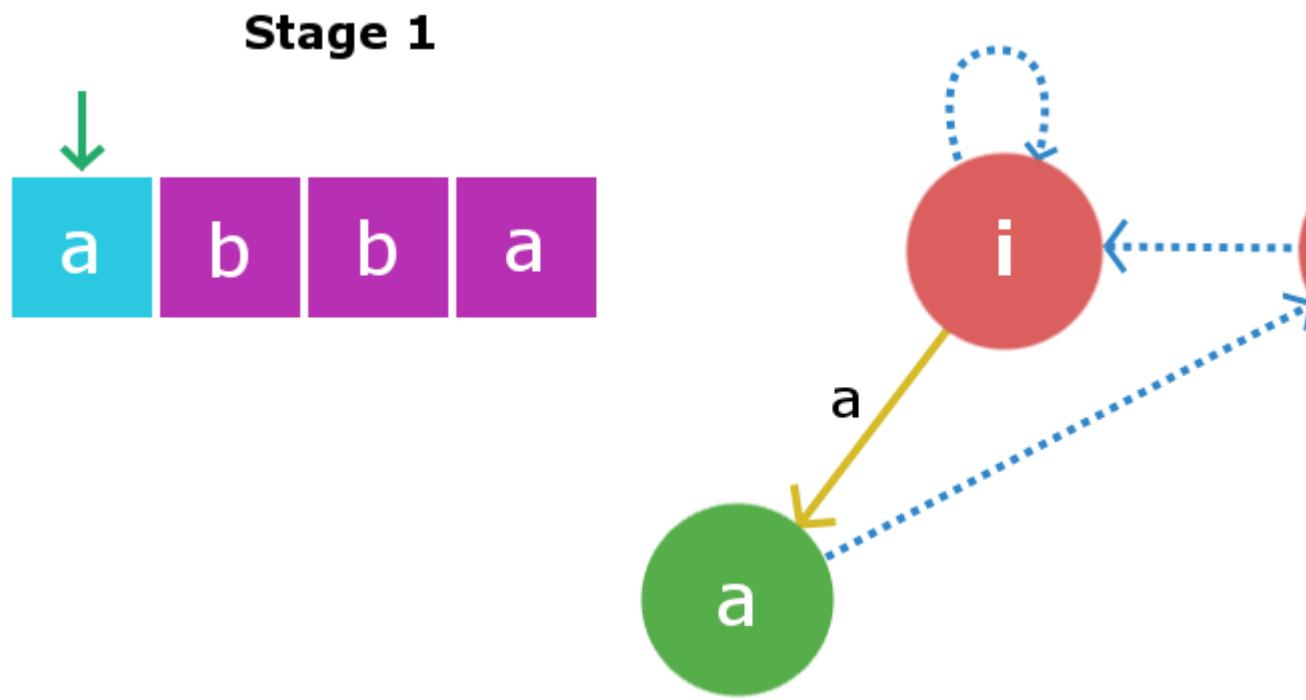
create a new node separately but will just link the insertion edge from X to already existing  $S[k+1]+X+S[k+1]$  node in the tree. We also need not to add the suffix link because the node will already contain its suffix link.

Consider a string  $s = \text{"abba"}$  with  $length = 4$ .

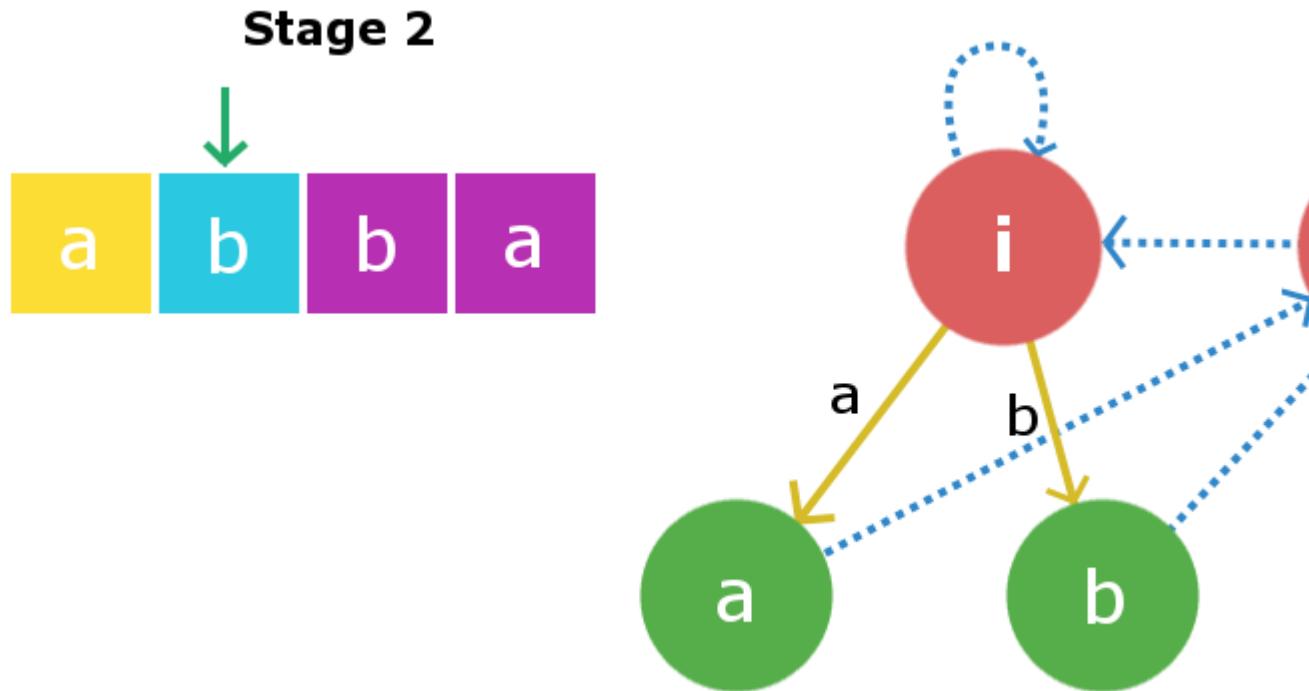
At initial state we will have our two dummy root nodes one with length -1 ( some imaginary string **i** ) and second a **null** string with length 0. At this point we haven't inserted any character in the tree. Root1 i.e root node with length -1 will be the current node from which insertion takes place.



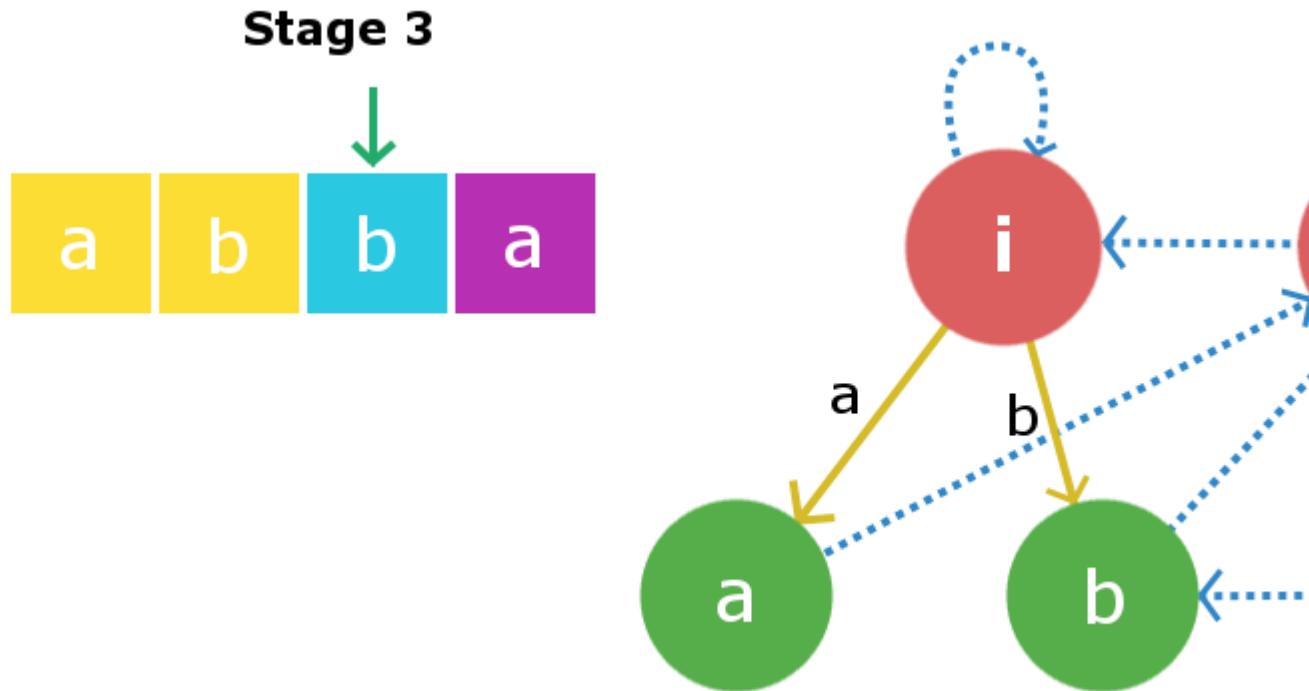
Stage 1: We will insert  $s[0]$  i.e 'a'. We will start checking from the current node i.e Root1. Inserting 'a' at start and end of a string with length -1 will yield to a string with length 1 and this string will be "a". Therefore, we create a new node "a" and direct insertion edge from root1 to this new node. Now, largest palindromic suffix string for string of length 1 will be a null string so its suffix link will be directed to root2 i.e null string. Now the current node will be this new node "a".



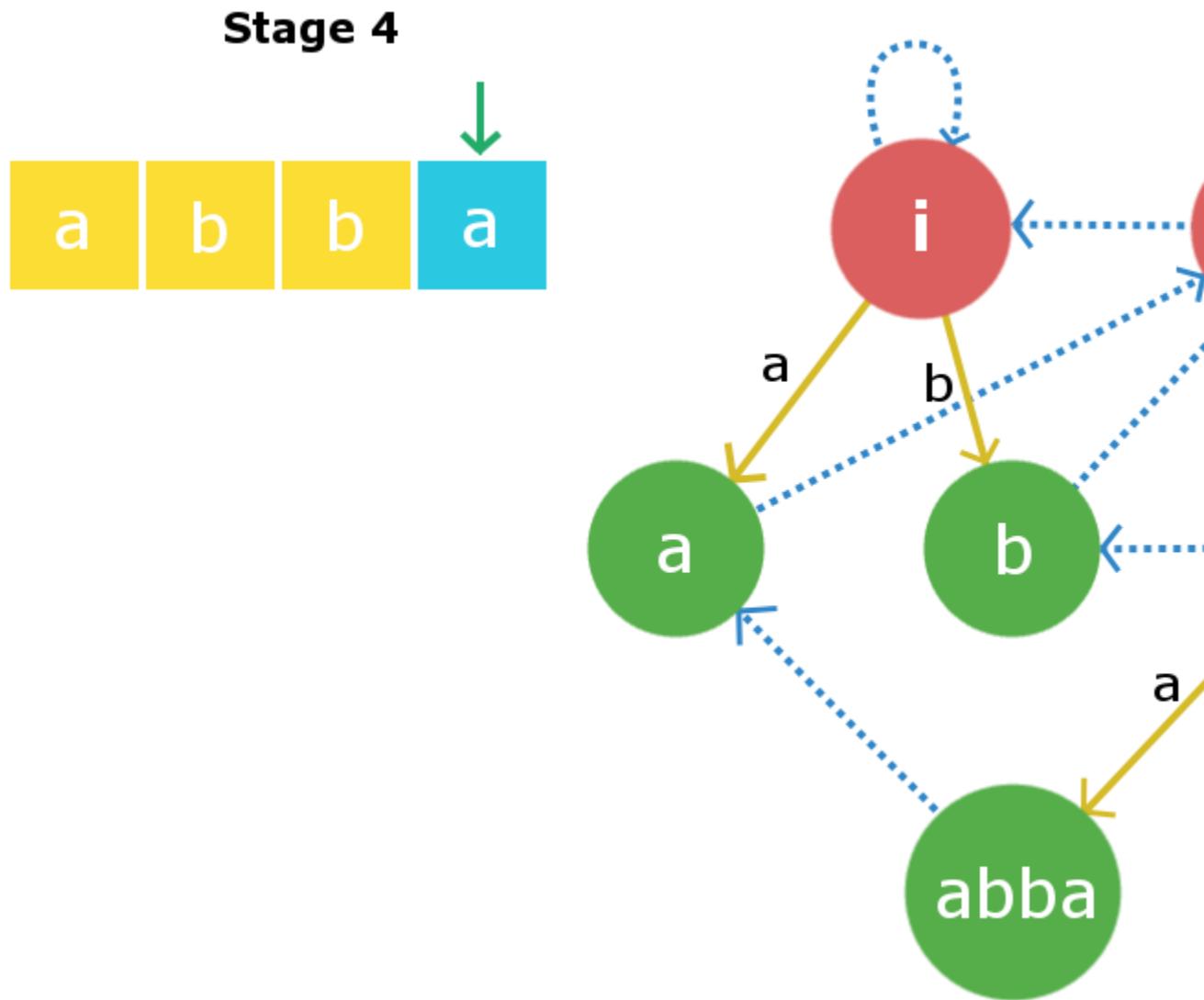
Stage 2: We will insert  $s[1]$  i.e ‘b’. Insertion process will start from current node i.e “a” node. We will traverse the suffix link chain starting from current node till we find suitable X string , So here traversing the suffix link we again found root1 as X string. Once again inserting ‘b’ to string of length -1 will yield a string of of length 1 i.e string “b”. Suffix link for this node will go to null string as described in above insertion. Now the current node will be this new node “b”.



Stage 3: We will insert  $s[2]$  i.e ‘b’. Once again starting from current node we will traverse its suffix link to find required X string. In this case it founds to be root2 i.e null string as adding ‘b’ at front and end of null string yields a palindrome “bb” of length 2. Therefore, we will create a new node “bb” and will direct the insertion edge from the null string to the newly created string. Now, the largest suffix palindrome for this current node will be node “b”. So, we will link the suffix edge from this newly created node to node “b”. Current node now becomes node “bb”.



Stage 4: We will insert  $s[3]$  i.e 'a'. Insertion process begins with current node and in this case the current node itself is the largest X string such that  $s[0] + X + s[3]$  is palindrome. Therefore, we will create a new node "abba" and link the insertion edge from the current node "bb" to this newly created node with edge weight 'a'. Now, the suffix the link from this newly created node will be linked to node "a" as that is the largest palindromic suffix.



The C++ implementation for the above implementation is given below :

```
// C++ program to demonstrate working of
// palindromic tree
#include "bits/stdc++.h"
using namespace std;

#define MAXN 1000

struct Node
```

```

{
    // store start and end indexes of current
    // Node inclusively
    int start, end;

    // stores length of substring
    int length;

    // stores insertion Node for all characters a-z
    int insertEdg[26];

    // stores the Maximum Palindromic Suffix Node for
    // the current Node
    int suffixEdg;
};

// two special dummy Nodes as explained above
Node root1, root2;

// stores Node information for constant time access
Node tree[MAXN];

// Keeps track the current Node while insertion
int currNode;
string s;
int ptr;

void insert(int idx)
{
    //STEP 1//

    /* search for Node X such that s[idx] X S[idx]
       is maximum palindrome ending at position idx
       iterate down the suffix link of currNode to
       find X */
    int tmp = currNode;
    while (true)
    {
        int curLength = tree[tmp].length;
        if (idx - curLength >= 1 and s[idx] == s[idx-curLength-1])
            break;
        tmp = tree[tmp].suffixEdg;
    }

    /* Now we have found X ....
     * X = string at Node tmp
     * Check : if s[idx] X s[idx] already exists or not*/
    if(tree[tmp].insertEdg[s[idx]-'a'] != 0)
}

```

```

{
    // s[idx] X s[idx] already exists in the tree
    currNode = tree[tmp].insertEdg[s[idx]-'a'];
    return;
}

// creating new Node
ptr++;

// making new Node as child of X with
// weight as s[idx]
tree[tmp].insertEdg[s[idx]-'a'] = ptr;

// calculating length of new Node
tree[ptr].length = tree[tmp].length + 2;

// updating end point for new Node
tree[ptr].end = idx;

// updating the start for new Node
tree[ptr].start = idx - tree[ptr].length + 1;

//STEP 2//

/* Setting the suffix edge for the newly created
Node tree[ptr]. Finding some String Y such that
s[idx] + Y + s[idx] is longest possible
palindromic suffix for newly created Node.*/

tmp = tree[tmp].suffixEdg;

// making new Node as current Node
currNode = ptr;
if (tree[currNode].length == 1)
{
    // if new palindrome's length is 1
    // making its suffix link to be null string
    tree[currNode].suffixEdg = 2;
    return;
}
while (true)
{
    int curLength = tree[tmp].length;
    if (idx-curLength >= 1 and s[idx] == s[idx-curLength-1])
        break;
    tmp = tree[tmp].suffixEdg;
}

```

```
// Now we have found string Y
// linking current Nodes suffix link with s[idx]+Y+s[idx]
tree[currNode].suffixEdg = tree[tmp].insertEdg[s[idx]-'a'];
}

// driver program
int main()
{
    // initializing the tree
    root1.length = -1;
    root1.suffixEdg = 1;
    root2.length = 0;
    root2.suffixEdg = 1;

    tree[1] = root1;
    tree[2] = root2;
    ptr = 2;
    currNode = 1;

    // given string
    s = "abcbab";
    int l = s.length();

    for (int i=0; i<l; i++)
        insert(i);

    // printing all of its distinct palindromic
    // substring
    cout << "All distinct palindromic substring for "
        << s << " : \n";
    for (int i=3; i<=ptr; i++)
    {
        cout << i-2 << " ) ";
        for (int j=tree[i].start; j<=tree[i].end; j++)
            cout << s[j];
        cout << endl;
    }

    return 0;
}
```

Output:

```
All distinct palindromic substring for abcbab :
1)a
2)b
```

- 3)c
- 4)bcb
- 5)abcba
- 6)bab

### Time Complexity

The time complexity for the building process will be **O(k\*n)**, here “n” is the length of the string and ‘k’ is the extra iterations required to find the string X and string Y in the suffix links every time we insert a character. Let’s try to approximate the constant ‘k’. We shall consider a worst case like  $s = \text{"aaaaaaaaabcccccccccdeeeeeeeeef"}$ . In this case for similar streak of continuous characters it will take extra 2 iterations per index to find both string X and Y in the suffix links , but as soon as it reaches some index  $i$  such that  $s[i] \neq s[i-1]$  the left most pointer for the maximum length suffix will reach its rightmost limit. Therefore, for all  $i$  when  $s[i] \neq s[i-1]$  , it will cost in total  $n$  iterations(summing over each iteration) and for rest  $i$  when  $s[i] == s[i-1]$  it takes 2 iteration which sums up over all such  $i$  and takes  $2*n$  iterations. Hence, approximately our complexity in this case will be  $O(3*n) \sim O(n)$ . So, we can roughly say that the constant factor ‘k’ will be very less. Therefore, we can consider the overall complexity to be linear **O(length of string)**. You may refer the reference links for better understanding.

### References :

- <http://codeforces.com/blog/entry/13959>
- <http://adilet.org/blog/25-09-14/>

### Source

<https://www.geeksforgeeks.org/palindromic-tree-introduction-implementation/>

## Chapter 300

# Path length having maximum number of bends

Path length having maximum number of bends - GeeksforGeeks

Given a binary tree, find the path length having maximum number of bends.

**Note :** Here, bend indicates switching from left to right or vice versa while traversing in the tree.

For example, consider below paths (L means moving leftwards, R means moving rightwards):

LLRRRR – 1 Bend

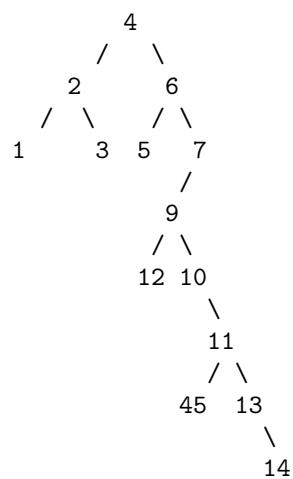
RLLLRR – 2 Bends

LRLRLR – 5 Bends

**Prerequisite :** [Finding Max path length in a Binary Tree](#)

Examples:

Input :



Output : 6

In the above example, the path 4-> 6-> 7-> 9-> 10-> 11-> 45

is having the maximum number of bends, i.e., 3.

The length of this path is 6.

**Approach :**

The idea is to traverse the tree for left and right subtrees of the root. While traversing, keep track of the direction of motion (left or right). Whenever, direction of motion changes from left to right or vice versa increment the number of bends in the current path by 1.

On reaching the leaf node, compare the number of bends in the current path with the maximum number of bends(i.e., maxBends) seen so far in a root-to-leaf path. If the number of bends in the current path is greater than the maxBends, then update the maxBends equal to the number of bends in the current path and update the maximum path length (i.e., len) also to the length of the current path.

**Implementation :**

C++

```
// C++ program to find path length
// having maximum number of bends
#include <bits/stdc++.h>
using namespace std;

// structure node
struct Node {
    int key;
    struct Node* left;
    struct Node* right;
};

// Utility function to create a new node
struct Node* newNode(int key)
{
    struct Node* node = new Node();
    node->left = NULL;
    node->right = NULL;
    node->key = key;

    return node;
}

/* Recursive function to calculate the path
length having maximum number of bends.
The following are parameters for this function.

node --> pointer to the current node
```

```
dir --> determines whether the current node
is left or right child of it's parent node
bends --> number of bends so far in the
current path.
maxBends --> maximum number of bends in a
path from root to leaf
soFar --> length of the current path so
far traversed
len --> length of the path having maximum
number of bends
*/
void findMaxBendsUtil(struct Node* node,
                      char dir, int bends,
                      int* maxBends, int soFar,
                      int* len)
{
    // Base Case
    if (node == NULL)
        return;

    // Leaf node
    if (node->left == NULL && node->right == NULL) {
        if (bends > *maxBends) {
            *maxBends = bends;
            *len = soFar;
        }
    }

    // Left child is NULL
    else if (node->left == NULL) {
        if (dir == 'r') {
            findMaxBendsUtil(node->right, dir,
                            bends, maxBends,
                            soFar + 1, len);
        }
        else {
            findMaxBendsUtil(node->right, 'r',
                            bends + 1, maxBends,
                            soFar + 1, len);
        }
    }

    // Right child is NULL
    else if (node->right == NULL) {
        if (dir == 'l') {
            findMaxBendsUtil(node->left, dir,
                            bends, maxBends,
                            soFar + 1, len);
        }
    }
}
```

```
        }
    else {
        findMaxBendsUtil(node->left, 'l',
                           bends + 1, maxBends,
                           soFar + 1, len);
    }
}
// Having both left and right child
else {
    if (dir == 'l') {
        findMaxBendsUtil(node->left, dir,
                           bends, maxBends,
                           soFar + 1, len);
        findMaxBendsUtil(node->right, 'r',
                           bends + 1, maxBends,
                           soFar + 1, len);
    }
    else {
        findMaxBendsUtil(node->right, dir,
                           bends, maxBends,
                           soFar + 1, len);
        findMaxBendsUtil(node->left, 'l',
                           bends + 1, maxBends,
                           soFar + 1, len);
    }
}
}

// Helper function to call findMaxBendsUtil()
int findMaxBends(struct Node* node)
{
    if (node == NULL)
        return 0;

    int len = 0, bends = 0, maxBends = -1;

    // Call for left subtree of the root
    if (node->left)
        findMaxBendsUtil(node->left, 'l',
                           bends, &maxBends, 1, &len);

    // Call for right subtree of the root
    if (node->right)
        findMaxBendsUtil(node->right, 'r', bends,
                           &maxBends, 1, &len);

    // Include the root node as well in the path length
    len++;
}
```

```
    return len;
}

// Driver code
int main()
{
    /* Constructed binary tree is
       10
      / \
     8   2
    / \  /
   3  5 2
      \
     1
    /
   9
*/
    struct Node* root = newNode(10);
    root->left = newNode(8);
    root->right = newNode(2);
    root->left->left = newNode(3);
    root->left->right = newNode(5);
    root->right->left = newNode(2);
    root->right->left->right = newNode(1);
    root->right->left->right->left = newNode(9);

    cout << findMaxBends(root) - 1;

    return 0;
}
```

Output:

4

## Source

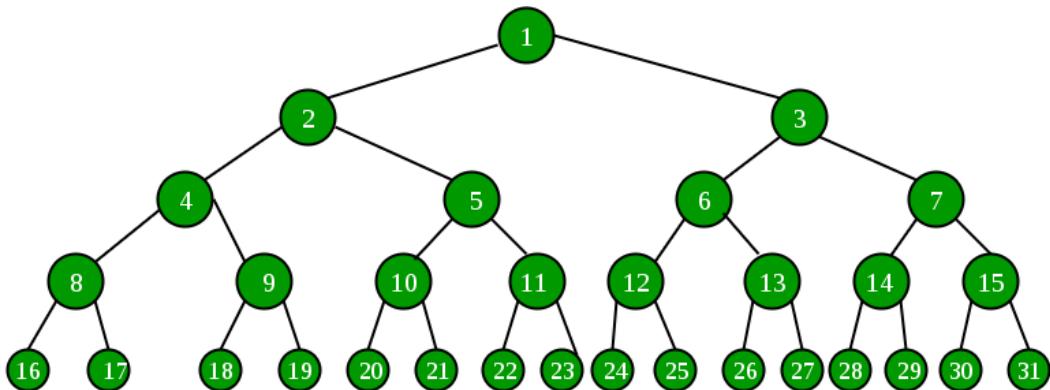
<https://www.geeksforgeeks.org/path-length-maximum-number-bends/>

## Chapter 301

# Perfect Binary Tree Specific Level Order Traversal

Perfect Binary Tree Specific Level Order Traversal - GeeksforGeeks

Given a [Perfect Binary Tree](#) like below:  
(click on image to get a clear view)



Print the level order of nodes in following specific manner:

1 2 3 4 7 5 6 8 15 9 14 10 13 11 12 16 31 17 30 18 29 19 28 20 27 21 26 22 25 23 24

i.e. print nodes in level order but nodes should be from left and right side alternatively.  
Here 1<sup>st</sup> and 2<sup>nd</sup> levels are trivial.

While 3<sup>rd</sup> level: 4(left), 7(right), 5(left), 6(right) are printed.

While 4<sup>th</sup> level: 8(left), 15(right), 9(left), 14(right), .. are printed.

While 5<sup>th</sup> level: 16(left), 31(right), 17(left), 30(right), .. are printed.

**We strongly recommend to minimize your browser and try this yourself first.**

In standard [Level Order Traversal](#), we enqueue root into a queue 1<sup>st</sup>, then we dequeue ONE node from queue, process (print) it, enqueue its children into queue. We keep doing this until queue is empty.

**Approach 1:**

We can do standard level order traversal here too but instead of printing nodes directly, we have to store nodes in current level in a temporary array or list 1<sup>st</sup> and then take nodes from alternate ends (left and right) and print nodes. Keep repeating this for all levels. This approach takes more memory than standard traversal.

**Approach 2:**

The standard level order traversal idea will slightly change here. Instead of processing ONE node at a time, we will process TWO nodes at a time. And while pushing children into queue, the enqueue order will be: 1<sup>st</sup> node's left child, 2<sup>nd</sup> node's right child, 1<sup>st</sup> node's right child and 2<sup>nd</sup> node's left child.

C++

```
/* C++ program for special order traversal */
#include <iostream>
#include <queue>
using namespace std;

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct Node
{
    int data;
    Node *left;
    Node *right;
};

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
Node *newNode(int data)
{
    Node *node = new Node;
    node->data = data;
    node->right = node->left = NULL;
    return node;
}

/* Given a perfect binary tree, print its nodes in specific
   level order */
void printSpecificLevelOrder(Node *root)
{
    if (root == NULL)
        return;
```

```
// Let us print root and next level first
cout << root->data;

// Since it is perfect Binary Tree, right is not checked
if (root->left != NULL)
    cout << " " << root->left->data << " " << root->right->data;

// Do anything more if there are nodes at next level in
// given perfect Binary Tree
if (root->left->left == NULL)
    return;

// Create a queue and enqueue left and right children of root
queue <Node *> q;
q.push(root->left);
q.push(root->right);

// We process two nodes at a time, so we need two variables
// to store two front items of queue
Node *first = NULL, *second = NULL;

// traversal loop
while (!q.empty())
{
    // Pop two items from queue
    first = q.front();
    q.pop();
    second = q.front();
    q.pop();

    // Print children of first and second in reverse order
    cout << " " << first->left->data << " " << second->right->data;
    cout << " " << first->right->data << " " << second->left->data;

    // If first and second have grandchildren, enqueue them
    // in reverse order
    if (first->left->left != NULL)
    {
        q.push(first->left);
        q.push(second->right);
        q.push(first->right);
        q.push(second->left);
    }
}

/* Driver program to test above functions*/
int main()
```

```
{  
    //Perfect Binary Tree of Height 4  
    Node *root = newNode(1);  
  
    root->left      = newNode(2);  
    root->right     = newNode(3);  
  
    root->left->left  = newNode(4);  
    root->left->right = newNode(5);  
    root->right->left  = newNode(6);  
    root->right->right = newNode(7);  
  
    root->left->left->left = newNode(8);  
    root->left->left->right = newNode(9);  
    root->left->right->left = newNode(10);  
    root->left->right->right = newNode(11);  
    root->right->left->left = newNode(12);  
    root->right->left->right = newNode(13);  
    root->right->right->left = newNode(14);  
    root->right->right->right = newNode(15);  
  
    root->left->left->left->left = newNode(16);  
    root->left->left->left->right = newNode(17);  
    root->left->left->right->left = newNode(18);  
    root->left->left->right->right = newNode(19);  
    root->left->right->left->left = newNode(20);  
    root->left->right->left->right = newNode(21);  
    root->left->right->right->left = newNode(22);  
    root->left->right->right->right = newNode(23);  
    root->right->left->left->left = newNode(24);  
    root->right->left->left->right = newNode(25);  
    root->right->left->right->left = newNode(26);  
    root->right->left->right->right = newNode(27);  
    root->right->right->left->left = newNode(28);  
    root->right->right->left->right = newNode(29);  
    root->right->right->right->left = newNode(30);  
    root->right->right->right->right = newNode(31);  
  
    cout << "Specific Level Order traversal of binary tree is \n";  
    printSpecificLevelOrder(root);  
  
    return 0;  
}  
}
```

### Java

```
// Java program for special level order traversal
```

```
import java.util.LinkedList;
import java.util.Queue;

/* Class containing left and right child of current
   node and key value*/
class Node
{
    int data;
    Node left, right;

    public Node(int item)
    {
        data = item;
        left = right = null;
    }
}

class BinaryTree
{
    Node root;

    /* Given a perfect binary tree, print its nodes in specific
       level order */
    void printSpecificLevelOrder(Node node)
    {
        if (node == null)
            return;

        // Let us print root and next level first
        System.out.print(node.data);

        // Since it is perfect Binary Tree, right is not checked
        if (node.left != null)
            System.out.print(" " + node.left.data + " " + node.right.data);

        // Do anything more if there are nodes at next level in
        // given perfect Binary Tree
        if (node.left.left == null)
            return;

        // Create a queue and enqueue left and right children of root
        Queue<Node> q = new LinkedList<Node>();
        q.add(node.left);
        q.add(node.right);

        // We process two nodes at a time, so we need two variables
        // to store two front items of queue
        Node first = null, second = null;
```

```
// traversal loop
while (!q.isEmpty())
{
    // Pop two items from queue
    first = q.peek();
    q.remove();
    second = q.peek();
    q.remove();

    // Print children of first and second in reverse order
    System.out.print(" " + first.left.data + " " +second.right.data);
    System.out.print(" " + first.right.data + " " +second.left.data);

    // If first and second have grandchildren, enqueue them
    // in reverse order
    if (first.left.left != null)
    {
        q.add(first.left);
        q.add(second.right);
        q.add(first.right);
        q.add(second.left);
    }
}

// Driver program to test for above functions
public static void main(String args[])
{
    BinaryTree tree = new BinaryTree();
    tree.root = new Node(1);
    tree.root.left = new Node(2);
    tree.root.right = new Node(3);

    tree.root.left.left = new Node(4);
    tree.root.left.right = new Node(5);
    tree.root.right.left = new Node(6);
    tree.root.right.right = new Node(7);

    tree.root.left.left.left = new Node(8);
    tree.root.left.left.right = new Node(9);
    tree.root.left.right.left = new Node(10);
    tree.root.left.right.right = new Node(11);
    tree.root.right.left.left = new Node(12);
    tree.root.right.left.right = new Node(13);
    tree.root.right.right.left = new Node(14);
    tree.root.right.right.right = new Node(15);
```

```
tree.root.left.left.left.left = new Node(16);
tree.root.left.left.left.right = new Node(17);
tree.root.left.left.right.left = new Node(18);
tree.root.left.left.right.right = new Node(19);
tree.root.left.right.left.left = new Node(20);
tree.root.left.right.left.right = new Node(21);
tree.root.left.right.right.left = new Node(22);
tree.root.left.right.right.right = new Node(23);
tree.root.right.left.left.left = new Node(24);
tree.root.right.left.left.right = new Node(25);
tree.root.right.left.right.left = new Node(26);
tree.root.right.left.right.right = new Node(27);
tree.root.right.right.left.left = new Node(28);
tree.root.right.right.left.right = new Node(29);
tree.root.right.right.right.left = new Node(30);
tree.root.right.right.right.right = new Node(31);

System.out.println("Specific Level Order traversal of binary"
                     +"tree is ");
tree.printSpecificLevelOrder(tree.root);
}

}

// This code has been contributed by Mayank Jaiswal
```

### Python

```
# Python program for special order traversal

# A binary tree node
class Node:
    # A constructor for making a new node
    def __init__(self, key):
        self.data = key
        self.left = None
        self.right = None

    # Given a perfect binary tree print its node in
    # specific order
    def printSpecificLevelOrder(root):
        if root is None:
            return

        # Let us print root and next level first
        print root.data,

        # Since it is perfect Binary tree,
        # one of the node is needed to be checked
```

```
if root.left is not None :
    print root.left.data,
    print root.right.data,

# Do anything more if there are nodes at next level
# in given perfect Binary Tree
if root.left.left is None:
    return

# Create a queue and enqueue left and right
# children of root
q = []
q.append(root.left)
q.append(root.right)

# We process two nodes at a time, so we need
# two variables to store two front items of queue
first = None
second = None

# Traversal loop
while(len(q) > 0):

    # Pop two items from queue
    first = q.pop(0)
    second = q.pop(0)

    # Print children of first and second in reverse order
    print first.left.data,
    print second.right.data,
    print first.right.data,
    print second.left.data,

    # If first and second have grandchildren,
    # enqueue them in reverse order
    if first.left.left is not None:
        q.append(first.left)
        q.append(second.right)
        q.append(first.right)
        q.append(second.left)

# Driver program to test above function

# Perfect Binary Tree of Height 4
root = Node(1)

root.left= Node(2)
root.right    = Node(3)
```

```
root.left.left = Node(4)
root.left.right = Node(5)
root.right.left = Node(6)
root.right.right = Node(7)

root.left.left.left = Node(8)
root.left.left.right = Node(9)
root.left.right.left = Node(10)
root.left.right.right = Node(11)
root.right.left.left = Node(12)
root.right.left.right = Node(13)
root.right.right.left = Node(14)
root.right.right.right = Node(15)

root.left.left.left.left = Node(16)
root.left.left.left.right = Node(17)
root.left.left.right.left = Node(18)
root.left.left.right.right = Node(19)
root.left.right.left.left = Node(20)
root.left.right.left.right = Node(21)
root.left.right.right.left = Node(22)
root.left.right.right.right = Node(23)
root.right.left.left.left = Node(24)
root.right.left.left.right = Node(25)
root.right.left.right.left = Node(26)
root.right.left.right.right = Node(27)
root.right.right.left.left = Node(28)
root.right.right.left.right = Node(29)
root.right.right.right.left = Node(30)
root.right.right.right.right = Node(31)

print "Specific Level Order traversal of binary tree is"
printSpecificLevelOrder(root);

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

Output:

```
Specific Level Order traversal of binary tree is
1 2 3 4 7 5 6 8 15 9 14 10 13 11 12 16 31 17 30 18 29 19 28 20 27 21 26 22 25 23 24
```

**Followup Questions:**

1. The above code prints specific level order from TOP to BOTTOM. How will you do specific level order traversal from BOTTOM to TOP ([Amazon Interview | Set 120 – Round 1 Last Problem](#))

2. What if tree is not perfect, but complete.
3. What if tree is neither perfect, nor complete. It can be any general binary tree.

This article is contributed by **Anurag Singh**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## Source

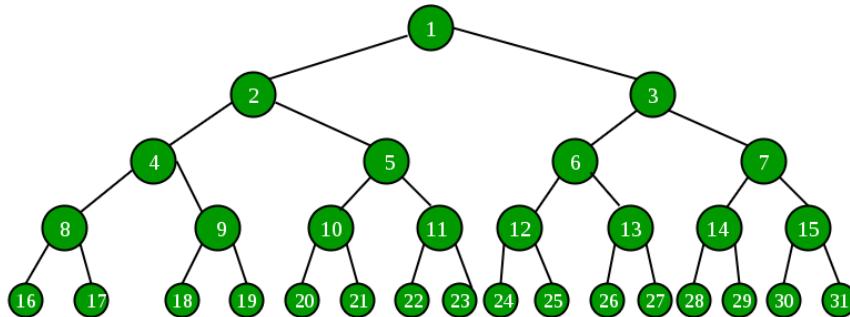
<https://www.geeksforgeeks.org/perfect-binary-tree-specific-level-order-traversal/>

## Chapter 302

# Perfect Binary Tree Specific Level Order Traversal | Set 2

Perfect Binary Tree Specific Level Order Traversal | Set 2 - GeeksforGeeks

Perfect Binary Tree using Specific Level Order Traversal in [Set 1](#). The earlier traversal was from Top to Bottom. In this post, Bottom to Top traversal (asked in [Amazon Interview | Set 120 – Round 1](#)) is discussed.



16 31 17 30 18 29 19 28 20 27 21 26 22 25 23 24 8 15 9 14 10 13 11 12 4 7 5 6 2 3 1

The task is to print nodes in level order but nodes should be from left and right side alternatively and from bottom – up manner

5th level: 16(left), 31(right), 17(left), 30(right), ... are printed.

4th level: 8(left), 15(right), 9(left), 14(right), ... are printed.

3rd level: 4(left), 7(right), 5(left), 6(right) are printed.

1st and 2nd levels are trivial.

We strongly recommend you to minimize your browser and try this yourself first.

The standard level order traversal idea slightly changes here.

1. Instead of processing ONE node at a time, we will process TWO nodes at a time.
2. For dequeued nodes, we push node's left and right child into stack in following manner
  - 2nd node's left child, 1st node's right child, 2nd node's right child and 1st node's left child.
3. And while pushing children into queue, the enqueue order will be: 1st node's right child, 2nd node's left child, 1st node's left child and 2nd node's right child. Also, when we process two queue nodes.
4. Finally pop all Nodes from stack and prints them.

C++

```
/* C++ program for special order traversal */
#include <bits/stdc++.h>
using namespace std;

/* A binary tree node has data, pointer to left child
and a pointer to right child */
struct Node
{
    int data;
    Node *left, *right;
};

/* Helper function that allocates a new node with the
given data and NULL left and right pointers. */
Node* newNode(int data)
{
    Node* node = new Node;
    node->data = data;
    node->right = node->left = NULL;
    return node;
}

void printSpecificLevelOrderUtil(Node* root, stack<Node*> &s)
{
    if (root == NULL)
        return;

    // Create a queue and enqueue left and right
    // children of root
    queue<Node*> q;

    q.push(root->left);
```

```
q.push(root->right);

// We process two nodes at a time, so we
// need two variables to store two front
// items of queue
Node *first = NULL, *second = NULL;

// traversal loop
while (!q.empty())
{
    // Pop two items from queue
    first = q.front();
    q.pop();
    second = q.front();
    q.pop();

    // Push first and second node's children
    // in reverse order
    s.push(second->left);
    s.push(first->right);
    s.push(second->right);
    s.push(first->left);

    // If first and second have grandchildren,
    // enqueue them in specific order
    if (first->left->left != NULL)
    {
        q.push(first->right);
        q.push(second->left);
        q.push(first->left);
        q.push(second->right);
    }
}

/* Given a perfect binary tree, print its nodes in
specific level order */
void printSpecificLevelOrder(Node* root)
{
    //create a stack and push root
    stack<Node*> s;

    //Push level 1 and level 2 nodes in stack
    s.push(root);

    // Since it is perfect Binary Tree, right is
    // not checked
    if (root->left != NULL)
```

```

{
    s.push(root->right);
    s.push(root->left);
}

// Do anything more if there are nodes at next
// level in given perfect Binary Tree
if (root->left->left != NULL)
    printSpecificLevelOrderUtil(root, s);

// Finally pop all Nodes from stack and prints
// them.
while (!s.empty())
{
    cout << s.top()->data << " ";
    s.pop();
}
}

/* Driver program to test above functions*/
int main()
{
    // Perfect Binary Tree of Height 4
    Node* root = newNode(1);

    root->left = newNode(2);
    root->right = newNode(3);

    /* root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->left = newNode(6);
    root->right->right = newNode(7);

    root->left->left->left = newNode(8);
    root->left->left->right = newNode(9);
    root->left->right->left = newNode(10);
    root->left->right->right = newNode(11);
    root->right->left->left = newNode(12);
    root->right->left->right = newNode(13);
    root->right->right->left = newNode(14);
    root->right->right->right = newNode(15);

    root->left->left->left->left = newNode(16);
    root->left->left->left->right = newNode(17);
    root->left->left->right->left = newNode(18);
    root->left->left->right->right = newNode(19);
    root->left->right->left->left = newNode(20);
    root->left->right->left->right = newNode(21);
```

```
root->left->right->right->left = newNode(22);
root->left->right->right->right = newNode(23);
root->right->left->left->left = newNode(24);
root->right->left->left->right = newNode(25);
root->right->left->right->left = newNode(26);
root->right->left->right->right = newNode(27);
root->right->right->left->left = newNode(28);
root->right->right->left->right = newNode(29);
root->right->right->right->left = newNode(30);
root->right->right->right->right = newNode(31);
*/
cout << "Specific Level Order traversal of binary "
     "tree is \n";
printSpecificLevelOrder(root);

return 0;
}
```

**Java**

```
// Java program for special order traversal

import java.util.*;

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
class Node
{
    int data;
    Node left, right;

    public Node(int data)
    {
        this.data = data;
        left = right = null;
    }
}

class BinaryTree
{
    Node root;

    void printSpecificLevelOrderUtil(Node root, Stack<Node> s)
    {
        if (root == null)
            return;

        // Create a queue and enqueue left and right
```

```
// children of root
Queue<Node> q = new LinkedList<Node>();

q.add(root.left);
q.add(root.right);

// We process two nodes at a time, so we
// need two variables to store two front
// items of queue
Node first = null, second = null;

// traversal loop
while (!q.isEmpty())
{
    // Pop two items from queue
    first = q.peek();
    q.poll();
    second = q.peek();
    q.poll();

    // Push first and second node's children
    // in reverse order
    s.push(second.left);
    s.push(first.right);
    s.push(second.right);
    s.push(first.left);

    // If first and second have grandchildren,
    // enqueue them in specific order
    if (first.left.left != null)
    {
        q.add(first.right);
        q.add(second.left);
        q.add(first.left);
        q.add(second.right);
    }
}

/* Given a perfect binary tree, print its nodes in
   specific level order */
void printSpecificLevelOrder(Node root)
{
    //create a stack and push root
    Stack<Node> s = new Stack<Node>();

    //Push level 1 and level 2 nodes in stack
    s.push(root);
```

```
// Since it is perfect Binary Tree, right is
// not checked
if (root.left != null)
{
    s.push(root.right);
    s.push(root.left);
}

// Do anything more if there are nodes at next
// level in given perfect Binary Tree
if (root.left.left != null)
    printSpecificLevelOrderUtil(root, s);

// Finally pop all Nodes from stack and prints
// them.
while (!s.empty())
{
    System.out.print(s.peek().data + " ");
    s.pop();
}
}

// Driver program to test the above functions
public static void main(String[] args)
{
    BinaryTree tree = new BinaryTree();
    tree.root = new Node(1);
    tree.root.left = new Node(2);
    tree.root.right = new Node(3);

    /* tree.root.left.left = new Node(4);
    tree.root.left.right = new Node(5);
    tree.root.right.left = new Node(6);
    tree.root.right.right = new Node(7);

    tree.root.left.left.left = new Node(8);
    tree.root.left.left.right = new Node(9);
    tree.root.left.right.left = new Node(10);
    tree.root.left.right.right = new Node(11);
    tree.root.right.left.left = new Node(12);
    tree.root.right.left.right = new Node(13);
    tree.root.right.right.left = new Node(14);
    tree.root.right.right.right = new Node(15);

    tree.root.left.left.left.left = new Node(16);
    tree.root.left.left.left.right = new Node(17);
    tree.root.left.left.right.left = new Node(18);
```

```
tree.root.left.left.right.right = new Node(19);
tree.root.left.right.left.left = new Node(20);
tree.root.left.right.left.right = new Node(21);
tree.root.left.right.right.left = new Node(22);
tree.root.left.right.right.right = new Node(23);
tree.root.right.left.left.left = new Node(24);
tree.root.right.left.left.right = new Node(25);
tree.root.right.left.right.left = new Node(26);
tree.root.right.left.right.right = new Node(27);
tree.root.right.right.left.left = new Node(28);
tree.root.right.right.left.right = new Node(29);
tree.root.right.right.right.left = new Node(30);
tree.root.right.right.right.right = new Node(31);
*/
System.out.println("Specific Level Order Traversal "
    + "of Binary Tree is ");
tree.printSpecificLevelOrder(tree.root);
}
}

// This code has been contributed by Mayank Jaiswal(mayank_24)
```

### Python3

```
# Python program for special order traversal

# A binary tree node
class Node:

    # Create queue and enqueue left
    # and right child of root
    s = []
    q = []

    # Variable to traverse the reversed array
    elements = 0

    # A constructor for making a new node
    def __init__(self, key):
        self.data = key
        self.left = None
        self.right = None

    # Given a perfect binary tree print
    # its node in specific order
    def printSpecificLevelOrder(self, root):
        self.s.append(root)
```

```
# Pop the element from the list
prnt = self.s.pop(0)
self.q.append(prnt.data)
if prnt.right:
    self.s.append(prnt.right)
if prnt.left:
    self.s.append(prnt.left)

# Traversal loop
while(len(self.s) > 0):

    # Pop two items from queue
    first = self.s.pop(0)
    self.q.append(first.data)
    second = self.s.pop(0)
    self.q.append(second.data)

    # Since it is perfect Binary tree,
    # one of the node is needed to be checked
    if first.left and second.right and first.right and second.left:

        # If first and second have grandchildren,
        # enqueue them in reverse order
        self.s.append(first.left)
        self.s.append(second.right)
        self.s.append(first.right)
        self.s.append(second.left)

    # Give a perfect binary tree print
    # its node in reverse order
    for elements in reversed(self.q):
        print(elements, end=" ")

# Driver Code
root = Node(1)

root.left = Node(2)
root.right = Node(3)

...
root.left.left = Node(4)
root.left.right = Node(5)
root.right.left = Node(6)
root.right.right = Node(7)

root.left.left.left = Node(8)
root.left.left.right = Node(9)
```

```
root.left.right.left = Node(10)
root.left.right.right = Node(11)
root.right.left.left = Node(12)
root.right.left.right = Node(13)
root.right.right.left = Node(14)
root.right.right.right = Node(15)

root.left.left.left.left = Node(16)
root.left.left.left.right = Node(17)
root.left.left.right.left = Node(18)
root.left.left.right.right = Node(19)
root.left.right.left.left = Node(20)
root.left.right.left.right = Node(21)
root.left.right.right.left = Node(22)
root.left.right.right.right = Node(23)
root.right.left.left.left = Node(24)
root.right.left.left.right = Node(25)
root.right.left.right.left = Node(26)
root.right.left.right.right = Node(27)
root.right.right.left.left = Node(28)
root.right.right.left.right = Node(29)
root.right.right.right.left = Node(30)
root.right.right.right.right = Node(31)
'''

print("Specific Level Order traversal of "
      "binary tree is")
root.printSpecificLevelOrder(root)

# This code is contributed by 'Vaibhav Kumar 12'
```

Output :

```
Specific Level Order traversal of binary tree is
2 3 1
```

This article is contributed by **Aditya Goel**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## Source

<https://www.geeksforgeeks.org/perfect-binary-tree-specific-level-order-traversal-set-2/>

## Chapter 303

# Persistent Segment Tree | Set 1 (Introduction)

Persistent Segment Tree | Set 1 (Introduction) - GeeksforGeeks

Prerequisite : Segment Tree  
Persistency in Data Structure

Segment Tree is itself a great data structure that comes into play in many cases. In this post we will introduce the concept of Persistency in this data structure. Persistency, simply means to retain the changes. But obviously, retaining the changes cause extra memory consumption and hence affect the Time Complexity.

Our aim is to apply persistency in segment tree and also to ensure that it does not take more than **O(log n)** time and space for each change.

Let's think in terms of versions i.e. for each change in our segment tree we create a new version of it.

We will consider our initial version to be Version-0. Now, as we do any update in the segment tree we will create a new version for it and in similar fashion track the record for all versions.

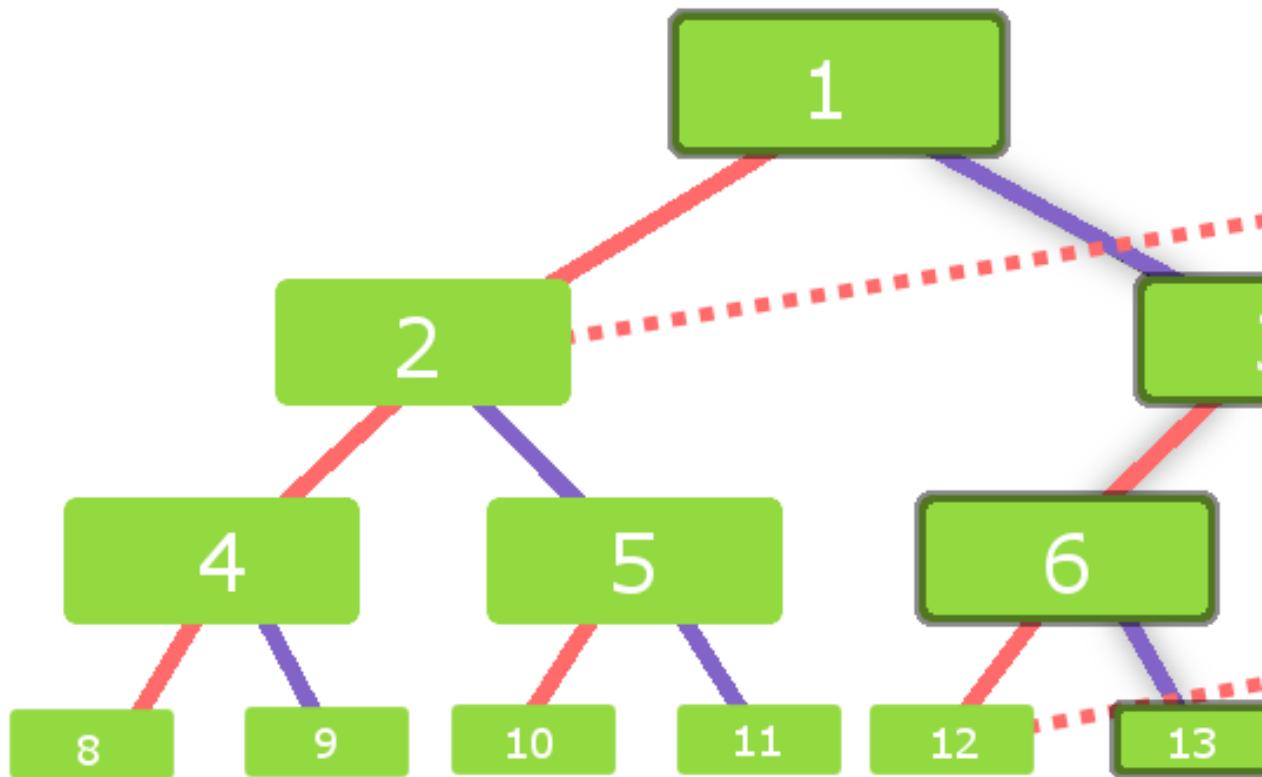
But creating the whole tree for every version will take  $O(n \log n)$  extra space and  $O(n \log n)$  time. So, this idea runs out of time and memory for large number of versions.

Let's exploit the fact that for each new update(say point update for simplicity) in segment tree, At max  $\log n$  nodes will be modified. So, our new version will only contain these  $\log n$  new nodes and rest nodes will be the same as previous version. Therefore, it is quite clear that for each new version we only need to create these  $\log n$  new nodes whereas the rest of nodes can be shared from the previous version.

Consider the below figure for better visualization(click on the image for better view) :-

# Persistency in

Version 0



Consider the segment tree with green nodes . Lets call this segment tree as **version-0**. The left child for each node is connected with solid red edge where as the right child for each node is connected with solid purple edge. Clearly, this segment tree consists of 15 nodes.

Now consider we need to make change in the leaf node 13 of version-0.

So, the affected nodes will be – **node 13 , node 6 , node 3 , node 1**.

Therefore, for the new version (**Version-1**) we need to create only these **4 new nodes**.

Now, lets construct version-1 for this change in segment tree. We need a new node 1 as it is affected by change done in node 13. So , we will first create a new **node 1** (yellow color) . The left child for node 1 will be the same for left child for node 1 in version-0. So, we connect the left child of node 1 with node 2 of version-0(red dashed line in figure). Let's now examine the right child for node 1 in version-1. We need to create a new node as it is affected . So we create a new node called node 3 and make it the right child for node 1 (solid purple edge connection).

In the similar fashion we will now examine for **node 3** . The left child is affected , So we create a new node called **node 6** and connect it with solid red edge with node 3 , where as the right child for node 3 will be the same as right child of node 3 in version-0. So, we will make the right child of node 3 in version-0 as the right child of node 3 in version-1(see the purple dash edge.)

Same procedure is done for node 6 and we see that the left child of node 6 will be the left child of node 6 in version-0(red dashed connection) and right child is newly created node called **node 13** (solid purple dashed edge).

Each **yellow color node** is a newly created node and dashed edges are the inter-connection between the different versions of the segment tree.

Now, the Question arises : **How to keep track of all the versions?**

– We only need to keep track the first root node for all the versions and this will serve the purpose to track all the newly created nodes in the different versions. For this purpose we can maintain an array of pointers to the first node of segment trees for all versions.

Let's consider a very basic problem to see how to implement persistence in segment tree

**Problem :** Given an array A[] and different point update operations.Considering each point operation to create a new version of the array. We need to answer the queries of type  
Q v l r : output the sum of elements in range l to r just after the v-th update.

We will create all the versions of the segment tree and keep track of their root node.Then for each range sum query we will pass the required version's root node in our query function and output the required sum.

**Below is the C++ implementation for the above problem:-**

```
// C++ program to implement persistent segment
// tree.
#include <bits/stdc++.h>
using namespace std;

#define MAXN 100

/* data type for individual
```

```
* node in the segment tree */
struct node
{
    // stores sum of the elements in node
    int val;

    // pointer to left and right children
    node* left, *right;

    // required constructors.....
    node() {}
    node(node* l, node* r, int v)
    {
        left = l;
        right = r;
        val = v;
    }
};

// input array
int arr[MAXN];

// root pointers for all versions
node* version[MAXN];

// Constructs Version-0
// Time Complexity : O(nlogn)
void build(node* n,int low,int high)
{
    if (low==high)
    {
        n->val = arr[low];
        return;
    }
    int mid = (low+high) / 2;
    n->left = new node(NULL, NULL, 0);
    n->right = new node(NULL, NULL, 0);
    build(n->left, low, mid);
    build(n->right, mid+1, high);
    n->val = n->left->val + n->right->val;
}

/**
 * Upgrades to new Version
 * @param prev : points to node of previous version
 * @param cur : points to node of current version
 * Time Complexity : O(logn)
 * Space Complexity : O(logn)  */

```

```

void upgrade(node* prev, node* cur, int low, int high,
            int idx, int value)
{
    if (idx > high or idx < low or low > high)
        return;

    if (low == high)
    {
        // modification in new version
        cur->val = value;
        return;
    }
    int mid = (low+high) / 2;
    if (idx <= mid)
    {
        // link to right child of previous version
        cur->right = prev->right;

        // create new node in current version
        cur->left = new node(NULL, NULL, 0);

        upgrade(prev->left, cur->left, low, mid, idx, value);
    }
    else
    {
        // link to left child of previous version
        cur->left = prev->left;

        // create new node for current version
        cur->right = new node(NULL, NULL, 0);

        upgrade(prev->right, cur->right, mid+1, high, idx, value);
    }

    // calculating data for current version
    // by combining previous version and current
    // modification
    cur->val = cur->left->val + cur->right->val;
}

int query(node* n, int low, int high, int l, int r)
{
    if (l > high or r < low or low > high)
        return 0;
    if (l <= low and high <= r)
        return n->val;
    int mid = (low+high) / 2;
    int p1 = query(n->left, low, mid, l, r);

```

```
int p2 = query(n->right,mid+1,high,l,r);
return p1+p2;
}

int main(int argc, char const *argv[])
{
    int A[] = {1,2,3,4,5};
    int n = sizeof(A)/sizeof(int);

    for (int i=0; i<n; i++)
        arr[i] = A[i];

    // creating Version-0
    node* root = new node(NULL, NULL, 0);
    build(root, 0, n-1);

    // storing root node for version-0
    version[0] = root;

    // upgrading to version-1
    version[1] = new node(NULL, NULL, 0);
    upgrade(version[0], version[1], 0, n-1, 4, 1);

    // upgrading to version-2
    version[2] = new node(NULL, NULL, 0);
    upgrade(version[1],version[2], 0, n-1, 2, 10);

    cout << "In version 1 , query(0,4) : ";
    cout << query(version[1], 0, n-1, 0, 4) << endl;

    cout << "In version 2 , query(3,4) : ";
    cout << query(version[2], 0, n-1, 3, 4) << endl;

    cout << "In version 0 , query(0,3) : ";
    cout << query(version[0], 0, n-1, 0, 3) << endl;
    return 0;
}
```

Output:

```
In version 1 , query(0,4) : 11
In version 2 , query(3,4) : 5
In version 0 , query(0,3) : 10
```

Note : The above problem can also be solved by processing the queries offline by sorting it with respect to the version and answering the queries just after the corresponding update.

**Time Complexity :** The time complexity will be the same as the query and point update operation in the segment tree as we can consider the extra node creation step to be done in  $O(1)$ . Hence, the overall Time Complexity per query for new version creation and range sum query will be  $O(\log n)$ .

## Source

<https://www.geeksforgeeks.org/persistent-segment-tree-set-1-introduction/>

## Chapter 304

# Populate Inorder Successor for all nodes

Populate Inorder Successor for all nodes - GeeksforGeeks

Given a Binary Tree where each node has following structure, write a function to populate next pointer for all nodes. The next pointer for every node should be set to point to inorder successor.

```
struct node
{
    int data;
    struct node* left;
    struct node* right;
    struct node* next;
}
```

Initially, all next pointers have NULL values. Your function should fill these next pointers so that they point to inorder successor.

### Solution (Use Reverse Inorder Traversal)

Traverse the given tree in reverse inorder traversal and keep track of previously visited node. When a node is being visited, assign previously visited node as next.

C

```
#include <stdio.h>
#include <stdlib.h>

struct node
{
    int data;
    struct node *left;
```

```
struct node *right;
struct node *next;
};

/* Set next of p and all descendants of p by traversing them in reverse Inorder */
void populateNext(struct node* p)
{
    // The first visited node will be the rightmost node
    // next of the rightmost node will be NULL
    static struct node *next = NULL;

    if (p)
    {
        // First set the next pointer in right subtree
        populateNext(p->right);

        // Set the next as previously visited node in reverse Inorder
        p->next = next;

        // Change the prev for subsequent node
        next = p;

        // Finally, set the next pointer in left subtree
        populateNext(p->left);
    }
}

/* UTILITY FUNCTIONS */
/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newnode(int data)
{
    struct node* node = (struct node*)
                           malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;
    node->next = NULL;

    return(node);
}

/* Driver program to test above functions*/
int main()
{

    /* Constructed binary tree is
       10
```

```
          /   \
          8     12
         /
         3
*/
struct node *root = newnode(10);
root->left      = newnode(8);
root->right     = newnode(12);
root->left->left = newnode(3);

// Populates nextRight pointer in all nodes
populateNext(root);

// Let us see the populated values
struct node *ptr = root->left->left;
while(ptr)
{
    // -1 is printed if there is no successor
    printf("Next of %d is %d \n", ptr->data, ptr->next? ptr->next->data: -1);
    ptr = ptr->next;
}

return 0;
}
```

**Java**

```
// Java program to populate inorder traversal of all nodes

// A binary tree node
class Node
{
    int data;
    Node left, right, next;

    Node(int item)
    {
        data = item;
        left = right = next = null;
    }
}

class BinaryTree
{
    Node root;
    static Node next = null;

    /* Set next of p and all descendants of p by traversing them in
```

```
    reverse Inorder */
void populateNext(Node node)
{
    // The first visited node will be the rightmost node
    // next of the rightmost node will be NULL
    if (node != null)
    {
        // First set the next pointer in right subtree
        populateNext(node.right);

        // Set the next as previously visited node in reverse Inorder
        node.next = next;

        // Change the prev for subsequent node
        next = node;

        // Finally, set the next pointer in left subtree
        populateNext(node.left);
    }
}

/* Driver program to test above functions*/
public static void main(String args[])
{
    /* Constructed binary tree is
       10
      /   \
     8     12
     /
    3   */
    BinaryTree tree = new BinaryTree();
    tree.root = new Node(10);
    tree.root.left = new Node(8);
    tree.root.right = new Node(12);
    tree.root.left.left = new Node(3);

    // Populates nextRight pointer in all nodes
    tree.populateNext(tree.root);

    // Let us see the populated values
    Node ptr = tree.root.left.left;
    while (ptr != null)
    {
        // -1 is printed if there is no successor
        int print = ptr.next != null ? ptr.next.data : -1;
        System.out.println("Next of " + ptr.data + " is: " + print);
        ptr = ptr.next;
    }
}
```

```
    }
}

// This code has been contributed by Mayank Jaiswal
```

We can avoid the use of static variable by passing reference to next as parameter.

### C

```
// An implementation that doesn't use static variable

// A wrapper over populateNextRecur
void populateNext(struct node *root)
{
    // The first visited node will be the rightmost node
    // next of the rightmost node will be NULL
    struct node *next = NULL;

    populateNextRecur(root, &next);
}

/* Set next of all descendants of p by traversing them in reverse Inorder */
void populateNextRecur(struct node* p, struct node **next_ref)
{
    if (p)
    {
        // First set the next pointer in right subtree
        populateNextRecur(p->right, next_ref);

        // Set the next as previously visited node in reverse Inorder
        p->next = *next_ref;

        // Change the prev for subsequent node
        *next_ref = p;

        // Finally, set the next pointer in right subtree
        populateNextRecur(p->left, next_ref);
    }
}
```

### Java

```
// A wrapper over populateNextRecur
void populateNext(Node node) {

    // The first visited node will be the rightmost node
    // next of the rightmost node will be NULL
    populateNextRecur(node, next);
```

```
}

/* Set next of all descendants of p by traversing them in reverse Inorder */
void populateNextRecur(Node p, Node next_ref) {
    if (p != null) {

        // First set the next pointer in right subtree
        populateNextRecur(p.right, next_ref);

        // Set the next as previously visited node in reverse Inorder
        p.next = next_ref;

        // Change the prev for subsequent node
        next_ref = p;

        // Finally, set the next pointer in right subtree
        populateNextRecur(p.left, next_ref);
    }
}
```

Time Complexity: O(n)

## Source

<https://www.geeksforgeeks.org/populate-inorder-successor-for-all-nodes/>

## Chapter 305

# Possible edges of a tree for given diameter, height and vertices

Possible edges of a tree for given diameter, height and vertices - GeeksforGeeks

Find a tree with the given values and print the edges of the tree. Print “-1”, if the tree is not possible.

Given three integers n, d and h.

```
n -> Number of vertices. [1, n]
d -> Diameter of the tree (largest
      distance between two vertices).
h -> Height of the tree (longest distance
      between vertex 1 and another vertex)
```

Examples :

Input : n = 5, d = 3, h = 2

Output : 1 2  
 2 3  
 1 4  
 1 5

Explanation :

We can see that the height of the tree is 2 (1 -> 2 --> 5) and diameter is 3 ( 3 -> 2 -> 1 -> 5).  
So our conditions are satisfied.

Input : n = 8, d = 4, h = 2  
Output : 1 2

```
2 3
1 4
4 5
1 6
1 7
1 8
```

Explanation :

1. Observe that when  $d = 1$ , we cannot construct a tree (if tree has more than 2 vertices).  
Also when  $d > 2^*h$ , we cannot construct a tree.
2. As we know that height is the longest path from vertex 1 to another vertex. So build that path from vertex 1 by adding edges up to h. Now, if  $d > h$ , we should add another path to satisfy diameter from vertex 1, with a length of  $d - h$ .
3. Our conditions for height and diameter are satisfied. But still some vertices may be left. Add the remaining vertices at any vertex other than the end points. This step will not alter our diameter and height. Choose vertex 1 to add the remaining vertices (you can choose any).
4. But when  $d == h$ , choose vertex 2 to add the remaining vertices.

## CPP

```
// CPP program to construct tree for given count
// width and height.
#include <iostream>
using namespace std;

// Function to construct the tree
void constructTree(int n, int d, int h)
{
    if (d == 1) {

        // Special case when d == 2, only one edge
        if (n == 2 && h == 1) {
            cout << "1 2" << endl;
            return;
        }
        cout << "-1" << endl; // Tree is not possible
        return;
    }

    if (d > 2 * h) {
        cout << "-1" << endl;
        return;
    }

    // Satisfy the height condition by add
```

```
// edges up to h
for (int i = 1; i <= h; i++)
    cout << i << " " << i + 1 << endl;

if (d > h) {

    // Add d - h edges from 1 to
    // satisfy diameter condition
    cout << "1"
        << " " << h + 2 << endl;
    for (int i = h + 2; i <= d; i++) {
        cout << i << " " << i + 1 << endl;
    }
}

// Remaining edges at vertex 1 or 2(d == h)
for (int i = d + 1; i < n; i++)
{
    int k = 1;
    if (d == h)
        k = 2;
    cout << k << " " << i + 1 << endl;
}
}

// Driver Code
int main()
{
    int n = 5, d = 3, h = 2;
    constructTree(n, d, h);
    return 0;
}
```

### Python3

```
# Python3 code to construct tree for given count
# width and height.

# Function to construct the tree
def constructTree(n, d, h):
    if d == 1:

        # Special case when d == 2, only one edge
        if n == 2 and h == 1:
            print("1 2")
            return 0

    print("-1")      # Tree is not possible
```

```
return 0

if d > 2 * h:
    print("-1")
    return 0

# Satisfy the height condition by add
# edges up to h
for i in range(1, h+1):
    print(i, " ", i + 1)

if d > h:

    # Add d - h edges from 1 to
    # satisfy diameter condition
    print(1, " ", h + 2)
    for i in range(h+2, d+1):
        print(i, " ", i + 1)

# Remaining edges at vertex 1 or 2(d == h)
for i in range(d+1, n):
    k = 1
    if d == h:
        k = 2
    print(k, " ", i + 1)

# Driver Code
n = 5
d = 3
h = 2
constructTree(n, d, h)

# This code is contributed by "Sharad_Bhardwaj".
```

Output :

```
1 2
2 3
1 4
1 5
```

## Source

<https://www.geeksforgeeks.org/print-possible-edges-tree-given-diameter-height-vertices/>

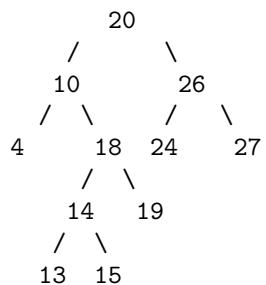
## Chapter 306

# Postorder predecessor of a Node in Binary Search Tree

Postorder predecessor of a Node in Binary Search Tree - GeeksforGeeks

Given a binary tree and a node in the binary tree, find Postorder predecessor of the given node.

Examples: Consider the following binary tree



Input : 4

Output : 10

Postorder traversal of given tree is 4, 13, 15,  
14, 19, 18, 10, 24, 27, 26, 20.

Input : 24

Output : 10

A **simple solution** is to first store Postorder traversal of the given tree in an array then linearly search given node and print node next to it.

Time Complexity :  $O(n)$

Auxiliary Space :  $O(n)$

An **efficient solution** is based on below observations.

1. If right child of given node exists, then the right child is postorder predecessor.
2. If right child does not exist and given node is left child of its parent, then its sibling is its postorder predecessor.
3. If none of above conditions are satisfied (left child does not exist and given node is not right child of its parent), then we move up using parent pointers until one of the following happens.
  - We reach root. In this case, postorder predecessor does not exist
  - Current node (one of the ancestors of given node) is right child of its parent, in this case postorder predecessor is sibling of current node.

```
// CPP program to find postorder predecessor of
// a node in Binary Tree.
#include <iostream>
using namespace std;

struct Node {
    struct Node *left, *right, *parent;
    int key;
};

Node* newNode(int key)
{
    Node* temp = new Node;
    temp->left = temp->right = temp->parent = NULL;
    temp->key = key;
    return temp;
}

Node* postorderPredecessor(Node* root, Node* n)
{
    // If right child exists, then it is postorder
    // predecessor.
    if (n->right)
        return n->right;

    // If right child does not exist, then
    // travel up (using parent pointers)
    // until we reach a node which is right
    // child of its parent.
    Node *curr = n, *parent = curr->parent;
    while (parent != NULL && parent->left == curr) {
        curr = curr->parent;
        parent = parent->parent;
    }

    // If we reached root, then the given
    // node has no postorder predecessor
    if (parent == NULL)
```

```
    return NULL;

    return parent->left;
}

int main()
{
    Node* root = newNode(20);
    root->parent = NULL;
    root->left = newNode(10);
    root->left->parent = root;
    root->left->left = newNode(4);
    root->left->left->parent = root->left;
    root->left->right = newNode(18);
    root->left->right->parent = root->left;
    root->right = newNode(26);
    root->right->parent = root;
    root->right->left = newNode(24);
    root->right->left->parent = root->right;
    root->right->right = newNode(27);
    root->right->right->parent = root->right;
    root->left->right->left = newNode(14);
    root->left->right->left->parent = root->left->right;
    root->left->right->left->left = newNode(13);
    root->left->right->left->left->parent = root->left->right->left;
    root->left->right->left->right = newNode(15);
    root->left->right->left->right->parent = root->left->right->left;
    root->left->right->right = newNode(19);
    root->left->right->right->parent = root->left->right;

    Node* res = postorderPredecessor(root, root->left->right->right);

    if (res) {
        printf("Postorder predecessor of %d is %d\n",
               root->left->right->right->key, res->key);
    }
    else {
        printf("Postorder predecessor of %d is NULL\n",
               root->left->right->right->key);
    }

    return 0;
}
```

**Output:**

```
Postorder predecessor of 19 is 14
```

Time Complexity :  $O(h)$  where  $h$  is height of given Binary Tree

Auxiliary Space :  $O(1)$

### **Source**

<https://www.geeksforgeeks.org/postorder-predecessor-node-binary-search-tree/>

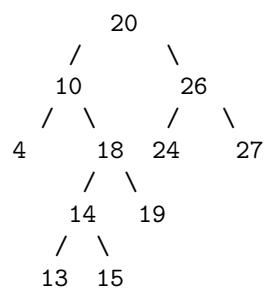
## Chapter 307

# Postorder successor of a Node in Binary Tree

Postorder successor of a Node in Binary Tree - GeeksforGeeks

Given a binary tree and a node in the binary tree, find Postorder successor of the given node.

Examples: Consider the following binary tree



Postorder traversal of given tree is 4, 13, 15, 14, 19, 18, 10, 24, 27, 26, 20.

Input : 24  
Output : 27

Input : 4  
Output : 13

A **simple solution** is to first store Postorder traversal of the given tree in an array then linearly search given node and print node next to it.

Time Complexity : O(n)  
Auxiliary Space : O(n)

An **efficient solution** is based on below observations.

1. If given node is root then postorder successor is NULL, since root is the last node print in a postorder traversal
2. If given node is right child of parent or right child of parent is NULL, then parent is postorder successor.
3. If given node is left child of parent and right child of parent is not NULL, then postorder successor is the leftmost node of parent's right subtree

```
// CPP program to find postorder successor of
// given node.
#include <iostream>
using namespace std;

struct Node {
    struct Node *left, *right, *parent;
    int value;
};

// Utility function to create a new node with
// given value.
struct Node* newNode(int value)
{
    Node* temp = new Node;
    temp->left = temp->right = temp->parent = NULL;
    temp->value = value;
    return temp;
}

Node* postorderSuccessor(Node* root, Node* n)
{
    // Root has no successor in postorder
    // traversal
    if (n == root)
        return NULL;

    // If given node is right child of its
    // parent or parent's right is empty, then
    // parent is postorder successor.
    Node* parent = n->parent;
    if (parent->right == NULL || parent->right == n)
        return parent;

    // In all other cases, find the leftmost
    // child in left subtree of parent.
    Node* curr = parent->right;
    while (curr->left != NULL)
        curr = curr->left;
```

```
        return curr;
    }

// Driver code
int main()
{
    struct Node* root = newNode(20);
    root->parent = NULL;
    root->left = newNode(10);
    root->left->parent = root;
    root->left->left = newNode(4);
    root->left->left->parent = root->left;
    root->left->right = newNode(18);
    root->left->right->parent = root->left;
    root->right = newNode(26);
    root->right->parent = root;
    root->right->left = newNode(24);
    root->right->left->parent = root->right;
    root->right->right = newNode(27);
    root->right->right->parent = root->right;
    root->left->right->left = newNode(14);
    root->left->right->left->parent = root->left->right;
    root->left->right->left->left = newNode(13);
    root->left->right->left->left->parent = root->left->right->left;
    root->left->right->left->right = newNode(15);
    root->left->right->left->right->parent = root->left->right->left;
    root->left->right->right = newNode(19);
    root->left->right->right->parent = root->left->right;

    struct Node* res = postorderSuccessor(root, root->left->right->right);
    if (res)
        printf("Postorder successor of %d is %d\n",
               root->left->right->right->value, res->value);
    else
        printf("Postorder successor of %d is NULL\n",
               root->left->right->right->value);

    return 0;
}
```

**Output:**

Postorder successor of 19 is 18

Time Complexity : O(h) where h is height of given Binary Tree  
Auxiliary Space : O(1) since no use of arrays, stacks, queues.

**Source**

<https://www.geeksforgeeks.org/postorder-successor-node-binary-tree/>

## Chapter 308

# Postorder traversal of Binary Tree without recursion and without stack

Postorder traversal of Binary Tree without recursion and without stack - GeeksforGeeks

Prerequisite – [Inorder/preorder/postorder traversal of tree](#)  
Given a binary tree, perform postorder traversal.

We have discussed below methods for postorder traversal.

- 1) [Recursive Postorder Traversal](#).
- 2) [Postorder traversal using Stack](#).
- 2) [Postorder traversal using two Stacks](#).

In this method a [DFS](#) based solution is discussed. We keep track of visited nodes in a hash table.

```
// CPP program for postorder traversal
#include <bits/stdc++.h>
using namespace std;

/* A binary tree node has data, pointer to left child
and a pointer to right child */
struct Node {
    int data;
    struct Node *left, *right;
};

/* Helper function that allocates a new node with the
given data and NULL left and right pointers. */
void postorder(struct Node* head)
{
```

```
struct Node* temp = head;
unordered_set<Node*> visited;
while (temp && visited.find(temp) == visited.end()) {

    // Visited left subtree
    if (temp->left &&
        visited.find(temp->left) == visited.end())
        temp = temp->left;

    // Visited right subtree
    else if (temp->right &&
              visited.find(temp->right) == visited.end())
        temp = temp->right;

    // Print node
    else {
        printf("%d ", temp->data);
        visited.insert(temp);
        temp = head;
    }
}

struct Node* newNode(int data)
{
    struct Node* node = new Node;
    node->data = data;
    node->left = NULL;
    node->right = NULL;
    return (node);
}

/* Driver program to test above functions*/
int main()
{
    struct Node* root = newNode(8);
    root->left = newNode(3);
    root->right = newNode(10);
    root->left->left = newNode(1);
    root->left->right = newNode(6);
    root->left->right->left = newNode(4);
    root->left->right->right = newNode(7);
    root->right->right = newNode(14);
    root->right->right->left = newNode(13);
    postorder(root);
    return 0;
}
```

Output:

```
1 4 7 6 3 13 14 10 8
```

**Alternate Solution:**

We can keep visited flag with every node instead of separate hash table.

```
// CPP program for postorder traversal
#include <bits/stdc++.h>
using namespace std;

/* A binary tree node has data, pointer to left child
and a pointer to right child */
struct Node {
    int data;
    struct Node *left, *right;
    bool visited;
};

void postorder(struct Node* head)
{
    struct Node* temp = head;
    while (temp && temp->visited == false) {

        // Visited left subtree
        if (temp->left && temp->left->visited == false)
            temp = temp->left;

        // Visited right subtree
        else if (temp->right && temp->right->visited == false)
            temp = temp->right;

        // Print node
        else {
            printf("%d ", temp->data);
            temp->visited = true;
            temp = head;
        }
    }
}

struct Node* newNode(int data)
{
    struct Node* node = new Node;
    node->data = data;
    node->left = NULL;
    node->right = NULL;
```

```
node->visited = false;
return (node);
}

/* Driver program to test above functions*/
int main()
{
    struct Node* root = newNode(8);
    root->left = newNode(3);
    root->right = newNode(10);
    root->left->left = newNode(1);
    root->left->right = newNode(6);
    root->left->right->left = newNode(4);
    root->left->right->right = newNode(7);
    root->right->right = newNode(14);
    root->right->right->left = newNode(13);
    postorder(root);
    return 0;
}
```

Output:

```
1 4 7 6 3 13 14 10 8
```

Time complexity of above solution is  $O(n^2)$  in worst case we move pointer back to head after visiting every node.

## Source

<https://www.geeksforgeeks.org/postorder-traversal-binary-tree-without-recursion-without-stack/>

## Chapter 309

# Practice questions on Height balanced/AVL Tree

Practice questions on Height balanced/AVL Tree - GeeksforGeeks

AVL tree is binary search tree with additional property that difference between height of left sub-tree and right sub-tree of any node can't be more than 1. Here are some key points about [AVL trees](#):

- If there are n nodes in AVL tree, minimum height of AVL tree is  $\text{floor}(\log_2 n)$ .
- If there are n nodes in AVL tree, maximum height can't exceed  $1.44 * \log_2 n$ .
- If height of AVL tree is h, maximum number of nodes can be  $2^{h+1} - 1$ .
- Minimum number of nodes in a tree with height h can be represented as:  
 $N(h) = N(h-1) + N(h-2) + 1$  for  $n > 2$  where  $N(0) = 1$  and  $N(1) = 2$ .
- The complexity of searching, inserting and deletion in AVL tree is  $O(\log n)$ .

We have discussed types of questions based on AVL trees.

### Type 1: Relationship between number of nodes and height of AVL tree –

Given number of nodes, the question can be asked to find minimum and maximum height of AVL tree. Also, given the height, maximum or minimum number of nodes can be asked.

**Que – 1.** What is the maximum height of any AVL-tree with 7 nodes? Assume that the height of a tree with a single node is 0.

- (A) 2
- (B) 3
- (C) 4
- (D) 5

**Solution:** For finding maximum height, the nodes should be minimum at each level. Assuming

height as 2, minimum number of nodes required:

$$N(h) = N(h-1) + N(h-2) + 1$$

$$N(2) = N(1) + N(0) + 1 = 2 + 1 + 1 = 4.$$

It means, height 2 is achieved using minimum 4 nodes.

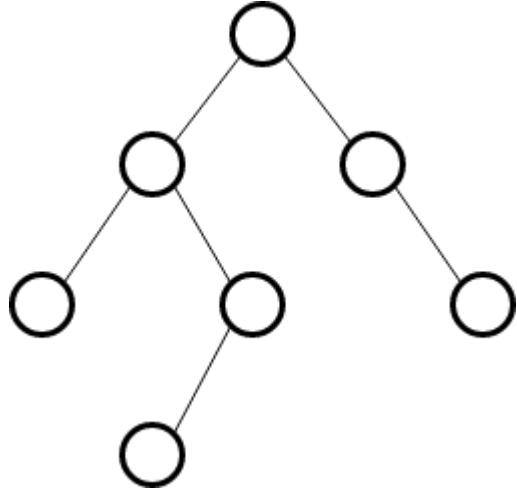
Assuming height as 3, minimum number of nodes required:

$$N(h) = N(h-1) + N(h-2) + 1$$

$$N(3) = N(2) + N(1) + 1 = 4 + 2 + 1 = 7.$$

It means, height 3 is achieved using minimum 7 nodes.

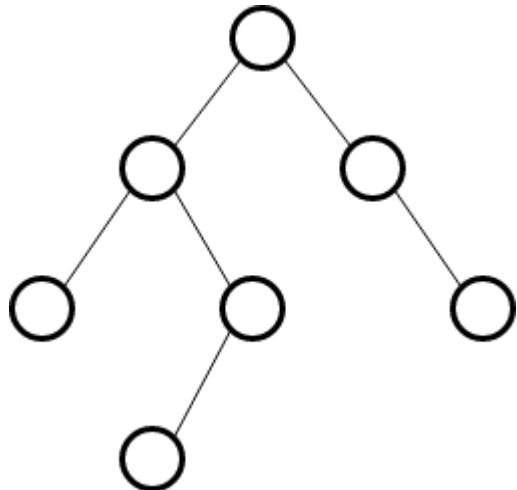
Therefore, using 7 nodes, we can achieve maximum height as 3. Following is the AVL tree with 7 nodes and height 3.



**Que – 2.** What is the worst case possible height of AVL tree?

- (A)  $2 * \log n$
- (B)  $1.44 * \log n$
- (C) Depends upon implementation
- (D)  $(n)$

**Solution:** The worst case possible height of AVL tree with  $n$  nodes is  $1.44 * \log n$ . This can be verified using AVL tree having 7 nodes and maximum height.



Checking for option (A),  $2 \log 7 = 5.6$ , however height of tree is 3.

Checking for option (B),  $1.44 \log 7 = 4$ , which is near to 3.

Checking for option (D),  $n = 7$ , however height of tree is 3.

Out of these, option (B) is the best possible answer.

**Type 2: Based on complexity of insertion, deletion and searching in AVL tree –**

**Que – 3.** Which of the following is TRUE?

- (A) The cost of searching an AVL tree is  $(\log n)$  but that of a binary search tree is  $O(n)$
- (B) The cost of searching an AVL tree is  $(\log n)$  but that of a complete binary tree is  $(n \log n)$
- (C) The cost of searching a binary search tree is  $O(\log n)$  but that of an AVL tree is  $(n)$
- (D) The cost of searching an AVL tree is  $(n \log n)$  but that of a binary search tree is  $O(n)$

**Solution:** AVL tree's time complexity of searching, insertion and deletion =  $O(\log n)$ . But a binary search tree, may be skewed tree, so in worst case BST searching, insertion and deletion complexity =  $O(n)$ .

**Que – 4.** The worst case running time to search for an element in a balanced in a binary search tree with  $n * 2^n$  elements is

- (A)  $\Theta(n \log n)$
- (B)  $\Theta(n^{2^n})$
- (C)  $\Theta(n)$
- (D)  $\Theta(\log n)$

**Solution:** Time taken to search an element is  $\Theta(\log n)$  where  $n$  is number of elements in AVL tree.

As number of elements given is  $n * 2^n$ , the searching complexity will be  $\Theta(\log(n * 2^n))$  which can be written as:

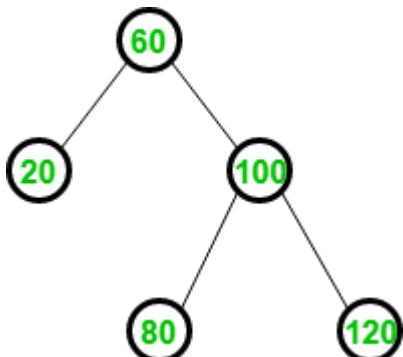
$$\begin{aligned} &= \Theta(\log(n * 2^n)) \\ &= \Theta(\log(n)) + \Theta(\log(2^n)) \\ &= \Theta(\log(n)) + \Theta(n \log(2)) \\ &= \Theta(\log(n)) + \Theta(n) \end{aligned}$$

As  $\log n$  is asymptotically smaller than  $n$ ,  $\Theta(\log(n)) + \Theta(n)$  can be written as  $\Theta(n)$  which matches option C.

**Type 3: Insertion and Deletion in AVL tree –**

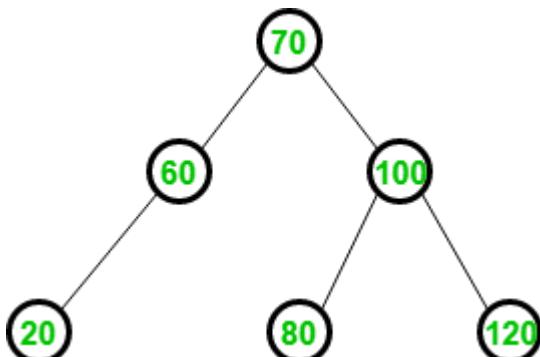
The question can be asked on the resultant tree when keys are inserted or deleted from AVL tree. Appropriate rotations need to be made if balance factor is disturbed.

**Que – 5.** Consider the following AVL tree.

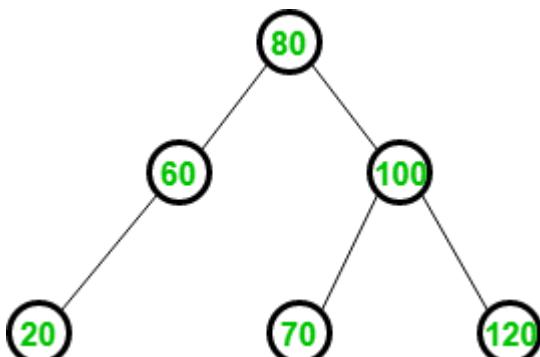


Which of the following is updated AVL tree after insertion of 70?

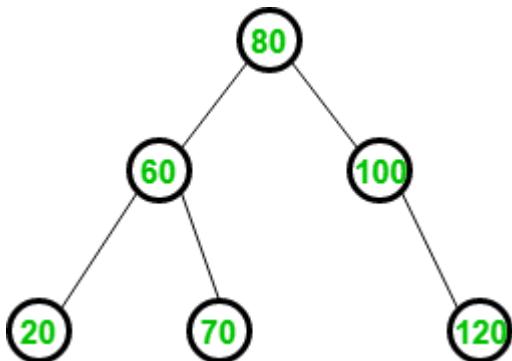
(A)



(B)

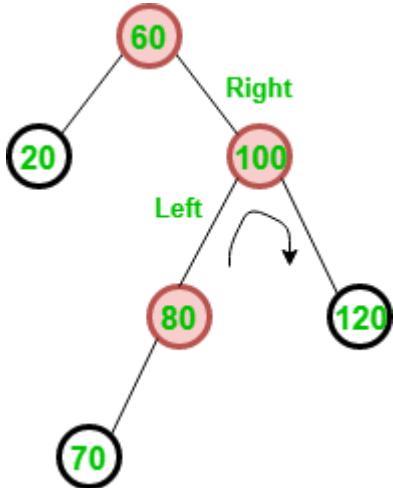


(C)

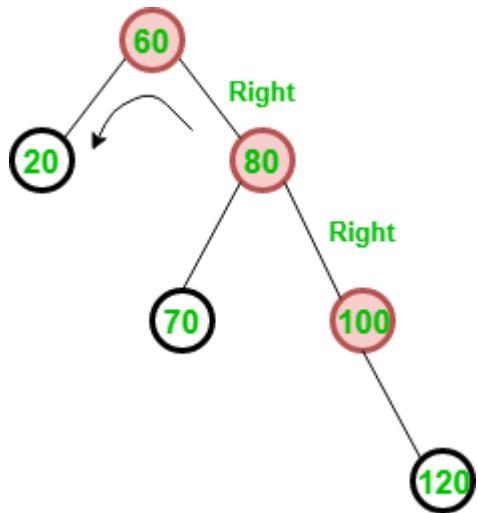


(D) None

**Solution:** The element is first inserted in the same way as BST. Therefore after insertion of 70, BST can be shown as:



However, balance factor is disturbed requiring RL rotation. To remove RL rotation, it is first converted into RR rotation as:



After removal of RR rotation, AVL tree generated is same as option (C).

### Source

<https://www.geeksforgeeks.org/practice-questions-height-balancedavl-tree/>

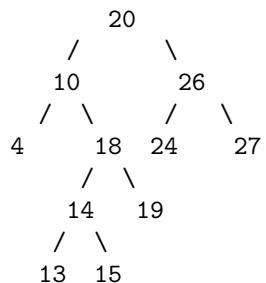
## Chapter 310

# Preorder Successor of a Node in Binary Tree

Preorder Successor of a Node in Binary Tree - GeeksforGeeks

Given a binary tree and a node in the binary tree, find preorder successor of the given node.

Examples: Consider the following binary tree



Input : 4

Output : 10

Preorder traversal of given tree is 20, 10, 4,  
18, 14, 13, 15, 19, 26, 24, 27.

Input : 19

Output : 15

A **simple solution** is to first store Preorder traversal of the given tree in an array then linearly search given node and print node next to it.

Time Complexity : O(n)

Auxiliary Space : O(n)

An **efficient solution** is based on below observations.

1. If left child of given node exists, then the left child is preorder successor.
2. If left child does not exist and given node is left child of its parent, then its sibling is its preorder successor.
3. If none of above conditions are satisfied (left child does not exist and given node is not left child of its parent), then we move up using parent pointers until one of the following happens.
  - We reach root. In this case, preorder successor does not exist.
  - Current node (one of the ancestors of given node) is left child of its parent, in this case preorder successor is sibling of current node.

```
// CPP program to find preorder successor of
// a node in Binary Tree.
#include <iostream>
using namespace std;

struct Node {
    struct Node *left, *right, *parent;
    int key;
};

Node* newNode(int key)
{
    Node* temp = new Node;
    temp->left = temp->right = temp->parent = NULL;
    temp->key = key;
    return temp;
}

Node* preorderSuccessor(Node* root, Node* n)
{
    // If left child exists, then it is preorder
    // successor.
    if (n->left)
        return n->left;

    // If left child does not exist, then
    // travel up (using parent pointers)
    // until we reach a node which is left
    // child of its parent.
    Node *curr = n, *parent = curr->parent;
    while (parent != NULL && parent->right == curr) {
        curr = curr->parent;
        parent = parent->parent;
    }

    // If we reached root, then the given
    // node has no preorder successor
    if (parent == NULL)
```

```
    return NULL;

    return parent->right;
}

int main()
{
    Node* root = newNode(20);
    root->parent = NULL;
    root->left = newNode(10);
    root->left->parent = root;
    root->left->left = newNode(4);
    root->left->left->parent = root->left;
    root->left->right = newNode(18);
    root->left->right->parent = root->left;
    root->right = newNode(26);
    root->right->parent = root;
    root->right->left = newNode(24);
    root->right->left->parent = root->right;
    root->right->right = newNode(27);
    root->right->right->parent = root->right;
    root->left->right->left = newNode(14);
    root->left->right->left->parent = root->left->right;
    root->left->right->left->left = newNode(13);
    root->left->right->left->left->parent = root->left->right->left;
    root->left->right->left->right = newNode(15);
    root->left->right->left->right->parent = root->left->right->left;
    root->left->right->right = newNode(19);
    root->left->right->right->parent = root->left->right;

    Node* res = preorderSuccessor(root, root->left->right->right);

    if (res) {
        printf("Preorder successor of %d is %d\n",
               root->left->right->right->key, res->key);
    }
    else {
        printf("Preorder successor of %d is NULL\n",
               root->left->right->right->key);
    }

    return 0;
}
```

**Output:**

```
Preorder successor of 19 is 26
```

Time Complexity :  $O(h)$  where  $h$  is height of given Binary Tree

Auxiliary Space :  $O(1)$

### **Source**

<https://www.geeksforgeeks.org/preorder-successor-node-binary-tree/>

## Chapter 311

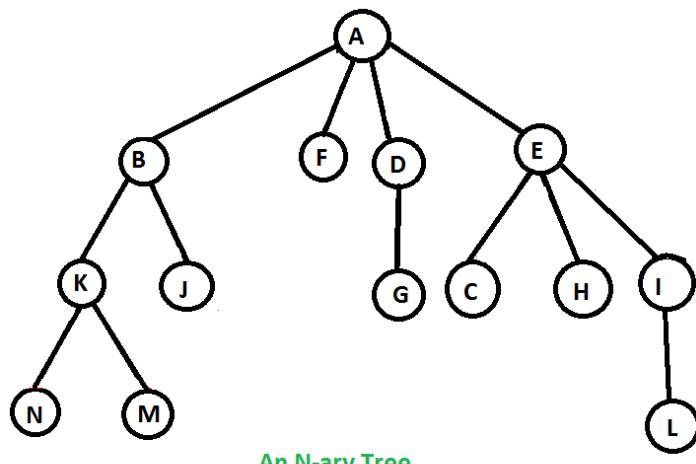
# Preorder Traversal of N-ary Tree Without Recursion

Preorder Traversal of N-ary Tree Without Recursion - GeeksforGeeks

Given an n-ary tree, print preorder traversal of it.

Example :

Preorder traversal of below tree is **A B K N M J F D G E C H I L**



The idea is to use stack like [iterative preorder traversal of binary tree](#).

- 1) Create an empty stack to store nodes.
- 2) Push the root node to the stack.
- 3) Run a loop while the stack is not empty
  - ....a) Pop the top node from stack.

- ....b) Print the popped node.
- ....c) Store all the children of popped node onto the stack. We must store children from right to left so that leftmost node is popped first.
- 4) If stack is empty then we are done.

```
// C++ program to traverse an N-ary tree
// without recursion
#include <bits/stdc++.h>
using namespace std;

// Structure of a node of an n-ary tree
struct Node {
    char key;
    vector<Node*> child;
};

// Utility function to create a new tree node
Node* newNode(int key)
{
    Node* temp = new Node;
    temp->key = key;
    return temp;
}

// Function to traverse tree without recursion
void traverse_tree(struct Node* root)
{
    // Stack to store the nodes
    stack<Node*> nodes;

    // push the current node onto the stack
    nodes.push(root);

    // loop while the stack is not empty
    while (!nodes.empty()) {

        // store the current node and pop it from the stack
        Node* curr = nodes.top();
        nodes.pop();

        // current node has been traversed
        cout << curr->key << " ";

        // store all the children of current node from
        // right to left.
        vector<Node*>::iterator it = curr->child.end();

        while (it != curr->child.begin()) {
```

```
        it--;
        nodes.push(*it);
    }
}
// Driver program
int main()
{
    /* Let us create below tree
     *      A
     *      /   \   \
     *      B   F   D   E
     *      / \       |   /|\
     *      K   J       G   C   H   I
     *      / \           |   |
     *      N   M           O   L
    */
    Node* root = newNode('A');
    (root->child).push_back(newNode('B'));
    (root->child).push_back(newNode('F'));
    (root->child).push_back(newNode('D'));
    (root->child).push_back(newNode('E'));
    (root->child[0]->child).push_back(newNode('K'));
    (root->child[0]->child).push_back(newNode('J'));
    (root->child[2]->child).push_back(newNode('G'));
    (root->child[3]->child).push_back(newNode('C'));
    (root->child[3]->child).push_back(newNode('H'));
    (root->child[3]->child).push_back(newNode('I'));
    (root->child[0]->child[0]->child).push_back(newNode('N'));
    (root->child[0]->child[0]->child).push_back(newNode('M'));
    (root->child[3]->child[0]->child).push_back(newNode('O'));
    (root->child[3]->child[2]->child).push_back(newNode('L'));

    traverse_tree(root);

    return 0;
}
```

**Output:**

A B K N M J F D G E C O H I L

**Source**

<https://www.geeksforgeeks.org/preorder-traversal-of-n-ary-tree-without-recursion/>

## Chapter 312

# Preorder from Inorder and Postorder traversals

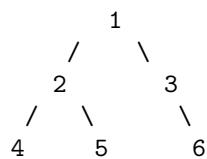
Preorder from Inorder and Postorder traversals - GeeksforGeeks

Given Inorder and Postorder traversals of a binary tree, print Preorder traversal.

**Example:**

```
Input: Postorder traversal post[] = {4, 5, 2, 6, 3, 1}
       Inorder traversal in[] = {4, 2, 5, 1, 3, 6}
Output: Preorder traversal 1, 2, 4, 5, 3, 6
```

Traversals in the above example represents following tree



A **naive method** is to first [construct the tree from given postorder and inorder](#), then use simple recursive method to print preorder traversal of the constructed tree.

InOrder(root) visits nodes in the following order:

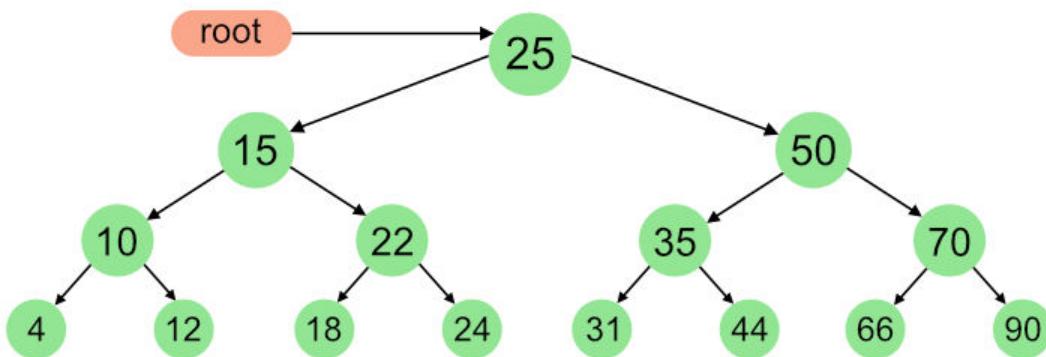
4, 10, 12, 15, 18, 22, 24, 25, 31, 35, 44, 50, 66, 70, 90

A Pre-order traversal visits nodes in the following order:

25, 15, 10, 4, 12, 22, 18, 24, 50, 35, 31, 44, 70, 66, 90

A Post-order traversal visits nodes in the following order:

4, 12, 10, 18, 24, 22, 15, 31, 44, 35, 66, 90, 70, 50, 25



We can print preorder traversal without constructing the tree. The idea is, root is always the first item in preorder traversal and it must be the last item in postorder traversal. We first push right subtree to a stack, then left subtree and finally we push root. Finally we print contents of stack. To find boundaries of left and right subtrees in post[] and in[], we search root in in[], all elements before root in in[] are elements of left subtree and all elements after root are elements of right subtree. In post[], all elements after index of root in in[] are elements of right subtree. And elements before index (including the element at index and excluding the first element) are elements of left subtree.

```

// Java program to print Postorder traversal from given
// Inorder and Preorder traversals.
import java.util.Stack;

public class PrintPre {

    static int postIndex;

    // Fills preorder traversal of tree with given
    // inorder and postorder traversals in a stack
    void fillPre(int[] in, int[] post, int inStrt,
  
```

```
        int inEnd, Stack<Integer> s)
{
    if (inStrt > inEnd)
        return;

    // Find index of next item in postorder traversal in
    // inorder.
    int val = post[postIndex];
    int inIndex = search(in, val);
    postIndex--;

    // traverse right tree
    fillPre(in, post, inIndex + 1, inEnd, s);

    // traverse left tree
    fillPre(in, post, inStrt, inIndex - 1, s);

    s.push(val);
}

// This function basically initializes postIndex
// as last element index, then fills stack with
// reverse preorder traversal using printPre
void printPreMain(int[] in, int[] post)
{
    int len = in.length;
    postIndex = len - 1;
    Stack<Integer> s = new Stack<Integer>();
    fillPre(in, post, 0, len - 1, s);
    while (s.empty() == false)
        System.out.print(s.pop() + " ");
}

// A utility function to search data in in[]
int search(int[] in, int data)
{
    int i = 0;
    for (i = 0; i < in.length; i++)
        if (in[i] == data)
            return i;
    return i;
}

// Driver code
public static void main(String args[])
{
    int in[] = { 4, 10, 12, 15, 18, 22, 24, 25,
                31, 35, 44, 50, 66, 70, 90 };
}
```

```
int post[] = { 4, 12, 10, 18, 24, 22, 15, 31,
               44, 35, 66, 90, 70, 50, 25 };
PrintPre tree = new PrintPre();
tree.printPreMain(in, post);
}
}
```

**Output:**

```
25 15 10 4 12 22 18 24 50 35 31 44 70 66 90
```

**Time Complexity:** The above function visits every node in array. For every visit, it calls search which takes  $O(n)$  time. Therefore, overall time complexity of the function is  $O(n^2)$

**$O(n)$  Solution**

We can further optimize above solution to first hash all items of inorder traversal so that we do not have to linearly search items. With hash table available to us, we can search an item in  $O(1)$  time.

```
// Java program to print Postorder traversal from given
// Inorder and Preorder traversals.
import java.util.Stack;
import java.util.HashMap;

public class PrintPre {

    static int postIndex;

    // Fills preorder traversal of tree with given
    // inorder and postorder traversals in a stack
    void fillPre(int[] in, int[] post, int inStrt, int inEnd,
                Stack<Integer> s, HashMap<Integer, Integer> hm)
    {
        if (inStrt > inEnd)
            return;

        // Find index of next item in postorder traversal in
        // inorder.
        int val = post[postIndex];
        int inIndex = hm.get(val);
        postIndex--;

        // traverse right tree
        fillPre(in, post, inIndex + 1, inEnd, s, hm);

        // traverse left tree
    }
}
```

```
        fillPre(in, post, inStrt, inIndex - 1, s, hm);

        s.push(val);
    }

// This function basically initializes postIndex
// as last element index, then fills stack with
// reverse preorder traversal using printPre
void printPreMain(int[] in, int[] post)
{
    int len = in.length;
    postIndex = len - 1;
    Stack<Integer> s = new Stack<Integer>();

    // Insert values in a hash map and their indexes.
    HashMap<Integer, Integer> hm =
        new HashMap<Integer, Integer>();
    for (int i = 0; i < in.length; i++)
        hm.put(in[i], i);

    // Fill preorder traversal in a stack
    fillPre(in, post, 0, len - 1, s, hm);

    // Print contents of stack
    while (s.empty() == false)
        System.out.print(s.pop() + " ");
}

// Driver code
public static void main(String args[])
{
    int in[] = { 4, 10, 12, 15, 18, 22, 24, 25,
                31, 35, 44, 50, 66, 70, 90 };
    int post[] = { 4, 12, 10, 18, 24, 22, 15, 31,
                  44, 35, 66, 90, 70, 50, 25 };
    PrintPre tree = new PrintPre();
    tree.printPreMain(in, post);
}
```

**Output:**

25 15 10 4 12 22 18 24 50 35 31 44 70 66 90

**Time Complexity:** O(n)

**Source**

<https://www.geeksforgeeks.org/preorder-from-inorder-and-postorder-traversals/>

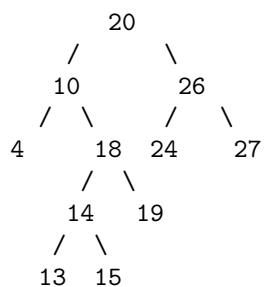
## Chapter 313

# Preorder predecessor of a Node in Binary Tree

Preorder predecessor of a Node in Binary Tree - GeeksforGeeks

Given a binary tree and a node in the binary tree, find Preorder predecessor of the given node.

Examples: Consider the following binary tree



Input : 4

Output : 10

Preorder traversal of given tree is 20, 10, 4,  
18, 14, 13, 15, 19, 26, 24, 27.

Input : 19

Output : 15

A **simple solution** is to first store Preorder traversal of the given tree in an array then linearly search given node and print node next to it.

Time Complexity :  $O(n)$

Auxiliary Space :  $O(n)$

An **efficient solution** is based on below observations.

1. If the given node is root, then return NULL as preorder predecessor.
2. If node is the left child of its parent or left child of parent is NULL, then return parent as its preorder predecessor.
3. If node is the right child of its parent and left child of parent exists, then predecessor would be the rightmost node (max value) of the left subtree of parent.

```
// CPP program to find preorder predecessor of
// given node.
#include <iostream>
using namespace std;

struct Node {
    struct Node *left, *right, *parent;
    int value;
};

// Utility function to create a new node with
// given value.
struct Node* newNode(int value)
{
    Node* temp = new Node;
    temp->left = temp->right = temp->parent = NULL;
    temp->value = value;
    return temp;
}

Node* preorderPredecessor(Node* root, Node* n)
{
    // Root has no predecessor in preorder
    // traversal
    if (n == root)
        return NULL;

    // If given node is left child of its
    // parent or parent's left is empty, then
    // parent is Preorder Predecessor.
    Node* parent = n->parent;
    if (parent->left == NULL || parent->left == n)
        return parent;

    // In all other cases, find the rightmost
    // child in left subtree of parent.
    Node* curr = parent->left;
    while (curr->right != NULL)
        curr = curr->right;

    return curr;
}
```

```
// Driver code
int main()
{
    struct Node* root = newNode(20);
    root->parent = NULL;
    root->left = newNode(10);
    root->left->parent = root;
    root->left->left = newNode(4);
    root->left->left->parent = root->left;
    root->left->right = newNode(18);
    root->left->right->parent = root->left;
    root->right = newNode(26);
    root->right->parent = root;
    root->right->left = newNode(24);
    root->right->left->parent = root->right;
    root->right->right = newNode(27);
    root->right->right->parent = root->right;
    root->left->right->left = newNode(14);
    root->left->right->left->parent = root->left->right;
    root->left->right->left->left = newNode(13);
    root->left->right->left->left->parent = root->left->right->left;
    root->left->right->left->right = newNode(15);
    root->left->right->left->right->parent = root->left->right->left;
    root->left->right->right = newNode(19);
    root->left->right->right->parent = root->left->right;

    struct Node* res = preorderPredecessor(root, root->left->right->right);
    if (res)
        printf("Preorder predecessor of %d is %d\n",
               root->left->right->right->value, res->value);
    else
        printf("Preorder predecessor of %d is NULL\n",
               root->left->right->right->value);

    return 0;
}
```

**Output:**

```
Preorder predecessor of 19 is 15
```

Time Complexity :  $O(h)$  where  $h$  is height of given Binary Tree  
Auxiliary Space :  $O(1)$  since no use of arrays, stacks, queues.

**Source**

<https://www.geeksforgeeks.org/preorder-predecessor-node-binary-tree/>

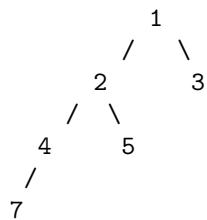
## Chapter 314

# Print Ancestors of a given node in Binary Tree

Print Ancestors of a given node in Binary Tree - GeeksforGeeks

Given a Binary Tree and a key, write a function that prints all the ancestors of the key in the given binary tree.

For example, if the given tree is following Binary Tree and key is 7, then your function should print 4, 2 and 1.



Thanks to Mike, Sambasiva and wgpshashank for their contribution.

C++

```
#include<iostream>
#include<stdio.h>
#include<stdlib.h>

using namespace std;

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
```

```
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

/* If target is present in tree, then prints the ancestors
   and returns true, otherwise returns false. */
bool printAncestors(struct node *root, int target)
{
    /* base cases */
    if (root == NULL)
        return false;

    if (root->data == target)
        return true;

    /* If target is present in either left or right subtree of this node,
       then print this node */
    if (printAncestors(root->left, target) ||
        printAncestors(root->right, target) )
    {
        cout << root->data << " ";
        return true;
    }

    /* Else return false */
    return false;
}

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newnode(int data)
{
    struct node* node = (struct node*)
                           malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;

    return(node);
}

/* Driver program to test above functions*/
int main()
{
```

```
/* Construct the following binary tree
      1
     /   \
    2     3
   / \   /
  4   5 7
 */
struct node *root = newnode(1);
root->left      = newnode(2);
root->right     = newnode(3);
root->left->left = newnode(4);
root->left->right = newnode(5);
root->left->left->left = newnode(7);

printAncestors(root, 7);

getchar();
return 0;
}
```

### Java

```
// Java program to print ancestors of given node

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
class Node
{
    int data;
    Node left, right, nextRight;

    Node(int item)
    {
        data = item;
        left = right = nextRight = null;
    }
}

class BinaryTree
{
    Node root;

    /* If target is present in tree, then prints the ancestors
       and returns true, otherwise returns false. */
    boolean printAncestors(Node node, int target)
    {
```

```
/* base cases */
if (node == null)
    return false;

if (node.data == target)
    return true;

/* If target is present in either left or right subtree
   of this node, then print this node */
if (printAncestors(node.left, target)
    || printAncestors(node.right, target))
{
    System.out.print(node.data + " ");
    return true;
}

/* Else return false */
return false;
}

/* Driver program to test above functions */
public static void main(String args[])
{
    BinaryTree tree = new BinaryTree();

    /* Construct the following binary tree
        1
       /   \
      2     3
     /   \
    4     5
   /
  7
*/
    tree.root = new Node(1);
    tree.root.left = new Node(2);
    tree.root.right = new Node(3);
    tree.root.left.left = new Node(4);
    tree.root.left.right = new Node(5);
    tree.root.left.left.left = new Node(7);

    tree.printAncestors(tree.root, 7);

}
}

// This code has been contributed by Mayank Jaiswal
```

## Python

```
# Python program to print ancestors of given node in
# binary tree

# A Binary Tree node
class Node:

    # Constructor to create a new node
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

    # If target is present in tree, then prints the ancestors
    # and returns true, otherwise returns false
    def printAncestors(root, target):

        # Base case
        if root == None:
            return False

        if root.data == target:
            return True

        # If target is present in either left or right subtree
        # of this node, then print this node
        if (printAncestors(root.left, target) or
            printAncestors(root.right, target)):
            print root.data,
            return True

        # Else return False
        return False

# Driver program to test above function
root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)
root.left.left.left = Node(7)

printAncestors(root, 7)

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

Output:

4 2 1

Time Complexity: O(n) where n is the number of nodes in the given Binary Tree.

### Source

<https://www.geeksforgeeks.org/print-ancestors-of-a-given-node-in-binary-tree/>

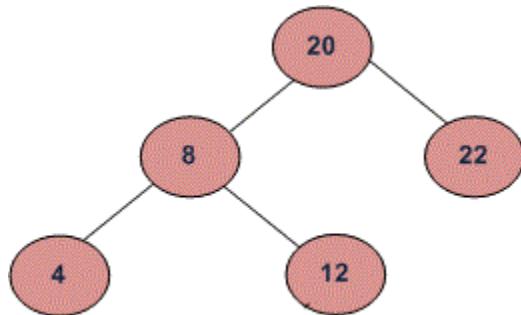
## Chapter 315

# Print BST keys in the given range

Print BST keys in the given range - GeeksforGeeks

Given two values  $k_1$  and  $k_2$  (where  $k_1 < k_2$ ) and a root pointer to a Binary Search Tree. Print all the keys of tree in range  $k_1$  to  $k_2$ . i.e. print all  $x$  such that  $k_1 \leq x \leq k_2$  and  $x$  is a key of given BST. Print all the keys in increasing order.

For example, if  $k_1 = 10$  and  $k_2 = 22$ , then your function should print 12, 20 and 22.



### Algorithm:

- 1) If value of root's key is greater than  $k_1$ , then recursively call in left subtree.
- 2) If value of root's key is in range, then print the root's key.
- 3) If value of root's key is smaller than  $k_2$ , then recursively call in right subtree.

### Implementation:

C

```
#include<stdio.h>
```

```
/* A tree node structure */
struct node
{
    int data;
    struct node *left;
    struct node *right;
};

/* The functions prints all the keys which in the given range [k1..k2].
   The function assumes than k1 < k2 */
void Print(struct node *root, int k1, int k2)
{
    /* base case */
    if ( NULL == root )
        return;

    /* Since the desired o/p is sorted, recurse for left subtree first
       If root->data is greater than k1, then only we can get o/p keys
       in left subtree */
    if ( k1 < root->data )
        Print(root->left, k1, k2);

    /* if root's data lies in range, then prints root's data */
    if ( k1 <= root->data && k2 >= root->data )
        printf("%d ", root->data );

    /* If root->data is smaller than k2, then only we can get o/p keys
       in right subtree */
    if ( k2 > root->data )
        Print(root->right, k1, k2);
}

/* Utility function to create a new Binary Tree node */
struct node* newNode(int data)
{
    struct node *temp = new struct node;
    temp->data = data;
    temp->left = NULL;
    temp->right = NULL;

    return temp;
}

/* Driver function to test above functions */
int main()
{
    struct node *root = new struct node;
    int k1 = 10, k2 = 25;
```

```
/* Constructing tree given in the above figure */
root = newNode(20);
root->left = newNode(8);
root->right = newNode(22);
root->left->left = newNode(4);
root->left->right = newNode(12);

Print(root, k1, k2);

getchar();
return 0;
}
```

**Java**

```
// Java program to print BST in given range

// A binary tree node
class Node {

    int data;
    Node left, right;

    Node(int d) {
        data = d;
        left = right = null;
    }
}

class BinaryTree {

    static Node root;

    /* The functions prints all the keys which in the given range [k1..k2].
     * The function assumes than k1 < k2 */
    void Print(Node node, int k1, int k2) {

        /* base case */
        if (node == null) {
            return;
        }

        /* Since the desired o/p is sorted, recurse for left subtree first
         * If root->data is greater than k1, then only we can get o/p keys
         * in left subtree */
        if (k1 < node.data) {
            Print(node.left, k1, k2);
        }

        /* If root->data is smaller than k2, then only we can get o/p keys
         * in right subtree */
        if (k2 > node.data) {
            Print(node.right, k1, k2);
        }
    }
}
```

```
}

/* if root's data lies in range, then prints root's data */
if (k1 <= node.data && k2 >= node.data) {
    System.out.print(node.data + " ");
}

/* If root->data is smaller than k2, then only we can get o/p keys
   in right subtree */
if (k2 > node.data) {
    Print(node.right, k1, k2);
}
}

public static void main(String[] args) {
    BinaryTree tree = new BinaryTree();
    int k1 = 10, k2 = 25;
    tree.root = new Node(20);
    tree.root.left = new Node(8);
    tree.root.right = new Node(22);
    tree.root.left.left = new Node(4);
    tree.root.left.right = new Node(12);

    tree.Print(root, k1, k2);
}
}

// This code has been contributed by Mayank Jaiswal
```

### Python

```
# Python program to find BST keys in given range

# A binary tree node
class Node:

    # Constructor to create a new node
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

    # The function prints all the keys in the gicven range
    # [k1..k2]. Assumes that k1 < k2
    def Print(root, k1, k2):

        # Base Case
```

```
if root is None:  
    return  
  
# Since the desired o/p is sorted, recurse for left  
# subtree first. If root.data is greater than k1, then  
# only we can get o/p keys in left subtree  
if k1 < root.data :  
    Print(root.left, k1, k2)  
  
# If root's data lies in range, then prints root's data  
if k1 <= root.data and k2 >= root.data:  
    print root.data,  
  
# If root.data is smaller than k2, then only we can get  
# o/p keys in right subtree  
if k2 > root.data:  
    Print(root.right, k1, k2)  
  
# Driver function to test above function  
k1 = 10 ; k2 = 25 ;  
root = Node(20)  
root.left = Node(8)  
root.right = Node(22)  
root.left.left = Node(4)  
root.left.right = Node(12)  
  
Print(root, k1, k2)  
  
# This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

Output:

12 20 22

Time Complexity: O(n) where n is the total number of keys in tree.

## Source

<https://www.geeksforgeeks.org/print-bst-keys-in-the-given-range/>

## Chapter 316

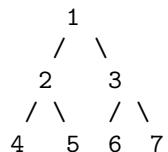
# Print Binary Tree in 2-Dimensions

Print Binary Tree in 2-Dimensions - GeeksforGeeks

Given a Binary Tree, print it in two dimension.

Examples:

Input : Pointer to root of below tree



Output :

```
7  
3  
6  
1  
5  
2  
4
```

We strongly recommend you to minimize your browser and try this yourself first.

If we take a closer look at the pattern, we can notice following.

- 1) Rightmost node is printed in first line and leftmost node is printed in last line.
- 2) Space count increases by a fixed amount at every level.

So we do a reverse inorder traversal (right – root – left) and print tree nodes. We increase space by a fixed amount at every level.

Below is C++ implementation.

```
// Program to print binary tree in 2D
#include<stdio.h>
#define COUNT 10

// A binary tree node
struct Node
{
    int data;
    Node* left, *right;
};

// Helper function to allocates a new node
Node* newNode(int data)
{
    Node* node = new Node;
    node->data = data;
    node->left = node->right = NULL;
    return node;
}

// Function to print binary tree in 2D
// It does reverse inorder traversal
void print2DUtil(Node *root, int space)
{
    // Base case
    if (root == NULL)
        return;

    // Increase distance between levels
    space += COUNT;

    // Process right child first
    print2DUtil(root->right, space);

    // Print current node after space
    // count
    printf("\n");
    for (int i = COUNT; i < space; i++)
        printf(" ");
    printf("%d\n", root->data);
}
```

```
// Process left child
print2DUtil(root->left, space);
}

// Wrapper over print2DUtil()
void print2D(Node *root)
{
    // Pass initial space count as 0
    print2DUtil(root, 0);
}

// Driver program to test above functions
int main()
{
    Node *root      = newNode(1);
    root->left     = newNode(2);
    root->right    = newNode(3);

    root->left->left  = newNode(4);
    root->left->right = newNode(5);
    root->right->left = newNode(6);
    root->right->right = newNode(7);

    root->left->left->left  = newNode(8);
    root->left->left->right = newNode(9);
    root->left->right->left = newNode(10);
    root->left->right->right = newNode(11);
    root->right->left->left = newNode(12);
    root->right->left->right = newNode(13);
    root->right->right->left = newNode(14);
    root->right->right->right = newNode(15);

    print2D(root);

    return 0;
}
```

Output :

3  
13  
6  
12  
1  
11  
5  
10  
2  
9  
4  
8

**Source**

<https://www.geeksforgeeks.org/print-binary-tree-2-dimensions/>

## Chapter 317

# Print Binary Tree levels in sorted order

Print Binary Tree levels in sorted order - GeeksforGeeks

Given a Binary tree, the task is to print its all level in sorted order

Examples:

```
Input :      7
          /   \
         6     5
        / \   / \
       4  3   2   1
Output :
7
5 6
1 2 3 4
```

```
Input :      7
          /   \
         16     1
        / \
       4   13
Output :
7
1 16
4 13
```

Here we can use two [Priority queue](#) for print in sorted order. We create an empty queue q and two priority queues, current\_level and next\_level. We use NULL as a separator between two levels. Whenever we encounter NULL in normal level order traversal, we swap current\_level and next\_level.

```
// CPP program to print levels in sorted order.
#include <iostream>
#include <queue>
#include <vector>
using namespace std;

// A Binary Tree Node
struct Node {
    int data;
    struct Node *left, *right;
};

// Iterative method to find height of Binary Tree
void printLevelOrder(Node* root)
{
    // Base Case
    if (root == NULL)
        return;

    // Create an empty queue for level order traversal
    queue<Node*> q;

    // A priority queue (or min heap) of integers for
    // to store all elements of current level.
    priority_queue<int, vector<int>, greater<int> > current_level;

    // A priority queue (or min heap) of integers for
    // to store all elements of next level.
    priority_queue<int, vector<int>, greater<int> > next_level;

    // push the root for traverse all next level nodes
    q.push(root);

    // for go level by level
    q.push(NULL);

    // push the first node data in previous_level queue
    current_level.push(root->data);

    while (q.empty() == false) {

        // Get top of priority queue
        int data = current_level.top();

        // Get top of queue
        Node* node = q.front();

        // if node == NULL (Means this is boundary
```

```
// between two levels), swap current_level
// next_level priority queues.
if (node == NULL) {
    q.pop();

    // here queue is empty represent
    // no element in the actual
    // queue
    if (q.empty())
        break;

    q.push(NULL);
    cout << "\n";

    // swap next_level to current_level level
    // for print in sorted order
    current_level.swap(next_level);

    continue;
}

// print the current_level data
cout << data << " ";

q.pop();
current_level.pop();

/* Enqueue left child */
if (node->left != NULL) {
    q.push(node->left);

    // Enqueue left child in next_level queue
    next_level.push(node->left->data);
}

/*Enqueue right child */
if (node->right != NULL) {
    q.push(node->right);

    // Enqueue right child in next_level queue
    next_level.push(node->right->data);
}

}

}

// Utility function to create a new tree node
Node* newNode(int data)
{
```

```
Node* temp = new Node;
temp->data = data;
temp->left = temp->right = NULL;
return temp;
}

// Driver program to test above functions
int main()
{
    // Let us create binary tree shown in above diagram
    Node* root = newNode(7);
    root->left = newNode(6);
    root->right = newNode(5);
    root->left->left = newNode(4);
    root->left->right = newNode(3);
    root->right->left = newNode(2);
    root->right->right = newNode(1);

    /*      7
           /   \
          6     5
         / \   / \
        4  3   2  1 */

    cout << "Level Order traversal of binary tree is \n";
    printLevelOrder(root);
    return 0;
}
```

**Output:**

```
Level Order traversal of binary tree is
7
5 6
1 2 3 4
```

**Source**

<https://www.geeksforgeeks.org/print-binary-tree-levels-sorted-order/>

## Chapter 318

# Print Binary Tree levels in sorted order | Set 2 (Using set)

Print Binary Tree levels in sorted order | Set 2 (Using set) - GeeksforGeeks

Given a tree, print the level order traversal in sorted order.

Examples :

Input :        7  
          /      \  
        6          5  
      / \        / \  
    4 3      2 1

Output :

7  
5 6  
1 2 3 4

Input :        7  
          /      \  
        16          1  
      / \  
    4 13

Output :

7  
1 16  
4 13

We have discussed a priority queue based solution in below post.

[Print Binary Tree levels in sorted order | Set 1 \(Using Priority Queue\)](#)

In this post, a [set](#) (which is implemented using balanced binary search tree) based solution is discussed.

**Approach :**

1. Start level order traversal of tree.
2. Store all the nodes in a set(or any other similar data structures).
3. Print elements of set.

C++

```
// CPP code to print level order
// traversal in sorted order
#include <bits/stdc++.h>
using namespace std;

struct Node {
    int data;
    Node* left;
    Node* right;
    Node(int dat = 0)
        : data(dat), left(nullptr),
          right(nullptr)
    {
    }
};

// Function to print sorted
// level order traversal
void sorted_level_order(Node* root)
{
    queue<Node*> q;
    set<int> s;

    q.push(root);
    q.push(nullptr);

    while (q.empty() == false) {
        Node* tmp = q.front();
        q.pop();

        if (tmp == nullptr) {
            if (s.empty() == true)
                break;
            for (set<int>::iterator it =
                  s.begin(); it != s.end(); ++it)
                cout << *it << " ";
            q.push(nullptr);
            s.clear();
        }
    }
}
```

```
        else {
            s.insert(tmp->data);

            if (tmp->left != nullptr)
                q.push(tmp->left);
            if (tmp->right != nullptr)
                q.push(tmp->right);
        }
    }

// Driver code
int main()
{
    Node* root = new Node(7);
    root->left = new Node(6);
    root->right = new Node(5);
    root->left->left = new Node(4);
    root->left->right = new Node(3);
    root->right->left = new Node(2);
    root->right->right = new Node(1);
    sorted_level_order(root);
    return 0;
}
```

**Output:**

7 5 6 1 2 3 4

**Source**

<https://www.geeksforgeeks.org/print-binary-tree-levels-sorted-order-2/>

## Chapter 319

# Print Common Nodes in Two Binary Search Trees

Print Common Nodes in Two Binary Search Trees - GeeksforGeeks

Given two Binary Search Trees, find common nodes in them. In other words, find intersection of two BSTs.

Example:

```
Input: root1 (Root of below tree)
      5
     /   \
    1     10
   / \   /
  0   4  7
     \
      9

root2 (Root of below tree)
      10
     /   \
    7     20
   / \
  4   9

Output: 4 7 9 10
```

**Method 1 (Simple Solution)** A simple way is to one by once search every node of first tree in second tree. Time complexity of this solution is  $O(m * h)$  where m is number of nodes in first tree and h is height of second tree.

**Method 2 (Linear Time)** We can find common elements in  $O(n)$  time.

- 1) Do inorder traversal of first tree and store the traversal in an auxiliary array  $ar1[]$ . See

sortedInorder() [here](#).

- 2) Do inorder traversal of second tree and store the traversal in an auxiliary array ar2[]
- 3) Find intersection of ar1[] and ar2[]. See [this](#)for details.

Time complexity of this method is  $O(m+n)$  where m and n are number of nodes in first and second tree respectively. This solution requires  $O(m+n)$  extra space.

**Method 3 (Linear Time and limited Extra Space)** We can find common elements in  $O(n)$  time and  $O(h_1 + h_2)$  extra space where  $h_1$  and  $h_2$  are heights of first and second BSTs respectively.

The idea is to use [iterative inorder traversal](#). We use two auxiliary stacks for two BSTs. Since we need to find common elements, whenever we get same element, we print it.

```
// Iterative traversal based method to find common elements
// in two BSTs.
#include<iostream>
#include<stack>
using namespace std;

// A BST node
struct Node
{
    int key;
    struct Node *left, *right;
};

// A utility function to create a new node
Node *newNode(int ele)
{
    Node *temp = new Node;
    temp->key = ele;
    temp->left = temp->right = NULL;
    return temp;
}

// Function two print common elements in given two trees
void printCommon(Node *root1, Node *root2)
{
    // Create two stacks for two inorder traversals
    stack<Node *> stack1, s1, s2;

    while (1)
    {
        // push the Nodes of first tree in stack s1
        if (root1)
        {
            s1.push(root1);
            root1 = root1->left;
        }
    }
}
```

```
// push the Nodes of second tree in stack s2
else if (root2)
{
    s2.push(root2);
    root2 = root2->left;
}

// Both root1 and root2 are NULL here
else if (!s1.empty() && !s2.empty())
{
    root1 = s1.top();
    root2 = s2.top();

    // If current keys in two trees are same
    if (root1->key == root2->key)
    {
        cout << root1->key << " ";
        s1.pop();
        s2.pop();

        // move to the inorder successor
        root1 = root1->right;
        root2 = root2->right;
    }

    else if (root1->key < root2->key)
    {
        // If Node of first tree is smaller, than that of
        // second tree, then its obvious that the inorder
        // successors of current Node can have same value
        // as that of the second tree Node. Thus, we pop
        // from s2
        s1.pop();
        root1 = root1->right;

        // root2 is set to NULL, because we need
        // new Nodes of tree 1
        root2 = NULL;
    }
    else if (root1->key > root2->key)
    {
        s2.pop();
        root2 = root2->right;
        root1 = NULL;
    }
}
```

```
// Both roots and both stacks are empty
else break;
}

// A utility function to do inorder traversal
void inorder(struct Node *root)
{
    if (root)
    {
        inorder(root->left);
        cout<<root->key<<" ";
        inorder(root->right);
    }
}

/* A utility function to insert a new Node with given key in BST */
struct Node* insert(struct Node* node, int key)
{
    /* If the tree is empty, return a new Node */
    if (node == NULL) return newNode(key);

    /* Otherwise, recur down the tree */
    if (key < node->key)
        node->left = insert(node->left, key);
    else if (key > node->key)
        node->right = insert(node->right, key);

    /* return the (unchanged) Node pointer */
    return node;
}

// Driver program
int main()
{
    // Create first tree as shown in example
    Node *root1 = NULL;
    root1 = insert(root1, 5);
    root1 = insert(root1, 1);
    root1 = insert(root1, 10);
    root1 = insert(root1, 0);
    root1 = insert(root1, 4);
    root1 = insert(root1, 7);
    root1 = insert(root1, 9);

    // Create second tree as shown in example
    Node *root2 = NULL;
    root2 = insert(root2, 10);
```

```
root2 = insert(root2, 7);
root2 = insert(root2, 20);
root2 = insert(root2, 4);
root2 = insert(root2, 9);

cout << "Tree 1 : ";
inorder(root1);
cout << endl;

cout << "Tree 2 : ";
inorder(root2);

cout << "\nCommon Nodes: ";
printCommon(root1, root2);

return 0;
}
```

Output:

```
4 7 9 10
```

This article is contributed by **Ekta Goel**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## Source

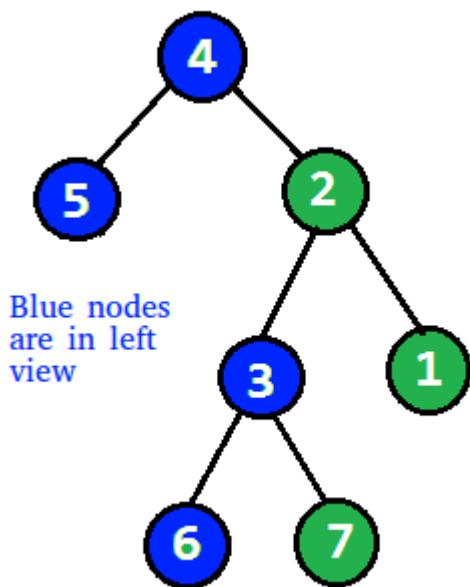
<https://www.geeksforgeeks.org/print-common-nodes-in-two-binary-search-trees/>

## Chapter 320

# Print Left View of a Binary Tree

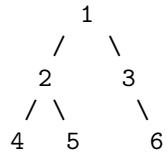
Print Left View of a Binary Tree - GeeksforGeeks

Given a Binary Tree, print left view of it. Left view of a Binary Tree is set of nodes visible when tree is visited from left side.



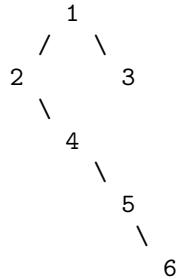
Examples:

Input :



Output : 1 2 4

Input :



Output : 1 2 4 5 6

The left view contains all nodes that are first nodes in their levels. A simple solution is to do **level order traversal** and print the first node in every level.

The problem can also be solved using **simple recursive traversal**. We can keep track of level of a node by passing a parameter to all recursive calls. The idea is to keep track of maximum level also. Whenever we see a node whose level is more than maximum level so far, we print the node because this is the first node in its level (Note that we traverse the left subtree before right subtree). Following is the implementation-

C

```
// C program to print left view of Binary Tree
#include<stdio.h>
#include<stdlib.h>

struct node
{
    int data;
    struct node *left, *right;
};

// A utility function to create a new Binary Tree node
struct node *newNode(int item)
{
    struct node *temp = (struct node *)malloc(sizeof(struct node));
    temp->data = item;
    temp->left = temp->right = NULL;
    return temp;
}
```

```
}

// Recursive function to print left view of a binary tree.
void leftViewUtil(struct node *root, int level, int *max_level)
{
    // Base Case
    if (root==NULL)  return;

    // If this is the first node of its level
    if (*max_level < level)
    {
        printf("%d\t", root->data);
        *max_level = level;
    }

    // Recur for left and right subtrees
    leftViewUtil(root->left, level+1, max_level);
    leftViewUtil(root->right, level+1, max_level);
}

// A wrapper over leftViewUtil()
void leftView(struct node *root)
{
    int max_level = 0;
    leftViewUtil(root, 1, &max_level);
}

// Driver Program to test above functions
int main()
{
    struct node *root = newNode(12);
    root->left = newNode(10);
    root->right = newNode(30);
    root->right->left = newNode(25);
    root->right->right = newNode(40);

    leftView(root);

    return 0;
}
```

### Java

```
// Java program to print left view of binary tree

/* Class containing left and right child of current
node and key value*/
class Node
```

```
{  
    int data;  
    Node left, right;  
  
    public Node(int item)  
    {  
        data = item;  
        left = right = null;  
    }  
}  
  
/* Class to print the left view */  
class BinaryTree  
{  
    Node root;  
    static int max_level = 0;  
  
    // recursive function to print left view  
    void leftViewUtil(Node node, int level)  
    {  
        // Base Case  
        if (node==null) return;  
  
        // If this is the first node of its level  
        if (max_level < level)  
        {  
            System.out.print(" " + node.data);  
            max_level = level;  
        }  
  
        // Recur for left and right subtrees  
        leftViewUtil(node.left, level+1);  
        leftViewUtil(node.right, level+1);  
    }  
  
    // A wrapper over leftViewUtil()  
    void leftView()  
    {  
        leftViewUtil(root, 1);  
    }  
  
    /* testing for example nodes */  
    public static void main(String args[])  
    {  
        /* creating a binary tree and entering the nodes */  
        BinaryTree tree = new BinaryTree();  
        tree.root = new Node(12);  
        tree.root.left = new Node(10);  
    }  
}
```

```
        tree.root.right = new Node(30);
        tree.root.right.left = new Node(25);
        tree.root.right.right = new Node(40);

        tree.leftView();
    }
}
```

### Python

```
# Python program to print left view of Binary Tree

# A binary tree node
class Node:

    # Constructor to create a new node
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

# Recursive function prtn left view of a binary tree
def leftViewUtil(root, level, max_level):

    # Base Case
    if root is None:
        return

    # If this is the first node of its level
    if (max_level[0] < level):
        print "%d\t" %(root.data),
        max_level[0] = level

    # Recur for left and right subtree
    leftViewUtil(root.left, level+1, max_level)
    leftViewUtil(root.right, level+1, max_level)

# A wrapper over leftViewUtil()
def leftView(root):
    max_level = [0]
    leftViewUtil(root, 1, max_level)

# Driver program to test above function
root = Node(12)
root.left = Node(10)
```

```
root.right = Node(20)
root.right.left = Node(25)
root.right.right = Node(40)

leftView(root)

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

Output:

```
12      10      25
```

Time Complexity: The function does a simple traversal of the tree, so the complexity is  $O(n)$ .

This article is contributed by Ramsai Chinthamani. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## Source

<https://www.geeksforgeeks.org/print-left-view-binary-tree/>

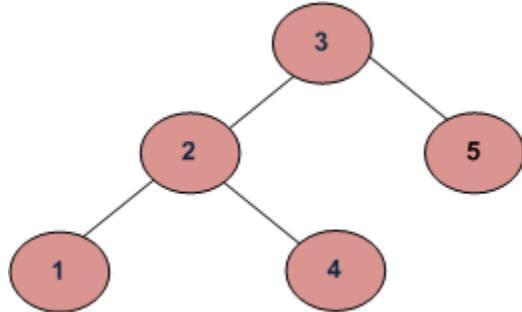
## Chapter 321

# Print Levels of all nodes in a Binary Tree

Print Levels of all nodes in a Binary Tree - GeeksforGeeks

Given a Binary Tree and a key, write a function that prints levels of all keys in given binary tree.

For example, consider the following tree. If the input key is 3, then your function should return 1. If the input key is 4, then your function should return 3. And for key which is not present in key, then your function should return 0.



Input:

```
3
 / \
2   5
 / \
1   4
```

Output:

```
Level of 1 is 3
Level of 2 is 2
```

```
Level of 3 is 1
Level of 4 is 3
Level of 5 is 2
```

We have discussed an recursive solution in below post.

[Get Level of a node in a Binary Tree](#)

In this post, an iterative solution based on [Level order traversal](#) is discussed. We store level of every node in queue together with the node while doing the traversal.

C++

```
// An iterative C++ program to print levels
// of all nodes
#include <bits/stdc++.h>
using namespace std;

/* A tree node structure */
struct Node {
    int data;
    struct Node* left;
    struct Node* right;
};

void printLevel(struct Node* root)
{
    if (!root)
        return;

    // queue to hold tree node with level
    queue<pair<struct Node*, int>> q;

    q.push({root, 1}); // let root node be at level 1

    pair<struct Node*, int> p;

    // Do level Order Traversal of tree
    while (!q.empty()) {
        p = q.front();
        q.pop();

        cout << "Level of " << p.first->data
            << " is " << p.second << "\n";

        if (p.first->left)
            q.push({p.first->left, p.second + 1});
        if (p.first->right)
            q.push({p.first->right, p.second + 1});
    }
}
```

```
    }
}

/* Utility function to create a new Binary Tree node */
struct Node* newNode(int data)
{
    struct Node* temp = new struct Node;
    temp->data = data;
    temp->left = temp->right = NULL;
    return temp;
}

/* Driver function to test above functions */
int main()
{
    struct Node* root = NULL;

    /* Constructing tree given in the above figure */
    root = newNode(3);
    root->left = newNode(2);
    root->right = newNode(5);
    root->left->left = newNode(1);
    root->left->right = newNode(4);

    printLevel(root);
    return 0;
}
```

### Java

```
// Java program to print
// levels of all nodes
import java.util.LinkedList;
import java.util.Queue;
public class Print_Level_Btree {

    /* A tree node structure */
    static class Node {
        int data;
        Node left;
        Node right;
        Node(int data){
            this.data = data;
            left = null;
            right = null;
        }
    }
}
```

```
// User defined class Pair to hold
// the node and its level
static class Pair{
    Node n;
    int i;
    Pair(Node n, int i){
        this.n = n;
        this.i = i;
    }
}

// function to print the nodes and
// its corresponding level
static void printLevel(Node root)
{
    if (root == null)
        return;

    // queue to hold tree node with level
    Queue<Pair> q = new LinkedList<Pair>();

    // let root node be at level 1
    q.add(new Pair(root, 1));

    Pair p;

    // Do level Order Traversal of tree
    while (!q.isEmpty()) {
        p = q.peek();
        q.remove();

        System.out.println("Level of " + p.n.data +
                           " is " + p.i);
        if (p.n.left != null)
            q.add(new Pair(p.n.left, p.i + 1));
        if (p.n.right != null)
            q.add(new Pair(p.n.right, p.i + 1));
    }
}

/* Driver function to test above
   functions */
public static void main(String args[])
{
    Node root = null;

    /* Constructing tree given in the
```

```
    above figure */
root = new Node(3);
root.left = new Node(2);
root.right = new Node(5);
root.left.left = new Node(1);
root.left.right = new Node(4);

printLevel(root);
}
}

// This code is contributed by Sumit Ghosh
```

Output :

```
Level of 3 is 1
Level of 2 is 2
Level of 5 is 2
Level of 1 is 3
Level of 4 is 3
```

Time Complexity: O(n) where n is the number of nodes in the given Binary Tree.

## Source

<https://www.geeksforgeeks.org/print-levels-nodes-binary-tree/>

## Chapter 322

# Print Nodes in Top View of Binary Tree

Print Nodes in Top View of Binary Tree - GeeksforGeeks

Top view of a binary tree is the set of nodes visible when the tree is viewed from the top. Given a binary tree, print the top view of it. The output nodes can be printed in any order. Expected time complexity is O(n)

A node x is there in output if x is the topmost node at its horizontal distance. Horizontal distance of left child of a node x is equal to horizontal distance of x minus 1, and that of right child is horizontal distance of x plus 1.

```
      1
     /   \
    2     3
   / \   / \
  4   5   6   7
Top view of the above binary tree is
4 2 1 3 7
```

```
      1
     /   \
    2     3
   \
  4
  \
  5
  \
  6
Top view of the above binary tree is
2 1 3 6
```

The idea is to do something similar to [vertical Order Traversal](#). Like [vertical Order Traversal](#), we need to nodes of same horizontal distance together. We do a level order traversal so that the topmost node at a horizontal node is visited before any other node of same horizontal distance below it. Hashing is used to check if a node at given horizontal distance is seen or not.

C++

```
// C++ program to print top
// view of binary tree
#include <bits/stdc++.h>
using namespace std;

// Structure of binary tree
struct Node {
    int data;
    struct Node *left, *right;
};

// function should print the topView of
// the binary tree
void topView(struct Node* root)
{
    if (root == NULL)
        return;

    unordered_map<int, int> m;
    queue<pair<Node*, int>> q;

    // push node and horizontal distance to queue
    q.push(make_pair(root, 0));

    while (!q.empty()) {
        pair<Node*, int> p = q.front();
        Node* n = p.first;
        int val = p.second;
        q.pop();

        // if horizontal value is not in the hashmap
        // that means it is the first value with that
        // horizontal distance so print it and store
        // this value in hashmap
        if (m.find(val) == m.end()) {
            m[val] = n->data;
            printf("%d ", n->data);
        }

        if (n->left != NULL)
            q.push(make_pair(n->left, val - 1));
    }
}
```

```
        if (n->right != NULL)
            q.push(make_pair(n->right, val + 1));
    }
}

// function to create a new node
struct Node* newNode(int key)
{
    struct Node* node = new Node;
    node->data = key;
    node->left = node->right = NULL;
    return node;
}

// main function
int main()
{
    /* Create following Binary Tree
       1
      / \
     2   3
      \   \
       4   5
          \   \
           6*/
    Node* root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->right = newNode(4);
    root->left->right->right = newNode(5);
    root->left->right->right->right = newNode(6);

    topView(root);
    return 0;
}

/* This code is contributed by Niteesh Kumar */
```

### Java

```
// Java program to print top view of Binary tree
import java.util.*;

// Class for a tree node
```

```
class TreeNode {
    // Members
    int key;
    TreeNode left, right;

    // Constructor
    public TreeNode(int key)
    {
        this.key = key;
        left = right = null;
    }
}

// A class to represent a queue item. The queue is used to do Level
// order traversal. Every Queue item contains node and horizontal
// distance of node from root
class QItem {
    TreeNode node;
    int hd;
    public QItem(TreeNode n, int h)
    {
        node = n;
        hd = h;
    }
}

// Class for a Binary Tree
class Tree {
    TreeNode root;

    // Constructors
    public Tree() { root = null; }
    public Tree(TreeNode n) { root = n; }

    // This method prints nodes in top view of binary tree
    public void printTopView()
    {
        // base case
        if (root == null) {
            return;
        }

        // Creates an empty hashset
        HashSet<Integer> set = new HashSet<>();

        // Create a queue and add root to it
        Queue<QItem> Q = new LinkedList<QItem>();
        Q.add(new QItem(root, 0)); // Horizontal distance of root is 0
```

```
// Standard BFS or level order traversal loop
while (!Q.isEmpty()) {
    // Remove the front item and get its details
    QItem qi = Q.remove();
    int hd = qi.hd;
    TreeNode n = qi.node;

    // If this is the first node at its horizontal distance,
    // then this node is in top view
    if (!set.contains(hd)) {
        set.add(hd);
        System.out.print(n.key + " ");
    }

    // Enqueue left and right children of current node
    if (n.left != null)
        Q.add(new QItem(n.left, hd - 1));
    if (n.right != null)
        Q.add(new QItem(n.right, hd + 1));
}
}

// Driver class to test above methods
public class Main {
    public static void main(String[] args)
    {
        /* Create following Binary Tree
           1
         /   \
        2     3
          \
         4
          \
         5
          \
         6*/
        TreeNode root = new TreeNode(1);
        root.left = new TreeNode(2);
        root.right = new TreeNode(3);
        root.left.right = new TreeNode(4);
        root.left.right.right = new TreeNode(5);
        root.left.right.right.right = new TreeNode(6);
        Tree t = new Tree(root);
        System.out.println("Following are nodes in top view of Binary Tree");
        t.printTopView();
    }
}
```

}

Output:

```
Following are nodes in top view of Binary Tree
1 2 3 6
```

Time Complexity of the above implementation is  $O(n)$  where  $n$  is number of nodes in given binary tree. The assumption here is that `add()` and `contains()` methods of HashSet work in  $O(1)$  time.

This article is contributed by **Rohan**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

**Improved By :** [Aarsee](#)

## Source

<https://www.geeksforgeeks.org/print-nodes-top-view-binary-tree/>

## Chapter 323

# Print Postorder traversal from given Inorder and Preorder traversals

Print Postorder traversal from given Inorder and Preorder traversals - GeeksforGeeks

Given Inorder and Preorder traversals of a binary tree, print Postorder traversal.

**Example:**

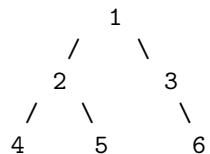
**Input:**

```
Inorder traversal in[] = {4, 2, 5, 1, 3, 6}  
Preorder traversal pre[] = {1, 2, 4, 5, 3, 6}
```

**Output:**

```
Postorder traversal is {4, 5, 2, 6, 3, 1}
```

Traversals in the above example represents following tree



A **naive method** is to first construct the tree, then use simple recursive method to print postorder traversal of the constructed tree.

We can print postorder traversal without constructing the tree. The idea is, root is always the first item in preorder traversal and it must be the last item in postorder

traversal. We first recursively print left subtree, then recursively print right subtree. Finally, print root. To find boundaries of left and right subtrees in pre[] and in[], we search root in in[], all elements before root in in[] are elements of left subtree and all elements after root are elements of right subtree. In pre[], all elements after index of root in in[] are elements of right subtree. And elements before index (including the element at index and excluding the first element) are elements of left subtree.

C++

```
// C++ program to print postorder traversal from preorder and inorder traversals
#include <iostream>
using namespace std;

// A utility function to search x in arr[] of size n
int search(int arr[], int x, int n)
{
    for (int i = 0; i < n; i++)
        if (arr[i] == x)
            return i;
    return -1;
}

// Prints postorder traversal from given inorder and preorder traversals
void printPostOrder(int in[], int pre[], int n)
{
    // The first element in pre[] is always root, search it
    // in in[] to find left and right subtrees
    int root = search(in, pre[0], n);

    // If left subtree is not empty, print left subtree
    if (root != 0)
        printPostOrder(in, pre + 1, root);

    // If right subtree is not empty, print right subtree
    if (root != n - 1)
        printPostOrder(in + root + 1, pre + root + 1, n - root - 1);

    // Print root
    cout << pre[0] << " ";
}

// Driver program to test above functions
int main()
{
    int in[] = { 4, 2, 5, 1, 3, 6 };
    int pre[] = { 1, 2, 4, 5, 3, 6 };
    int n = sizeof(in) / sizeof(in[0]);
    cout << "Postorder traversal " << endl;
    printPostOrder(in, pre, n);
}
```

```
    return 0;
}
```

### Python3

```
# Python program to print postorder
# traversal from preorder and
# inorder traversals
def printpostorder(inorder, preorder, n):
    if preorder[0] in inorder:
        root = inorder.index(preorder[0])

    if root != 0: # left subtree exists
        printpostorder(inorder[:root],
                        preorder[1:root + 1],
                        len(inorder[:root]))

    if root != n - 1: # right subtree exists
        printpostorder(inorder[root + 1:],
                        preorder[root + 1:],
                        len(inorder[root + 1:]))

    print preorder[0], 

# Driver Code
inorder = [4, 2, 5, 1, 3, 6];
preorder = [1, 2, 4, 5, 3, 6];
n = len(inorder)
print "Postorder traversal "
printpostorder(inorder, preorder, n)

# This code is contributed by SaiNath
```

### Output:

```
Postorder traversal
4 5 2 6 3 1
```

Below is Java implementation.

```
// Java program to print Postorder traversal from given Inorder
// and Preorder traversals.

public class PrintPost {
    static int preIndex = 0;
```

```
void printPost(int[] in, int[] pre, int inStrt, int inEnd)
{
    if (inStrt > inEnd)
        return;

    // Find index of next item in preorder traversal in
    // inorder.
    int inIndex = search(in, inStrt, inEnd, pre[preIndex++]);

    // traverse left tree
    printPost(in, pre, inStrt, inIndex - 1);

    // traverse right tree
    printPost(in, pre, inIndex + 1, inEnd);

    // print root node at the end of traversal
    System.out.print(in[inIndex] + " ");
}

int search(int[] in, int startIn, int endIn, int data)
{
    int i = 0;
    for (i = startIn; i < endIn; i++)
        if (in[i] == data)
            return i;
    return i;
}

// Driver code
public static void main(String args[])
{
    int in[] = { 4, 2, 5, 1, 3, 6 };
    int pre[] = { 1, 2, 4, 5, 3, 6 };
    int len = in.length;
    PrintPost tree = new PrintPost();
    tree.printPost(in, pre, 0, len - 1);
}
```

**Output:**

4 5 2 6 3 1

**Time Complexity:** The above function visits every node in array. For every visit, it calls search which takes  $O(n)$  time. Therefore, overall time complexity of the function is  $O(n^2)$

The above solution can be optimized using hashing. We use a HashMap to store elements and their indexes so that we can quickly find index of an element.

```
// Java program to print Postorder traversal from
// given Inorder and Preorder traversals.
import java.util.*;

public class PrintPost {
    static int preIndex = 0;
    void printPost(int[] in, int[] pre, int inStrt,
                  int inEnd, HashMap<Integer, Integer> hm)
    {
        if (inStrt > inEnd)
            return;

        // Find index of next item in preorder traversal in
        // inorder.
        int inIndex = hm.get(pre[preIndex++]);

        // traverse left tree
        printPost(in, pre, inStrt, inIndex - 1, hm);

        // traverse right tree
        printPost(in, pre, inIndex + 1, inEnd, hm);

        // print root node at the end of traversal
        System.out.print(in[inIndex] + " ");
    }

    void printPostMain(int[] in, int[] pre)
    {
        int n = pre.length;
        HashMap<Integer, Integer> hm = new HashMap<Integer, Integer>();
        for (int i=0; i<n; i++)
            hm.put(in[i], i);

        printPost(in, pre, 0, n-1, hm);
    }

    // Driver code
    public static void main(String args[])
    {
        int in[] = { 4, 2, 5, 1, 3, 6 };
        int pre[] = { 1, 2, 4, 5, 3, 6 };
        PrintPost tree = new PrintPost();
        tree.printPostMain(in, pre);
    }
}
```

**Output:**

4 5 2 6 3 1

Time complexity : O(n)

**Improved By :** [sainathcvs](#)

## Source

<https://www.geeksforgeeks.org/print-postorder-from-given-inorder-and-preorder-traversals/>

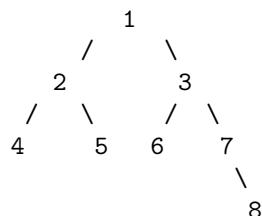
## Chapter 324

# Print Right View of a Binary Tree

Print Right View of a Binary Tree - GeeksforGeeks

Given a Binary Tree, print Right view of it. Right view of a Binary Tree is set of nodes visible when tree is visited from Right side.

Right view of following tree is 1 3 7 8



The Right view contains all nodes that are last nodes in their levels. A simple solution is to do [level order traversal](#) and print the last node in every level.

The problem can also be solved using simple recursive traversal. We can keep track of level of a node by passing a parameter to all recursive calls. The idea is to keep track of maximum level also. And traverse the tree in a manner that right subtree is visited before left subtree. Whenever we see a node whose level is more than maximum level so far, we print the node because this is the last node in its level (Note that we traverse the right subtree before left subtree). Following is C implementation of this approach.

C

```
// C program to print right view of Binary Tree
#include<stdio.h>
```

```
#include<stdlib.h>

struct Node
{
    int data;
    struct Node *left, *right;
};

// A utility function to create a new Binary Tree Node
struct Node *newNode(int item)
{
    struct Node *temp = (struct Node *)malloc(sizeof(struct Node));
    temp->data = item;
    temp->left = temp->right = NULL;
    return temp;
}

// Recursive function to print right view of a binary tree.
void rightViewUtil(struct Node *root, int level, int *max_level)
{
    // Base Case
    if (root==NULL) return;

    // If this is the last Node of its level
    if (*max_level < level)
    {
        printf("%d\t", root->data);
        *max_level = level;
    }

    // Recur for right subtree first, then left subtree
    rightViewUtil(root->right, level+1, max_level);
    rightViewUtil(root->left, level+1, max_level);
}

// A wrapper over rightViewUtil()
void rightView(struct Node *root)
{
    int max_level = 0;
    rightViewUtil(root, 1, &max_level);
}

// Driver Program to test above functions
int main()
{
    struct Node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
```

```
root->left->left = newNode(4);
root->left->right = newNode(5);
root->right->left = newNode(6);
root->right->right = newNode(7);
root->right->left->right = newNode(8);

rightView(root);

return 0;
}
```

**Java**

```
// Java program to print right view of binary tree

// A binary tree node
class Node {

    int data;
    Node left, right;

    Node(int item) {
        data = item;
        left = right = null;
    }
}

// class to access maximum level by reference
class Max_level {

    int max_level;
}

class BinaryTree {

    Node root;
    Max_level max = new Max_level();

    // Recursive function to print right view of a binary tree.
    void rightViewUtil(Node node, int level, Max_level max_level) {

        // Base Case
        if (node == null)
            return;

        // If this is the last Node of its level
        if (max_level.max_level < level) {
            System.out.print(node.data + " ");
            max_level.max_level = level;
        }

        rightViewUtil(node.left, level + 1, max_level);
        rightViewUtil(node.right, level + 1, max_level);
    }
}
```

```
    max_level.max_level = level;
}

// Recur for right subtree first, then left subtree
rightViewUtil(node.right, level + 1, max_level);
rightViewUtil(node.left, level + 1, max_level);
}

void rightView()
{
    rightView(root);
}

// A wrapper over rightViewUtil()
void rightView(Node node) {

    rightViewUtil(node, 1, max);
}

// Driver program to test the above functions
public static void main(String args[]) {
    BinaryTree tree = new BinaryTree();
    tree.root = new Node(1);
    tree.root.left = new Node(2);
    tree.root.right = new Node(3);
    tree.root.left.left = new Node(4);
    tree.root.left.right = new Node(5);
    tree.root.right.left = new Node(6);
    tree.root.right.right = new Node(7);
    tree.root.right.left.right = new Node(8);

    tree.rightView();
}

}

// This code has been contributed by Mayank Jaiswal
```

### Python

```
# Python program to print right view of Binary Tree

# A binary tree node
class Node:
    # A constructor to create a new Binary tree Node
    def __init__(self, item):
        self.data = item
        self.left = None
```

```
        self.right = None

# Recursive function to print right view of Binary Tree
# used max_level as reference list ..only max_level[0]
# is helpful to us
def rightViewUtil(root, level, max_level):

    # Base Case
    if root is None:
        return

    # If this is the last node of its level
    if (max_level[0] < level):
        print "%d " %(root.data),
        max_level[0] = level

    # Recur for right subtree first, then left subtree
    rightViewUtil(root.right, level+1, max_level)
    rightViewUtil(root.left, level+1, max_level)

def rightView(root):
    max_level = [0]
    rightViewUtil(root, 1, max_level)

# Driver program to test above function
root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)
root.right.left = Node(6)
root.right.right = Node(7)
root.right.left.right = Node(8)

rightView(root)

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

Output:

```
1      3      7      8
```

#### Right view of Binary Tree using Queue

Time Complexity: The function does a simple traversal of the tree, so the complexity is  $O(n)$ .

This article is contributed by **Shalki Agarwal**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

**Source**

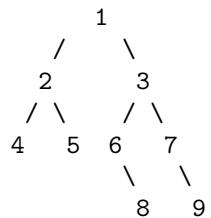
<https://www.geeksforgeeks.org/print-right-view-binary-tree-2/>

## Chapter 325

# Print a Binary Tree in Vertical Order | Set 1

Print a Binary Tree in Vertical Order | Set 1 - GeeksforGeeks

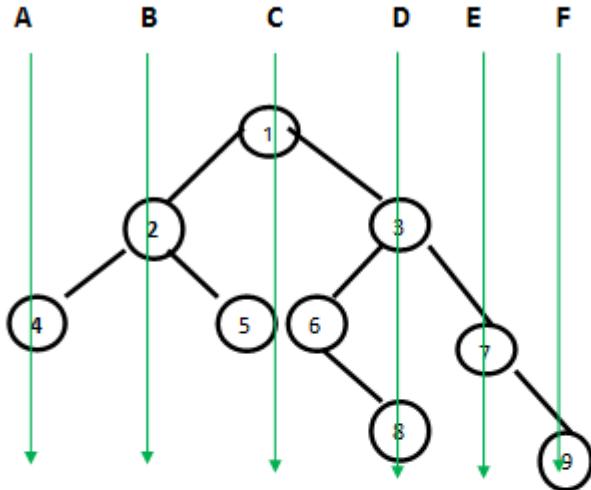
Given a binary tree, print it vertically. The following example illustrates vertical order traversal.



The output of print this tree vertically will be:

```
4
2
1 5 6
3 8
7
9
```

### Vertical Lines



**Vertical order traversal is:**

- A- 4
- B- 2
- C- 1 5 6
- D- 3 8
- E- 7
- F- 9

The idea is to traverse the tree once and get the minimum and maximum horizontal distance with respect to root. For the tree shown above, minimum distance is -2 (for node with value 4) and maximum distance is 3 (For node with value 9).

Once we have maximum and minimum distances from root, we iterate for each vertical line at distance minimum to maximum from root, and for each vertical line traverse the tree and print the nodes which lie on that vertical line.

**Algorithm:**

```

// min --> Minimum horizontal distance from root
// max --> Maximum horizontal distance from root
// hd --> Horizontal distance of current node from root
findMinMax(tree, min, max, hd)
    if tree is NULL then return;
    if hd is less than min then
  
```

```
        min = hd;
else if hd is greater than max then
    *max = hd;

findMinMax(tree->left, min, max, hd-1);
findMinMax(tree->right, min, max, hd+1);

printVerticalLine(tree, line_no, hd)
if tree is NULL then return;

if hd is equal to line_no, then
    print(tree->data);
printVerticalLine(tree->left, line_no, hd-1);
printVerticalLine(tree->right, line_no, hd+1);
```

**Implementation:**

Following is the implementation of above algorithm.

C++

```
#include <iostream>
using namespace std;

// A node of binary tree
struct Node
{
    int data;
    struct Node *left, *right;
};

// A utility function to create a new Binary Tree node
Node* newNode(int data)
{
    Node *temp = new Node;
    temp->data = data;
    temp->left = temp->right = NULL;
    return temp;
}

// A utility function to find min and max distances with respect
// to root.
void findMinMax(Node *node, int *min, int *max, int hd)
{
    // Base case
    if (node == NULL) return;

    // Update min and max
```

```

        if (hd < *min) *min = hd;
        else if (hd > *max) *max = hd;

        // Recur for left and right subtrees
        findMinMax(node->left, min, max, hd-1);
        findMinMax(node->right, min, max, hd+1);
    }

    // A utility function to print all nodes on a given line_no.
    // hd is horizontal distance of current node with respect to root.
    void printVerticalLine(Node *node, int line_no, int hd)
    {
        // Base case
        if (node == NULL) return;

        // If this node is on the given line number
        if (hd == line_no)
            cout << node->data << " ";

        // Recur for left and right subtrees
        printVerticalLine(node->left, line_no, hd-1);
        printVerticalLine(node->right, line_no, hd+1);
    }

    // The main function that prints a given binary tree in
    // vertical order
    void verticalOrder(Node *root)
    {
        // Find min and max distances with respect to root
        int min = 0, max = 0;
        findMinMax(root, &min, &max, 0);

        // Iterate through all possible vertical lines starting
        // from the leftmost line and print nodes line by line
        for (int line_no = min; line_no <= max; line_no++)
        {
            printVerticalLine(root, line_no, 0);
            cout << endl;
        }
    }

    // Driver program to test above functions
    int main()
    {
        // Create binary tree shown in above figure
        Node *root = newNode(1);
        root->left = newNode(2);
        root->right = newNode(3);
    }
}

```

```
root->left->left = newNode(4);
root->left->right = newNode(5);
root->right->left = newNode(6);
root->right->right = newNode(7);
root->right->left->right = newNode(8);
root->right->right->right = newNode(9);

cout << "Vertical order traversal is \n";
verticalOrder(root);

return 0;
}
```

**Java**

```
// Java program to print binary tree in reverse order

// A binary tree node
class Node
{
    int data;
    Node left, right;

    Node(int item)
    {
        data = item;
        left = right = null;
    }
}

class Values
{
    int max, min;
}

class BinaryTree
{
    Node root;
    Values val = new Values();

    // A utility function to find min and max distances with respect
    // to root.
    void findMinMax(Node node, Values min, Values max, int hd)
    {
        // Base case
        if (node == null)
            return;
```

```
// Update min and max
if (hd < min.min)
    min.min = hd;
else if (hd > max.max)
    max.max = hd;

// Recur for left and right subtrees
findMinMax(node.left, min, max, hd - 1);
findMinMax(node.right, min, max, hd + 1);
}

// A utility function to print all nodes on a given line_no.
// hd is horizontal distance of current node with respect to root.
void printVerticalLine(Node node, int line_no, int hd)
{
    // Base case
    if (node == null)
        return;

    // If this node is on the given line number
    if (hd == line_no)
        System.out.print(node.data + " ");

    // Recur for left and right subtrees
    printVerticalLine(node.left, line_no, hd - 1);
    printVerticalLine(node.right, line_no, hd + 1);
}

// The main function that prints a given binary tree in
// vertical order
void verticalOrder(Node node)
{
    // Find min and max distances with respect to root
    findMinMax(node, val, val, 0);

    // Iterate through all possible vertical lines starting
    // from the leftmost line and print nodes line by line
    for (int line_no = val.min; line_no <= val.max; line_no++)
    {
        printVerticalLine(node, line_no, 0);
        System.out.println("");
    }
}

// Driver program to test the above functions
public static void main(String args[])
{
    BinaryTree tree = new BinaryTree();
```

```
/* Let us construct the tree shown in above diagram */
tree.root = new Node(1);
tree.root.left = new Node(2);
tree.root.right = new Node(3);
tree.root.left.left = new Node(4);
tree.root.left.right = new Node(5);
tree.root.right.left = new Node(6);
tree.root.right.right = new Node(7);
tree.root.right.left.right = new Node(8);
tree.root.right.right.right = new Node(9);

System.out.println("vertical order traversal is :");
tree.verticalOrder(tree.root);
}

}

// This code has been contributed by Mayank Jaiswal
```

### Python

```
# Program to print binary tree in vertical order

# A binary tree
class Node:
    # Constructor to create a new node
    def __init__(self, key):
        self.data = key
        self.left = None
        self.right = None

# A utility function to find min and max distances with
# respect to root
def findMinMax(node, minimum, maximum, hd):

    # Base Case
    if node is None:
        return

    # Update min and max
    if hd < minimum[0] :
        minimum[0] = hd
    elif hd > maximum[0]:
        maximum[0] = hd

    # Recur for left and right subtrees
    findMinMax(node.left, minimum, maximum, hd-1)
    findMinMax(node.right, minimum, maximum, hd+1)
```

```
# A utility function to print all nodes on a given line_no
# hd is horizontal distance of current node with respect to root
def printVerticalLine(node, line_no, hd):

    # Base Case
    if node is None:
        return

    # If this node is on the given line number
    if hd == line_no:
        print node.data,

    # Recur for left and right subtrees
    printVerticalLine(node.left, line_no, hd-1)
    printVerticalLine(node.right, line_no, hd+1)

def verticalOrder(root):

    # Find min and max distances with respect to root
    minimum = [0]
    maximum = [0]
    findMinMax(root, minimum, maximum, 0)

    # Iterate through all possible lines starting
    # from the leftmost line and print nodes line by line
    for line_no in range(minimum[0], maximum[0]+1):
        printVerticalLine(root, line_no, 0)
        print

# Driver program to test above function
root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)
root.right.left = Node(6)
root.right.right = Node(7)
root.right.left.right = Node(8)
root.right.right.right = Node(9)

print "Vertical order traversal is"
verticalOrder(root)

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

Output:

Vertical order traversal is

```
4  
2  
1 5 6  
3 8  
7  
9
```

**Time Complexity:** Time complexity of above algorithm is  $O(w^*n)$  where w is width of Binary Tree and n is number of nodes in Binary Tree. In worst case, the value of w can be  $O(n)$  (consider a complete tree for example) and time complexity can become  $O(n^2)$ .

This problem can be solved more efficiently using the technique discussed in [this post](#). We will soon be discussing complete algorithm and implementation of more efficient method.

This article is contributed by **Shalki Agarwal**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## Source

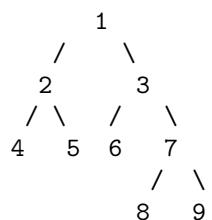
<https://www.geeksforgeeks.org/print-binary-tree-vertical-order/>

## Chapter 326

# Print a Binary Tree in Vertical Order | Set 2 (Map based Method)

Print a Binary Tree in Vertical Order | Set 2 (Map based Method) - GeeksforGeeks

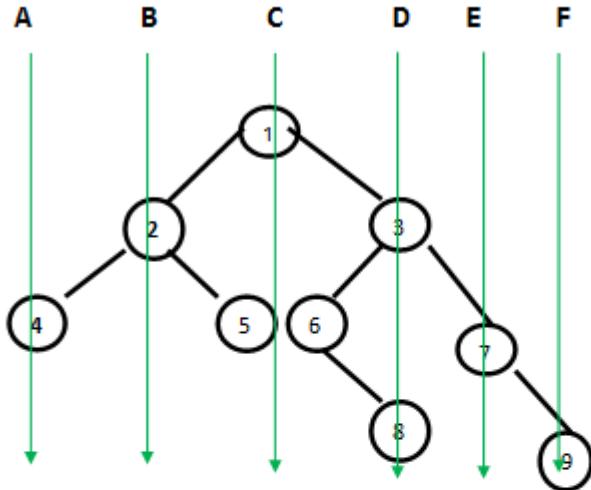
Given a binary tree, print it vertically. The following example illustrates vertical order traversal.



The output of print this tree vertically will be:

```
4
2
1 5 6
3 8
7
9
```

### Vertical Lines



**Vertical order traversal is:**

- A- 4
- B- 2
- C- 1 5 6
- D- 3 8
- E- 7
- F- 9

We have discussed a  $O(n^2)$  solution in the [previous post](#). In this post, an efficient solution based on hash map is discussed. We need to check the Horizontal Distances from root for all nodes. If two nodes have the same Horizontal Distance (HD), then they are on same vertical line. The idea of HD is simple. HD for root is 0, a right edge (edge connecting to right subtree) is considered as +1 horizontal distance and a left edge is considered as -1 horizontal distance. For example, in the above tree, HD for Node 4 is at -2, HD for Node 2 is -1, HD for 5 and 6 is 0 and HD for node 7 is +2.

We can do preorder traversal of the given Binary Tree. While traversing the tree, we can recursively calculate HDs. We initially pass the horizontal distance as 0 for root. For left subtree, we pass the Horizontal Distance as Horizontal distance of root minus 1. For right subtree, we pass the Horizontal Distance as Horizontal Distance of root plus 1. For every HD value, we maintain a list of nodes in a hash map. Whenever we see a node in traversal, we go to the hash map entry and add the node to the hash map using HD as a key in map.

Following is C++ implementation of the above method. Thanks to Chirag for providing the below C++ implementation.

C++

```
// C++ program for printing vertical order of a given binary tree
#include <iostream>
#include <vector>
#include <map>
using namespace std;

// Structure for a binary tree node
struct Node
{
    int key;
    Node *left, *right;
};

// A utility function to create a new node
struct Node* newNode(int key)
{
    struct Node* node = new Node;
    node->key = key;
    node->left = node->right = NULL;
    return node;
}

// Utility function to store vertical order in map 'm'
// 'hd' is horizontal distance of current node from root.
// 'hd' is initially passed as 0
void getVerticalOrder(Node* root, int hd, map<int, vector<int>> &m)
{
    // Base case
    if (root == NULL)
        return;

    // Store current node in map 'm'
    m[hd].push_back(root->key);

    // Store nodes in left subtree
    getVerticalOrder(root->left, hd-1, m);

    // Store nodes in right subtree
    getVerticalOrder(root->right, hd+1, m);
}

// The main function to print vertical order of a binary tree
// with given root
void printVerticalOrder(Node* root)
{
    // Create a map and store vertical order in map using
```

```
// function getVerticalOrder()
map < int,vector<int> > m;
int hd = 0;
getVerticalOrder(root, hd,m);

// Traverse the map and print nodes at every horigontal
// distance (hd)
map< int,vector<int> > :: iterator it;
for (it=m.begin(); it!=m.end(); it++)
{
    for (int i=0; i<it->second.size(); ++i)
        cout << it->second[i] << " ";
    cout << endl;
}
}

// Driver program to test above functions
int main()
{
    Node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->left = newNode(6);
    root->right->right = newNode(7);
    root->right->left->right = newNode(8);
    root->right->right->right = newNode(9);
    cout << "Vertical order traversal is n";
    printVerticalOrder(root);
    return 0;
}
```

### Java

```
// Java program for printing vertical order of a given binary tree
import java.util.TreeMap;
import java.util.Vector;
import java.util.Map.Entry;

public class VerticalOrderBtree
{
    // Tree node
    static class Node
    {
        int key;
        Node left;
        Node right;
```

```
// Constructor
Node(int data)
{
    key = data;
    left = null;
    right = null;
}
}

// Utility function to store vertical order in map 'm'
// 'hd' is horizontal distance of current node from root.
// 'hd' is initially passed as 0
static void getVerticalOrder(Node root, int hd,
                             TreeMap<Integer,Vector<Integer>> m)
{
    // Base case
    if(root == null)
        return;

    //get the vector list at 'hd'
    Vector<Integer> get = m.get(hd);

    // Store current node in map 'm'
    if(get == null)
    {
        get = new Vector<>();
        get.add(root.key);
    }
    else
        get.add(root.key);

    m.put(hd, get);

    // Store nodes in left subtree
    getVerticalOrder(root.left, hd-1, m);

    // Store nodes in right subtree
    getVerticalOrder(root.right, hd+1, m);
}

// The main function to print vertical oder of a binary tree
// with given root
static void printVerticalOrder(Node root)
{
    // Create a map and store vertical oder in map using
    // function getVerticalOrder()
    TreeMap<Integer,Vector<Integer>> m = new TreeMap<>();
```

```
int hd =0;
getVerticalOrder(root,hd,m);

// Traverse the map and print nodes at every horizontal
// distance (hd)
for (Entry<Integer, Vector<Integer>> entry : m.entrySet())
{
    System.out.println(entry.getValue());
}
}

// Driver program to test above functions
public static void main(String[] args) {

    // TO DO Auto-generated method stub
    Node root = new Node(1);
    root.left = new Node(2);
    root.right = new Node(3);
    root.left.left = new Node(4);
    root.left.right = new Node(5);
    root.right.left = new Node(6);
    root.right.right = new Node(7);
    root.right.left.right = new Node(8);
    root.right.right.right = new Node(9);
    System.out.println("Vertical Order traversal is");
    printVerticalOrder(root);
}
}

// This code is contributed by Sumit Ghosh
```

### Python

```
# Python program for printing vertical order of a given
# binary tree

# A binary tree node
class Node:
    # Constructor to create a new node
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None

    # Utility function to store vertical order in map 'm'
    # 'hd' is horizontal distance of current node from root
    # 'hd' is initially passed as 0
    def getVerticalOrder(self, hd, m):
```

```
# Base Case
if root is None:
    return

# Store current node in map 'm'
try:
    m[hd].append(root.key)
except:
    m[hd] = [root.key]

# Store nodes in left subtree
getVerticalOrder(root.left, hd-1, m)

# Store nodes in right subtree
getVerticalOrder(root.right, hd+1, m)

# The main function to print vertical order of a binary
# tree with given root
def printVerticalOrder(root):

    # Create a map and store vertical order in map using
    # function getVerticalOrder()
    m = dict()
    hd = 0
    getVerticalOrder(root, hd, m)

    # Traverse the map and print nodes at every horizontal
    # distance (hd)
    for index, value in enumerate(sorted(m)):
        for i in m[value]:
            print i,
        print

# Driver program to test above function
root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)
root.right.left = Node(6)
root.right.right = Node(7)
root.right.left.right = Node(8)
root.right.right.right = Node(9)
print "Vertical order traversal is"
printVerticalOrder(root)

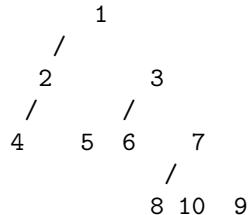
# This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

Output:

```
Vertical order traversal is
4
2
1 5 6
3 8
7
9
```

**Time Complexity** of hashing based solution can be considered as  $O(n)$  under the assumption that we have good hashing function that allows insertion and retrieval operations in  $O(1)$  time. In the above C++ implementation, [map of STL](#) is used. map in STL is typically implemented using a Self-Balancing Binary Search Tree where all operations take  $O(\log n)$  time. Therefore time complexity of above implementation is  $O(n \log n)$ .

**Note that the above solution may print nodes in same vertical order as they appear in tree.** For example, the above program prints 12 before 9. See [this](#) for a sample run.



11

12

Refer below post for level order traversal based solution. The below post makes sure that nodes of a vertical line are printed in same order as they appear in tree.

[Print a Binary Tree in Vertical Order | Set 3 \(Using Level Order Traversal\)](#)

## Source

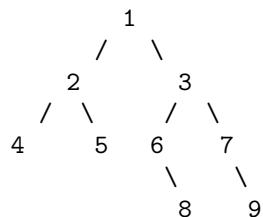
<https://www.geeksforgeeks.org/print-binary-tree-vertical-order-set-2/>

## Chapter 327

# Print a Binary Tree in Vertical Order | Set 3 (Using Level Order Traversal)

Print a Binary Tree in Vertical Order | Set 3 (Using Level Order Traversal) - GeeksforGeeks

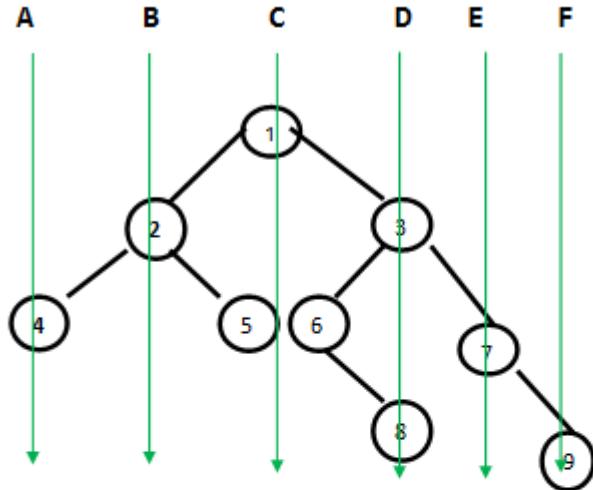
Given a binary tree, print it vertically. The following example illustrates vertical order traversal.



The output of print this tree vertically will be:

```
4
2
1 5 6
3 8
7
9
```

### Vertical Lines



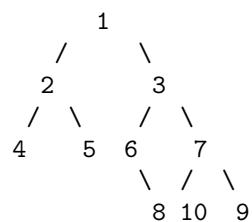
Vertical order traversal is:

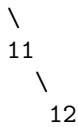
- A- 4
- B- 2
- C- 1 5 6
- D- 3 8
- E- 7
- F- 9

We have discussed an efficient approach in below post.

[Print a Binary Tree in Vertical Order | Set 2 \(Hashmap based Method\)](#)

The above solution uses preorder traversal and Hashmap to store nodes according to horizontal distances. Since above approach uses preorder traversal, nodes in a vertical line may not be printed in same order as they appear in tree. For example, the above solution prints 12 before 9 in below tree. See [this](#) for a sample run.





If we use [level order traversal](#), we can make sure that if a node like 12 comes below in same vertical line, it is printed after a node like 9 which comes above in vertical line.

1. To maintain a hash for the branch of each node.
2. Traverse the tree in level order fashion.
3. In level order traversal, maintain a queue which holds, node and its vertical branch.
  - \* pop from queue.
  - \* add this node's data in vector corresponding to its branch in the hash.
  - \* if this node has left child, insert in the queue, left with branch - 1.
  - \* if this node has right child, insert in the queue, right with branch + 1.

C++

```
// C++ program for printing vertical order  
// of a given binary tree using BFS.  
#include<bits/stdc++.h>  
  
using namespace std;  
  
// Structure for a binary tree node  
struct Node  
{  
    int key;  
    Node *left, *right;  
};  
  
// A utility function to create a new node  
Node* newNode(int key)  
{  
    Node* node = new Node;  
    node->key = key;  
    node->left = node->right = NULL;  
    return node;  
}  
  
// The main function to print vertical order of a  
// binary tree with given root
```

```
void printVerticalOrder(Node* root)
{
    // Base case
    if (!root)
        return;

    // Create a map and store vertical order in
    // map using function getVerticalOrder()
    map < int, vector<int> > m;
    int hd = 0;

    // Create queue to do level order traversal.
    // Every item of queue contains node and
    // horizontal distance.
    queue<pair<Node*, int> > que;
    que.push(make_pair(root, hd));

    while (!que.empty())
    {
        // pop from queue front
        pair<Node *, int> temp = que.front();
        que.pop();
        hd = temp.second;
        Node* node = temp.first;

        // insert this node's data in vector of hash
        m[hd].push_back(node->key);

        if (node->left != NULL)
            que.push(make_pair(node->left, hd-1));
        if (node->right != NULL)
            que.push(make_pair(node->right, hd+1));
    }

    // Traverse the map and print nodes at
    // every horizontal distance (hd)
    map< int, vector<int> > :: iterator it;
    for (it=m.begin(); it!=m.end(); it++)
    {
        for (int i=0; i<it->second.size(); ++i)
            cout << it->second[i] << " ";
        cout << endl;
    }
}

// Driver program to test above functions
int main()
{
```

```
Node *root = newNode(1);
root->left = newNode(2);
root->right = newNode(3);
root->left->left = newNode(4);
root->left->right = newNode(5);
root->right->left = newNode(6);
root->right->right = newNode(7);
root->right->left->right = newNode(8);
root->right->right->right = newNode(9);
root->right->right->left= newNode(10);
root->right->right->left->right= newNode(11);
root->right->right->left->right->right= newNode(12);
cout << "Vertical order traversal is \n";
printVerticalOrder(root);
return 0;
}
```

### Python3

```
#python3 Program to print zigzag traversal of binary tree
import collections
# Binary tree node
class Node:
    # Constructor to create a new node
    def __init__(self, data):
        self.data = data
        self.left = self.right = None

# function to print vertical order traversal of binary tree
def verticalTraverse(root):

    # Base case
    if root is None:
        return

    # Create empty queue for level order traversal
    queue = []

    # create a map to store nodes at a particular
    # horizontal distance
    m = {}

    # map to store horizontal distance of nodes
    hd_node = {}

    # enqueue root
    queue.append(root)
    # store the horizontal distance of root as 0
```

```
hd_node[root] = 0

m[0] = [root.data]

# loop will run while queue is not empty
while len(queue) > 0:

    # dequeue node from queue
    temp = queue.pop(0)

    if temp.left:
        # Enqueue left child
        queue.append(temp.left)

        # Store the horizontal distance of left node
        # hd(left child) = hd(parent) -1
        hd_node[temp.left] = hd_node[temp] - 1
        hd = hd_node[temp.left]

        if m.get(hd) is None:
            m[hd] = []

        m[hd].append(temp.left.data)

    if temp.right:
        # Enqueue right child
        queue.append(temp.right)

        # store the horizontal distance of right child
        # hd(right child) = hd(parent) + 1
        hd_node[temp.right] = hd_node[temp] + 1
        hd = hd_node[temp.right]

        if m.get(hd) is None:
            m[hd] = []

        m[hd].append(temp.right.data)

# Sort the map according to horizontal distance
sorted_m = collections.OrderedDict(sorted(m.items()))

# Traverse the sorted map and print nodes at each horizontal distance
for i in sorted_m.values():
    for j in i:
        print(j, " ", end="")
    print()

# Driver program to check above function
```

```
"""
Constructed binary tree is
      1
     / \
    2   3
   / \ / \
  4   5 6   7
         \ / \
        8 10 9
           \
          11
             \
            12

"""
root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)
root.right.left = Node(6)
root.right.right = Node(7)
root.right.left.right = Node(8)
root.right.right.left = Node(10)
root.right.right.right = Node(9)
root.right.right.left.right = Node(11)
root.right.right.left.right.right = Node(12)
print("Vertical order traversal is ")
verticalTraverse(root)

# This code is contributed by Shweta Singh
```

Output:

```
Vertical order traversal is
4
2
1 5 6
3 8 10
7 11
9 12
```

Time Complexity of above implementation is  $O(n \log n)$ . Note that above implementation uses map which is implemented using self-balancing BST.

We can reduce time complexity to  $O(n)$  using unordered\_map. To print nodes in desired order, we can have 2 variables denoting min and max horizontal distance. We can simply

Chapter 327. Print a Binary Tree in Vertical Order | Set 3 (Using Level Order Traversal)

iterate from min to max horizontal distance and get corresponding values from Map. So it is  $O(n)$

Auxiliary Space :  $O(n)$

Improved By : [shweta44](#)

## Source

<https://www.geeksforgeeks.org/print-a-binary-tree-in-vertical-order-set-3-using-level-order-traversal/>

## Chapter 328

# Print all full nodes in a Binary Tree

Print all full nodes in a Binary Tree - GeeksforGeeks

Given a binary tree, print all nodes will are full nodes. **Full Nodes** are nodes which has both left and right children as non-empty.

Examples:

```
Input :      10
          /   \
         8     2
        / \   /
       3   5  7
Output : 10 8
```

```
Input :      1
          / \
         2   3
        / \
       4   6
Output : 1 3
```

This is a simple problem. We do any of the traversals ([Inorder](#), [Preorder](#), [Postorder](#), level order traversal) and keep printing nodes that have mode left and right children as non-NULL.

```
// A C++ program to find the all full nodes in
// a given binary tree
#include <iostream>
using namespace std;
```

```
struct Node
{
    int data;
    struct Node *left, *right;
};

Node *newNode(int data)
{
    Node *temp = new Node;
    temp->data = data;
    temp->left = temp->right = NULL;
    return temp;
}

// Traverses given tree in Inorder fashion and
// prints all nodes that have both children as
// non-empty.
void findFullNode(Node *root)
{
    if (root != NULL)
    {
        findFullNode(root->left);
        if (root->left != NULL && root->right != NULL)
            cout << root->data << " ";
        findFullNode(root->right);
    }
}

// Driver program to test above function
int main()
{
    Node* root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->right->left = newNode(5);
    root->right->right = newNode(6);
    root->right->left->right = newNode(7);
    root->right->right->right = newNode(8);
    root->right->left->right->left = newNode(9);
    findFullNode(root);
    return 0;
}
```

Output:

1 3

Time Complexity :  $O(n)$

**Source**

<https://www.geeksforgeeks.org/print-full-nodes-binary-tree/>

## Chapter 329

# Print all k-sum paths in a binary tree

Print all k-sum paths in a binary tree - GeeksforGeeks

A binary tree and a number k are given. Print every path in the tree with sum of the nodes in the path as k.

A path can start from any node and end at any node and must be downward only, i.e. they need not be root node and leaf node; and negative numbers can also be there in the tree.

Examples:

```
Input : k = 5
        Root of below binary tree:
              1
            /   \
          3     -1
         / \   / \
        2   1   4   5
       / \ / \   \
      1   1  2   6
```

```
Output :
3 2
3 1 1
1 3 1
4 1
1 -1 4 1
-1 4 2
5
1 -1 5
```

Source : [Amazon Interview Experience Set-323](#)

Kindly note that this problem is significantly different from [finding k-sum path from root to leaves](#). Here each node can be treated as root, hence the path can start and end at any node.

The basic idea to solve the problem is to do a preorder traversal of the given tree. We also need a container (vector) to keep track of the path that led to that node. At each node we check if there are any path that sums to k, if any we print the path and proceed recursively to print each path.

Below is the C++ program for the same.

```
// C++ program to print all paths with sum k.
#include <bits/stdc++.h>
using namespace std;

//utility function to print contents of
//a vector from index i to it's end
void printVector(const vector<int>& v, int i)
{
    for (int j=i; j<v.size(); j++)
        cout << v[j] << " ";
    cout << endl;
}

// binary tree node
struct Node
{
    int data;
    Node *left,*right;
    Node(int x)
    {
        data = x;
        left = right = NULL;
    }
};

// This function prints all paths that have sum k
void printKPathUtil(Node *root, vector<int>& path,
                     int k)
{
    // empty node
    if (!root)
        return;

    // add current node to the path
    path.push_back(root->data);

    // If current node is leaf and sum of path is k
    if (path.size() == k)
        cout << path[0] << " ";
    else
        printKPathUtil(root->left, path, k);
        printKPathUtil(root->right, path, k);

    // Remove current node from path
    path.pop_back();
}
```

```
// check if there's any k sum path
// in the left sub-tree.
printKPathUtil(root->left, path, k);

// check if there's any k sum path
// in the right sub-tree.
printKPathUtil(root->right, path, k);

// check if there's any k sum path that
// terminates at this node
// Traverse the entire path as
// there can be negative elements too
int f = 0;
for (int j=path.size()-1; j>=0; j--)
{
    f += path[j];

    // If path sum is k, print the path
    if (f == k)
        printVector(path, j);
}

// Remove the current element from the path
path.pop_back();
}

// A wrapper over printKPathUtil()
void printKPath(Node *root, int k)
{
    vector<int> path;
    printKPathUtil(root, path, k);
}

// Driver code
int main()
{
    Node *root = new Node(1);
    root->left = new Node(3);
    root->left->left = new Node(2);
    root->left->right = new Node(1);
    root->left->right->left = new Node(1);
    root->right = new Node(-1);
    root->right->left = new Node(4);
    root->right->left->left = new Node(1);
    root->right->left->right = new Node(2);
    root->right->right = new Node(5);
    root->right->right->right = new Node(2);
```

```
int k = 5;
printKPath(root, k);

return 0;
}
```

Output:

```
3 2
3 1 1
1 3 1
4 1
1 -1 4 1
-1 4 2
5
1 -1 5
```

## Source

<https://www.geeksforgeeks.org/print-k-sum-paths-binary-tree/>

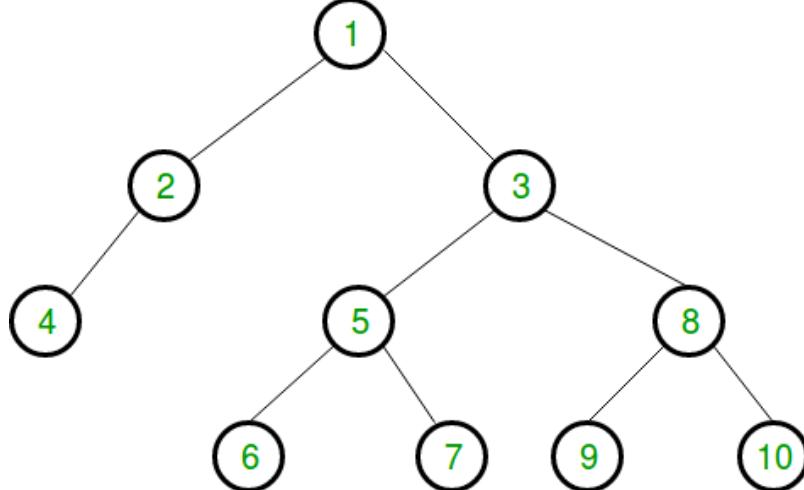
## Chapter 330

# Print all leaf nodes of a Binary Tree from left to right

Print all leaf nodes of a Binary Tree from left to right - GeeksforGeeks

Given a binary tree, we need to write a program to print all leaf nodes of the given binary tree from left to right. That is, the nodes should be printed in the order they appear from left to right in the given tree.

For Example,



For the above binary tree, output will be as shown below:

4 6 7 9 10

The idea to do this is similar to [DFS algorithm](#). Below is step by step algorithm to do this:

1. Check if given node is null. If null, then return from the function.

2. Check if it is a leaf node. If the node is a leaf node, then print its data.
3. If in above step, node is not a leaf node then check if left and right childs of node exists. If yes then call function for left and right childs of the node recursively.

Below is C++ implementation of above approach.

```
/* C++ program to print leaf nodes from left
   to right */
#include <iostream>
using namespace std;

// A Binary Tree Node
struct Node
{
    int data;
    struct Node *left, *right;
};

// function to print leaf
// nodes from left to right
void printLeafNodes(Node *root)
{
    // if node is null, return
    if (!root)
        return;

    // if node is leaf node, print its data
    if (!root->left && !root->right)
    {
        cout << root->data << " ";
        return;
    }

    // if left child exists, check for leaf
    // recursively
    if (root->left)
        printLeafNodes(root->left);

    // if right child exists, check for leaf
    // recursively
    if (root->right)
        printLeafNodes(root->right);
}

// Utility function to create a new tree node
Node* newNode(int data)
{
    Node *temp = new Node;
```

```
temp->data = data;
temp->left = temp->right = NULL;
return temp;
}

// Driver program to test above functions
int main()
{
    // Let us create binary tree shown in
    // above diagram
    Node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->right->left = newNode(5);
    root->right->right = newNode(8);
    root->right->left->left = newNode(6);
    root->right->left->right = newNode(7);
    root->right->right->left = newNode(9);
    root->right->right->right = newNode(10);

    // print leaf nodes of the given tree
    printLeafNodes(root);

    return 0;
}
```

Output:

4 6 7 9 10

**Time Complexity:**  $O(n)$ , where n is number of nodes in the binary tree.

## Source

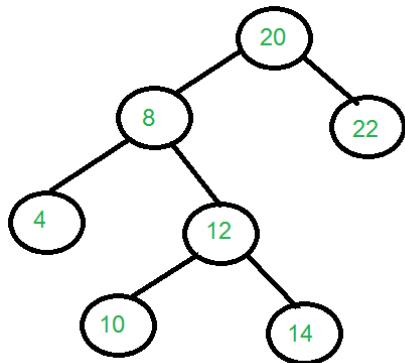
<https://www.geeksforgeeks.org/print-leaf-nodes-left-right-binary-tree/>

## Chapter 331

# Print all nodes at distance k from a given node

Print all nodes at distance k from a given node - GeeksforGeeks

Given a binary tree, a target node in the binary tree, and an integer value k, print all the nodes that are at distance k from the given target node. No parent pointers are available.



Consider the tree shown in diagram

Input: target = pointer to node with data 8.

root = pointer to node with data 20.

k = 2.

Output : 10 14 22

If target is 14 and k is 3, then output should be “4 20”

There are two types of nodes to be considered.

1) Nodes in the subtree rooted with target node. For example if the target node is 8 and k is 2, then such nodes are 10 and 14.

**2)** Other nodes, may be an ancestor of target, or a node in some other subtree. For target node 8 and k is 2, the node 22 comes in this category.

Finding the first type of nodes is easy to implement. Just traverse subtrees rooted with the target node and decrement k in recursive call. When the k becomes 0, print the node currently being traversed (See [this](#) for more details). Here we call the function as *printkdistanceNodeDown()*.

How to find nodes of second type? For the output nodes not lying in the subtree with the target node as the root, we must go through all ancestors. For every ancestor, we find its distance from target node, let the distance be d, now we go to other subtree (if target was found in left subtree, then we go to right subtree and vice versa) of the ancestor and find all nodes at k-d distance from the ancestor.

Following is the implementation of the above approach.

C++

```
#include <iostream>
using namespace std;

// A binary Tree node
struct node
{
    int data;
    struct node *left, *right;
};

/* Recursive function to print all the nodes at distance k in the
   tree (or subtree) rooted with given root. See  */
void printkdistanceNodeDown(node *root, int k)
{
    // Base Case
    if (root == NULL || k < 0)  return;

    // If we reach a k distant node, print it
    if (k==0)
    {
        cout << root->data << endl;
        return;
    }

    // Recur for left and right subtrees
    printkdistanceNodeDown(root->left, k-1);
    printkdistanceNodeDown(root->right, k-1);
}

// Prints all nodes at distance k from a given target node.
// The k distant nodes may be upward or downward. This function
// Returns distance of root from target node, it returns -1 if target
```

```
// node is not present in tree rooted with root.
int printkdistanceNode(node* root, node* target , int k)
{
    // Base Case 1: If tree is empty, return -1
    if (root == NULL) return -1;

    // If target is same as root. Use the downward function
    // to print all nodes at distance k in subtree rooted with
    // target or root
    if (root == target)
    {
        printkdistanceNodeDown(root, k);
        return 0;
    }

    // Recur for left subtree
    int dl = printkdistanceNode(root->left, target, k);

    // Check if target node was found in left subtree
    if (dl != -1)
    {
        // If root is at distance k from target, print root
        // Note that dl is Distance of root's left child from target
        if (dl + 1 == k)
            cout << root->data << endl;

        // Else go to right subtree and print all k-dl-2 distant nodes
        // Note that the right child is 2 edges away from left child
        else
            printkdistanceNodeDown(root->right, k-dl-2);

        // Add 1 to the distance and return value for parent calls
        return 1 + dl;
    }

    // MIRROR OF ABOVE CODE FOR RIGHT SUBTREE
    // Note that we reach here only when node was not found in left subtree
    int dr = printkdistanceNode(root->right, target, k);
    if (dr != -1)
    {
        if (dr + 1 == k)
            cout << root->data << endl;
        else
            printkdistanceNodeDown(root->left, k-dr-2);
        return 1 + dr;
    }

    // If target was neither present in left nor in right subtree
```

```
        return -1;
    }

// A utility function to create a new binary tree node
node *newnode(int data)
{
    node *temp = new node;
    temp->data = data;
    temp->left = temp->right = NULL;
    return temp;
}

// Driver program to test above functions
int main()
{
    /* Let us construct the tree shown in above diagram */
    node * root = newnode(20);
    root->left = newnode(8);
    root->right = newnode(22);
    root->left->left = newnode(4);
    root->left->right = newnode(12);
    root->left->right->left = newnode(10);
    root->left->right->right = newnode(14);
    node * target = root->left->right;
    printkdistanceNode(root, target, 2);
    return 0;
}
```

### Java

```
// Java program to print all nodes at a distance k from given node

// A binary tree node
class Node
{
    int data;
    Node left, right;

    Node(int item)
    {
        data = item;
        left = right = null;
    }
}

class BinaryTree
{
    Node root;
```

```
/* Recursive function to print all the nodes at distance k in
tree (or subtree) rooted with given root. */

void printkdistanceNodeDown(Node node, int k)
{
    // Base Case
    if (node == null || k < 0)
        return;

    // If we reach a k distant node, print it
    if (k == 0)
    {
        System.out.print(node.data);
        System.out.println("");
        return;
    }

    // Recur for left and right subtrees
    printkdistanceNodeDown(node.left, k - 1);
    printkdistanceNodeDown(node.right, k - 1);
}

// Prints all nodes at distance k from a given target node.
// The k distant nodes may be upward or downward. This function
// Returns distance of root from target node, it returns -1
// if target node is not present in tree rooted with root.
int printkdistanceNode(Node node, Node target, int k)
{
    // Base Case 1: If tree is empty, return -1
    if (node == null)
        return -1;

    // If target is same as root. Use the downward function
    // to print all nodes at distance k in subtree rooted with
    // target or root
    if (node == target)
    {
        printkdistanceNodeDown(node, k);
        return 0;
    }

    // Recur for left subtree
    int dl = printkdistanceNode(node.left, target, k);

    // Check if target node was found in left subtree
    if (dl != -1)
    {
```

```
// If root is at distance k from target, print root
// Note that dl is Distance of root's left child from
// target
if (dl + 1 == k)
{
    System.out.print(node.data);
    System.out.println("");
}

// Else go to right subtree and print all k-dl-2 distant nodes
// Note that the right child is 2 edges away from left child
else
    printkdistanceNodeDown(node.right, k - dl - 2);

// Add 1 to the distance and return value for parent calls
return 1 + dl;
}

// MIRROR OF ABOVE CODE FOR RIGHT SUBTREE
// Note that we reach here only when node was not found in left
// subtree
int dr = printkdistanceNode(node.right, target, k);
if (dr != -1)
{
    if (dr + 1 == k)
    {
        System.out.print(node.data);
        System.out.println("");
    }
    else
        printkdistanceNodeDown(node.left, k - dr - 2);
    return 1 + dr;
}

// If target was neither present in left nor in right subtree
return -1;
}

// Driver program to test the above functions
public static void main(String args[])
{
    BinaryTree tree = new BinaryTree();

    /* Let us construct the tree shown in above diagram */
    tree.root = new Node(20);
    tree.root.left = new Node(8);
    tree.root.right = new Node(22);
    tree.root.left.left = new Node(4);
```

```
tree.root.left.right = new Node(12);
tree.root.left.right.left = new Node(10);
tree.root.left.right.right = new Node(14);
Node target = tree.root.left.right;
tree.printkdistanceNode(tree.root, target, 2);
}
}

// This code has been contributed by Mayank Jaiswal
```

### Python

```
# Python program to print nodes at distance k from a given node

# A binary tree node
class Node:
    # A constructor to create a new node
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

    # Recursive function to print all the nodes at distance k
    # int the tree(or subtree) rooted with given root. See
    def printkDistanceNodeDown(root, k):

        # Base Case
        if root is None or k < 0 :
            return

        # If we reach a k distant node, print it
        if k == 0 :
            print root.data
            return

        # Recur for left and right subtree
        printkDistanceNodeDown(root.left, k-1)
        printkDistanceNodeDown(root.right, k-1)

    # Prints all nodes at distance k from a given target node
    # The k distant nodes may be upward or downward. This function
    # returns distance of root from target node, it returns -1
    # if target node is not present in tree rooted with root
    def printkDistanceNode(root, target, k):

        # Base Case 1 : IF tree is empty return -1
        if root is None:
```

```
        return -1

    # If target is same as root. Use the downward function
    # to print all nodes at distance k in subtree rooted with
    # target or root
    if root == target:
        printkDistanceNodeDown(root, k)
        return 0

    # Recur for left subtree
    dl = printkDistanceNode(root.left, target, k)

    # Check if target node was found in left subtree
    if dl != -1:

        # If root is at distance k from target, print root
        # Note: dl is distance of root's left child
        # from target
        if dl +1 == k :
            print root.data

        # Else go to right subtreee and print all k-dl-2
        # distant nodes
        # Note: that the right child is 2 edges away from
        # left chlid
        else:
            printkDistanceNodeDown(root.right, k-dl-2)

        # Add 1 to the distance and return value for
        # for parent calls
        return 1 + dl

    # MIRROR OF ABOVE CODE FOR RIGHT SUBTREE
    # Note that we reach here only when node was not found
    # in left subtree
    dr = printkDistanceNode(root.right, target, k)
    if dr != -1:
        if (dr+1 == k):
            print root.data
        else:
            printkDistanceNodeDown(root.left, k-dr-2)
    return 1 + dr

    # If target was neither present in left nor in right subtree
    return -1

# Driver program to test above function
root = Node(20)
```

```
root.left = Node(8)
root.right = Node(22)
root.left.left = Node(4)
root.left.right = Node(12)
root.left.right.left = Node(10)
root.left.right.right = Node(14)
target = root.left.right
printkDistanceNode(root, target, 2)

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

Output:

```
4
20
```

Time Complexity: At first look the time complexity looks more than  $O(n)$ , but if we take a closer look, we can observe that no node is traversed more than twice. Therefore the time complexity is  $O(n)$ .

This article is contributed by **Prasant Kumar**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

**Improved By :** [HarshitGargjpr](#), [zoozoo](#)

## Source

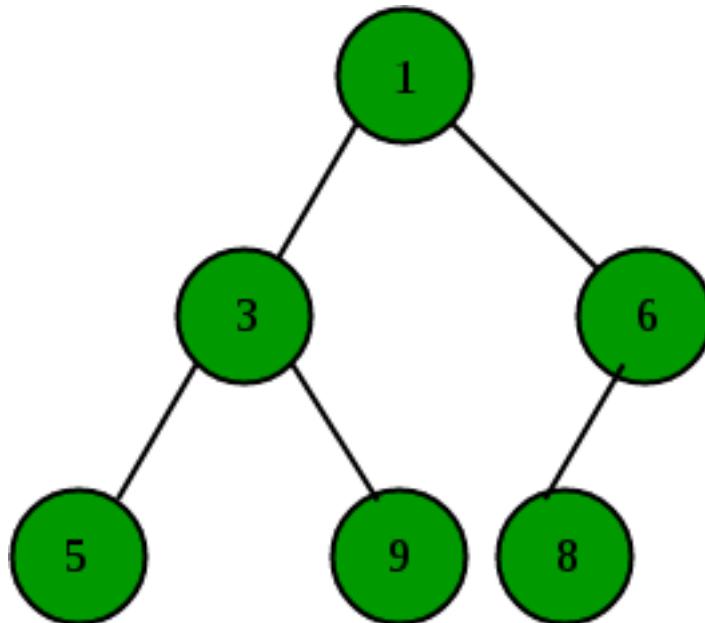
<https://www.geeksforgeeks.org/print-nodes-distance-k-given-node-binary-tree/>

## Chapter 332

# Print all nodes in a binary tree having K leaves

Print all nodes in a binary tree having K leaves - GeeksforGeeks

Given a binary tree and a integer value K, the task is to find all nodes in given binary tree having K leaves in subtree rooted with them.



Examples :

```
// For above binary tree
Input : k = 2
Output: {3}
```

```
// here node 3 have k = 2 leaves
```

```
Input : k = 1
```

```
Output: {6}
```

```
// here node 6 have k = 1 leave
```

Here any node having K leaves means sum of leaves in left subtree and in right subtree must be equal to K. So to solve this problem we use Postorder traversal of tree. First we calculate leaves in left subtree then in right subtree and if sum is equal to K, then print current node. In each recursive call we return sum of leaves of left subtree and right subtree to its ancestor.

```
// C++ program to count all nodes having k leaves
// in subtree rooted with them
#include<bits/stdc++.h>
using namespace std;

/* A binary tree node */
struct Node
{
    int data ;
    struct Node * left, * right ;
};

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct Node * newNode(int data)
{
    struct Node * node = new Node;
    node->data = data;
    node->left = node->right = NULL;
    return (node);
}

// Function to print all nodes having k leaves
int kLeaves(struct Node *ptr,int k)
{
    // Base Conditions : No leaves
    if (ptr == NULL)
        return 0;

    // if node is leaf
    if (ptr->left == NULL && ptr->right == NULL)
        return 1;

    // total leaves in subtree rooted with this
    // node
    int total = kLeaves(ptr->left, k) +
```

```
kLeaves(ptr->right, k);

// Print this node if total is k
if (k == total)
    cout << ptr->data << " ";

return total;
}

// Driver program to run the case
int main()
{
    struct Node *root = newNode(1);
    root->left      = newNode(2);
    root->right     = newNode(4);
    root->left->left  = newNode(5);
    root->left->right = newNode(6);
    root->left->left->left = newNode(9);
    root->left->left->right = newNode(10);
    root->right->right = newNode(8);
    root->right->left   = newNode(7);
    root->right->left->left = newNode(11);
    root->right->left->right = newNode(12);

    kLeaves(root, 2);

    return 0;
}
```

Output:

5 7

Time complexity : O(n)

## Source

<https://www.geeksforgeeks.org/print-nodes-binary-tree-k-leaves/>

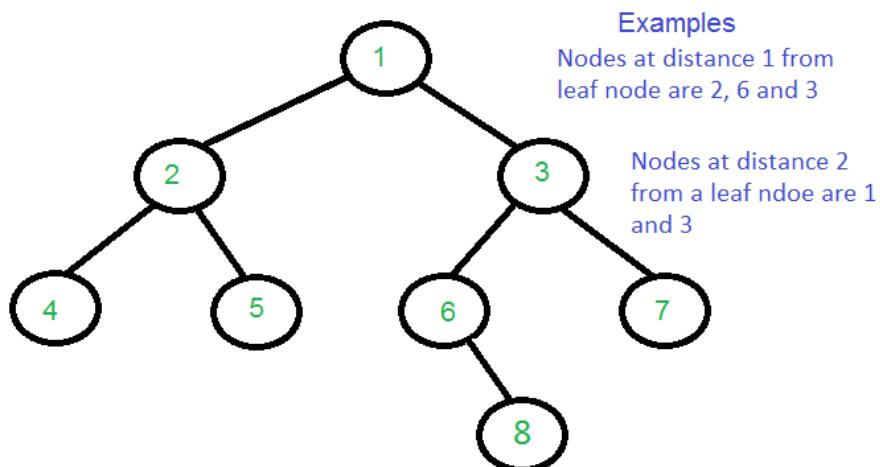
## Chapter 333

# Print all nodes that are at distance k from a leaf node

Print all nodes that are at distance k from a leaf node - GeeksforGeeks

Given a Binary Tree and a positive integer k, print all nodes that are distance k from a leaf node.

Here the meaning of distance is different from [previous post](#). Here k distance from a leaf means k levels higher than a leaf node. For example if k is more than height of Binary Tree, then nothing should be printed. Expected time complexity is O(n) where n is the number nodes in the given Binary Tree.



The idea is to traverse the tree. Keep storing all ancestors till we hit a leaf node. When we reach a leaf node, we print the ancestor at distance k. We also need to keep track of nodes that are already printed as output. For that we use a boolean array `visited[]`.

C++

```
/* Program to print all nodes which are at distance k from a leaf */
#include <iostream>
using namespace std;
#define MAX_HEIGHT 10000

struct Node
{
    int key;
    Node *left, *right;
};

/* utility that allocates a new Node with the given key */
Node* newNode(int key)
{
    Node* node = new Node;
    node->key = key;
    node->left = node->right = NULL;
    return (node);
}

/* This function prints all nodes that are distance k from a leaf node
   path[] --> Store ancestors of a node
   visited[] --> Stores true if a node is printed as output. A node may be k
                  distance away from many leaves, we want to print it once */
void kDistantFromLeafUtil(Node* node, int path[], bool visited[],
                           int pathLen, int k)
{
    // Base case
    if (node==NULL) return;

    /* append this Node to the path array */
    path[pathLen] = node->key;
    visited[pathLen] = false;
    pathLen++;

    /* it's a leaf, so print the ancestor at distance k only
       if the ancestor is not already printed */
    if (node->left == NULL && node->right == NULL &&
        pathLen-k-1 >= 0 && visited[pathLen-k-1] == false)
    {
        cout << path[pathLen-k-1] << " ";
        visited[pathLen-k-1] = true;
        return;
    }

    /* If not leaf node, recur for left and right subtrees */
    kDistantFromLeafUtil(node->left, path, visited, pathLen, k);
    kDistantFromLeafUtil(node->right, path, visited, pathLen, k);
}
```

```
}

/* Given a binary tree and a number k, print all nodes that are k
   distant from a leaf*/
void printKDistantfromLeaf(Node* node, int k)
{
    int path[MAX_HEIGHT];
    bool visited[MAX_HEIGHT] = {false};
    kDistantFromLeafUtil(node, path, visited, 0, k);
}

/* Driver program to test above functions*/
int main()
{
    // Let us create binary tree given in the above example
    Node * root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->left = newNode(6);
    root->right->right = newNode(7);
    root->right->left->right = newNode(8);

    cout << "Nodes at distance 2 are: ";
    printKDistantfromLeaf(root, 2);

    return 0;
}
```

### Java

```
// Java program to print all nodes at a distance k from leaf
// A binary tree node
class Node
{
    int data;
    Node left, right;

    Node(int item)
    {
        data = item;
        left = right = null;
    }
}

class BinaryTree
{
```

```
Node root;

/* This function prints all nodes that are distance k from a leaf node
path[] --> Store ancestors of a node
visited[] --> Stores true if a node is printed as output. A node may
be k distance away from many leaves, we want to print it once */
void kDistantFromLeafUtil(Node node, int path[], boolean visited[],
                           int pathLen, int k)
{
    // Base case
    if (node == null)
        return;

    /* append this Node to the path array */
    path[pathLen] = node.data;
    visited[pathLen] = false;
    pathLen++;

    /* it's a leaf, so print the ancestor at distance k only
       if the ancestor is not already printed */
    if (node.left == null && node.right == null
        && pathLen - k - 1 >= 0 && visited[pathLen - k - 1] == false)
    {
        System.out.print(path[pathLen - k - 1] + " ");
        visited[pathLen - k - 1] = true;
        return;
    }

    /* If not leaf node, recur for left and right subtrees */
    kDistantFromLeafUtil(node.left, path, visited, pathLen, k);
    kDistantFromLeafUtil(node.right, path, visited, pathLen, k);
}

/* Given a binary tree and a number k, print all nodes that are k
distant from a leaf*/
void printKDistantfromLeaf(Node node, int k)
{
    int path[] = new int[1000];
    boolean visited[] = new boolean[1000];
    kDistantFromLeafUtil(node, path, visited, 0, k);
}

// Driver program to test the above functions
public static void main(String args[])
{
    BinaryTree tree = new BinaryTree();

    /* Let us construct the tree shown in above diagram */
}
```

```
tree.root = new Node(1);
tree.root.left = new Node(2);
tree.root.right = new Node(3);
tree.root.left.left = new Node(4);
tree.root.left.right = new Node(5);
tree.root.right.left = new Node(6);
tree.root.right.right = new Node(7);
tree.root.right.left.right = new Node(8);

System.out.println(" Nodes at distance 2 are :");
tree.printKDistantfromLeaf(tree.root, 2);
}

}

// This code has been contributed by Mayank Jaiswal
```

Output:

```
Nodes at distance 2 are: 3 1
```

Time Complexity: Time Complexity of above code is  $O(n)$  as the code does a simple tree traversal.

## Source

<https://www.geeksforgeeks.org/print-nodes-distance-k-leaf-node/>

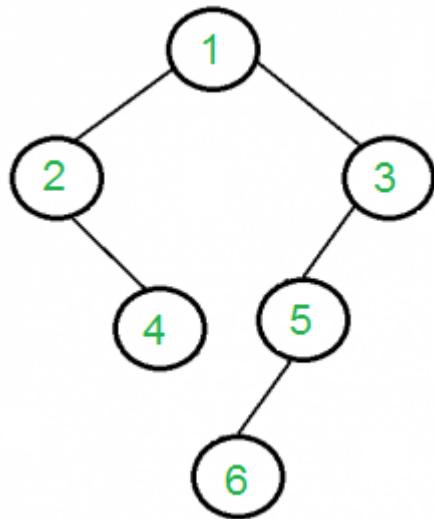
## Chapter 334

# Print all nodes that don't have sibling

Print all nodes that don't have sibling - GeeksforGeeks

Given a Binary Tree, print all nodes that don't have a sibling (a sibling is a node that has same parent. In a Binary Tree, there can be at most one sibling). Root should not be printed as root cannot have a sibling.

For example, the output should be “4 5 6” for the following tree.



This is a typical tree traversal question. We start from root and check if the node has one child, if yes then print the only child of that node. If node has both children, then recur for both the children.

C++

```
/* Program to find singles in a given binary tree */
#include <iostream>
using namespace std;

// A Binary Tree Node
struct node
{
    struct node *left, *right;
    int key;
};

// Utility function to create a new tree node
node* newNode(int key)
{
    node *temp = new node;
    temp->key = key;
    temp->left = temp->right = NULL;
    return temp;
}

// Function to print all non-root nodes that don't have a sibling
void printSingles(struct node *root)
{
    // Base case
    if (root == NULL)
        return;

    // If this is an internal node, recur for left
    // and right subtrees
    if (root->left != NULL && root->right != NULL)
    {
        printSingles(root->left);
        printSingles(root->right);
    }

    // If left child is NULL and right is not, print right child
    // and recur for right child
    else if (root->right != NULL)
    {
        cout << root->right->key << " ";
        printSingles(root->right);
    }

    // If right child is NULL and left is not, print left child
    // and recur for left child
    else if (root->left != NULL)
    {
        cout << root->left->key << " ";
```

```
        printSingles(root->left);
    }
}

// Driver program to test above functions
int main()
{
    // Let us create binary tree given in the above example
    node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->right = newNode(4);
    root->right->left = newNode(5);
    root->right->left->left = newNode(6);
    printSingles(root);
    return 0;
}
```

### Java

```
// Java program to print all nodes that don't have sibling

// A binary tree node
class Node
{
    int data;
    Node left, right;

    Node(int item)
    {
        data = item;
        left = right = null;
    }
}

class BinaryTree
{
    Node root;

    // Function to print all non-root nodes that don't have a sibling
    void printSingles(Node node)
    {
        // Base case
        if (node == null)
            return;

        // If this is an internal node, recur for left
        // and right subtrees
        printSingles(node->left);
        printSingles(node->right);
    }
}
```

```
if (node.left != null && node.right != null)
{
    printSingles(node.left);
    printSingles(node.right);
}

// If left child is NULL and right is not, print right child
// and recur for right child
else if (node.right != null)
{
    System.out.print(node.right.data + " ");
    printSingles(node.right);
}

// If right child is NULL and left is not, print left child
// and recur for left child
else if (node.left != null)
{
    System.out.print( node.left.data + " ");
    printSingles(node.left);
}

// Driver program to test the above functions
public static void main(String args[])
{
    BinaryTree tree = new BinaryTree();

    /* Let us construct the tree shown in above diagram */
    tree.root = new Node(1);
    tree.root.left = new Node(2);
    tree.root.right = new Node(3);
    tree.root.left.right = new Node(4);
    tree.root.right.left = new Node(5);
    tree.root.right.left.right = new Node(6);
    tree.printSingles(tree.root);
}

}

// This code has been contributed by Mayank Jaiswal
```

### Python

```
# Program to find singles in a given binary tree

# A Binary Tree Node
class Node:

    # A constructor to create new tree node
```

```
def __init__(self, key):
    self.key = key
    self.left = None
    self.right = None

# Function to print all non-root nodes that don't have
# a sibling
def printSingles(root):

    # Base Case
    if root is None:
        return

    # If this is an internal node , recur for left
    # and right subtrees
    if root.left is not None and root.right is not None:
        printSingles(root.left)
        printSingles(root.right)

    # If left child is NULL, and right is not, print
    # right child and recur for right child
    elif root.right is not None:
        print root.right.key,
        printSingles(root.right)

    # If right child is NULL and left is not, print
    # left child and recur for left child
    elif root.left is not None:
        print root.left.key,
        printSingles(root.left)

# Driver program to test above function
root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.right = Node(4)
root.right.left = Node(5)
root.right.left.left = Node(6)
printSingles(root)

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

Output:

4 5 6

Time Complexity of above code is O(n) as the code does a simple tree traversal.

This article is compiled by **Aman Gupta**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## Source

<https://www.geeksforgeeks.org/print-nodes-dont-sibling-binary-tree/>

## Chapter 335

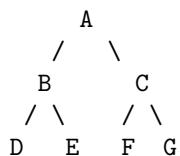
# Print all root to leaf paths with their relative positions

Print all root to leaf paths with their relative positions - GeeksforGeeks

Given a binary tree, print the root to the leaf path, but add “\_” to indicate the relative position.

Example:

Input : Root of below tree



Output : All root to leaf paths

```
_ _ A  
_ B  
D
```

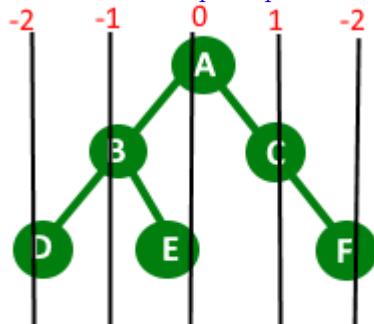
```
_ A  
B  
_ E
```

```
A  
_ C  
F
```

```
A  
_ C  
_ _ G
```

Asked In: **Google Interview**

The idea base on [print path in vertical order](#).

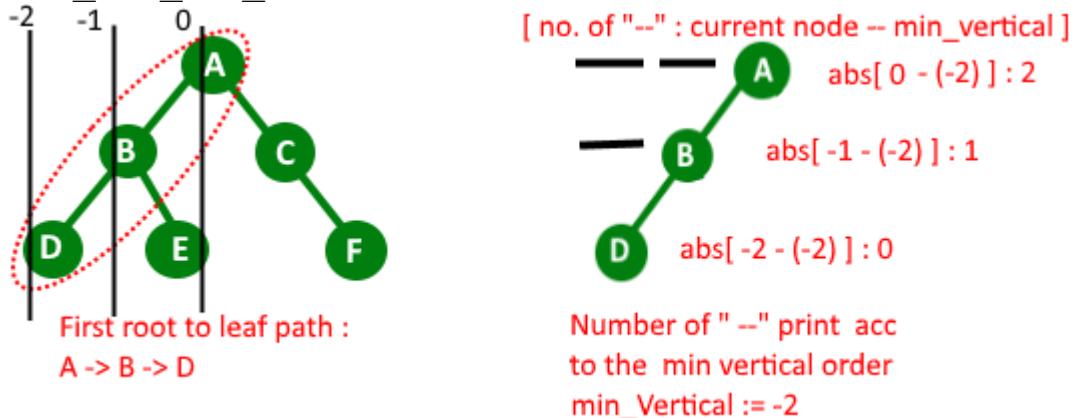


**Vertical Order**

Below is complete algorithm :

- 1) We do Preorder traversal of the given Binary Tree. While traversing the tree, we can recursively calculate [horizontal distances or HDs](#). We initially pass the horizontal distance as 0 for root. For left subtree, we pass the Horizontal Distance as Horizontal distance of root minus 1. For right subtree, we pass the Horizontal Distance as Horizontal Distance of root plus 1. For every HD value, we maintain a list of nodes in a vector ("") that will store information of current node horizontal distance and key value of root "").we also maintain the order of node (order in which they appear in path from root to leaf). for maintaining the order,here we used vector.
- 2) While we reach to leaf node during traverse we print that path with underscore “\_”

**Print\_Path\_with\_underscore function**



- .....a) First find the minimum Horizontal distance of the current path.
- .....b) After that we traverse current path
- .....First Print number of underscore “\_” : abs (current\_node\_HD – minimum-HD)
- .....Print current node value.

We do this process for all root to leaf path

Bellow is C++ implementations of above idea.

```
// C++ program to print all root to leaf paths
// with there relative position
#include<bits/stdc++.h>
using namespace std;

#define MAX_PATH_SIZE 1000

// tree structure
struct Node
{
    char data;
    Node *left, *right;
};

// function create new node
Node * newNode(char data)
{
    struct Node *temp = new Node;
    temp->data = data;
    temp->left = temp->right = NULL;
    return temp;
}

// store path information
struct PATH
{
    int Hd; // horigontal distance of node from root.
    char key; // store key
};

// Prints given root to leaf path with underscores
void printPath(vector < PATH > path, int size)
{
    // Find the minimum horizontal distance value
    // in current root to leaf path
    int minimum_Hd = INT_MAX;

    PATH p;

    // find minimum horizontal distance
    for (int it=0; it<size; it++)
    {
        p = path[it];
        minimum_Hd = min(minimum_Hd, p.Hd);
    }

    // print the root to leaf path with "_"
    // that indicate the related position
```

```
for (int it=0; it < size; it++)
{
    // current tree node
    p = path[it];
    int noOfUnderScores = abs(p.Hd - minimum_Hd);

    // print underscore
    for (int i = 0; i < noOfUnderScores; i++)
        cout << "_" ;

    // print current key
    cout << p.key << endl;
}
cout << "======" << endl;

// a utility function print all path from root to leaf
// working of this function is similar to function of
// "Print_vertical_order" : Print paths of binary tree
// in vertical order
// https://www.geeksforgeeks.org/print-binary-tree-vertical-order-set-2/
void printAllPathsUtil(Node *root,
                      vector < PATH > &AllPath,
                      int HD, int order )
{
    // base case
    if(root == NULL)
        return;

    // leaf node
    if (root->left == NULL && root->right == NULL)
    {
        // add leaf node and then print path
        AllPath[order] = (PATH { HD, root->data });
        printPath(AllPath, order+1);
        return;
    }

    // store current path information
    AllPath[order] = (PATH { HD, root->data });

    // call left sub_tree
    printAllPathsUtil(root->left, AllPath, HD-1, order+1);

    //call left sub_tree
    printAllPathsUtil(root->right, AllPath, HD+1, order+1);
}
```

```
void printAllPaths(Node *root)
{
    // base case
    if (root == NULL)
        return;

    vector<PATH> Allpaths(MAX_PATH_SIZE);
    printAllPathsUtil(root, Allpaths, 0, 0);
}

// Driver program to test above function
int main()
{
    Node *root = newNode('A');
    root->left = newNode('B');
    root->right = newNode('C');
    root->left->left = newNode('D');
    root->left->right = newNode('E');
    root->right->left = newNode('F');
    root->right->right = newNode('G');
    printAllPaths(root);
    return 0;
}
```

Output:

```
      A
     _ B
    D
=====
      A
     _ B
    _ E
=====
      A
     _ C
    F
=====
      A
     _ C
    _ _ G
```

## Source

<https://www.geeksforgeeks.org/print-all-root-to-leaf-paths-with-there-relative-positions/>

## Chapter 336

# Print all the paths from root, with a specified sum in Binary tree

Print all the paths from root, with a specified sum in Binary tree - GeeksforGeeks

Given a Binary tree and a sum  $S$ , print all the paths, starting from root, that sums upto the given sum.

Note that this problem is different from [root to leaf paths](#). Here path doesn't need to end on a leaf node.

Examples:

```
Input :  
Input : sum = 8,  
        Root of tree  
          1  
         /   \  
       20     3  
         /   \  
       4     15  
      / \   / \  
    6   7   8   9
```

```
Output :  
Path: 1 3 4
```

```
Input : sum = 38,  
        Root of tree  
          10  
         /   \
```

```
28      13
      /   \
    14     15
   / \   / \
  21 22 23 24
Output : Path found: 10 28
          Path found: 10 13 15
```

For this problem, preorder traversal is best suited as we have to add up a key value as we land on that node.

We start from the root and start traversing by preorder traversal, adding key value to the *sum\_so\_far* and checking whether it is equal to the required sum.

Also, as tree node doesn't have a pointer pointing to its parent, we have to explicitly save from where we have moved. We use a vector *path* to store the path for this.

Every node in this path contributes to the *sum\_so\_far*.

```
// C++ program to print all paths beginning with
// root and sum as given sum
#include<bits/stdc++.h>
using namespace std;

// A Tree node
struct Node
{
    int key;
    struct Node *left, *right;
};

// Utility function to create a new node
Node* newNode(int key)
{
    Node* temp = new Node;
    temp->key = key;
    temp->left = temp->right = NULL;
    return (temp);
}

void printPathsUtil(Node* curr_node, int sum,
                    int sum_so_far, vector<int> &path)
{
    if (curr_node == NULL)
        return;

    // add the current node's value
    sum_so_far += curr_node->key;
```

```
// add the value to path
path.push_back(curr_node->key);

// print the required path
if (sum_so_far == sum )
{
    cout << "Path found: ";
    for (int i=0; i<path.size(); i++)
        cout << path[i] << " ";

    cout << endl;
}

// if left child exists
if (curr_node->left != NULL)
    printPathsUtil(curr_node->left, sum, sum_so_far, path);

// if right child exists
if (curr_node->right != NULL)
    printPathsUtil(curr_node->right, sum, sum_so_far, path);

// Remove last element from path
// and move back to parent
path.pop_back();
}

// Wrapper over printPathsUtil
void printPaths(Node *root, int sum)
{
    vector<int> path;
    printPathsUtil(root, sum, 0, path);
}

// Driver program
int main ()
{
    /*      10
           /   \
          28   13
             /   \
            14   15
           / \   / \
          21 22 23 24*/
    Node *root = newNode(10);
    root->left = newNode(28);
    root->right = newNode(13);
```

```
root->right->left    = newNode(14);
root->right->right   = newNode(15);

root->right->left->left   = newNode(21);
root->right->left->right = newNode(22);
root->right->right->left = newNode(23);
root->right->right->right = newNode(24);

int sum = 38;

printPaths(root, sum);

return 0;
}
```

Output:

```
Path found: 10 28
Path found: 10 13 15
```

## Source

<https://www.geeksforgeeks.org/print-paths-root-specified-sum-binary-tree/>

## Chapter 337

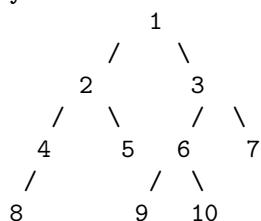
# Print common nodes on path from root (or common ancestors)

Print common nodes on path from root (or common ancestors) - GeeksforGeeks

Given a binary tree and two nodes, the task is to Print all the nodes that are common for 2 given nodes in a binary tree.

Examples:

Given binary tree is :



Given nodes 9 and 7, so the common nodes are:-  
1, 3

Asked in : [Amazon](#)

1. Find the [LCA](#) of given two nodes.
2. Print all ancestors of the LCA as done in this [post](#), also print the LCA.

[C/C++](#)

```
// C++ Program to find common nodes for given two nodes
#include <iostream>
using namespace std;

// A Binary Tree Node
struct Node {
    struct Node* left, *right;
    int key;
};

// Utility function to create a new tree Node
Node* newNode(int key)
{
    Node* temp = new Node;
    temp->key = key;
    temp->left = temp->right = NULL;
    return temp;
}

// Utility function to find the LCA of two given values
// n1 and n2.
struct Node* findLCA(struct Node* root, int n1, int n2)
{
    // Base case
    if (root == NULL)
        return NULL;

    // If either n1 or n2 matches with root's key,
    // report the presence by returning root (Note
    // that if a key is ancestor of other, then the
    // ancestor key becomes LCA
    if (root->key == n1 || root->key == n2)
        return root;

    // Look for keys in left and right subtrees
    Node* left_lca = findLCA(root->left, n1, n2);
    Node* right_lca = findLCA(root->right, n1, n2);

    // If both of the above calls return Non-NULL, then
    // one key is present in once subtree and other is
    // present in other, So this node is the LCA
    if (left_lca && right_lca)
        return root;

    // Otherwise check if left subtree or right
    // subtree is LCA
    return (left_lca != NULL) ? left_lca : right_lca;
}
```

```
// Utility Function to print all ancestors of LCA
bool printAncestors(struct Node* root, int target)
{
    /* base cases */
    if (root == NULL)
        return false;

    if (root->key == target) {
        cout << root->key << " ";
        return true;
    }

    /* If target is present in either left or right
       subtree of this node, then print this node */
    if (printAncestors(root->left, target) ||
        printAncestors(root->right, target)) {
        cout << root->key << " ";
        return true;
    }

    /* Else return false */
    return false;
}

// Function to find nodes common to given two nodes
bool findCommonNodes(struct Node* root, int first,
                     int second)
{
    struct Node* LCA = findLCA(root, first, second);
    if (LCA == NULL)
        return false;

    printAncestors(root, LCA->key);
}

// Driver program to test above functions
int main()
{
    // Let us create binary tree given in the above
    // example
    Node* root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->left = newNode(6);
    root->right->right = newNode(7);
```

```
root->left->left->left = newNode(8);
root->right->left->left = newNode(9);
root->right->left->right = newNode(10);

if (findCommonNodes(root, 9, 7) == false)
    cout << "No Common nodes";

return 0;
}
```

**Java**

```
// Java Program to find common nodes for given two nodes
import java.util.LinkedList;

// Class to represent Tree node
class Node
{
    int data;
    Node left, right;

    public Node(int item)
    {
        data = item;
        left = null;
        right = null;
    }
}

// Class to count full nodes of Tree
class BinaryTree
{
    static Node root;
    // Utility function to find the LCA of two given values
    // n1 and n2.
    static Node findLCA(Node root, int n1, int n2)
    {
        // Base case
        if (root == null)
            return null;

        // If either n1 or n2 matches with root's key,
        // report the presence by returning root (Note
        // that if a key is ancestor of other, then the
        // ancestor key becomes LCA
        if (root.data == n1 || root.data == n2)
            return root;
```

```
// Look for keys in left and right subtrees
Node left_lca = findLCA(root.left, n1, n2);
Node right_lca = findLCA(root.right, n1, n2);

// If both of the above calls return Non-NULL, then
// one key is present in once subtree and other is
// present in other, So this node is the LCA
if (left_lca!=null && right_lca!=null)
    return root;

// Otherwise check if left subtree or right
// subtree is LCA
return (left_lca != null) ? left_lca : right_lca;
}

// Utility Function to print all ancestors of LCA
static boolean printAncestors(Node root, int target)
{
    /* base cases */
    if (root == null)
        return false;

    if (root.data == target) {
        System.out.print(root.data+ " ");
        return true;
    }

    /* If target is present in either left or right
    subtree of this node, then print this node */
    if (printAncestors(root.left, target) ||
        printAncestors(root.right, target)) {
        System.out.print(root.data+ " ");
        return true;
    }

    /* Else return false */
    return false;
}

// Function to find nodes common to given two nodes
static boolean findCommonNodes(Node root, int first,
                               int second)
{
    Node LCA = findLCA(root, first, second);
    if (LCA == null)
        return false;

    printAncestors(root, LCA.data);
```

```
    return true;
}

// Driver program to test above functions
public static void main(String args[])
{
/*Let us create Binary Tree shown in
above example */

BinaryTree tree = new BinaryTree();
tree.root = new Node(1);
tree.root.left = new Node(2);
tree.root.right = new Node(3);
tree.root.left.left = new Node(4);
tree.root.left.right = new Node(5);
tree.root.right.left = new Node(6);
tree.root.right.right = new Node(7);
tree.root.left.left.left = new Node(8);
tree.root.right.left.left = new Node(9);
tree.root.right.left.right = new Node(10);

if (findCommonNodes(root, 9, 7) == false)
System.out.println("No Common nodes");

}
}

// This code is contributed by Mr Somesh Awasthi
```

Output:

3 1

## Source

<https://www.geeksforgeeks.org/print-common-nodes-path-root-common-ancestors/>

## Chapter 338

# Print cousins of a given node in Binary Tree

Print cousins of a given node in Binary Tree - GeeksforGeeks

Given a binary tree and a node, print all cousins of given node. Note that siblings should not be printed.

Example:

```
Input : root of below tree
        1
       /   \
      2     3
     / \   / \
    4   5   6   7
and pointer to a node say 5.
```

```
Output : 6, 7
```

The idea to first find level of given node using the approach discussed [here](#). Once we have found level, we can print all nodes at a given level using the approach discussed [here](#). The only thing to take care of is, sibling should not be printed. To handle this, we change the printing function to first check for sibling and print node only if it is not sibling.

Below is C++ implementation of above idea.

```
// C program to print cousins of a node
#include <stdio.h>
#include <stdlib.h>

// A Binary Tree Node
```

```
struct Node
{
    int data;
    Node *left, *right;
};

// A utility function to create a new Binary
// Tree Node
Node *newNode(int item)
{
    Node *temp = new Node;
    temp->data = item;
    temp->left = temp->right = NULL;
    return temp;
}

/* It returns level of the node if it is present
   in tree, otherwise returns 0.*/
int getLevel(Node *root, Node *node, int level)
{
    // base cases
    if (root == NULL)
        return 0;
    if (root == node)
        return level;

    // If node is present in left subtree
    int downlevel = getLevel(root->left, node, level+1);
    if (downlevel != 0)
        return downlevel;

    // If node is not present in left subtree
    return getLevel(root->right, node, level+1);
}

/* Print nodes at a given level such that sibling of
   node is not printed if it exists */
void printGivenLevel(Node* root, Node *node, int level)
{
    // Base cases
    if (root == NULL || level < 2)
        return;

    // If current node is parent of a node with
    // given level
    if (level == 2)
    {
        if (root->left == node || root->right == node)
```

```
        return;
    if (root->left)
        printf("%d ", root->left->data);
    if (root->right)
        printf("%d ", root->right->data);
}

// Recur for left and right subtrees
else if (level > 2)
{
    printGivenLevel(root->left, node, level-1);
    printGivenLevel(root->right, node, level-1);
}
}

// This function prints cousins of a given node
void printCousins(Node *root, Node *node)
{
    // Get level of given node
    int level = getLevel(root, node, 1);

    // Print nodes of given level.
    printGivenLevel(root, node, level);
}

// Driver Program to test above functions
int main()
{
    Node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->left->right->right = newNode(15);
    root->right->left = newNode(6);
    root->right->right = newNode(7);
    root->right->left->right = newNode(8);

    printCousins(root, root->left->right);

    return 0;
}
```

Output :

6 7

Time Complexity : O(n)

Can we solve this problem using single traversal? Please refer below article

[Print cousins of a given node in Binary Tree | Single Traversal](#)

### Source

<https://www.geeksforgeeks.org/print-cousins-of-a-given-node-in-binary-tree/>

## Chapter 339

# Print cousins of a given node in Binary Tree | Single Traversal

Print cousins of a given node in Binary Tree | Single Traversal - GeeksforGeeks

Given a binary tree and a node, print all cousins of given node. Note that siblings should not be printed.

**Examples:**

```
Input : root of below tree
        1
       /   \
      2     3
     / \   / \
    4   5   6   7
and pointer to a node say 5.
```

```
Output : 6, 7
```

Note that it is the same problem as given at [Print cousins of a given node in Binary Tree](#) which consists of two traversals recursively. In this post, a single level traversal approach is discussed.

The idea is to go for level order traversal of the tree, as the cousins and siblings of a node can be found in its level order traversal. Run the traversal till the level containing the node is not found, and if found, print the given level.

**How to print the cousin nodes instead of siblings and how to get the nodes of that level in the queue?** During the level order, when for the parent node, if parent->left == Node\_to\_find, or parent->right == Node\_to\_find, then the children of this parent must not be pushed into the queue (as one is the node and other will be its sibling). Push the remaining nodes at the same level in the queue and then exit the loop. The current

queue will have the nodes at the next level (the level of the node being searched, except the node and its sibling). Now, print the queue.

Following is the implementation of the above algorithm.

C++

```
// CPP program to print cousins of a node
#include <iostream>
#include <queue>
using namespace std;

// A Binary Tree Node
struct Node {
    int data;
    Node *left, *right;
};

// A utility function to create a new Binary
// Tree Node
Node* newNode(int item)
{
    Node* temp = new Node;
    temp->data = item;
    temp->left = temp->right = NULL;
    return temp;
}

// function to print cousins of the node
void printCousins(Node* root, Node* node_to_find)
{
    // if the given node is the root itself,
    // then no nodes would be printed
    if (root == node_to_find) {
        cout << "Cousin Nodes : None" << endl;
        return;
    }

    queue<Node*> q;
    bool found = false;
    int size_;
    Node* p;
    q.push(root);

    // the following loop runs until found is
    // not true, or q is not empty.
    // if found has become true => we have found
    // the level in which the node is present
    // and the present queue will contain all the
```

```
// cousins of that node
while (!q.empty() && !found) {

    size_ = q.size();
    while (size_) {
        p = q.front();
        q.pop();

        // if current node's left or right child
        // is the same as the node to find,
        // then make found = true, and don't push
        // any of them into the queue, as
        // we don't have to print the siblings
        if ((p->left == node_to_find ||
            p->right == node_to_find)) {
            found = true;
        }
        else {
            if (p->left)
                q.push(p->left);
            if (p->right)
                q.push(p->right);
        }
        size_--;
    }
}

// if found == true then the queue will contain the
// cousins of the given node
if (found) {
    cout << "Cousin Nodes : ";
    size_ = q.size();

    // size_ will be 0 when the node was at the
    // level just below the root node.
    if (size_ == 0)
        cout << "None";
    for (int i = 0; i < size_; i++) {
        p = q.front();
        q.pop();
        cout << p->data << " ";
    }
}
else {
    cout << "Node not found";
}
cout << endl;
```

```
    return;
}

// Driver Program to test above function
int main()
{
    Node* root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->left->right->right = newNode(15);
    root->right->left = newNode(6);
    root->right->right = newNode(7);
    root->right->left->right = newNode(8);

    Node* x = newNode(43);

    printCousins(root, x);
    printCousins(root, root);
    printCousins(root, root->right);
    printCousins(root, root->left);
    printCousins(root, root->left->right);

    return 0;
}
```

**Java**

```
// Java program to print
// cousins of a node
import java.io.*;
import java.util.*;
import java.lang.*;

// A Binary Tree Node
class Node
{
    int data;
    Node left, right;
    Node(int key)
    {
        data = key;
        left = right = null;
    }
}
```

```
class GFG
{
    // function to print
    // cousins of the node
    static void printCousins(Node root,
                             Node node_to_find)
    {
        // if the given node
        // is the root itself,
        // then no nodes would
        // be printed
        if (root == node_to_find)
        {
            System.out.print("Cousin Nodes : " +
                            " None" + "\n");
            return;
        }

        Queue<Node> q = new LinkedList<Node>();
        boolean found = false;
        int size_ = 0;
        Node p = null;
        q.add(root);

        // the following loop runs
        // until found is not true,
        // or q is not empty. if
        // found has become true => we
        // have found the level in
        // which the node is present
        // and the present queue will
        // contain all the cousins of
        // that node
        while (q.isEmpty() == false &&
               found == false)
        {

            size_ = q.size();
            while (size_ -- > 0)
            {
                p = q.peek();
                q.remove();

                // if current node's left
                // or right child is the
                // same as the node to find,
                // then make found = true,
            }
        }
    }
}
```

```
// and don't push any of them
// into the queue, as we don't
// have to print the siblings
if ((p.left == node_to_find ||
     p.right == node_to_find))
{
    found = true;
}
else
{
    if (p.left != null)
        q.add(p.left);
    if (p.right!= null)
        q.add(p.right);
}

}

// if found == true then the
// queue will contain the
// cousins of the given node
if (found == true)
{
    System.out.print("Cousin Nodes : ");
    size_ = q.size();

    // size_ will be 0 when
    // the node was at the
    // level just below the
    // root node.
    if (size_ == 0)
        System.out.print("None");

    for (int i = 0; i < size_; i++)
    {
        p = q.peek();
        q.poll();

        System.out.print(p.data + " ");
    }
}
else
{
    System.out.print("Node not found");
}

System.out.println("");
```

```
    return;
}

// Driver code
public static void main(String[] args)
{
    Node root = new Node(1);
    root.left = new Node(2);
    root.right = new Node(3);
    root.left.left = new Node(4);
    root.left.right = new Node(5);
    root.left.right.right = new Node(15);
    root.right.left = new Node(6);
    root.right.right = new Node(7);
    root.right.left.right = new Node(8);

    Node x = new Node(43);

    printCousins(root, x);
    printCousins(root, root);
    printCousins(root, root.right);
    printCousins(root, root.left);
    printCousins(root, root.left.right);
}
}
```

**Output:**

```
Node not found
Cousin Nodes : None
Cousin Nodes : None
Cousin Nodes : None
Cousin Nodes : 6 7
```

Time Complexity : This is a single level order traversal, hence time complexity =  $O(n)$ , and Auxiliary space =  $O(n)$  (See [this](#)).

**Source**

<https://www.geeksforgeeks.org/print-cousins-of-a-given-node-in-binary-tree-single-traversal/>

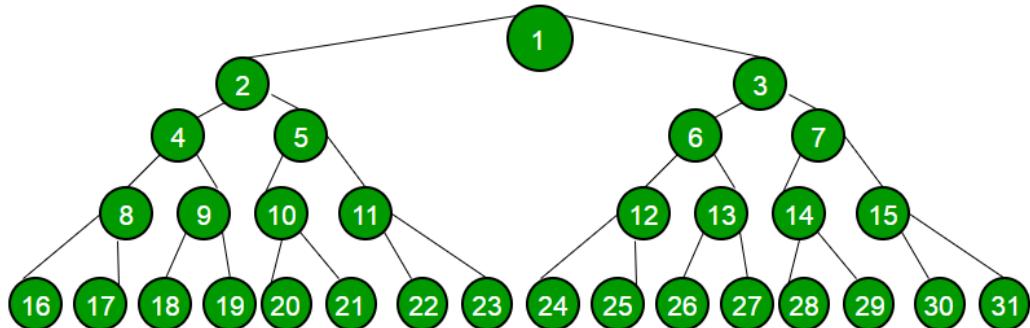
## Chapter 340

**Print extreme nodes of each level of Binary Tree in alternate order**

Print extreme nodes of each level of Binary Tree in alternate order - GeeksforGeeks

Given a binary tree, print nodes of extreme corners of each level but in alternate order.

Example:



For above tree, the output can be

**1 2 7 8 31**

- print rightmost node of 1st level
- print leftmost node of 2nd level
- print rightmost node of 3rd level
- print leftmost node of 4th level
- print rightmost node of 5th level

OR

**1 3 4 15 16**

- print leftmost node of 1st level
- print rightmost node of 2nd level
- print leftmost node of 3rd level
- print rightmost node of 4th level
- print leftmost node of 5th level

The idea is to traverse tree level by level. For each level, we count number of nodes in it and print its leftmost or the rightmost node based on value of a Boolean flag. We dequeue all nodes of current level and enqueue all nodes of next level and invert value of Boolean flag when switching levels.

Below is C++ implementation of above idea –

```
/* C++ program to print nodes of extreme corners
of each level in alternate order */
#include <bits/stdc++.h>
using namespace std;

/* A binary tree node has data, pointer to left child
and a pointer to right child */
struct Node
{
    int data;
    Node *left, *right;
};

/* Helper function that allocates a new node with the
given data and NULL left and right pointers. */
Node* newNode(int data)
{
    Node* node = new Node;
    node->data = data;
    node->right = node->left = NULL;
    return node;
}

/* Function to print nodes of extreme corners
of each level in alternate order */
void printExtremeNodes(Node* root)
{
    if (root == NULL)
        return;

    // Create a queue and enqueue left and right
    // children of root
    queue<Node*> q;
    q.push(root);
```

```
// flag to indicate whether leftmost node or
// the rightmost node has to be printed
bool flag = false;
while (!q.empty())
{
    // nodeCount indicates number of nodes
    // at current level.
    int nodeCount = q.size();
    int n = nodeCount;

    // Dequeue all nodes of current level
    // and Enqueue all nodes of next level
    while (n--)
    {
        Node* curr = q.front();

        // Enqueue left child
        if (curr->left)
            q.push(curr->left);

        // Enqueue right child
        if (curr->right)
            q.push(curr->right);

        // Dequeue node
        q.pop();

        // if flag is true, print leftmost node
        if (flag && n == nodeCount - 1)
            cout << curr->data << " ";

        // if flag is false, print rightmost node
        if (!flag && n == 0)
            cout << curr->data << " ";
    }
    // invert flag for next level
    flag = !flag;
}

/* Driver program to test above functions */
int main()
{
    // Binary Tree of Height 4
    Node* root = newNode(1);

    root->left = newNode(2);
    root->right = newNode(3);
```

```
root->left->left = newNode(4);
root->left->right = newNode(5);
root->right->right = newNode(7);

root->left->left->left = newNode(8);
root->left->left->right = newNode(9);
root->left->right->left = newNode(10);
root->left->right->right = newNode(11);
root->right->right->left = newNode(14);
root->right->right->right = newNode(15);

root->left->left->left->left = newNode(16);
root->left->left->left->right = newNode(17);
root->right->right->right->right = newNode(31);

printExtremeNodes(root);

return 0;
}
```

Output:

1 2 7 8 31

**Time complexity** of above solution is  $O(n)$  where  $n$  is total number of nodes in given binary tree.

**Exercise** – Print nodes of extreme corners of each level from bottom to top in alternate order.

## Source

<https://www.geeksforgeeks.org/print-extreme-nodes-of-each-level-of-binary-tree-in-alternate-order/>

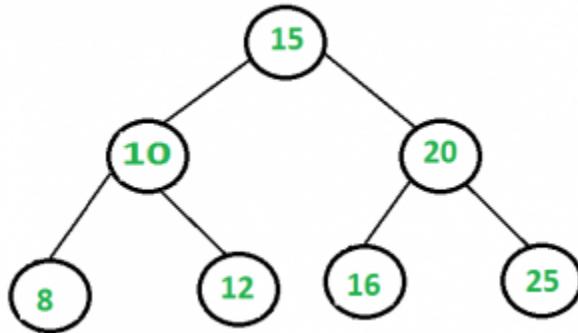
## Chapter 341

# Print leftmost and rightmost nodes of a Binary Tree

Print leftmost and rightmost nodes of a Binary Tree - GeeksforGeeks

Given a Binary Tree, Print the corner nodes at each level. The node at the leftmost and the node at the rightmost.

For example, output for following is **15, 10, 20, 8, 25**.



A Simple Solution is to do two traversals using the approaches discussed for [printing left view](#) and [right view](#).

**Can we print all corner nodes using one traversal?**

The idea is to use [Level Order Traversal](#). To find first node, we use a variable **isFirst**. To separate levels, we enqueue NULL after every level. So in level order traversal, if we see a NULL, we know next node would be first node of its level and therefore we set **isFirst**.

A special case to consider is, a tree like below.

```
1  
 \
```



The output for above tree should be **1, 2, 3**. We need make sure that the levels having only one node are handled and the node is printed only once. For this purpose, we maintain a separate variable **isOne**.

C++

```
// C/C++ program to print corner node at each level
// of binary tree
#include <bits/stdc++.h>
using namespace std;

/* A binary tree node has key, pointer to left
   child and a pointer to right child */
struct Node
{
    int key;
    struct Node* left, *right;
};

/* To create a newNode of tree and return pointer */
struct Node* newNode(int key)
{
    Node* temp = new Node;
    temp->key = key;
    temp->left = temp->right = NULL;
    return (temp);
}

/* Function to print corner node at each level */
void printCorner(Node *root)
{
    // star node is for keeping track of levels
    queue<Node *> q;

    // pushing root node and star node
    q.push(root);
    q.push(NULL);

    // if isFirst = true then left most node of that level
    // will be printed
    bool isFirst = false;

    // if isOne = true then that level has only one node
```

```
bool isOne = false;

// last will store right most node of that level
int last;

// Do level order traversal of Binary Tree
while (!q.empty())
{
    // dequeue the front node from the queue
    Node *temp = q.front();
    q.pop();

    // if isFirst is true, then temp is leftmost node
    if (isFirst)
    {
        cout << temp->key << " ";

        if (temp->left)
            q.push(temp->left);
        if (temp->right)
            q.push(temp->right);

        // make isFirst as false and one = 1
        isFirst = false;
        isOne = true;
    }

    // Else if temp is a separator between two levels
    else if (temp == NULL)
    {
        // Insert new separator if there are items in queue
        if (q.size() >= 1)
            q.push(NULL);

        // making isFirst as true because next node will be
        // leftmost node of that level
        isFirst = true;

        // printing last node, only if that level
        // doesn't contain single node otherwise
        // that single node will be printed twice
        if (!isOne)
            cout << last << " ";
    }
    else
    {
        // Store current key as last
        last = temp->key;
    }
}
```

```
// Here we are making isOne = false to signify
// that level has more than one node
isOne = false;
if (temp->left)
    q.push(temp->left);
if (temp->right)
    q.push(temp->right);
}
}
}

// Driver program to test above function
int main ()
{
    Node *root = newNode(15);
    root->left = newNode(10);
    root->right = newNode(20);
    root->left->left = newNode(8);
    root->left->right = newNode(12);
    root->right->left = newNode(16);
    root->right->right = newNode(25);
    printCorner(root);
    return 0;
}
```

### Java

```
// Java program to print corner node at each level in a binary tree

import java.util.*;

/* A binary tree node has key, pointer to left
   child and a pointer to right child */
class Node
{
    int key;
    Node left, right;

    public Node(int key)
    {
        this.key = key;
        left = right = null;
    }
}

class BinaryTree
{
```

```
Node root;

/* Function to print corner node at each level */
void printCorner(Node root)
{
    // star node is for keeping track of levels
    Queue<Node> q = new LinkedList<Node>();

    // pushing root node and star node
    q.add(root);
    q.add(null);

    // if isFirst = true then left most node of that level
    // will be printed
    boolean isFirst = false;

    // if isOne = true then that level has only one node
    boolean isOne = false;

    // last will store right most node of that level
    int last = 0;

    // Do level order traversal of Binary Tree
    while (!q.isEmpty())
    {
        // dequeue the front node from the queue
        Node temp = q.peek();
        q.poll();

        // if isFirst is true, then temp is leftmost node
        if (isFirst)
        {
            System.out.print(temp.key + " ");

            if (temp.left != null)
                q.add(temp.left);
            if (temp.right != null)
                q.add(temp.right);
        }

        // make isFirst as false and one = 1
        isFirst = false;
        isOne = true;
    }

    // Else if temp is a separator between two levels
    else if (temp == null)
    {
        // Insert new separator if there are items in queue
    }
}
```

```
if (q.size() >= 1)
    q.add(null);

// making isFirst as true because next node will be
// leftmost node of that level
isFirst = true;

// printing last node, only if that level
// doesn't contain single node otherwise
// that single node will be printed twice
if (!isOne)
    System.out.print(last + " ");
}
else
{
    // Store current key as last
    last = temp.key;

    // Here we are making isOne = false to signify
    // that level has more than one node
    isOne = false;
    if (temp.left != null)
        q.add(temp.left);
    if (temp.right != null)
        q.add(temp.right);
}
}

// Driver program to test above functions
public static void main(String[] args)
{
    BinaryTree tree = new BinaryTree();
    tree.root = new Node(15);
    tree.root.left = new Node(10);
    tree.root.right = new Node(20);
    tree.root.left.left = new Node(8);
    tree.root.left.right = new Node(12);
    tree.root.right.left = new Node(16);
    tree.root.right.right = new Node(25);

    tree.printCorner(tree.root);
}
}

// This code has been contributed by Mayank Jaiswal(mayank_24)
```

Output :

15 10 20 8 25

Time Complexity :  $O(n)$  where  $n$  is number of nodes in Binary Tree.

This article is contributed by Utkarsh Trivedi. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## Source

<https://www.geeksforgeeks.org/print-leftmost-and-rightmost-nodes-of-a-binary-tree/>

## Chapter 342

# Print level order traversal line by line | Set 1

Print level order traversal line by line | Set 1 - GeeksforGeeks

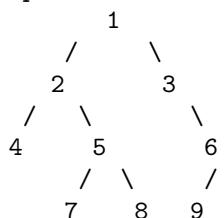
Given a binary tree, print level order traversal in a way that nodes of all levels are printed in separate lines.

For example consider the following tree

Example 1:

```
Output for above tree should be
20
8 22
4 12
10 14
```

Example 2:



```
Output for above tree should be
```

```
1
2 3
4 5 6
7 8 9<
```

Note that this is different from [simple level order traversal](#) where we need to print all nodes together. Here we need to print nodes of different levels in different lines.

A simple solution is to print use the recursive function discussed in the [level order traversal post](#) and print a new line after every call to [printGivenLevel\(\)](#).

C++

```
/* Function to line by line print level order traversal a tree*/
void printLevelOrder(struct node* root)
{
    int h = height(root);
    int i;
    for (i=1; i<=h; i++)
    {
        printGivenLevel(root, i);
        printf("\n");
    }
}

/* Print nodes at a given level */
void printGivenLevel(struct node* root, int level)
{
    if (root == NULL)
        return;
    if (level == 1)
        printf("%d ", root->data);
    else if (level > 1)
    {
        printGivenLevel(root->left, level-1);
        printGivenLevel(root->right, level-1);
    }
}
```

Java

```
/* Function to line by line print level order traversal a tree*/
static void printLevelOrder(Node root)
{
    int h = height(root);
    int i;
    for (i=1; i<=h; i++)
    {
        printGivenLevel(root, i);
        System.out.println();
    }
}

/* Print nodes at a given level */
```

```
void printGivenLevel(Node root, int level)
{
    if (root == null)
        return;
    if (level == 1)
        System.out.println(root.data);
    else if (level > 1)
    {
        printGivenLevel(root.left, level-1);
        printGivenLevel(root.right, level-1);
    }
}
```

The time complexity of the above solution is  $O(n^2)$

**How to modify the iterative level order traversal (Method 2 of this) to levels line by line?**

The idea is similar to [this](#) post. We count the nodes at current level. And for every node, we enqueue its children to queue.

C++

```
/* Iterative program to print levels line by line */
#include <iostream>
#include <queue>
using namespace std;

// A Binary Tree Node
struct node
{
    struct node *left;
    int data;
    struct node *right;
};

// Iterative method to do level order traversal line by line
void printLevelOrder(node *root)
{
    // Base Case
    if (root == NULL)  return;

    // Create an empty queue for level order traversal
    queue<node *> q;

    // Enqueue Root and initialize height
    q.push(root);

    while (1)
    {
```

```
// nodeCount (queue size) indicates number of nodes
// at current level.
int nodeCount = q.size();
if (nodeCount == 0)
    break;

// Dequeue all nodes of current level and Enqueue all
// nodes of next level
while (nodeCount > 0)
{
    node *node = q.front();
    cout << node->data << " ";
    q.pop();
    if (node->left != NULL)
        q.push(node->left);
    if (node->right != NULL)
        q.push(node->right);
    nodeCount--;
}
cout << endl;
}

// Utility function to create a new tree node
node* newNode(int data)
{
    node *temp = new node;
    temp->data = data;
    temp->left = NULL;
    temp->right = NULL;
    return temp;
}

// Driver program to test above functions
int main()
{
    // Let us create binary tree shown above
    node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->right = newNode(6);

    printLevelOrder(root);
    return 0;
}
```

### Java

```
/* An Iterative Java program to print levels line by line */

import java.util.LinkedList;
import java.util.Queue;

public class LevelOrder
{
    // A Binary Tree Node
    static class Node
    {
        int data;
        Node left;
        Node right;

        // constructor
        Node(int data){
            this.data = data;
            left = null;
            right =null;
        }
    }

    // Iterative method to do level order traversal line by line
    static void printLevelOrder(Node root)
    {
        // Base Case
        if(root == null)
            return;

        // Create an empty queue for level order tarversal
        Queue<Node> q =new LinkedList<Node>();

        // Enqueue Root and initialize height
        q.add(root);

        while(true)
        {

            // nodeCount (queue size) indicates number of nodes
            // at current level.
            int nodeCount = q.size();
            if(nodeCount == 0)
                break;

            // Dequeue all nodes of current level and Enqueue all
```

```
// nodes of next level
while(nodeCount > 0)
{
    Node node = q.peek();
    System.out.print(node.data + " ");
    q.remove();
    if(node.left != null)
        q.add(node.left);
    if(node.right != null)
        q.add(node.right);
    nodeCount--;
}
System.out.println();

}

// Driver program to test above functions
public static void main(String[] args)
{
    // Let us create binary tree shown in above diagram
    /*
           1
          /   \
         2     3
        / \   \
       4   5   6
    */
    Node root = new Node(1);
    root.left = new Node(2);
    root.right = new Node(3);
    root.left.left = new Node(4);
    root.left.right = new Node(5);
    root.right.right = new Node(6);

    printLevelOrder(root);
}

//This code is contributed by Sumit Ghosh
```

Output:

```
1
2 3
4 5 6
```

Time complexity of this method is  $O(n)$  where  $n$  is number of nodes in given binary tree.

[Level order traversal line by line | Set 2 \(Using Two Queues\)](#)

### **Source**

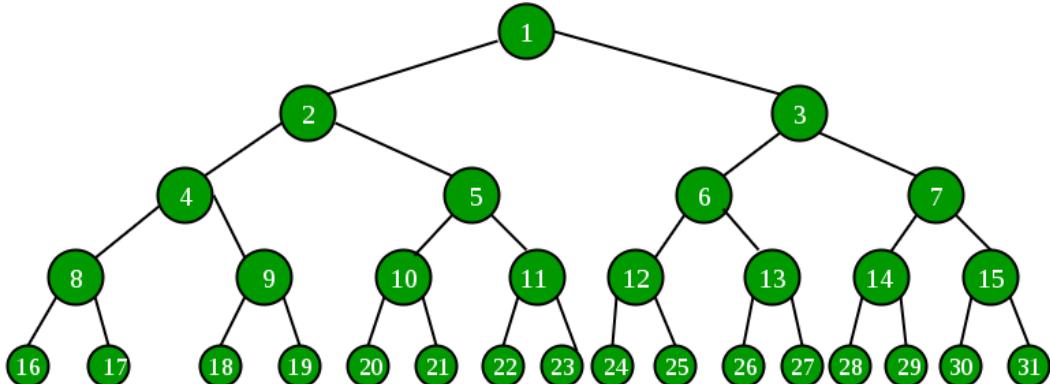
<https://www.geeksforgeeks.org/print-level-order-traversal-line-line/>

## Chapter 343

# Print middle level of perfect binary tree without finding height

Print middle level of perfect binary tree without finding height - GeeksforGeeks

Given a [perfect binary tree](#), print nodes of middle level without computing its height. A perfect binary tree is a binary tree in which all interior nodes have two children and all leaves have the same depth or same level.



Output : 4 5 6 7

The idea is similar to method 2 of [finding middle of singly linked list](#).

Use fast and slow (or tortoise) pointers in each route of the tree.

1. Advance fast pointer towards leaf by 2.
2. Advance slow pointer towards lead by 1.
3. If fast pointer reaches the leaf print value at slow pointer
4. Call recursively the next route.

C++

```
#include <bits/stdc++.h>
using namespace std;

/* A binary tree node has key, pointer to left
   child and a pointer to right child */
struct Node
{
    int key;
    struct Node* left, *right;
};

/* To create a newNode of tree and return pointer */
struct Node* newNode(int key)
{
    Node* temp = new Node;
    temp->key = key;
    temp->left = temp->right = NULL;
    return (temp);
}

// Takes two parameters - same initially and
// calls recursively
void printMiddleLevelUtil(Node* a, Node* b)
{
    // Base case e
    if (a == NULL || b == NULL)
        return;

    // Fast pointer has reached the leaf so print
    // value at slow pointer
    if ((b->left == NULL) && (b->right == NULL))
    {
        cout << a->key << " ";
        return;
    }

    // Recursive call
    // root.left.left and root.left.right will
    // print same value
    // root.right.left and root.right.right
    // will print same value
    // So we use any one of the condition
    printMiddleLevelUtil(a->left, b->left->left);
    printMiddleLevelUtil(a->right, b->left->left);
}
```

```
// Main printing method that take a Tree as input
void printMiddleLevel(Node* node)
{
    printMiddleLevelUtil(node, node);
}

// Driver program to test above functions
int main()
{
    Node* n1 = newNode(1);
    Node* n2 = newNode(2);
    Node* n3 = newNode(3);
    Node* n4 = newNode(4);
    Node* n5 = newNode(5);
    Node* n6 = newNode(6);
    Node* n7 = newNode(7);

    n2->left = n4;
    n2->right = n5;
    n3->left = n6;
    n3->right = n7;
    n1->left = n2;
    n1->right = n3;

    printMiddleLevel(n1);
}

// This code is contributed by Prasad Kshirsagar
```

**Java**

```
// Tree node definition
class Node
{
    public int key;
    public Node left;
    public Node right;
    public Node(int val)
    {
        this.left = null;
        this.right = null;
        this.key = val;
    }
}

public class PrintMiddle
```

```
{  
    // Takes two parameters - same initially and  
    // calls recursively  
    private static void printMiddleLevelUtil(Node a,  
                                              Node b)  
{  
    // Base case e  
    if (a == null || b == null)  
        return;  
  
    // Fast pointer has reached the leaf so print  
    // value at slow pointer  
    if ((b.left == null) && (b.right == null))  
    {  
        System.out.print(a.key + " ");  
        return;  
    }  
  
    // Recursive call  
    // root.left.left and root.left.right will  
    // print same value  
    // root.right.left and root.right.right  
    // will print same value  
    // So we use any one of the condition  
    printMiddleLevelUtil(a.left, b.left.left);  
    printMiddleLevelUtil(a.right, b.left.left);  
}  
  
// Main printing method that take a Tree as input  
public static void printMiddleLevel(Node node)  
{  
    printMiddleLevelUtil(node, node);  
}  
  
public static void main(String[] args)  
{  
    Node n1 = new Node(1);  
    Node n2 = new Node(2);  
    Node n3 = new Node(3);  
    Node n4 = new Node(4);  
    Node n5 = new Node(5);  
    Node n6 = new Node(6);  
    Node n7 = new Node(7);  
  
    n2.left = n4;  
    n2.right = n5;  
    n3.left = n6;  
    n3.right = n7;
```

```
n1.left = n2;
n1.right = n3;

    printMiddleLevel(n1);
}
}
```

Output:

2 3

**Improved By :** [Prasad\\_Kshirsagar](#)

## Source

<https://www.geeksforgeeks.org/print-middle-level-perfect-binary-tree-without-finding-height/>

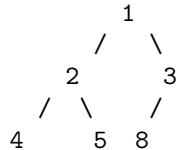
## Chapter 344

# Print nodes at k distance from root

Print nodes at k distance from root - GeeksforGeeks

Given a root of a tree, and an integer k. Print all the nodes which are at k distance from root.

For example, in the below tree, 4, 5 & 8 are at distance 2 from root.



The problem can be solved using recursion. Thanks to eldho for suggesting the solution.

C

```
#include <stdio.h>
#include <stdlib.h>

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};
```

```
void printKDistant(struct node *root , int k)
{
    if(root == NULL)
        return;
    if( k == 0 )
    {
        printf( "%d ", root->data );
        return ;
    }
    else
    {
        printKDistant( root->left, k-1 ) ;
        printKDistant( root->right, k-1 ) ;
    }
}

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* node = (struct node*)
                           malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;

    return(node);
}

/* Driver program to test above functions*/
int main()
{

    /* Constructed binary tree is
           1
         /   \
        2     3
       / \   /
      4   5  8
    */
    struct node *root = newNode(1);
    root->left      = newNode(2);
    root->right     = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->left = newNode(8);

    printKDistant(root, 2);
}
```

```
        getchar();
        return 0;
    }

Java

// Java program to print nodes at k distance from root

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
class Node
{
    int data;
    Node left, right;

    Node(int item)
    {
        data = item;
        left = right = null;
    }
}

class BinaryTree
{
    Node root;

    void printKDistant(Node node, int k)
    {
        if (node == null)
            return;
        if (k == 0)
        {
            System.out.print(node.data + " ");
            return;
        }
        else
        {
            printKDistant(node.left, k - 1);
            printKDistant(node.right, k - 1);
        }
    }

    /* Driver program to test above functions */
    public static void main(String args[])
    {
        BinaryTree tree = new BinaryTree();

        /* Constructed binary tree is
           50
         /   \
        30   70
      /   \   /   \
     20  40  60  80
    /  \  /  \
   10  30  50  70
  /  \
 10  20

```

```
        1
       /   \
      2     3
     / \   /
    4   5  8
*/
tree.root = new Node(1);
tree.root.left = new Node(2);
tree.root.right = new Node(3);
tree.root.left.left = new Node(4);
tree.root.left.right = new Node(5);
tree.root.right.left = new Node(8);

tree.printKDistant(tree.root, 2);
}
}

// This code has been contributed by Mayank Jaiswal
```

### Python

```
# Python program to find the nodes at k distance from root

# A Binary tree node
class Node:

    # Constructor to create a new node
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

    def printKDistant(self, k):

        if self is None:
            return
        if k == 0:
            print self.data,
        else:
            self.left.printKDistant(k-1)
            self.right.printKDistant(k-1)

# Driver program to test above function
"""
Constructed binary tree is
        1
       /   \
      2     3

```

```
      /   \   /
     4     5   8
     """
root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)
root.right.left = Node(8)

printKDistant(root, 2)

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

The above program prints 4, 5 and 8.

Time Complexity: O(n) where n is number of nodes in the given binary tree.

## Source

<https://www.geeksforgeeks.org/print-nodes-at-k-distance-from-root/>

## Chapter 345

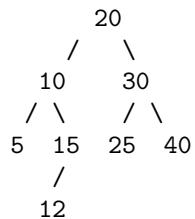
# Print nodes at k distance from root | Iterative

Print nodes at k distance from root | Iterative - GeeksforGeeks

Given a root of a tree, and an integer k. Print all the nodes which are at k distance from root.

For example :

Input :



and k = 3

Root is at level 1.

Output :

5 15 25 40

Recursive approach to this problem is discussed [here](#)

Following is the iterative approach.

The solution is similar to [Getting level of node in Binary Tree](#)

```
// CPP program to print all nodes of level k
// iterative approach
```

```
/* binary tree
root is at level 1

      20
     /   \
    10   30
   / \   / \
  5  15  25  40
   /
  12 */

#include <bits/stdc++.h>
using namespace std;

// Node of binary tree
struct Node {
    int data;
    Node* left, * right;
};

// Function to add a new node
Node* newNode(int data)
{
    Node* newnode = new Node();
    newnode->data = data;
    newnode->left = newnode->right = NULL;
}

// Function to print nodes of given level
bool printKDistant(Node* root, int klevel)
{
    queue<Node*> q;
    int level = 1;
    bool flag = false;
    q.push(root);

    // extra NULL is pushed to keep track
    // of all the nodes to be pushed before
    // level is incremented by 1
    q.push(NULL);
    while (!q.empty()) {
        Node* temp = q.front();

        // print when level is equal to k
        if (level == klevel && temp != NULL) {
            flag = true;
            cout << temp->data << " ";
        }
        q.pop();
        if (temp == NULL) {
            if (flag)
                cout << endl;
            level++;
            q.push(NULL);
        } else {
            q.push(temp->left);
            q.push(temp->right);
        }
    }
}
```

```
if (temp == NULL) {
    if (q.front())
        q.push(NULL);
    level += 1;

    // break the loop if level exceeds
    // the given level number
    if (level > klevel)
        break;
} else {
    if (temp->left)
        q.push(temp->left);

    if (temp->right)
        q.push(temp->right);
}
cout << endl;

return flag;
}

// Driver code
int main()
{
    // create a binary tree
    Node* root = newNode(20);
    root->left = newNode(10);
    root->right = newNode(30);
    root->left->left = newNode(5);
    root->left->right = newNode(15);
    root->left->right->left = newNode(12);
    root->right->left = newNode(25);
    root->right->right = newNode(40);

    cout << "data at level 1 : ";
    int ret = printKDistant(root, 1);
    if (ret == false)
        cout << "Number exceeds total number of levels\n";

    cout << "data at level 2 : ";
    ret = printKDistant(root, 2);
    if (ret == false)
        cout << "Number exceeds total number of levels\n";

    cout << "data at level 3 : ";
    ret = printKDistant(root, 3);
    if (ret == false)
```

```
cout << "Number exceeds total number of levels\n";

cout << "data at level 6 : ";
ret = printKDistant(root, 6);
if (ret == false)
    cout << "Number exceeds total number of levels\n";

return 0;
}
```

Output:

```
data at level 1 : 20
data at level 2 : 10 30
data at level 3 : 5 15 25 40
data at level 6 :
Number exceeds total number of levels
```

## Source

<https://www.geeksforgeeks.org/print-nodes-k-distance-root-iterative/>

## Chapter 346

# Print nodes between two given level numbers of a binary tree

Print nodes between two given level numbers of a binary tree - GeeksforGeeks

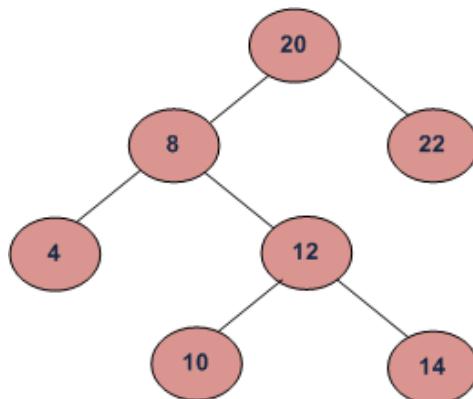
Given a binary tree and two level numbers ‘low’ and ‘high’, print nodes from level low to level high.

For example consider the binary tree given in below diagram.

Input: Root of below tree, low = 2, high = 4

Output:

8 22  
4 12  
10 14



A **Simple Method** is to first write a recursive function that prints nodes of a given level number. Then call recursive function in a loop from low to high. Time complexity of this

method is  $O(n^2)$

We can print nodes **in  $O(n)$  time** using queue based iterative level order traversal. The idea is to do simple queue based level order traversal. While doing inorder traversal, add a marker node at the end. Whenever we see a marker node, we increase level number. If level number is between low and high, then print nodes.

The following is the implementation of above idea.

C++

```
// A C++ program to print Nodes level by level berween given two levels.
#include <iostream>
#include <queue>
using namespace std;

/* A binary tree Node has key, pointer to left and right children */
struct Node
{
    int key;
    struct Node* left, *right;
};

/* Given a binary tree, print nodes from level number 'low' to level
   number 'high'*/
void printLevels(Node* root, int low, int high)
{
    queue <Node *> Q;

    Node *marker = new Node; // Marker node to indicate end of level

    int level = 1; // Initialize level number

    // Enqueue the only first level node and marker node for end of level
    Q.push(root);
    Q.push(marker);

    // Simple level order traversal loop
    while (Q.empty() == false)
    {
        // Remove the front item from queue
        Node *n = Q.front();
        Q.pop();

        // Check if end of level is reached
        if (n == marker)
        {
            // print a new line and increment level number
            cout << endl;
            level++;
        }
        else
        {
            cout << n->key << " ";
            if (n->left)
                Q.push(n->left);
            if (n->right)
                Q.push(n->right);
        }
    }
}
```

```
// Check if marker node was last node in queue or
// level number is beyond the given upper limit
if (Q.empty() == true || level > high) break;

// Enqueue the marker for end of next level
Q.push(marker);

// If this is marker, then we don't need print it
// and enqueue its children
continue;
}

// If level is equal to or greater than given lower level,
// print it
if (level >= low)
    cout << n->key << " ";

// Enqueue children of non-marker node
if (n->left != NULL) Q.push(n->left);
if (n->right != NULL) Q.push(n->right);
}

}

/* Helper function that allocates a new Node with the
   given key and NULL left and right pointers. */
Node* newNode(int key)
{
    Node* temp = new Node;
    temp->key = key;
    temp->left = temp->right = NULL;
    return (temp);
}

/* Driver program to test above functions*/
int main()
{
    // Let us construct the BST shown in the above figure
    struct Node *root      = newNode(20);
    root->left           = newNode(8);
    root->right          = newNode(22);
    root->left->left    = newNode(4);
    root->left->right   = newNode(12);
    root->left->right->left = newNode(10);
    root->left->right->right = newNode(14);

    cout << "Level Order traversal between given two levels is";
    printLevels(root, 2, 3);
```

```
    return 0;
}
```

**Java**

```
// Java program to print Nodes level by level between given two levels
import java.util.LinkedList;
import java.util.Queue;

/* A binary tree Node has key, pointer to left and right children */
class Node
{
    int data;
    Node left, right;

    public Node(int item)
    {
        data = item;
        left = right = null;
    }
}

class BinaryTree
{
    Node root;

    /* Given a binary tree, print nodes from level number 'low' to level
       number 'high'*/
    void printLevels(Node node, int low, int high)
    {
        Queue<Node> Q = new LinkedList<>();

        Node marker = new Node(4); // Marker node to indicate end of level

        int level = 1; // Initialize level number

        // Enqueue the only first level node and marker node for end of level
        Q.add(node);
        Q.add(marker);

        // Simple level order traversal loop
        while (Q.isEmpty() == false)
        {
            // Remove the front item from queue
            Node n = Q.peek();
            Q.remove();

```

```
// Check if end of level is reached
if (n == marker)
{
    // print a new line and increment level number
    System.out.println("");
    level++;

    // Check if marker node was last node in queue or
    // level number is beyond the given upper limit
    if (Q.isEmpty() == true || level > high)
        break;

    // Enqueue the marker for end of next level
    Q.add(marker);

    // If this is marker, then we don't need print it
    // and enqueue its children
    continue;
}

// If level is equal to or greater than given lower level,
// print it
if (level >= low)
    System.out.print( n.data + " ");

// Enqueue children of non-marker node
if (n.left != null)
    Q.add(n.left);

if (n.right != null)
    Q.add(n.right);

}
}

// Driver program to test for above functions
public static void main(String args[])
{
    BinaryTree tree = new BinaryTree();
    tree.root = new Node(20);
    tree.root.left = new Node(8);
    tree.root.right = new Node(22);

    tree.root.left.left = new Node(4);
    tree.root.left.right = new Node(12);
    tree.root.left.right.left = new Node(10);
    tree.root.left.right.right = new Node(14);
```

```
System.out.print("Level Order traversal between given two levels is ");
tree.printLevels(tree.root, 2, 3);

}

}

// This code has been contributed by Mayank Jaiswal
```

### Python

```
# Python program to print nodes level by level between
# given two levels

# A binary tree node
class Node:
    # Constructor to create a new node
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None

# Given a binary tree, print nodes from level number 'low'
# to level number 'high'

def printLevels(root, low, high):
    Q = []

    marker = Node(11114) # Marker node to indicate end of level

    level = 1 # Initialize level number

    # Enqueue the only first level node and marker node for
    # end of level
    Q.append(root)
    Q.append(marker)

    #print Q
    # Simple level order traversal loop
    while(len(Q) >0):
        # Remove the front item from queue
        n = Q[0]
        Q.pop(0)
        #print Q
        # Check if end of level is reached
        if n == marker:
            # print a new line and increment level number
            print
            level += 1
```

```
# Check if marker node was last node in queue
# or level number is beyond the given upper limit
if len(Q) == 0 or level > high:
    break

# Enqueue the marker for end of next level
Q.append(marker)

# If this is marker, then we don't need print it
# and enqueue its children
continue
if level >= low:
    print n.key,

# Enqueue children of non-marker node
if n.left is not None:
    Q.append(n.left)
    Q.append(n.right)

# Driver program to test the above function
root = Node(20)
root.left = Node(8)
root.right = Node(22)
root.left.left = Node(4)
root.left.right = Node(12)
root.left.right.left = Node(10)
root.left.right.right = Node(14)

print "Level Order Traversal between given two levels is",
printLevels(root,2,3)

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

```
Level Order traversal between given two levels is
8 22
4 12
```

Time complexity of above method is O(n) as it does a simple level order traversal.

This article is contributed by **Frank**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## Source

<https://www.geeksforgeeks.org/given-binary-tree-print-nodes-two-given-level-numbers/>

## Chapter 347

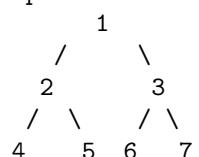
# Print nodes in top view of Binary Tree | Set 2

Print nodes in top view of Binary Tree | Set 2 - GeeksforGeeks

Top view of a binary tree is the set of nodes visible when the tree is viewed from the top. Given a binary tree, print the top view of it. The output nodes should be printed from **left to right**.

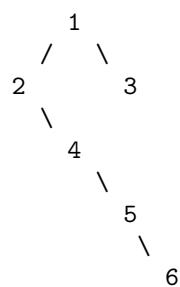
**Note:** A node x is there in output if x is the topmost node at its horizontal distance. Horizontal distance of the left child of a node x is equal to the horizontal distance of x minus 1, and that of right child is the horizontal distance of x plus 1.

Input:



Output: Top view: 4 2 1 3 7

Input:



Output: Top view: 2 1 3 6

The idea is to do something similar to [Vertical Order Traversal](#). Like [Vertical Order Traversal](#), we need to group nodes of same horizontal distance together. We do a level order traversal so that the topmost node at a horizontal node is visited before any other node of same horizontal distance below it. A [Map](#) is used to map the horizontal distance of the node with the node's Data and vertical distance of the node.

Below is the implementation of the above approach:

```
// C++ Program to print Top View of Binary Tree
// using hashmap and recursion
#include <bits/stdc++.h>
using namespace std;

// Node structure
struct Node {
    // Data of the node
    int data;

    // Horizontal Distance of the node
    int hd;

    // Reference to left node
    struct Node* left;

    // Reference to right node
    struct Node* right;
};

// Initialising node
struct Node* newNode(int data)
{
    struct Node* node = new Node;
    node->data = data;
    node->hd = INT_MAX;
    node->left = NULL;
    node->right = NULL;
    return node;
}

void printTopViewUtil(Node* root, int height,
                     int hd, map<int, pair<int, int> &m)
{
    // Base Case
    if (root == NULL)
        return;

    // If the node for particular horizontal distance
    // is not present in the map, add it.
    // For top view, we consider the first element
    m[hd].first = root->data;
    m[hd].second = height;
}
```

```
// at horizontal distance in level order traversal
if (m.find(hd) == m.end()) {
    m[hd] = make_pair(root->data, height);
}
else{
    pair<int, int> p = (m.find(hd))->second;

    if (p.second > height) {
        m.erase(hd);
        m[hd] = make_pair(root->data, height);
    }
}

// Recur for left and right subtree
printTopViewUtil(root->left, height + 1, hd - 1, m);
printTopViewUtil(root->right, height + 1, hd + 1, m);
}

void printTopView(Node* root)
{
    // Map to store horizontal distance,
    // height and node's data
    map<int, pair<int, int> > m;
    printTopViewUtil(root, 0, 0, m);

    // Print the node's value stored by printTopViewUtil()
    for (map<int, pair<int, int> >::iterator it = m.begin();
                                     it != m.end(); it++) {
        pair<int, int> p = it->second;
        cout << p.first << " ";
    }
}

int main()
{
    Node* root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->right = newNode(4);
    root->left->right->right = newNode(5);
    root->left->right->right->right = newNode(6);

    cout << "Top View : ";
    printTopView(root);

    return 0;
}
```

**Output:**

Top View : 2 1 3 6

**Source**

<https://www.geeksforgeeks.org/print-nodes-in-top-view-of-binary-tree-set-2/>

## Chapter 348

# Print path from root to a given node in a binary tree

Print path from root to a given node in a binary tree - GeeksforGeeks

Given a binary tree with distinct nodes(no two nodes have the same have data values). The problem is to print the path from root to a given node **x**. If node **x** is not present then print “No Path”.

Examples:

Input :  
        1  
      /    \  
    2    3  
  / \  / \  
4  5  6  7

x = 5

Output : 1->2->5

**Approach:** Create a recursive function that traverses the different path in the binary tree to find the required node **x**. If node **x** is present then it returns true and accumulates the path nodes in some array **arr[]**. Else it returns false.

Following are the cases during the traversal:

1. If **root = NULL**, return false.
2. push the root's data into **arr[]**.
3. if **root's data = x**, return true.
4. if node **x** is present in root's left or right subtree, return true.
5. Else remove root's data value from **arr[]** and return false.

This recursive function can be accessed from other function to check whether node **x** is present or not and if it is present, then the path nodes can be accessed from **arr[]**. You can define **arr[]** globally or pass its reference to the recursive function.

```
// C++ implementation to print the path from root
// to a given node in a binary tree
#include <bits/stdc++.h>
using namespace std;

// structure of a node of binary tree
struct Node
{
    int data;
    Node *left, *right;
};

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct Node* getNode(int data)
{
    struct Node *newNode = new Node;
    newNode->data = data;
    newNode->left = newNode->right = NULL;
    return newNode;
}

// Returns true if there is a path from root
// to the given node. It also populates
// 'arr' with the given path
bool hasPath(Node *root, vector<int>& arr, int x)
{
    // if root is NULL
    // there is no path
    if (!root)
        return false;

    // push the node's value in 'arr'
    arr.push_back(root->data);

    // if it is the required node
    // return true
    if (root->data == x)
        return true;

    // else check whether the required node lies
    // in the left subtree or right subtree of
    // the current node
    if (hasPath(root->left, arr, x) ||
        hasPath(root->right, arr, x))
        arr.push_back(x);

    return false;
}
```

```
    hasPath(root->right, arr, x))
    return true;

    // required node does not lie either in the
    // left or right subtree of the current node
    // Thus, remove current node's value from
    // 'arr' and then return false
    arr.pop_back();
    return false;
}

// function to print the path from root to the
// given node if the node lies in the binary tree
void printPath(Node *root, int x)
{
    // vector to store the path
    vector<int> arr;

    // if required node 'x' is present
    // then print the path
    if (hasPath(root, arr, x))
    {
        for (int i=0; i<arr.size()-1; i++)
            cout << arr[i] << "->";
        cout << arr[arr.size() - 1];
    }

    // 'x' is not present in the binary tree
    else
        cout << "No Path";
}

// Driver program to test above
int main()
{
    // binary tree formation
    struct Node *root = getNode(1);
    root->left = getNode(2);
    root->right = getNode(3);
    root->left->left = getNode(4);
    root->left->right = getNode(5);
    root->right->left = getNode(6);
    root->right->right = getNode(7);

    int x = 5;
    printPath(root, x);
    return 0;
}
```

Output:

1->2->5

Time complexity:  $O(n)$  in worst case, where  $n$  is the number of nodes in the binary tree.

### Source

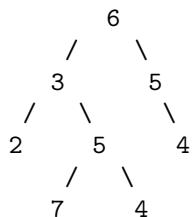
<https://www.geeksforgeeks.org/print-path-root-given-node-binary-tree/>

## Chapter 349

# Print root to leaf paths without using recursion

Print root to leaf paths without using recursion - GeeksforGeeks

Given a binary tree, print all its root to leaf paths without using recursion. For example, consider the following Binary Tree.



There are 4 leaves, hence 4 root to leaf paths -

```
6->3->2  
6->3->5->7  
6->3->5->4  
6->5>4
```

We strongly recommend you to minimize your browser and try this yourself first.

We can traverse tree iteratively (we have used [iterative preorder](#)). The question is, how to extend the traversal to print root to leaf paths? The idea is to maintain a map to store parent pointers of binary tree nodes. Now whenever we encounter a leaf node while doing iterative preorder traversal, we can easily print root to leaf path using parent pointer. Below is C++ implementation of this idea.

```
// C++ program to Print root to leaf path WITHOUT
```

```
// using recursion
#include <bits/stdc++.h>
using namespace std;

/* A binary tree */
struct Node
{
    int data;
    struct Node *left, *right;
};

/* Helper function that allocates a new node
   with the given data and NULL left and right
   pointers.*/
Node* newNode(int data)
{
    Node* node = new Node;
    node->data = data;
    node->left = node->right = NULL;
    return node;
}

/* Function to print root to leaf path for a leaf
   using parent nodes stored in map */
void printTopToBottomPath(Node* curr,
                         map<Node*, Node*> parent)
{
    stack<Node*> stk;

    // start from leaf node and keep on pushing
    // nodes into stack till root node is reached
    while (curr)
    {
        stk.push(curr);
        curr = parent[curr];
    }

    // Start popping nodes from stack and print them
    while (!stk.empty())
    {
        curr = stk.top();
        stk.pop();
        cout << curr->data << " ";
    }
    cout << endl;
}

/* An iterative function to do preorder traversal
```

```
of binary tree and print root to leaf path
without using recursion */
void printRootToLeaf(Node* root)
{
    // Corner Case
    if (root == NULL)
        return;

    // Create an empty stack and push root to it
    stack<Node*> nodeStack;
    nodeStack.push(root);

    // Create a map to store parent pointers of binary
    // tree nodes
    map<Node*, Node*> parent;

    // parent of root is NULL
    parent[root] = NULL;

    /* Pop all items one by one. Do following for
       every popped item
       a) push its right child and set its parent
          pointer
       b) push its left child and set its parent
          pointer
       Note that right child is pushed first so that
       left is processed first */
    while (!nodeStack.empty())
    {
        // Pop the top item from stack
        Node* current = nodeStack.top();
        nodeStack.pop();

        // If leaf node encountered, print Top To
        // Bottom path
        if (!(current->left) && !(current->right))
            printTopToBottomPath(current, parent);

        // Push right & left children of the popped node
        // to stack. Also set their parent pointer in
        // the map
        if (current->right)
        {
            parent[current->right] = current;
            nodeStack.push(current->right);
        }
        if (current->left)
        {
```

```
parent[current->left] = current;
nodeStack.push(current->left);
}
}

// Driver program to test above functions
int main()
{
    /* Constructed binary tree is
       10
      / \
     8   2
    / \   /
   3 5 2 */
    Node* root = newNode(10);
    root->left = newNode(8);
    root->right = newNode(2);
    root->left->left = newNode(3);
    root->left->right = newNode(5);
    root->right->left = newNode(2);

    printRootToLeaf(root);

    return 0;
}
```

Output :

```
10 8 3
10 8 5
10 2 2
```

This article is contributed by **Aditya Goel**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## Source

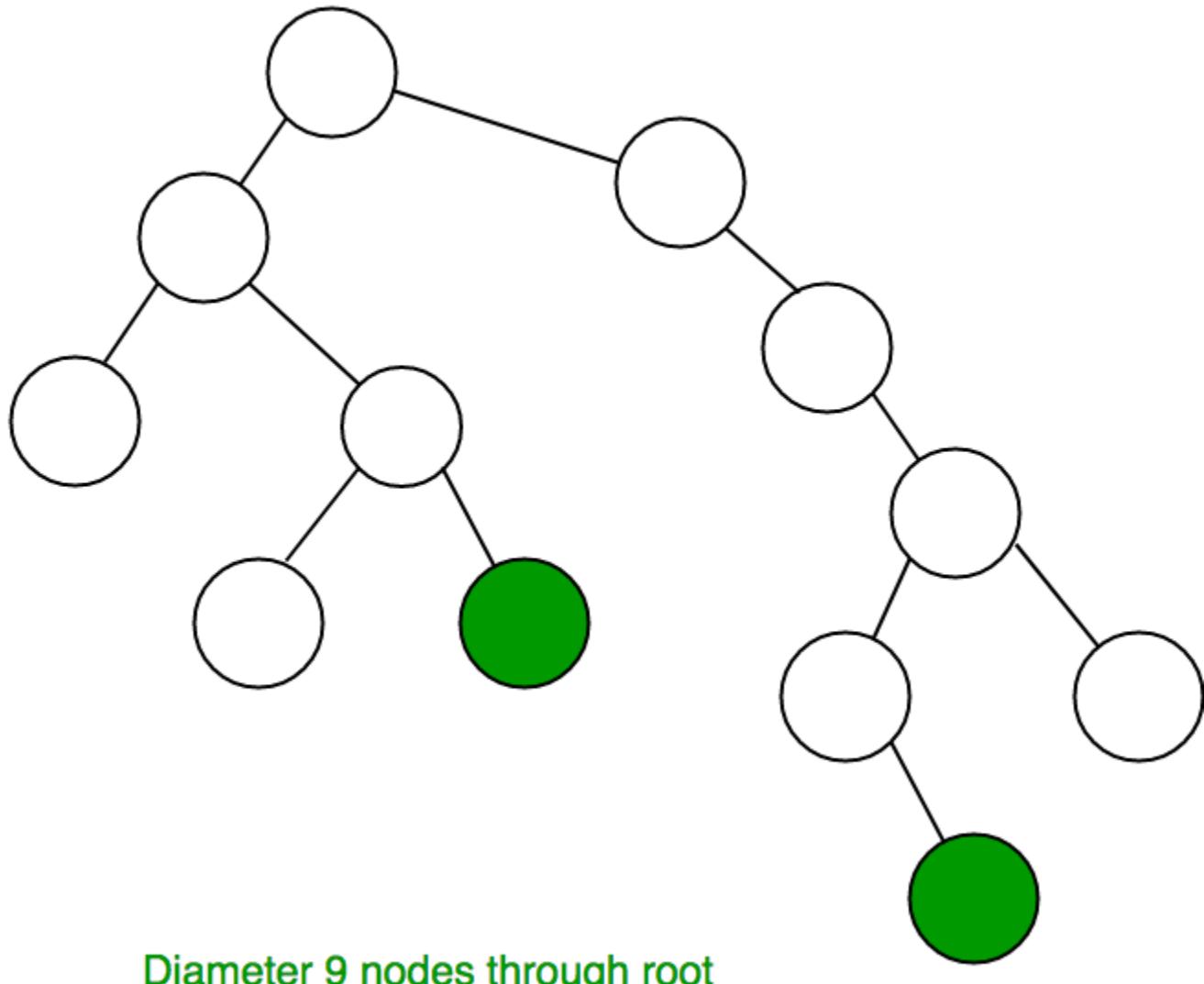
<https://www.geeksforgeeks.org/print-root-leaf-path-without-using-recursion/>

## Chapter 350

# Print the longest leaf to leaf path in a Binary tree

Print the longest leaf to leaf path in a Binary tree - GeeksforGeeks

The diameter of a tree (sometimes called the width) is the number of nodes on the longest path between two end nodes. In this post, we will see how to print the nodes involved in the diameter of the tree. The diagram below shows two trees each with diameter nine, the leaves that form the ends of the longest path are shaded (note that there is more than one path in each tree of length nine, but no path longer than nine nodes).



Examples:

Input:

```
      1
     / \
    2   3
   / \ 
  4   5
```

Output : 4 2 1 3  
or 5 2 1 3

Input:        1  
          /     \  
        2        3  
      /    \     \\  
    4      5       6

Output : 4 2 1 3 6  
or 5 2 1 3 6

We have already discussed how to find the diameter of a binary tree.[Diameter of a Binary tree](#)

We know that Diameter of a tree can be calculated by only using the height function because the diameter of a tree is nothing but the maximum value of ( $\text{left\_height} + \text{right\_height} + 1$ ) for each node.

Now for the node which has the maximum value of ( $\text{left\_height} + \text{right\_height} + 1$ ), we find the longest root to leaf path on the left side and similarly on the right side. Finally, we print left side path, root and right side path.

Time Complexity is  $O(N)$ . N is the number of nodes in the tree.

```
// CPP program to print the longest leaf to leaf
// path
#include <bits/stdc++.h>
using namespace std;

// Tree node structure used in the program
struct Node {
    int data;
    Node *left, *right;
};

struct Node* newNode(int data)
{
    struct Node* node = new Node;
    node->data = data;
    node->left = node->right = NULL;

    return (node);
}

// Function to find height of a tree
int height(Node* root, int& ans, Node*&k), int& lh, int& rh,
int& f)
{
    if (root == NULL)
        return 0;

    lh = height(root->left, ans, k, lh, rh, f);
    rh = height(root->right, ans, k, lh, rh, f);

    if (lh > rh)
        f = 1;
    else if (rh > lh)
        f = 2;

    if (f == 1)
        ans = max(ans, lh + rh + 1);
    else if (f == 2)
        ans = max(ans, rh + lh + 1);

    return max(lh, rh) + 1;
}
```

```
if (root == NULL)
    return 0;

int left_height = height(root->left, ans, k, lh, rh, f);

int right_height = height(root->right, ans, k, lh, rh, f);

// update the answer, because diameter of a
// tree is nothing but maximum value of
// (left_height + right_height + 1) for each node

if (ans < 1 + left_height + right_height) {

    ans = 1 + left_height + right_height;

    // save the root, this will help us finding the
    // left and the right part of the diameter
    k = root;

    // save the height of left & right subtree as well.
    lh = left_height;
    rh = right_height;
}

return 1 + max(left_height, right_height);
}

// prints the root to leaf path
void printArray(int ints[], int len, int f)
{
    int i;

    // print left part of the path in reverse order
    if (f == 0) {
        for (i = len - 1; i >= 0; i--) {
            printf("%d ", ints[i]);
        }
    }

    // print right part of the path
    else if (f == 1) {
        for (i = 0; i < len; i++) {
            printf("%d ", ints[i]);
        }
    }
}

// this function finds out all the root to leaf paths
```

```
void printPathsRecur(Node* node, int path[], int pathLen,
                      int max, int& f)
{
    if (node == NULL)
        return;

    // append this node to the path array
    path[pathLen] = node->data;
    pathLen++;

    // If it's a leaf, so print the path that led to here
    if (node->left == NULL && node->right == NULL) {

        // print only one path which is equal to the
        // height of the tree.
        if (pathLen == max && (f == 0 || f == 1)) {
            printArray(path, pathLen, f);
            f = 2;
        }
    }

    else {

        // otherwise try both subtrees
        printPathsRecur(node->left, path, pathLen, max, f);
        printPathsRecur(node->right, path, pathLen, max, f);
    }
}

// Computes the diameter of a binary tree with given root.
void diameter(Node* root)
{
    if (root == NULL)
        return;

    // lh will store height of left subtree
    // rh will store height of right subtree
    int ans = INT_MIN, lh = 0, rh = 0;

    // f is a flag whose value helps in printing
    // left & right part of the diameter only once
    int f = 0;
    Node* k;
    int height_of_tree = height(root, ans, k, lh, rh, f);
    int lPath[100], pathlen = 0;

    // print the left part of the diameter
    printPathsRecur(k->left, lPath, pathlen, lh, f);
```

```
printf("%d ", k->data);
int rPath[100];
f = 1;

// print the right part of the diameter
printPathsRecur(k->right, rPath, pathlen, rh, f);
}

// Driver code
int main()
{
    // Enter the binary tree ...
    //
    //      1
    //      / \
    //      2   3
    //      / \
    //      4   5
    //      \   / \
    //      8 6   7
    //      /
    //      9

    struct Node* root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->left->right->left = newNode(6);
    root->left->right->right = newNode(7);
    root->left->left->right = newNode(8);
    root->left->left->right->left = newNode(9);

    diameter(root);

    return 0;
}
```

**Output:**

9 8 4 2 5 6

**Source**

<https://www.geeksforgeeks.org/print-longest-leaf-leaf-path-binary-tree/>

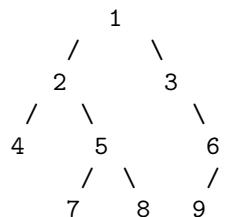
## Chapter 351

# Print the nodes at odd levels of a tree

Print the nodes at odd levels of a tree - GeeksforGeeks

Given a binary tree, print nodes of odd level in any order. Root is considered at level 1.

For example consider the following tree



Output 1 4 5 6

### Method 1 (Recursive)

The idea is to pass initial level as odd and switch level flag in every recursive call. For every node, if odd flag is set, then print it.

```
// Recursive C++ program to print odd level nodes
#include <bits/stdc++.h>
using namespace std;

struct Node {
    int data;
    Node* left, *right;
};
```

```
void printOddNodes(Node *root, bool isOdd = true)
{
    // If empty tree
    if (root == NULL)
        return;

    // If current node is of odd level
    if (isOdd)
        cout << root->data << " ";

    // Recur for children with isOdd
    // switched.
    printOddNodes(root->left, !isOdd);
    printOddNodes(root->right, !isOdd);
}

// Utility method to create a node
struct Node* newNode(int data)
{
    struct Node* node = new Node;
    node->data = data;
    node->left = node->right = NULL;
    return (node);
}

// Driver code
int main()
{
    struct Node* root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    printOddNodes(root);

    return 0;
}
```

Output:

1 4 5

Time complexity : O(n)

### Method 2 (Iterative)

The above code prints nodes in preorder way. If we wish to print nodes level by level, we can use level order traversal. The idea is based on [Print level order traversal line by line](#)

We traverse nodes level by level. We switch odd level flag after every level.

```
// Iterative C++ program to print odd level nodes
#include <bits/stdc++.h>
using namespace std;

struct Node {
    int data;
    Node* left, *right;
};

// Iterative method to do level order traversal line by line
void printOddNodes(Node *root)
{
    // Base Case
    if (root == NULL) return;

    // Create an empty queue for level
    // order traversal
    queue<Node *> q;

    // Enqueue root and initialize level as odd
    q.push(root);
    bool isOdd = true;

    while (1)
    {
        // nodeCount (queue size) indicates
        // number of nodes at current level.
        int nodeCount = q.size();
        if (nodeCount == 0)
            break;

        // Dequeue all nodes of current level
        // and Enqueue all nodes of next level
        while (nodeCount > 0)
        {
            Node *node = q.front();
            if (isOdd)
                cout << node->data << " ";
            q.pop();
            if (node->left != NULL)
                q.push(node->left);
            if (node->right != NULL)
                q.push(node->right);
            nodeCount--;
        }

        isOdd = !isOdd;
    }
}
```

```
}

// Utility method to create a node
struct Node* newNode(int data)
{
    struct Node* node = new Node;
    node->data = data;
    node->left = node->right = NULL;
    return (node);
}

// Driver code
int main()
{
    struct Node* root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    printOddNodes(root);

    return 0;
}
```

Output:

1 4 5

Time complexity : O(n)

## Source

<https://www.geeksforgeeks.org/print-nodes-odd-levels-tree/>

## Chapter 352

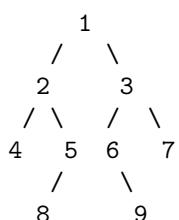
# Print the path common to the two paths from the root to the two given nodes

Print the path common to the two paths from the root to the two given nodes - Geeks-forGeeks

Given a binary tree with distinct nodes(no two nodes have the same have data values). The problem is to print the path common to the two paths from the root to the two given nodes **n1** and **n2**. If either of the nodes are not present then print “No Common Path”.

Examples:

Input :



**n1 = 4, n2 = 8**

Output : 1->2

Path form root to n1:

1->2->4

Path form root to n2:

1->2->5->8

Common Path:

1->2

**Approach:**The following steps are:

1. Find the **LCA**(Lowest Common Ancestor) of the two nodes **n1** and **n2**. Refer [this](#).
2. If **LCA** exists then print the path from the root to LCA. Refer [this](#). Else print “No Common Path”.

```
// C++ implementation to print the path common to the
// two paths from the root to the two given nodes
#include <bits/stdc++.h>

using namespace std;

// structure of a node of binary tree
struct Node
{
    int data;
    Node *left, *right;
};

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct Node* getNode(int data)
{
    struct Node *newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->left = newNode->right = NULL;
    return newNode;
}

// This function returns pointer to LCA of two given values n1 and n2.
// v1 is set as true by this function if n1 is found
// v2 is set as true by this function if n2 is found
struct Node *findLCAUtil(struct Node* root, int n1, int n2, bool &v1, bool &v2)
{
    // Base case
    if (root == NULL) return NULL;

    // If either n1 or n2 matches with root's data, report the presence
    // by setting v1 or v2 as true and return root (Note that if a key
    // is ancestor of other, then the ancestor key becomes LCA)
    if (root->data == n1)
    {
        v1 = true;
        return root;
    }
}
```

```
if (root->data == n2)
{
    v2 = true;
    return root;
}

// Look for nodes in left and right subtrees
Node *left_lca = findLCAUtil(root->left, n1, n2, v1, v2);
Node *right_lca = findLCAUtil(root->right, n1, n2, v1, v2);

// If both of the above calls return Non-NULL, then one node
// is present in one subtree and other is present in other,
// So this current node is the LCA
if (left_lca && right_lca) return root;

// Otherwise check if left subtree or right subtree is LCA
return (left_lca != NULL)? left_lca: right_lca;
}

// Returns true if key k is present in tree rooted with root
bool find(Node *root, int k)
{
    // Base Case
    if (root == NULL)
        return false;

    // If key k is present at root, or in left subtree
    // or right subtree, return true
    if (root->data == k || find(root->left, k) || find(root->right, k))
        return true;

    // Else return false
    return false;
}

// This function returns LCA of n1 and n2 only if both n1 and n2
// are present in tree, otherwise returns NULL
Node *findLCA(Node *root, int n1, int n2)
{
    // Initialize n1 and n2 as not visited
    bool v1 = false, v2 = false;

    // Find lca of n1 and n2
    Node *lca = findLCAUtil(root, n1, n2, v1, v2);

    // Return LCA only if both n1 and n2 are present in tree
    if (v1 && v2 || v1 && find(lca, n2) || v2 && find(lca, n1))
        return lca;
}
```

```
// Else return NULL
    return NULL;
}

// function returns true if
// there is a path from root to
// the given node. It also populates
// 'arr' with the given path
bool hasPath(Node *root, vector<int>& arr, int x)
{
    // if root is NULL
    // there is no path
    if (!root)
        return false;

    // push the node's value in 'arr'
    arr.push_back(root->data);

    // if it is the required node
    // return true
    if (root->data == x)
        return true;

    // else check whether there      the required node lies in the
    // left subtree or right subtree of the current node
    if (hasPath(root->left, arr, x) ||
        hasPath(root->right, arr, x))
        return true;

    // required node does not lie either in the
    // left or right subtree of the current node
    // Thus, remove current node's value from 'arr'
    // and then return false;
    arr.pop_back();
    return false;
}

// function to print the path common
// to the two paths from the root
// to the two given nodes if the nodes
// lie in the binary tree
void printCommonPath(Node *root, int n1, int n2)
{
    // vector to store the common path
    vector<int> arr;

    // LCA of node n1 and n2
```

Chapter 352. Print the path common to the two paths from the root to the two given nodes

```
Node *lca = findLCA(root, n1, n2);

// if LCA of both n1 and n2 exists
if (lca)
{
    // then print the path from root to
    // LCA node
    if (hasPath(root, arr, lca->data))
    {
        for (int i=0; i<arr.size()-1; i++)
            cout << arr[i] << "->";
        cout << arr[arr.size() - 1];
    }
}

// LCA is not present in the binary tree
// either n1 or n2 or both are not present
else
    cout << "No Common Path";
}

// Driver program to test above
int main()
{
    // binary tree formation
    struct Node *root = getNode(1);
    root->left = getNode(2);
    root->right = getNode(3);
    root->left->left = getNode(4);
    root->left->right = getNode(5);
    root->right->left = getNode(6);
    root->right->right = getNode(7);
    root->left->right->left = getNode(8);
    root->right->left->right = getNode(9);

    int n1 = 4, n2 = 8;
    printCommonPath(root, n1, n2);
    return 0;
}
```

Output:

1->2

Time complexity: O(n), where n is the number of nodes in the binary tree.

*Chapter 352. Print the path common to the two paths from the root to the two given nodes*

---

**Source**

<https://www.geeksforgeeks.org/print-path-common-two-paths-root-two-given-nodes/>

## Chapter 353

# Product of nodes at k-th level in a tree represented as string

Product of nodes at k-th level in a tree represented as string - GeeksforGeeks

Given an integer ‘K’ and a binary tree in string format. Every node of a tree has value in range from 0 to 9. We need to find product of elements at K-th level from root. The root is at level 0.

**Note :** Tree is given in the form: (node value(left subtree)(right subtree))

**Examples:**

```
Input : tree = "(0(5(6()())(4()(9()())))(7(1()())(3()())))"  
       k = 2  
Output : 72  
Its tree representation is shown below
```

```
Elements at level k = 2 are 6, 4, 1, 3  
sum of the digits of these elements = 6 * 4 * 1 * 3 = 72
```

```
Input : tree = "(8(3(2()())(6(5()())()))(5(10()())(7(13()())())))"  
       k = 3  
Output : 15  
Elements at level k = 3 are 5, 1 and 3  
sum of digits of these elements = 5 * 1 * 3 = 15
```

**Approach :**

1. Input 'tree' in string format and level k

2. Initialize level = -1 and product = 1
3. for each character 'ch' in 'tree'
  - 3.1 if ch == '(' then  
--> level++
  - 3.2 else if ch == ')' then  
--> level--
  - 3.3 else  
if level == k then  
product = product \* (ch-'0')
4. Print product

C++

```
// C++ implementation to find product of
// digits of elements at k-th level
#include <bits/stdc++.h>
using namespace std;

// Function to find product of digits
// of elements at k-th level
int productAtKthLevel(string tree, int k)
{
    int level = -1;
    int product = 1; // Initialize result
    int n = tree.length();

    for (int i = 0; i < n; i++) {
        // increasing level number
        if (tree[i] == '(')
            level++;

        // decreasing level number
        else if (tree[i] == ')')
            level--;

        else {
            // check if current level is
            // the desired level or not
            if (level == k)
                product *= (tree[i] - '0');
        }
    }

    // required product
    return product;
}

// Driver program
```

```
int main()
{
    string tree = "(0(5(6()())(4()(9()())))(7(1()())(3()())))";
    int k = 2;
    cout << productAtKthLevel(tree, k);
    return 0;
}
```

**Java**

```
// Java implementation to find product of
// digits of elements at k-th level

class GFG
{
    // Function to find product of digits
    // of elements at k-th level
    static int productAtKthLevel(String tree, int k)
    {
        int level = -1;

        // Initialize result
        int product = 1;

        int n = tree.length();

        for (int i = 0; i < n; i++)
        {
            // increasing level number
            if (tree.charAt(i) == '(')
                level++;

            // decreasing level number
            else if (tree.charAt(i) == ')')
                level--;

            else
            {
                // check if current level is
                // the desired level or not
                if (level == k)
                    product *= (tree.charAt(i) - '0');
            }
        }

        // required product
        return product;
    }
}
```

```
// Driver program
public static void main(String[] args)
{
    String tree = "(0(5(6()())(4()(9()())))(7(1()())(3()())))";
    int k = 2;
    System.out.println(productAtKthLevel(tree, k));
}
}

// This code is contributed
// by Smitha Dinesh Semwal.
```

### Python3

```
# Python 3 implementation
# to find product of
# digits of elements
# at k-th level

# Function to find
# product of digits
# of elements at
# k-th level
def productAtKthLevel(tree, k):

    level = -1

        # Initialize result
    product = 1
    n = len(tree)

    for i in range(0, n):

        # increasing level number
        if (tree[i] == '('):
            level+=1

        # decreasing level number
        elif (tree[i] == ')'):
            level-=1

        else:
            # check if current level is
            # the desired level or not
            if (level == k):
                product *= (int(tree[i]) - int('0'))
```

```
# required product
return product

# Driver program
tree = "(0(5(6()())(4()(9()())))(7(1()())(3()())))"
k = 2

print(productAtKthLevel(tree, k))

# This code is contributed by
# Smitha Dinesh Semwal
```

C#

```
// C# implementation to find
// product of digits of
// elements at k-th level
using System;

class GFG
{
    // Function to find product
    // of digits of elements
    // at k-th level
    static int productAtKthLevel(string tree,
                                  int k)
    {
        int level = -1;

        // Initialize result
        int product = 1;

        int n = tree.Length;

        for (int i = 0; i < n; i++)
        {
            // increasing
            // level number
            if (tree[i] == '(')
                level++;

            // decreasing
            // level number
            else if (tree[i] == ')')
                level--;
        }
    }
}
```

```
        else
        {
            // check if current level is
            // the desired level or not
            if (level == k)
                product *= (tree[i] - '0');
        }
    }

    // required product
    return product;
}

// Driver Code
static void Main()
{
    string tree = "(0(5(6()())(4()(9()())))(7(1()())(3()())))";
    int k = 2;
    Console.WriteLine(productAtKthLevel(tree, k));
}
}

// This code is contributed by Sam007
```

## PHP

```
<?php
// php implementation to find product of
// digits of elements at k-th level

// Function to find product of digits
// of elements at k-th level
function productAtKthLevel($tree, $k)
{
    $level = -1;
    $product = 1; // Initialize result
    $n = strlen($tree);

    for ($i = 0; $i < $n; $i++)
    {

        // increasing level number
        if ($tree[$i] == '(')
            $level++;

        // decreasing level number
        else if ($tree[$i] == ')')
```

```
$level--;

else
{
    // check if current level is
    // the desired level or not
    if ($level == $k)
        $product *= (ord($tree[$i]) -
                     ord('0'));

}

// required product
return $product;
}

// Driver Code
$tree = "(0(5(6()())(4()(9()())))(7(1()())(3()())))";
$k = 2;
echo productAtKthLevel($tree, $k);

//This code is contributed by mits
?>
```

**Output:**

72

**Time Complexity:** O(n)

**Improved By :** [Mithun Kumar](#), [Sam007](#)

**Source**

<https://www.geeksforgeeks.org/product-nodes-k-th-level-tree-represented-string/>

## Chapter 354

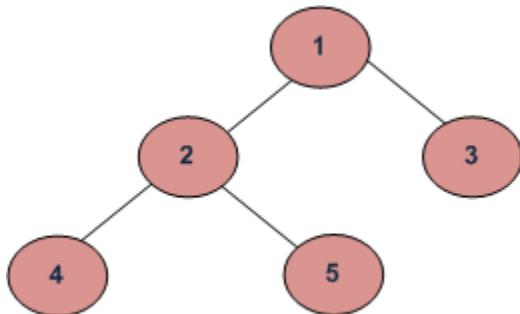
# Program to count leaf nodes in a binary tree

Program to count leaf nodes in a binary tree - GeeksforGeeks

A node is a leaf node if both left and right child nodes of it are NULL.

Here is an algorithm to get the leaf node count.

```
getLeafCount(node)
1) If node is NULL then return 0.
2) Else If left and right child nodes are NULL return 1.
3) Else recursively calculate leaf count of the tree using below formula.
   Leaf count of a tree = Leaf count of left subtree +
                           Leaf count of right subtree
```



Example Tree

Leaf count for the above tree is 3.

**Implementation:**

C

```
#include <stdio.h>
#include <stdlib.h>

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

/* Function to get the count of leaf nodes in a binary tree*/
unsigned int getLeafCount(struct node* node)
{
    if(node == NULL)
        return 0;
    if(node->left == NULL && node->right==NULL)
        return 1;
    else
        return getLeafCount(node->left)+getLeafCount(node->right);
}

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* node = (struct node*)
                           malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;

    return(node);
}

/*Driver program to test above functions*/
int main()
{
    /*create a tree*/
    struct node *root = newNode(1);
    root->left      = newNode(2);
    root->right     = newNode(3);
    root->left->left  = newNode(4);
    root->left->right = newNode(5);

    /*get leaf count of the above created tree*/
}
```

```
    printf("Leaf count of the tree is %d", getLeafCount(root));  
  
    getchar();  
    return 0;  
}
```

**Java**

```
//Java implementation to find leaf count of a given Binary tree  
  
/* Class containing left and right child of current  
node and key value*/  
class Node  
{  
    int data;  
    Node left, right;  
  
    public Node(int item)  
    {  
        data = item;  
        left = right = null;  
    }  
}  
  
public class BinaryTree  
{  
    //Root of the Binary Tree  
    Node root;  
  
    /* Function to get the count of leaf nodes in a binary tree*/  
    int getLeafCount()  
    {  
        return getLeafCount(root);  
    }  
  
    int getLeafCount(Node node)  
    {  
        if (node == null)  
            return 0;  
        if (node.left == null && node.right == null)  
            return 1;  
        else  
            return getLeafCount(node.left) + getLeafCount(node.right);  
    }  
  
    /* Driver program to test above functions */  
    public static void main(String args[])  
    {
```

```
/* create a tree */
BinaryTree tree = new BinaryTree();
tree.root = new Node(1);
tree.root.left = new Node(2);
tree.root.right = new Node(3);
tree.root.left.left = new Node(4);
tree.root.left.right = new Node(5);

/* get leaf count of the above tree */
System.out.println("The leaf count of binary tree is : "
+ tree.getLeafCount());
}

}

// This code has been contributed by Mayank Jaiswal(mayank_24)
```

### Python

```
# Python program to count leaf nodes in Binary Tree

# A Binary tree node
class Node:

    # Constructor to create a new node
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

    # Function to get the count of leaf nodes in binary tree
    def getLeafCount(node):
        if node is None:
            return 0
        if(node.left is None and node.right is None):
            return 1
        else:
            return getLeafCount(node.left) + getLeafCount(node.right)

# Driver program to test above function
root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)

print "Leaf count of the tree is %d" %(getLeafCount(root))
```

#This code is contributed by Nikhil Kumar Singh(nickzuck\_007)

**Time & Space Complexities:** Since this program is similar to traversal of tree, time and space complexities will be same as Tree traversal (Please see our [Tree Traversal](#) post for details)

## Source

<https://www.geeksforgeeks.org/write-a-c-program-to-get-count-of-leaf-nodes-in-a-binary-tree/>

## Chapter 355

# Prufer Code to Tree Creation

Prufer Code to Tree Creation - GeeksforGeeks

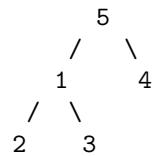
### What is Prufer Code?

Given a tree (represented as graph, not as a rooted tree) with n labeled nodes with labels from 1 to n, a Prufer code uniquely identifies the tree. The sequence has n-2 values.

### How to get Prufer Code of a tree?

1. Initialize Prufer code as empty.
2. Start with a leaf of lowest label say x. Find the vertex connecting it to the rest of tree say y. Remove x from the tree and add y to the Prufer Code
3. Repeat above step 2 until we are left with two nodes.

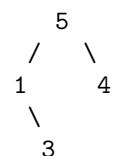
A tree with labels from 1 to n.



PruferCode = []

The lowest label leaf is 2, we remove it from tree  
and add the other vertex (connecting it to the tree)  
to Prufer code

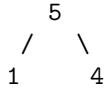
Tree now becomes



Prufer Code becomes = {1}

The lowest label leaf is 3, we remove it from tree  
and add the other vertex (connecting it to the tree)  
to Prufer code

Tree now becomes



Prufer Code becomes = {1, 1}

The lowest label leaf is 1, we remove it from tree  
and add the other vertex (connecting it to the tree)  
to Prufer code

Tree now becomes



Prufer Code becomes = {1, 1, 5}

We have only two nodes left now, so we stop.

### How to construct a tree from given Prufer Code?

Input : (4, 1, 3, 4)

Output : Edges of following tree  
 2----4----3----1----5  
     |  
     6

Input : (1, 3, 5)

Output : Edges of following tree  
 2----1----3----5----4

Let the length of given Prufer code be m. The idea is to create an empty graph of m+2 vertices. We remove first element from sequence. Let first element of current sequence be x. Then we find the least value which is not present in the given sequence and not yet added to the tree. Let this value be y. We add an edge from x to y and repeat this step.

Let us understand algorithm to construct tree with above first example:

Input : (4, 1, 3, 4)

Step 1: First we create an empty graph of 6 vertices  
and get 4 from the sequence.

Step 2: Out of 1 to 6, the least vertex not in  
Prufer sequence is 2.

Step 3: We form an edge between 2 and 4.

2----4    1    3    5    6

Step 4: Next in the sequence is 1 and corresponding vertex with least degree is 5 (as 2 has been considered).

2----4    1---5    3    6

Step 5: Next in the sequence is 3 and corresponding vertex with least degree is 1 (as 1 is now not part of remaining Prufer sequence)

2----4    3---1---5    6

Step 6: Next in the sequence is 4 and corresponding vertex with least degree is 3 (as 3 has not been considered as is not present further in sequence)

2----4----3---1---5    6

Step 7: Finally two vertices are left out from 1 to 6 (4 and 6) so we join them.

2----4----3---1---5  
|  
6

This is the required tree on 6 vertices.

Following is C++ implementation.

```
// C++ program to construct tree from given Prufer Code
#include<bits/stdc++.h>
using namespace std;

// Prints edges of tree represented by give Prufer code
void printTreeEdges(int prufer[], int m)
{
    int vertices = m + 2;
    int vertex_set[vertices];

    // Initialize the array of vertices
    for (int i=0; i<vertices; i++)
        vertex_set[i]=0;

    // Number of occurrences of vertex in code
    for (int i=0; i<vertices-2; i++)
        vertex_set[prufer[i]-1] += 1;

    cout<<"\nThe edge set E(G) is :\n";

    // Find the smallest label not present in
    // prufer[] .
    int j = 0;
    for (int i=0; i<vertices-2; i++)
    {
```

```

for (j=0; j<vertices; j++)
{
    // If j+1 is not present in prufer set
    if (vertex_set[j] == 0)
    {
        // Remove from Prufer set and print
        // pair.
        vertex_set[j] = -1;
        cout << "(" << (j+1) << "," 
            << prufer[i] << ")  ";
        vertex_set[prufer[i]-1]--;
        break;
    }
}
}

// For the last element
for (int i=0; i<vertices; i++)
{
    if (vertex_set[i] == 0 && j == 0 )
    {
        cout << "(" << (i+1) << ",";
        j++;
    }
    else if (vertex_set[i] == 0 && j == 1 )
        cout << (i+1) << ")\n";
}
}

// Driver code
int main()
{
    int prufer[] = {4, 1, 3, 4};
    int n = sizeof(prufer)/sizeof(prufer[0]);
    printTreeEdges(prufer, n);
    return 0;
}

```

Output:

The edge set E(G) is :  
(2,4) (5,1) (1,3) (3,4) (4,6)

**Source**

<https://www.geeksforgeeks.org/prufer-code-tree-creation/>

# Chapter 356

## Quad Tree

Quad Tree - GeeksforGeeks

Quadtrees are trees used to efficiently store data of points on a two-dimensional space. In this tree, each node has at most four children.

We can construct a quadtree from a two-dimensional area using the following steps:

1. Divide the current two dimensional space into four boxes.
2. If a box contains one or more points in it, create a child object, storing in it the two dimensional space of the box
3. If a box does not contain any points, do not create a child for it
4. Recurse for each of the children.

Quadtrees are used in image compression, where each node contains the average colour of each of its children. The deeper you traverse in the tree, the more the detail of the image. Quadtrees are also used in searching for nodes in a two-dimensional area. For instance, if you wanted to find the closest point to given coordinates, you can do it using quadtrees.

### Insert Function

The insert functions is used to insert a node into an existing Quad Tree. This function first checks whether the given node is within the boundaries of the current quad. If it is not, then we immediately cease the insertion. If it is within the boundaries, we select the appropriate child to contain this node based on its location.

This function is  $O(\log N)$  where  $N$  is the size of distance.

### Search Function

The search function is used to locate a node in the given quad. It can also be modified to return the closest node to the given point. This function is implemented by taking the given point, comparing with the boundaries of the child quads and recursing.

This function is  $O(\log N)$  where  $N$  is size of distance.

The program given below demonstrates storage of nodes in a quadtree.

```
// C++ Implementation of Quad Tree
```

```
#include <iostream>
#include <cmath>
using namespace std;

// Used to hold details of a point
struct Point
{
    int x;
    int y;
    Point(int _x, int _y)
    {
        x = _x;
        y = _y;
    }
    Point()
    {
        x = 0;
        y = 0;
    }
};

// The objects that we want stored in the quadtree
struct Node
{
    Point pos;
    int data;
    Node(Point _pos, int _data)
    {
        pos = _pos;
        data = _data;
    }
    Node()
    {
        data = 0;
    }
};

// The main quadtree class
class Quad
{
    // Hold details of the boundary of this node
    Point topLeft;
    Point botRight;

    // Contains details of node
    Node *n;

    // Children of this tree
};
```

```

Quad *topLeftTree;
Quad *topRightTree;
Quad *botLeftTree;
Quad *botRightTree;

public:
    Quad()
    {
        topLeft = Point(0, 0);
        botRight = Point(0, 0);
        n = NULL;
        topLeftTree = NULL;
        topRightTree = NULL;
        botLeftTree = NULL;
        botRightTree = NULL;
    }
    Quad(Point topL, Point botR)
    {
        n = NULL;
        topLeftTree = NULL;
        topRightTree = NULL;
        botLeftTree = NULL;
        botRightTree = NULL;
        topLeft = topL;
        botRight = botR;
    }
    void insert(Node*);
    Node* search(Point);
    bool inBoundary(Point);
};

// Insert a node into the quadtree
void Quad::insert(Node *node)
{
    if (node == NULL)
        return;

    // Current quad cannot contain it
    if (!inBoundary(node->pos))
        return;

    // We are at a quad of unit area
    // We cannot subdivide this quad further
    if (abs(topLeft.x - botRight.x) <= 1 &&
        abs(topLeft.y - botRight.y) <= 1)
    {
        if (n == NULL)
            n = node;
    }
}

```

```

        return;
    }

    if ((topLeft.x + botRight.x) / 2 >= node->pos.x)
    {
        // Indicates topLeftTree
        if ((topLeft.y + botRight.y) / 2 >= node->pos.y)
        {
            if (topLeftTree == NULL)
                topLeftTree = new Quad(
                    Point(topLeft.x, topLeft.y),
                    Point((topLeft.x + botRight.x) / 2,
                          (topLeft.y + botRight.y) / 2));
            topLeftTree->insert(node);
        }

        // Indicates botLeftTree
        else
        {
            if (botLeftTree == NULL)
                botLeftTree = new Quad(
                    Point(topLeft.x,
                          (topLeft.y + botRight.y) / 2),
                    Point((topLeft.x + botRight.x) / 2,
                          botRight.y));
            botLeftTree->insert(node);
        }
    }
    else
    {
        // Indicates topRightTree
        if ((topLeft.y + botRight.y) / 2 >= node->pos.y)
        {
            if (topRightTree == NULL)
                topRightTree = new Quad(
                    Point((topLeft.x + botRight.x) / 2,
                          topLeft.y),
                    Point(botRight.x,
                          (topLeft.y + botRight.y) / 2));
            topRightTree->insert(node);
        }

        // Indicates botRightTree
        else
        {
            if (botRightTree == NULL)
                botRightTree = new Quad(
                    Point((topLeft.x + botRight.x) / 2,

```

```

        (topLeft.y + botRight.y) / 2),
        Point(botRight.x, botRight.y));
    botRightTree->insert(node);
}
}

// Find a node in a quadtree
Node* Quad::search(Point p)
{
    // Current quad cannot contain it
    if (!inBoundary(p))
        return NULL;

    // We are at a quad of unit length
    // We cannot subdivide this quad further
    if (n != NULL)
        return n;

    if ((topLeft.x + botRight.x) / 2 >= p.x)
    {
        // Indicates topLeftTree
        if ((topLeft.y + botRight.y) / 2 >= p.y)
        {
            if (topLeftTree == NULL)
                return NULL;
            return topLeftTree->search(p);
        }

        // Indicates botLeftTree
        else
        {
            if (botLeftTree == NULL)
                return NULL;
            return botLeftTree->search(p);
        }
    }
    else
    {
        // Indicates topRightTree
        if ((topLeft.y + botRight.y) / 2 >= p.y)
        {
            if (topRightTree == NULL)
                return NULL;
            return topRightTree->search(p);
        }

        // Indicates botRightTree
    }
}

```

```
    else
    {
        if (botRightTree == NULL)
            return NULL;
        return botRightTree->search(p);
    }
};

// Check if current quadtree contains the point
bool Quad::inBoundary(Point p)
{
    return (p.x >= topLeft.x &&
            p.x <= botRight.x &&
            p.y >= topLeft.y &&
            p.y <= botRight.y);
}

// Driver program
int main()
{
    Quad center(Point(0, 0), Point(8, 8));
    Node a(Point(1, 1), 1);
    Node b(Point(2, 5), 2);
    Node c(Point(7, 6), 3);
    center.insert(&a);
    center.insert(&b);
    center.insert(&c);
    cout << "Node a: " <<
        center.search(Point(1, 1))->data << "\n";
    cout << "Node b: " <<
        center.search(Point(2, 5))->data << "\n";
    cout << "Node c: " <<
        center.search(Point(7, 6))->data << "\n";
    cout << "Non-existing node: "
        << center.search(Point(5, 5));
    return 0;
}
```

Output:

```
Node a: 1
Node b: 2
Node c: 3
Non-existing node: 0
```

Exercise:

Implement a Quad Tree which returns 4 closest nodes to a given point.

**Further References:**

<http://jimkang.com/quadtreenode/>  
<https://en.wikipedia.org/wiki/Quadtree>

**Source**

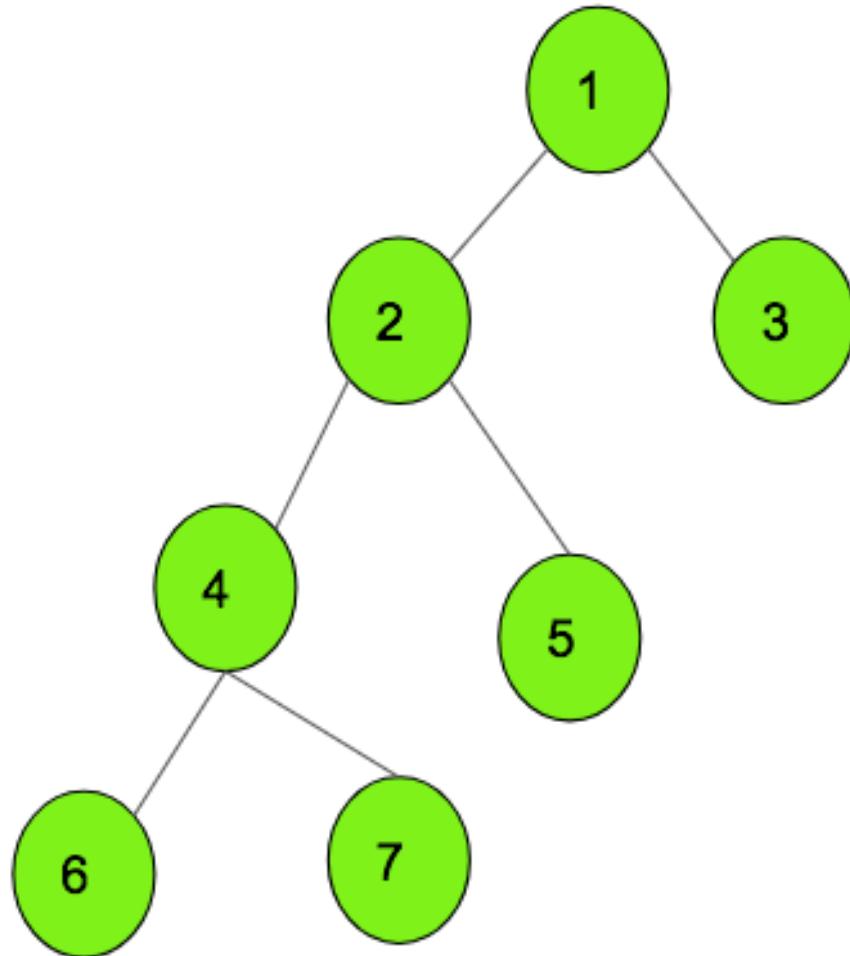
<https://www.geeksforgeeks.org/quad-tree/>

## Chapter 357

# Queries for DFS of a subtree in a tree

Queries for DFS of a subtree in a tree - GeeksforGeeks

Given a tree of N nodes and N-1 edges. The task is to print the DFS of the subtree of a given node for multiple queries. The DFS must include the given node as the root of the subtree.



In the above tree, if 1 is given as the node, then the DFS of subtree will be **1 2 4 6 7 5 3.**

If 2 is given as the node, then the DFS of the subtree will be **2 4 6 7 5..**

#### Approach:

- Add the edges between the nodes in an adjacency list.
- Call DFS function to generate the DFS of the complete tree.
- Use a `under[]` array to store the height of the subtree under the given node including the node.
- In the DFS function, keep incrementing the size of subtree on every recursive call.
- Mark the node index in the DFS of complete using hashing.
- The DFS of a subtree of a node will always be a contiguous subarray starting from the node(say `index ind`) to (`ind+height of subtree`).
- Get the index of node which has been stored using hashing and print the nodes from original DFS till `index = ind + height of subtree` which has been stored in `under[node]`.

Below is the implementation of the above approach.

```
// C++ program for Queries
// for DFS of subtree of a node in a tree
#include <bits/stdc++.h>
using namespace std;
const int N = 100000;

// Adjacency list to store the
// tree nodes connection
vector<int> v[N];

// stores the index of node in DFS
unordered_map<int, int> mp;

// stores the index of node in
// original node
vector<int> a;

// Function to call DFS and count nodes
// under that subtree
void dfs(int under[], int child, int parent)
{
    // stores the DFS of tree
    a.push_back(child);

    // height of subtree
    under[child] = 1;

    // iterate for children
    for (auto it : v[child]) {
        // if not equal to parent
        // so that it does not traverse back
        if (it != parent) {

            // call DFS for subtree
            dfs(under, it, child);

            // add the height
            under[child] += under[it];
        }
    }
}

// Function to print the DFS of subtree of nodec
void printDFSofSubtree(int node, int under[])
```

```
{  
    // index of node in the original DFS  
    int ind = mp[node];  
  
    // height of subtree of node  
    int height = under[node];  
  
    cout << "The DFS of subtree " << node << ": ";  
  
    // print the DFS of subtree  
    for (int i = ind; i < ind + under[node]; i++) {  
        cout << a[i] << " ";  
    }  
    cout << endl;  
}  
  
// Function to add edges to a tree  
void addEdge(int x, int y)  
{  
    v[x].push_back(y);  
    v[y].push_back(x);  
}  
  
// Marks the index of node in original DFS  
void markIndexDfs()  
{  
    int size = a.size();  
  
    // marks the index  
    for (int i = 0; i < size; i++) {  
        mp[a[i]] = i;  
    }  
}  
  
// Driver Code  
int main()  
{  
    int n = 7;  
  
    // add edges of a tree  
    addEdge(1, 2);  
    addEdge(1, 3);  
    addEdge(2, 4);  
    addEdge(2, 5);  
    addEdge(4, 6);  
    addEdge(4, 7);  
  
    // array to store the height of subtree
```

```
// of every node in a tree
int under[n + 1];

// Call the function DFS to generate the DFS
dfs(under, 1, 0);

// Function call to mark the index of node
markIndexDfs();

// Query 1
printDFSofSubtree(2, under);

// Query 1
printDFSofSubtree(4, under);

return 0;
}
```

**Time Complexity:**  $O(N + M)$ , where N is the number of nodes and M is the number of edges for pre-calculation and  $O(N)$  for queries in worst case.

**Auxiliary Space:**  $O(N)$

## Source

<https://www.geeksforgeeks.org/queries-for-dfs-of-a-subtree-in-a-tree/>

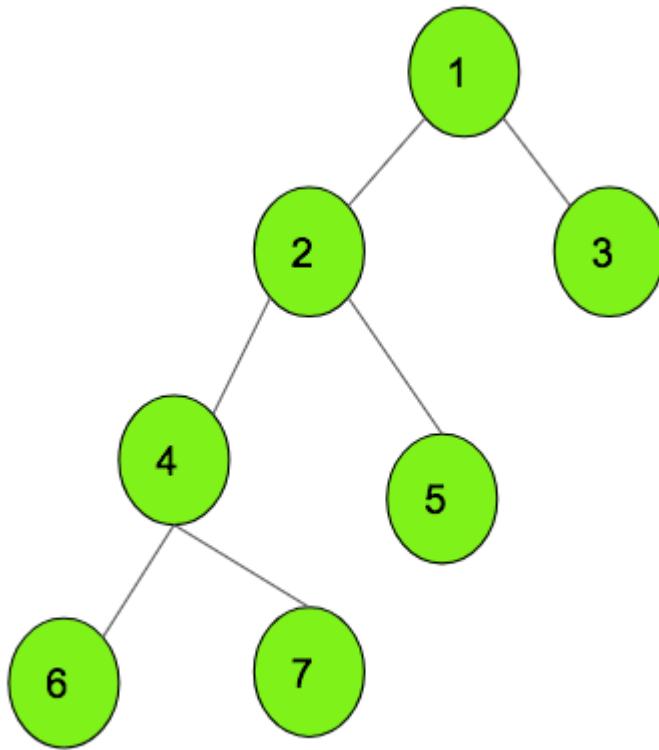
## Chapter 358

# Queries for M-th node in the DFS of subtree

Queries for M-th node in the DFS of subtree - GeeksforGeeks

Given a tree of N nodes and N-1 edges. Also given an integer M and a node, the task is to print the M-th node in the DFS of the subtree of a given node for multiple queries.

**Note:** M will not be greater than the number of nodes in the subtree of the given node.



**Input:** M = 3, node = 1

**Output:** 4

In the above example if 1 is given as the node, then the DFS of subtree will be **1 2 4 6 7 5 3**, hence if M is 3, then the 3rd node is 4

**Input:** M = 4, node = 2

**Output:** 7

If 2 is given as the node, then the DFS of the subtree will be **2 4 6 7 5.**, hence if M is 4 then the 4th node is 7.

#### Approach:

- Add the edges between the nodes in an adjacency list.
- Call [DFS function](#) to generate the DFS of the complete tree.
- Use an under[] array to store the height of the subtree under the given node including the node.
- In the DFS function, keep incrementing the size of subtree on every recursive call.
- Mark the node index in the DFS of complete using hashing.
- Let index of given node in the DFS of the tree be **ind**, then the M-th node will be at index **ind + M - 1** as the DFS of a subtree of a node will always be a contiguous subarray starting from the node.

Below is the implementation of the above approach.

```
// C++ program for Queries
// for DFS of subtree of a node in a tree
#include <bits/stdc++.h>
using namespace std;
const int N = 100000;

// Adjacency list to store the
// tree nodes connection
vector<int> v[N];

// stores the index of node in DFS
unordered_map<int, int> mp;

// stores the index of node in
// original node
vector<int> a;

// Function to call DFS and count nodes
// under that subtree
void dfs(int under[], int child, int parent)
{
    // stores the DFS of tree
    a.push_back(child);

    // height of subtree
    under[child] = 1;

    // iterate for children
    for (auto it : v[child]) {

        // if not equal to parent
        // so that it does not traverse back
        if (it != parent) {

            // call DFS for subtree
            dfs(under, it, child);

            // add the height
            under[child] += under[it];
        }
    }
}

// Function to return the DFS of subtree of nodec
int printnodeDFSofSubtree(int node, int under[], int m)
```

```
{  
    // index of node in the original DFS  
    int ind = mp[node];  
  
    // height of subtree of node  
    return a[ind + m - 1];  
}  
  
// Function to add edges to a tree  
void addEdge(int x, int y)  
{  
    v[x].push_back(y);  
    v[y].push_back(x);  
}  
  
// Marks the index of node in original DFS  
void markIndexDfs()  
{  
    int size = a.size();  
  
    // marks the index  
    for (int i = 0; i < size; i++) {  
        mp[a[i]] = i;  
    }  
}  
  
// Driver Code  
int main()  
{  
    int n = 7;  
  
    // add edges of a tree  
    addEdge(1, 2);  
    addEdge(1, 3);  
    addEdge(2, 4);  
    addEdge(2, 5);  
    addEdge(4, 6);  
    addEdge(4, 7);  
  
    // array to store the height of subtree  
    // of every node in a tree  
    int under[n + 1];  
  
    // Call the function DFS to generate the DFS  
    dfs(under, 1, 0);  
  
    // Function call to mark the index of node  
    markIndexDfs();
```

```
int m = 3;

// Query 1
cout << printnodeDFSofSubtree(1, under, m) << endl;

// Query 2
m = 4;
cout << printnodeDFSofSubtree(2, under, m);

return 0;
}
```

**Time Complexity:** O(1), for processing each query.

**Auxiliary Space:** O(N)

## Source

<https://www.geeksforgeeks.org/queries-for-m-th-node-in-the-dfs-of-subtree/>

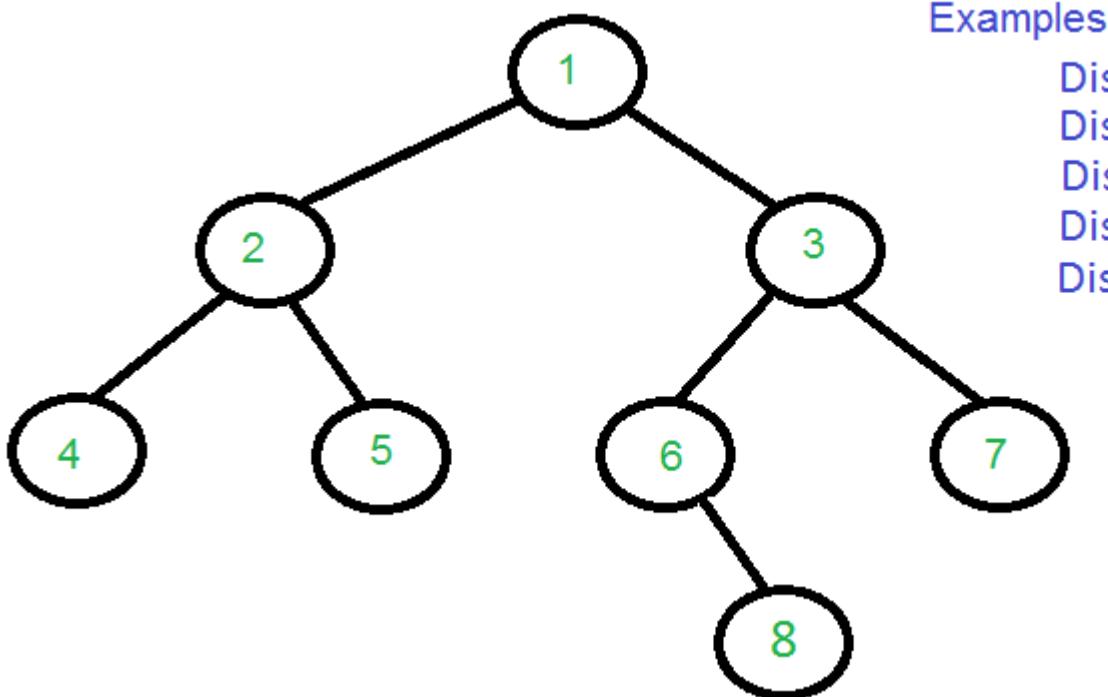
## Chapter 359

# Queries to find distance between two nodes of a Binary tree

Queries to find distance between two nodes of a Binary tree - GeeksforGeeks

Given a binary tree, the task is to find the distance between two keys in a binary tree, no parent pointers are given. The distance between two nodes is the minimum number of edges to be traversed to reach one node from other.

It has been already discussed in [this](#) for a single query in  $O(\log n)$  time, here the task is to reduce multiple queries time to  $O(1)$  by compromising with space complexity to  $O(N \log n)$ . In this post, we will use **Sparse table** instead of segment tree for finding the minimum in given range, which uses dynamic programming and bit manipulation to achieve  $O(1)$  query time.



A sparse table will preprocess the minimum values given for an array in  $N\log n$  space i.e. each node will contain chain of values of  $\log(i)$  length where  $i$  is the index of the  $i$ th node in  $L$  array. Each entry in the sparse table says  $M[i][j]$  will represent the index of the minimum value in the subarray starting at  $i$  having  $2^j$ .

The distance between two nodes can be obtained in terms of lowest common ancestor.

$$\text{Dist}(n_1, n_2) = \text{Level}[n_1] + \text{Level}[n_2] - 2 * \text{Level}[\text{lca}]$$

This problem can be breakdown into **finding levels of each node**, **finding the Euler tour of binary tree** and **building sparse table** for LCA, these steps are explained below :

1. Find the levels of each node by applying [level order traversal](#).
2. Find the LCA of two nodes in the binary tree in  $O(\log n)$  by Storing [Euler tour of tree](#) in array and computing two other arrays with the help of levels of each node and Euler tour.

These steps are shown below:

- (I) First, find [Euler Tour of binary tree](#).

Euler	1	2	4	2	5	2	1	3	6	8	6	3
	1	2	3	4	5	6	7	8	9	10	11	12
Euler tour of Binary Tree												

(II) Then, store levels of each node in Euler array.

L	0	1	2	1	2	1	0	1	2	3	2	1
	1	2	3	4	5	6	7	8	9	10	11	12
Levels of each node in Euler tour												

(III) Then, store First occurrences of all nodes of binary tree in Euler array.

H	1	2	8	3	5	9	13	10
	1	2	3	4	5	6	7	8
First occurrences of each node in Euler tree								

3. Then build sparse table on L array and find the minimum value say X in range ( $H[A]$  to  $H[B]$ ). Then use the index of value X as an index to Euler array to get LCA, i.e.  $Euler[\text{index}(X)]$ .

Let, A=8 and B=5.

(I)  $H[8]=1$  and  $H[5]=2$

(II) we get min value in L array between 1 and 2 as X=0, index=7

(III) Then, LCA= Euler[7], i.e LCA=1.

4. Finally, apply distance formula discussed above to get the distance between two nodes.

## Source

<https://www.geeksforgeeks.org/queries-find-distance-two-nodes-binary-tree/>

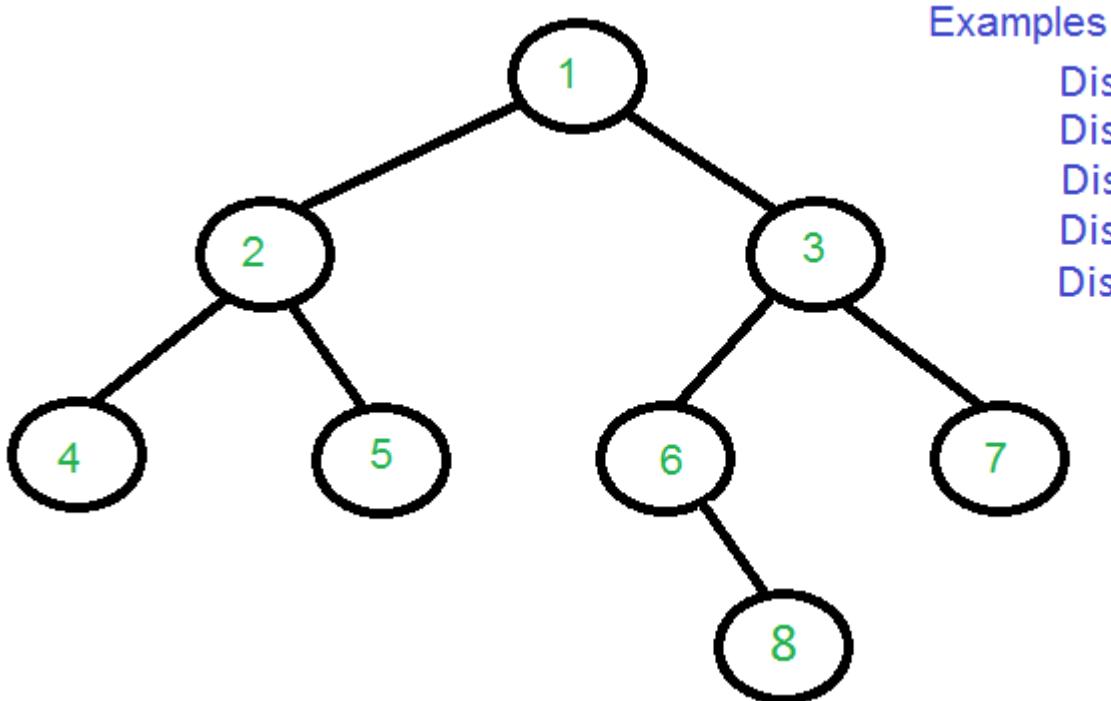
## Chapter 360

# Queries to find distance between two nodes of a Binary tree – O(logn) method

Queries to find distance between two nodes of a Binary tree - O(logn) method - GeeksforGeeks

Given a binary tree, the task is to find the distance between two keys in a binary tree, no parent pointers are given. Distance between two nodes is the minimum number of edges to be traversed to reach one node from other.

This problem has been already discussed in [previous post](#) but it uses **three traversals** of the Binary tree, one for finding Lowest Common Ancestor(LCA) of two nodes(let A and B) and then two traversals for finding distance between LCA and A and LCA and B which has  $O(n)$  time complexity. In this post, a method will be discussed that requires the  **$O(\log(n))$**  time to find LCA of two nodes.



The distance between two nodes can be obtained in terms of lowest common ancestor. Following is the formula.

$\text{Dist}(n_1, n_2) = \text{Dist}(\text{root}, n_1) + \text{Dist}(\text{root}, n_2) - 2 * \text{Dist}(\text{root}, \text{lca})$   
 'n1' and 'n2' are the two given keys  
 'root' is root of given Binary Tree.  
 'lca' is lowest common ancestor of n1 and n2  
 $\text{Dist}(n_1, n_2)$  is the distance between n1 and n2.

Above formula can also be written as:

$$\text{Dist}(n_1, n_2) = \text{Level}[n_1] + \text{Level}[n_2] - 2 * \text{Level}[\text{lca}]$$

This problem can be breakdown into:

1. Finding levels of each node
2. Finding the Euler tour of binary tree
3. Building segment tree for LCA,

These steps are explained below :

1. Find the levels of each node by applying [level order traversal](#).

2. Find the LCA of two nodes in binary tree in  $O(\log n)$  by Storing Euler tour of Binary tree in array and computing two other arrays with the help of levels of each node and Euler tour.

These steps are shown below:

- (I) First, find Euler Tour of binary tree.

Euler	1	2	4	2	5	2	1	3	6	8	6	
	1	2	3	4	5	6	7	8	9	10	11	
Euler tour of Binary Tree												

Euler tour of binary tree in example

- (II) Then, store levels of each node in Euler array.

L	0	1	2	1	2	1	0	1	2	3	2	
	1	2	3	4	5	6	7	8	9	10	11	
Levels of each node in Euler tour												

- (III) Then, store First occurrences of all nodes of binary tree in Euler array. H stores the indices of nodes from Euler array, so that range of query for finding minimum can be minimized and thereby further optimizing the query time.

H	1	2	8	3	5	9	13	10	
	1	2	3	4	5	6	7	8	
First occurrences of each node in Euler tree									

3. Then **build segment tree on L array** and take the low and high values from H array that will give us the first occurrences of say Two nodes(A and B). Then, we query segment tree to find the minimum value say X in range ( $H[A]$  to  $H[B]$ ). Then we use the index of value X as index to Euler array to get LCA, i.e. Euler[index(X)].

Let, A = 8 and B = 5.

- (I)  $H[8] = 1$  and  $H[5] = 2$

(II) Querying on Segment tree, we get min value in L array between 1 and 2 as X=0, index=7

- (III) Then, LCA= Euler[7], i.e LCA = 1.

4. Finally, we apply distance formula discussed above to get distance between two nodes.

```
// C++ program to find distance between
// two nodes for multiple queries
#include <bits/stdc++.h>
#define MAX 100001
using namespace std;

/* A tree node structure */
struct Node {
    int data;
    struct Node* left;
    struct Node* right;
};

/* Utility function to create a new Binary Tree node */
struct Node* newNode(int data)
{
    struct Node* temp = new struct Node;
    temp->data = data;
    temp->left = temp->right = NULL;
    return temp;
}

// Array to store level of each node
int level[MAX];

// Utility Function to store level of all nodes
void FindLevels(struct Node* root)
{
    if (!root)
        return;

    // queue to hold tree node with level
    queue<pair<struct Node*, int>> q;

    // let root node be at level 0
    q.push({root, 0});

    pair<struct Node*, int> p;

    // Do level Order Traversal of tree
    while (!q.empty()) {
        p = q.front();
        q.pop();

        // Node p.first is on level p.second
```

```
level[p.first->data] = p.second;

// If left child exists, put it in queue
// with current_level +1
if (p.first->left)
    q.push({ p.first->left, p.second + 1 });

// If right child exists, put it in queue
// with current_level +1
if (p.first->right)
    q.push({ p.first->right, p.second + 1 });
}

// Stores Euler Tour
int Euler[MAX];

// index in Euler array
int idx = 0;

// Find Euler Tour
void eulerTree(struct Node* root)
{
    // store current node's data
    Euler[++idx] = root->data;

    // If left node exists
    if (root->left) {

        // traverse left subtree
        eulerTree(root->left);

        // store parent node's data
        Euler[++idx] = root->data;
    }

    // If right node exists
    if (root->right) {
        // traverse right subtree
        eulerTree(root->right);

        // store parent node's data
        Euler[++idx] = root->data;
    }
}

// checks for visited nodes
```

```
int vis[MAX];

// Stores level of Euler Tour
int L[MAX];

// Stores indices of first occurrence
// of nodes in Euler tour
int H[MAX];

// Preprocessing Euler Tour for finding LCA
void preprocessEuler(int size)
{
    for (int i = 1; i <= size; i++) {
        L[i] = level[Euler[i]];

        // If node is not visited before
        if (vis[Euler[i]] == 0) {
            // Add to first occurrence
            H[Euler[i]] = i;

            // Mark it visited
            vis[Euler[i]] = 1;
        }
    }
}

// Stores values and positions
pair<int, int> seg[4 * MAX];

// Utility function to find minimum of
// pair type values
pair<int, int> min(pair<int, int> a,
                    pair<int, int> b)
{
    if (a.first <= b.first)
        return a;
    else
        return b;
}

// Utility function to build segment tree
pair<int, int> buildSegTree(int low, int high, int pos)
{
    if (low == high) {
        seg[pos].first = L[low];
        seg[pos].second = low;
        return seg[pos];
    }
}
```

```
int mid = low + (high - low) / 2;
buildSegTree(low, mid, 2 * pos);
buildSegTree(mid + 1, high, 2 * pos + 1);

seg[pos] = min(seg[2 * pos], seg[2 * pos + 1]);
}

// Utility function to find LCA
pair<int, int> LCA(int qlow, int qhigh, int low,
                     int high, int pos)
{
    if (qlow <= low && qhigh >= high)
        return {seg[pos], 0};

    if (qlow > high || qhigh < low)
        return {INT_MAX, 0};

    int mid = low + (high - low) / 2;

    return min(LCA(qlow, qhigh, low, mid, 2 * pos),
               LCA(qlow, qhigh, mid + 1, high, 2 * pos + 1));
}

// Function to return distance between
// two nodes n1 and n2
int findDistance(int n1, int n2, int size)
{
    // Maintain original Values
    int prevn1 = n1, prevn2 = n2;

    // Get First Occurrence of n1
    n1 = H[n1];

    // Get First Occurrence of n2
    n2 = H[n2];

    // Swap if low > high
    if (n2 < n1)
        swap(n1, n2);

    // Get position of minimum value
    int lca = LCA(n1, n2, 1, size, 1).second;

    // Extract value out of Euler tour
    lca = Euler[lca];

    // return calculated distance
    return level[prevn1] + level[prevn2] - 2 * level[lca];
}
```

```
}  
  
void preProcessing(Node* root, int N)  
{  
    // Build Tree  
    eulerTree(root);  
  
    // Store Levels  
    FindLevels(root);  
  
    // Find L and H array  
    preprocessEuler(2 * N - 1);  
  
    // Build segment Tree  
    buildSegTree(1, 2 * N - 1, 1);  
}  
  
/* Driver function to test above functions */  
int main()  
{  
    int N = 8; // Number of nodes  
  
    /* Constructing tree given in the above figure */  
    Node* root = newNode(1);  
    root->left = newNode(2);  
    root->right = newNode(3);  
    root->left->left = newNode(4);  
    root->left->right = newNode(5);  
    root->right->left = newNode(6);  
    root->right->right = newNode(7);  
    root->right->left->right = newNode(8);  
  
    // Function to do all preprocessing  
    preProcessing(root, N);  
  
    cout << "Dist(4, 5) = " << findDistance(4, 5, 2 * N - 1) << "\n";  
    cout << "Dist(4, 6) = " << findDistance(4, 6, 2 * N - 1) << "\n";  
    cout << "Dist(3, 4) = " << findDistance(3, 4, 2 * N - 1) << "\n";  
    cout << "Dist(2, 4) = " << findDistance(2, 4, 2 * N - 1) << "\n";  
    cout << "Dist(8, 5) = " << findDistance(8, 5, 2 * N - 1) << "\n";  
  
    return 0;  
}
```

**Output:**

```
Dist(4, 5) = 2
```

```
Dist(4, 6) = 4  
Dist(3, 4) = 3  
Dist(2, 4) = 1  
Dist(8, 5) = 5
```

Time Complexity:  $O(\log N)$

Space Complexity:  $O(N)$

[Queries to find distance between two nodes of a Binary tree – O\(1\) method](#)

## Source

<https://www.geeksforgeeks.org/queries-find-distance-two-nodes-binary-tree-ologn-method/>

## Chapter 361

# Query for ancestor-descendant relationship in a tree

Query for ancestor-descendant relationship in a tree - GeeksforGeeks

Given a rooted tree with N vertices and N-1 edges. We will be given many pairs of vertices u and v, we need to tell whether u is an ancestor of v or not. Given tree will be rooted at the vertex with index 0.

Examples:

```
u = 1      v = 6
we can see from above tree that node
1 is ancestor of node 6 so the answer
will be yes.
```

```
u = 1      v = 7
we can see from above tree that node 1
is not an ancestor of node 7 so the
answer will be no.
```

We can solve this problem using depth first search of the tree. While doing dfs we can observe a relation between the order in which we visit a node and its ancestors. If we assign in-time and out-time to each node when entering and leaving that node in dfs then we can see that for each pair of ancestor-descendant the in-time of ancestor is less than that of descendant and out-time of ancestor is more than that of descendant, so using this relation we can find the result for each pair of node in O(1) time.

So time complexity for preprocessing will be O(N) and for the query it will be O(1).

```
// C/C++ program to query whether two node has
// ancestor-descendant relationship or not
```

```
#include <bits/stdc++.h>
using namespace std;

// Utility dfs method to assign in and out time
// to each node
void dfs(vector<int> g[], int u, int parent,
          int timeIn[], int timeOut[], int& cnt)
{
    // assign In-time to node u
    timeIn[u] = cnt++;

    // call dfs over all neighbors except parent
    for (int i = 0; i < g[u].size(); i++) {
        int v = g[u][i];
        if (v != parent)
            dfs(g, v, u, timeIn, timeOut, cnt);
    }

    // assign Out-time to node u
    timeOut[u] = cnt++;
}

// method to preprocess all nodes for assigning time
void preProcess(int edges[][][2], int V, int timeIn[],
                int timeOut[])
{
    vector<int> g[V];

    // construct array of vector data structure
    // for tree
    for (int i = 0; i < V - 1; i++) {
        int u = edges[i][0];
        int v = edges[i][1];

        g[u].push_back(v);
        g[v].push_back(u);
    }

    int cnt = 0;

    // call dfs method from root
    dfs(g, 0, -1, timeIn, timeOut, cnt);
}

// method returns "yes" if u is a ancestor
// node of v
string isAncestor(int u, int v, int timeIn[],
                   int timeOut[])
```

```
{  
    bool b = (timeIn[u] <= timeIn[v] &&  
              timeOut[v] <= timeOut[u]);  
    return (b ? "yes" : "no");  
}  
  
// Driver code to test abovea methods  
int main()  
{  
    int edges[][] [2] = {  
        { 0, 1 },  
        { 0, 2 },  
        { 1, 3 },  
        { 1, 4 },  
        { 2, 5 },  
        { 4, 6 },  
        { 5, 7 }  
    };  
  
    int E = sizeof(edges) / sizeof(edges[0]);  
    int V = E + 1;  
  
    int timeIn[V], timeOut[V];  
    preprocess(edges, V, timeIn, timeOut);  
  
    int u = 1;  
    int v = 6;  
    cout << isAncestor(u, v, timeIn, timeOut) << endl;  
  
    u = 1;  
    v = 7;  
    cout << isAncestor(u, v, timeIn, timeOut) << endl;  
  
    return 0;  
}
```

Output:

```
yes  
no
```

## Source

<https://www.geeksforgeeks.org/query-ancestor-descendant-relationship-tree/>

## Chapter 362

# Range LCM Queries

Range LCM Queries - GeeksforGeeks

Given an array of integers, evaluate queries of the form  $\text{LCM}(l, r)$ . There might be many queries, hence evaluate the queries efficiently.

LCM ( $l, r$ ) denotes the LCM of array elements  
that lie between the index  $l$  and  $r$   
(inclusive of both indices)

Mathematically,  
 $\text{LCM}(l, r) = \text{LCM}(\text{arr}[l], \text{arr}[l+1], \dots, \text{arr}[r-1], \text{arr}[r])$

Examples:

```
Inputs : Array = {5, 7, 5, 2, 10, 12 ,11, 17, 14, 1, 44}
         Queries: LCM(2, 5), LCM(5, 10), LCM(0, 10)
Outputs: 60 15708 78540
Explanation : In the first query LCM(5, 2, 10, 12) = 60,
              similarly in other queries.
```

A naive solution would be to traverse the array for every query and calculate the answer by using,

$$\text{LCM}(a, b) = (a * b) / \text{GCD}(a, b)$$

However as the number of queries can be large, this solution would be impractical.

An efficient solution would be to use [segment tree](#). Recall that in this case, where no update is required, we can build the tree once and can use that repeatedly to answer the queries. Each node in the tree should store the LCM value for that particular segment and we can

use the same formula as above to combine the segments. Hence we can answer each query efficiently!

Below is a C++ solution for the same.

```
// LCM of given range queries using Segment Tree
#include <bits/stdc++.h>
using namespace std;

#define MAX 1000

// allocate space for tree
int tree[4*MAX];

// declaring the array globally
int arr[MAX];

// Function to return gcd of a and b
int gcd(int a, int b)
{
    if (a == 0)
        return b;
    return gcd(b%a, a);
}

//utility function to find lcm
int lcm(int a, int b)
{
    return a*b/gcd(a,b);
}

// Function to build the segment tree
// Node starts beginning index of current subtree.
// start and end are indexes in arr[] which is global
void build(int node, int start, int end)
{
    // If there is only one element in current subarray
    if (start==end)
    {
        tree[node] = arr[start];
        return;
    }

    int mid = (start+end)/2;

    // build left and right segments
    build(2*node, start, mid);
    build(2*node+1, mid+1, end);
```

```
// build the parent
int left_lcm = tree[2*node];
int right_lcm = tree[2*node+1];

tree[node] = lcm(left_lcm, right_lcm);
}

// Function to make queries for array range [l, r].
// Node is index of root of current segment in segment
// tree (Note that indexes in segment tree begin with 1
// for simplicity).
// start and end are indexes of subarray covered by root
// of current segment.
int query(int node, int start, int end, int l, int r)
{
    // Completely outside the segment, returning
    // 1 will not affect the lcm;
    if (end<l || start>r)
        return 1;

    // completely inside the segment
    if (l<=start && r>=end)
        return tree[node];

    // partially inside
    int mid = (start+end)/2;
    int left_lcm = query(2*node, start, mid, l, r);
    int right_lcm = query(2*node+1, mid+1, end, l, r);
    return lcm(left_lcm, right_lcm);
}

//driver function to check the above program
int main()
{
    //initialize the array
    arr[0] = 5;
    arr[1] = 7;
    arr[2] = 5;
    arr[3] = 2;
    arr[4] = 10;
    arr[5] = 12;
    arr[6] = 11;
    arr[7] = 17;
    arr[8] = 14;
    arr[9] = 1;
    arr[10] = 44;

    // build the segment tree
```

```
build(1, 0, 10);

// Now we can answer each query efficiently

// Print LCM of (2, 5)
cout << query(1, 0, 10, 2, 5) << endl;

// Print LCM of (5, 10)
cout << query(1, 0, 10, 5, 10) << endl;

// Print LCM of (0, 10)
cout << query(1, 0, 10, 0, 10) << endl;

return 0;
}
```

Output:

```
60
15708
78540
```

## Source

<https://www.geeksforgeeks.org/range-lcm-queries/>

## Chapter 363

# Range and Update Query for Chessboard Pieces

Range and Update Query for Chessboard Pieces - GeeksforGeeks

Given N pieces of chessboard all being ‘white’ and a number of queries Q. There are two types of queries :

1. **Update** : Given indices of a range [L, R]. Paint all the pieces with their respective opposite color between L and R (i.e. white pieces should be painted with black color and black pieces should be painted with white color).
2. **Get** : Given indices of a range [L, R]. Find out the number of black pieces between L and R.

Let us represent ‘white’ pieces with ‘0’ and ‘black’ pieces with ‘1’.

**Prerequisites:** [Segment Trees](#) | [Lazy Propagation](#)

Examples :

```
Input : N = 4, Q = 5
        Get : L = 0, R = 3
        Update : L = 1, R = 2
        Get : L = 0, R = 1
        Update : L = 0, R = 3
        Get : L = 0, R = 3
Output : 0
         1
         2
```

**Explanation :**

Query1 : A[] = { 0, 0, 0, 0 } Since initially all pieces are white, number of black pieces will

be zero.

Query2 :  $A[] = \{ 0, 1, 1, 0 \}$

Query3 : Number of black pieces in  $[0, 1] = 1$

Query4 : Change the color to its opposite color in  $[0, 3]$ ,  $A[] = \{ 1, 0, 0, 1 \}$

Query5 : Number of black pieces in  $[0, 3] = 2$

#### Naive Approach :

**Update(L, R)** : Iterate over the subarray from L to R and change the color of all the pieces (i.e. change 0 to 1 and 1 to 0)

**Get(L, R)** : To get the number of black pieces, simply count the number of ones in range  $[L, R]$ .

Both update and getBlackPieces() function will have  $O(N)$  time complexity. The time complexity in worst case is  $O(Q * N)$  where Q is number of queries and N is number of chessboard pieces.

#### Efficient Approach :

An efficient method to solve this problem is by using **Segment Trees** which can reduce the time complexity of update and getBlackPieces functions to  $O(\log N)$ .

**Build Structure:** Each leaf node of segment tree will contain either 0 or 1 depending upon the color of the piece (i.e. if the piece is black, node will contain 1 otherwise 0). Internal nodes will contain the sum of ones or number of black pieces of its left child and right child. Thus, the root node will give us the total number of black pieces in the whole array  $[0..N-1]$

**Update Structure :** Point updates takes  $O(\log(N))$  time but when there are range updates, optimize the updates using **Lazy Propagation**. Below is the modified update method.

```
UpdateRange(ss, se)
1. If current node's range lies completely in update query range.
...a) Value of current node becomes the difference of total count
    of black pieces in the subtree of current node and current
    value of node, i.e. tree[curNode] = (se - ss + 1) - tree[curNode]
...b) Provide the lazy value to its children by setting
    lazy[2*curNode] = 1 - lazy[2*curNode]
    lazy[2*curNode + 1] = 1 - lazy[2*curNode + 1]

2. If the current node's lazy value is not zero, first update
    it and provide lazy value to children.

3. Partial Overlap of current node's range with query range
...a) Recurse for left and right child
...b) Combine the results of step (a)
```

**Query Structure :** Query Structure will also change a bit in the same way as update structure by checking pending updates and updating them to get the correct query output.

Below is the implementation of above approach in C++.

```
// C code for queries on chessboard
#include <bits/stdc++.h>

using namespace std;

// A utility function to get the
// middle index from corner indexes.
int getMid(int s, int e)
{
    return s + (e - s) / 2;
}

/* A recursive function to get the
   sum of values in given range of
   the array. The following are
   parameters for this function.
si --> Index of current node in
      the segment tree. Initially
      0 is passed as root is always
      at index 0
ss & se --> Starting and ending
            indexes of the segment
            represented by current
            node, i.e., tree[si]
qs & qe --> Starting and ending
            indexes of query range */
int getSumUtil(int* tree, int* lazy, int ss,
               int se, int qs, int qe, int si)
{
    // If lazy flag is set for current node
    // of segment tree, then there are some
    // pending updates. So we need to make
    // sure that the pending updates are done
    // before processing the sub sum query
    if (lazy[si] != 0)
    {
        // Make pending updates to this node.
        // Note that this node represents
        // sum of elements in arr[ss..se]
        tree[si] = (se - ss + 1) - tree[si];

        // checking if it is not leaf node
        // because if it is leaf node then
        // we cannot go further
        if (ss != se)
        {
            // Since we are not yet updating
            // children os si, we need to set
```

```

// lazy values for the children
lazy[si * 2 + 1] =
    1 - lazy[si * 2 + 1];

lazy[si * 2 + 2] =
    1 - lazy[si * 2 + 2];
}

// unset the lazy value for current
// node as it has been updated
lazy[si] = 0;
}

// Out of range
if (ss > se || ss > qe || se < qs)
    return 0;

// At this point we are sure that pending
// lazy updates are done for current node.
// So we can return value (same as it was
// for query in our previous post)

// If this segment lies in range
if (ss >= qs && se <= qe)
    return tree[si];

// If a part of this segment overlaps
// with the given range
int mid = (ss + se) / 2;
return getSumUtil(tree, lazy, ss, mid,
                  qs, qe, 2 * si + 1) +
       getSumUtil(tree, lazy, mid + 1,
                  se, qs, qe, 2 * si + 2);
}

// Return sum of elements in range from index
// qs (query start) to qe (query end). It
// mainly uses getSumUtil()
int getSum(int* tree, int* lazy, int n,
           int qs, int qe)
{
    // Check for erroneous input values
    if (qs < 0 || qe > n - 1 || qs > qe)
    {
        printf("Invalid Input");
        return -1;
    }
}

```

```

        return getSumUtil(tree, lazy, 0, n - 1,
                           qs, qe, 0);
    }

/* si -> index of current node in segment tree
   ss and se -> Starting and ending indexes of
                  elements for which current
                  nodes stores sum.
   us and ue -> starting and ending indexes
                  of update query */
void updateRangeUtil(int* tree, int* lazy, int si,
                     int ss, int se, int us, int ue)
{
    // If lazy value is non-zero for current node
    // of segment tree, then there are some
    // pending updates. So we need to make sure that
    // the pending updates are done before making
    // new updates. Because this value may be used by
    // parent after recursive calls (See last line
    // of this function)
    if (lazy[si] != 0) {

        // Make pending updates using value stored
        // in lazy nodes
        tree[si] = (se - ss + 1) - tree[si];

        // checking if it is not leaf node because if
        // it is leaf node then we cannot go further
        if (ss != se)
        {
            // We can postpone updating children
            // we don't need their new values now.
            // Since we are not yet updating children
            // of si, we need to set lazy flags for
            // the children
            lazy[si * 2 + 1] = 1 - lazy[si * 2 + 1];
            lazy[si * 2 + 2] = 1 - lazy[si * 2 + 2];
        }

        // Set the lazy value for current node
        // as 0 as it has been updated
        lazy[si] = 0;
    }

    // out of range
    if (ss > se || ss > ue || se < us)
        return;
}

```

```

// Current segment is fully in range
if (ss >= us && se <= ue) {

    // Add the difference to current node
    tree[si] = (se - ss + 1) - tree[si];

    // same logic for checking leaf
    // node or not
    if (ss != se)
    {
        // This is where we store values in
        // lazy nodes, rather than updating
        // the segment tree itself. Since we
        // don't need these updated values now
        // we postpone updates by storing
        // values in lazy[]
        lazy[si * 2 + 1] = 1 - lazy[si * 2 + 1];
        lazy[si * 2 + 2] = 1 - lazy[si * 2 + 2];
    }
    return;
}

// If not completely in rang, but overlaps,
// recur for children
int mid = (ss + se) / 2;
updateRangeUtil(tree, lazy, si * 2 + 1,
                ss, mid, us, ue);
updateRangeUtil(tree, lazy, si * 2 + 2,
                mid + 1, se, us, ue);

// And use the result of children calls
// to update this node
tree[si] = tree[si * 2 + 1] + tree[si * 2 + 2];
}

// Function to update a range of values
// in segment tree
/* us and eu -> starting and ending indexes
   of update query ue -> ending index
   of update query, diff -> which we need
   to add in the range us to ue */
void updateRange(int* tree, int* lazy,
                 int n, int us, int ue)
{
    updateRangeUtil(tree, lazy, 0, 0, n - 1, us, ue);
}

// A recursive function that constructs

```

```

// Segment Tree for array[ss..se]. si is
// index of current node in segment tree st
int constructSTUtil(int arr[], int ss, int se,
                    int* tree, int si)
{
    // If there is one element in array, store
    // it in current node of segment tree and return
    if (ss == se)
    {
        tree[si] = arr[ss];
        return arr[ss];
    }

    // If there are more than one elements, then
    // recur for left and right subtrees and
    // store the sum of values in this node
    int mid = getMid(ss, se);
    tree[si] = constructSTUtil(arr, ss, mid,
                               tree, si * 2 + 1) +
               constructSTUtil(arr, mid + 1,
                               se, tree, si * 2 + 2);
    return tree[si];
}

/* Function to construct segment tree from
   given array. This function allocates
   memory for segment tree and calls
   constructSTUtil() to fill the
   allocated memory */
int* constructST(int arr[], int n)
{
    // Allocate memory for segment tree

    // Height of segment tree
    int x = (int)(ceil(log2(n)));

    // Maximum size of segment tree
    int max_size = 2 * (int)pow(2, x) - 1;

    // Allocate memory
    int* tree = new int[max_size];

    // Fill the allocated memory st
    constructSTUtil(arr, 0, n - 1, tree, 0);

    // Return the constructed segment tree
    return tree;
}

```

```
/* Function to construct lazy array for
   segment tree. This function allocates
   memory for lazy array */
int* constructLazy(int arr[], int n)
{
    // Allocate memory for lazy array

    // Height of lazy array
    int x = (int)(ceil(log2(n)));

    // Maximum size of lazy array
    int max_size = 2 * (int)pow(2, x) - 1;

    // Allocate memory
    int* lazy = new int[max_size];

    // Return the lazy array
    return lazy;
}

// Driver program to test above functions
int main()
{
    // Intialize the array to zero
    // since all pieces are white
    int arr[] = { 0, 0, 0, 0 };
    int n = sizeof(arr) / sizeof(arr[0]);

    // Build segment tree from given array
    int* tree = constructST(arr, n);

    // Allocate memory for Lazy array
    int* lazy = constructLazy(arr, n);

    // Print number of black pieces
    // from index 0 to 3
    cout << "Black Pieces in given range = "
        << getSum(tree, lazy, n, 0, 3) << endl;

    // UpdateRange: Change color of pieces
    // from index 1 to 2
    updateRange(tree, lazy, n, 1, 2);

    // Print number of black pieces
    // from index 0 to 1
    cout << "Black Pieces in given range = "
        << getSum(tree, lazy, n, 0, 1) << endl;
}
```

```
// UpdateRange: Change color of
// pieces from index 0 to 3
updateRange(tree, lazy, n, 0, 3);

// Print number of black pieces
// from index 0 to 3
cout << "Black Pieces in given range = "
    << getSum(tree, lazy, n, 0, 3) << endl;

return 0;
}
```

**Output:**

```
Black Pieces in given range = 0
Black Pieces in given range = 1
Black Pieces in given range = 2
```

**Time Complexity :** Each query and each update will take  $O(\log(N))$  time, where N is the number of chessboard pieces. Hence for Q queries, worst case complexity will be  $(Q * \log(N))$

**Source**

<https://www.geeksforgeeks.org/range-update-query-chessboard-pieces/>

## Chapter 364

# Range query for Largest Sum Contiguous Subarray

Range query for Largest Sum Contiguous Subarray - GeeksforGeeks

Given a number N, and Q queries of two types 1 and 2. Task is to write a code for the given query where, in type-1, given l and r, and task is to print the [Largest sum Contiguous Subarray](#) and for type 2, given type, index, and value, update value to  $A_{\text{index}}$ .

**Examples :**

```
Input : a = {-2, -3, 4, -1, -2, 1, 5, -3}
        1st query : 1 5 8
        2nd query : 2 1 10
        3rd query : 1 1 3
Output : Answer to 1st query : 6
Answer to 3rd query : 11
```

**Explanation :** In the first query, task is to print the largest sum of a contiguous subarray in range 5-8, which consists of  $\{-2, 1, 5, -3\}$ . The largest sum is 6, which is formed by the subarray  $\{1, 5\}$ . In the second query, an update operation is done, which updates  $a[1]$  to 10, hence the sequence is  $\{10, -3, 4, -1, -2, 1, 5, -3\}$ . In the third query, task is to print the largest sum of a contiguous subarray in range 1-3, which consists of  $\{10, -3, 4\}$ . The largest sum is 11, which is formed by the subarray  $\{10, -3, 4\}$ .

A **naive approach** is to use [Kadane's algorithm](#) for every type-1 query. The complexity of every type-1 query is  $O(n)$ . The type-2 query is done in  $O(1)$ .

**Efficient Approach :**

An efficient approach is to build a segment tree where each node stores four values(sum, prefixsum, suffixsum, maxsum), and do a range query on it to find the answer to every query. The nodes of segment tree store the four values as mentioned above. The parent will store the merging of left and right child. The parent node stores the value as mentioned below :

```

parent.sum = left.sum + right.sum
parent.prefixsum = max(left.prefixsum, left.sum + right.prefixsum)
parent.suffixsum = max(right.suffixsum, right.sum + left.suffixsum)
parent.maxsum = max(parent.prefixsum, parent.suffixsum, left.maxsum,
right.maxsum, left.suffixsum + right.prefixsum)

```

Parent node stores the following :

- Parent node's sum is the summation of left and right child sum.
- Parent node's prefix sum will be equivalent to maximum of left child's prefix sum or left child sum + right child prefix sum.
- Parent node's suffix sum will be equal to right child suffix sum or right child sum + suffix sum of left child
- Parent node's maxsum will be the maximum of prefixsum or suffix sum of parent or the left or right child's maxsum or the summation of suffixsum of left child and prefixsum of right child.

#### **Representation of Segment trees :**

1. Leaf Nodes are the elements of the input array.
2. Each internal node represents some merging of the leaf nodes. The merging may be different for different problems. For this problem, merging is done as given above.

An array representation of tree is used to represent Segment Trees. For each node at index  $i$ , the left child is at index  $2 * i + 1$ , right child at  $2 * i + 2$  and the parent is at  $(i - 1) / 2$ .

#### **Construction of Segment Tree from given array :**

Start with a segment arr[0 . . . n-1]. and every time divide the current segment into two halves(if it has not yet become a segment of length 1), and then call the same procedure on both halves, and for each such segment, store the values in all the four variables as given in the formulae above.

#### **Update a given value in array and segment Tree :**

Start with the complete segment of the array provided to us. Every time divide the array into two halves, **ignore the half in which the index to be updated is not present**. Keep on ignoring halves at every step until reach the leaf node, where update the value to the given index. Now, merge the updated values according to the given formulae to all the nodes that are present in the path we have traversed.

#### **Answering a query:**

For every query, move to the left and right halves of the tree. Whenever the given range completely overlaps any halve of a tree, return the Node from that half without traversing further in that region. When a halve of the tree completely lies outside the given range, return INT\_MIN. On partial overlapping of range, traverse in left and right halves and return accordingly.

Below is the implementation of the above idea :

```

// CPP program to find Largest Sum Contiguous
// Subarray in a given range with updates

```

```
#include <bits/stdc++.h>
using namespace std;

// Structure to store
// 4 values that are to be stored
// in the nodes
struct node {
    int sum, prefixsum, suffixsum, maxsum;
};

// array to store the segment tree
node tree[4 * 100];

// function to build the tree
void build(int arr[], int low, int high, int index)
{
    // the leaf node
    if (low == high) {
        tree[index].sum = arr[low];
        tree[index].prefixsum = arr[low];
        tree[index].suffixsum = arr[low];
        tree[index].maxsum = arr[low];
    }
    else {
        int mid = (low + high) / 2;

        // left subtree
        build(arr, low, mid, 2 * index + 1);

        // right subtree
        build(arr, mid + 1, high, 2 * index + 2);

        // parent node's sum is the summation
        // of left and right child
        tree[index].sum = tree[2 * index + 1].sum +
                         tree[2 * index + 2].sum;

        // parent node's prefix sum will be equivalent
        // to maximum of left child's prefix sum or left
        // child sum + right child prefix sum.
        tree[index].prefixsum =
            max(tree[2 * index + 1].prefixsum,
                tree[2 * index + 1].sum +
                tree[2 * index + 2].prefixsum);

        // parent node's suffix sum will be equal to right
        // child suffix sum or right child sum + suffix
        // sum of left child
    }
}
```

```

tree[index].suffixsum =
    max(tree[2 * index + 2].suffixsum,
        tree[2 * index + 2].sum +
        tree[2 * index + 1].suffixsum);

// maxum will be the maximum of prefix, suffix of
// parent or maximum of left child or right child
// and summation of left child's suffix and right
// child's prefix.
tree[index].maxsum =
    max(tree[index].prefixsum,
        max(tree[index].suffixsum,
            max(tree[2 * index + 1].maxsum,
                max(tree[2 * index + 2].maxsum,
                    tree[2 * index + 1].suffixsum +
                    tree[2 * index + 2].prefixsum))));

}

}

// function to update the tree
void update(int arr[], int index, int low, int high,
            int idx, int value)
{
    // the node to be updated
    if (low == high) {
        tree[index].sum = value;
        tree[index].prefixsum = value;
        tree[index].suffixsum = value;
        tree[index].maxsum = value;
    }
    else {

        int mid = (low + high) / 2;

        // if node to be updated is in left subtree
        if (idx <= mid)
            update(arr, 2 * index + 1, low, mid, idx, value);

        // if node to be updated is in right subtree
        else
            update(arr, 2 * index + 2, mid + 1,
                  high, idx, value);

        // parent node's sum is the summation of left
        // and rigth child
        tree[index].sum = tree[2 * index + 1].sum +
                        tree[2 * index + 2].sum;
    }
}

```

```

// parent node's prefix sum will be equivalent
// to maximum of left child's prefix sum or left
// child sum + right child prefix sum.
tree[index].prefixsum =
    max(tree[2 * index + 1].prefixsum,
        tree[2 * index + 1].sum +
        tree[2 * index + 2].prefixsum);

// parent node's suffix sum will be equal to right
// child suffix sum or right child sum + suffix
// sum of left child
tree[index].suffixsum =
    max(tree[2 * index + 2].suffixsum,
        tree[2 * index + 2].sum +
        tree[2 * index + 1].suffixsum);

// maxsum will be the maximum of prefix, suffix of
// parent or maximum of left child or right child
// and summation of left child's suffix and
// right child's prefix.
tree[index].maxsum =
    max(tree[index].prefixsum,
        max(tree[index].suffixsum,
            max(tree[2 * index + 1].maxsum,
                max(tree[2 * index + 2].maxsum,
                    tree[2 * index + 1].suffixsum +
                    tree[2 * index + 2].prefixsum))));

}

}

// function to return answer to every type-1 query
node query(int arr[], int index, int low,
           int high, int l, int r)
{
    // initially all the values are INT_MIN
    node result;
    result.sum = result.prefixsum =
        result.suffixsum =
        result.maxsum = INT_MIN;

    // range does not lies in this subtree
    if (r < low || high < l)
        return result;

    // complete overlap of range
    if (l <= low && high <= r)
        return tree[index];
}

```

```
int mid = (low + high) / 2;

// right subtree
if (l > mid)
    return query(arr, 2 * index + 2,
                 mid + 1, high, l, r);

// left subtree
if (r <= mid)
    return query(arr, 2 * index + 1,
                 low, mid, l, r);

node left = query(arr, 2 * index + 1,
                   low, mid, l, r);
node right = query(arr, 2 * index + 2,
                    mid + 1, high, l, r);

// finding the maximum and returning it
result.sum = left.sum + right.sum;
result.prefixsum = max(left.prefixsum, left.sum +
                      right.prefixsum);

result.suffixsum = max(right.suffixsum,
                      right.sum + left.suffixsum);
result.maxsum = max(result.prefixsum,
                     max(result.suffixsum,
                         max(left.maxsum,
                             max(right.maxsum,
                                 left.suffixsum + right.prefixsum))));

return result;
}

// Driver Code
int main()
{
    int a[] = { -2, -3, 4, -1, -2, 1, 5, -3 };
    int n = sizeof(a) / sizeof(a[0]);

    // build the tree
    build(a, 0, n - 1, 0);

    // 1st query type-1
    int l = 5, r = 8;
    cout << query(a, 0, 0, n - 1, l - 1, r - 1).maxsum;
    cout << endl;

    // 2nd type-2 query
```

```
int index = 1;
int value = 10;
a[index - 1] = value;
update(a, 0, 0, n - 1, index - 1, value);

// 3rd type-1 query
l = 1, r = 3;
cout << query(a, 0, 0, n - 1, l - 1, r - 1).maxsum;

return 0;
}
```

**Output:**

```
6
11
```

**Time Complexity :**  $O(n \log n)$  for building the tree,  $O(\log n)$  for every type-1 query,  $O(1)$  for type-2 query.

**Improved By :** [FelipeNoronha](#)

**Source**

<https://www.geeksforgeeks.org/range-query-largest-sum-contiguous-subarray/>

## Chapter 365

# Relationship between number of nodes and height of binary tree

Relationship between number of nodes and height of binary tree - GeeksforGeeks

Prerequisite – [Binary Tree Data Structure](#)

In this article, we will discuss various cases for relationship between number of nodes and height of binary tree. Before understanding this article, you should have basic idea about binary trees and their properties.

The **height** of the binary tree is the longest path from root **node to any leaf** node in the tree. For example, the height of binary tree shown in Figure 1(b) is 2 as longest path from root node to node 2 is 2. Also, the height of binary tree shown in Figure 1(a) is 4.

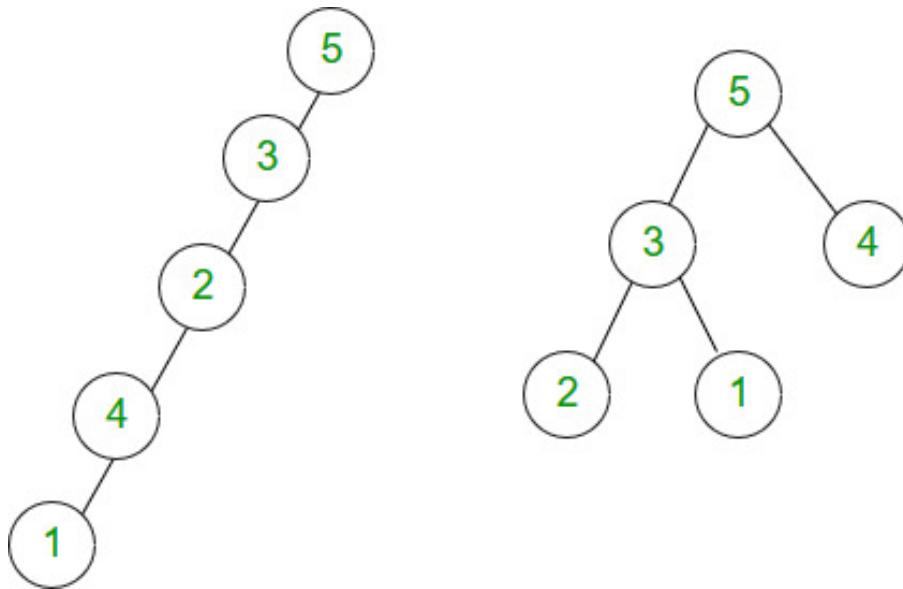
**Binary Tree –**

In a binary tree, a node can have maximum two children.

**Calculating minimum and maximum height from number of nodes –**

If there are  $n$  nodes in binary tree, **maximum height** of the binary tree is  $n-1$  and **minimum height** is  $\text{floor}(\log_2 n)$ .

For example, left skewed binary tree shown in Figure 1(a) with 5 nodes has height  $5-1 = 4$  and binary tree shown in Figure 1(b) with 5 nodes has height  $\text{floor}(\log_2 5) = 2$ .



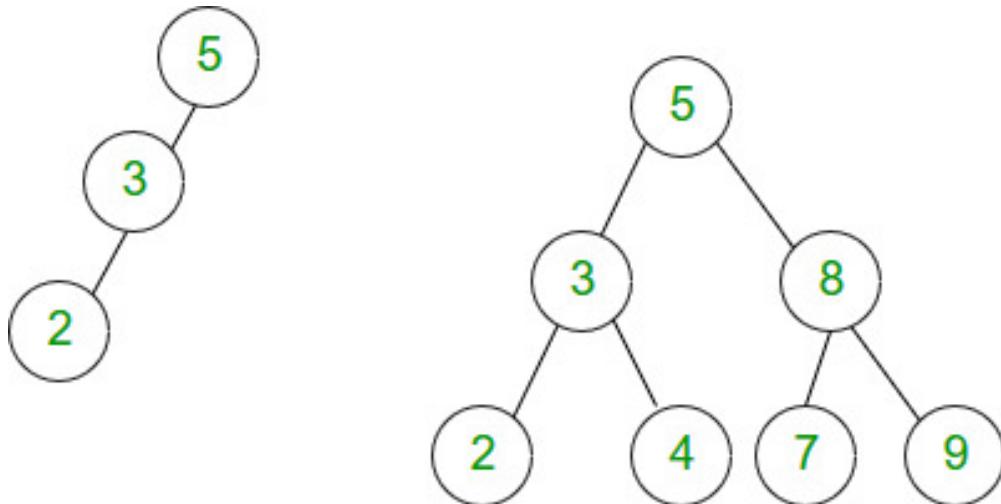
**Calculating minimum and maximum number of nodes from height –**

If binary tree has **height h**, minimum number of **nodes** is  $n+1$  (in case of left skewed and right skewed binary tree).

For example, the binary tree shown in Figure 2(a) with height 2 has 3 nodes.

If binary tree has height  $h$ , maximum number of nodes will be when all levels are completely full. Total number of nodes will be  $2^0 + 2^1 + \dots + 2^h = 2^{(h+1)-1}$ .

For example, the binary tree shown in Figure 2(b) with height 2 has  $2^{(2+1)-1} = 7$  nodes.



**Binary Search Tree –**

In a binary search tree, left child of a node has value less than the parent and right child has value greater than parent.

**Calculating minimum and maximum height from the number of nodes –**

If there are  $n$  nodes in a binary search tree, maximum height of the binary search tree is  $n-1$  and minimum height is  $\text{floor}(\log_2 n)$ .

**Calculating minimum and maximum number of nodes from height –**

If binary search tree has **height  $h$** , minimum number of **nodes is  $n+1$**  (in case of left skewed and right skewed binary search tree).

If binary search tree has **height  $h$** , maximum number of nodes will be when all levels are completely full. Total number of nodes will be  $2^0 + 2^1 + \dots + 2^h = 2^{h+1} - 1$ .

All the rules in BST are same as in binary tree and can be visualized in the same way.

**Que-1.** The height of a tree is the length of the longest root-to-leaf path in it. The maximum and the minimum number of nodes in a binary tree of height 5 are:

- (A) 63 and 6, respectively
- (B) 64 and 5, respectively
- (C) 32 and 6, respectively
- (D) 31 and 5, respectively

**Solution:** According to formula discussed,

$$\text{max number of nodes} = 2^{h+1} - 1 = 2^6 - 1 = 63.$$

$$\text{min number of nodes} = h+1 = 5+1 = 6.$$

**Que-2.** Which of the following height is not possible for a binary tree with 50 nodes?

- (A) 4
- (B) 5
- (C) 6
- (D) None

**Solution:** According to formula discussed,

$$\text{Minimum height with 50 nodes} = \text{floor}(\log_2 50) = 5. \text{ Therefore, height 4 is not possible.}$$

**Improved By :** [cssd1983](#)

## Source

<https://www.geeksforgeeks.org/relationship-number-nodes-height-binary-tree/>

## Chapter 366

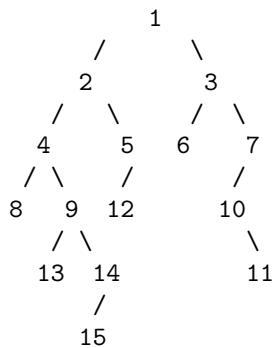
# Remove all nodes which don't lie in any path with sum $\geq k$

Remove all nodes which don't lie in any path with sum $\geq k$  - GeeksforGeeks

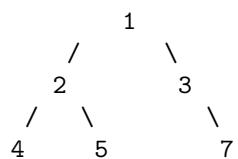
Given a binary tree, a complete path is defined as a path from root to a leaf. The sum of all nodes on that path is defined as the sum of that path. Given a number K, you have to remove (prune the tree) all nodes which don't lie in any path with sum $\geq k$ .

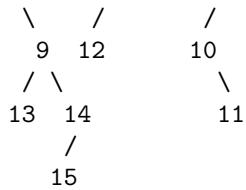
Note: A node can be part of multiple paths. So we have to delete it only in case when all paths from it have sum less than K.

Consider the following Binary Tree

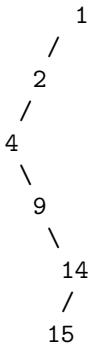


For input  $k = 20$ , the tree should be changed to following  
(Nodes with values 6 and 8 are deleted)





For input  $k = 45$ , the tree should be changed to following.



The idea is to traverse the tree and delete nodes in bottom up manner. While traversing the tree, recursively calculate the sum of nodes from root to leaf node of each path. For each visited node, checks the total calculated sum against given sum “ $k$ ”. If sum is less than  $k$ , then free(delete) that node (leaf node) and return the sum back to the previous node. Since the path is from root to leaf and nodes are deleted in bottom up manner, a node is deleted only when all of its descendants are deleted. Therefore, when a node is deleted, it must be a leaf in the current Binary Tree.

Following is the implementation of the above approach.

C++

```
#include <stdio.h>
#include <stdlib.h>

// A utility function to get maximum of two integers
int max(int l, int r) { return (l > r ? l : r); }

// A Binary Tree Node
struct Node
{
    int data;
    struct Node *left, *right;
};

// A utility function to create a new Binary Tree node with given data
```

```
struct Node* newNode(int data)
{
    struct Node* node = (struct Node*) malloc(sizeof(struct Node));
    node->data = data;
    node->left = node->right = NULL;
    return node;
}

// print the tree in LVR (Inorder traversal) way.
void print(struct Node *root)
{
    if (root != NULL)
    {
        print(root->left);
        printf("%d ",root->data);
        print(root->right);
    }
}

/* Main function which truncates the binary tree. */
struct Node *pruneUtil(struct Node *root, int k, int *sum)
{
    // Base Case
    if (root == NULL)  return NULL;

    // Initialize left and right sums as sum from root to
    // this node (including this node)
    int lsum = *sum + (root->data);
    int rsum = lsum;

    // Recursively prune left and right subtrees
    root->left = pruneUtil(root->left, k, &lsum);
    root->right = pruneUtil(root->right, k, &rsum);

    // Get the maximum of left and right sums
    *sum = max(lsum, rsum);

    // If maximum is smaller than k, then this node
    // must be deleted
    if (*sum < k)
    {
        free(root);
        root = NULL;
    }

    return root;
}
```

```
// A wrapper over pruneUtil()
struct Node *prune(struct Node *root, int k)
{
    int sum = 0;
    return pruneUtil(root, k, &sum);
}

// Driver program to test above function
int main()
{
    int k = 45;
    struct Node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->left = newNode(6);
    root->right->right = newNode(7);
    root->left->left->left = newNode(8);
    root->left->left->right = newNode(9);
    root->left->right->left = newNode(12);
    root->right->right->left = newNode(10);
    root->right->right->left->right = newNode(11);
    root->left->left->right->left = newNode(13);
    root->left->left->right->right = newNode(14);
    root->left->left->right->right->left = newNode(15);

    printf("Tree before truncation\n");
    print(root);

    root = prune(root, k); // k is 45

    printf("\n\nTree after truncation\n");
    print(root);

    return 0;
}
```

### Java

```
// Java program to implement
// the above approach
import java.util.*;
class GFG
{

// A utility function to get
// maximum of two integers
```

```
static int max(int l, int r)
{
    return (l > r ? l : r);
}

// A Binary Tree Node
static class Node
{
    int data;
    Node left, right;
};

static class INT
{
    int v;
    INT(int a)
    {
        v = a;
    }
}

// A utility function to create
// a new Binary Tree node with
// given data
static Node newNode(int data)
{
    Node node = new Node();
    node.data = data;
    node.left = node.right = null;
    return node;
}

// print the tree in LVR
// (Inorder traversal) way.
static void print(Node root)
{
    if (root != null)
    {
        print(root.left);
        System.out.print(root.data + " ");
        print(root.right);
    }
}

// Main function which
// truncates the binary tree.
static Node pruneUtil(Node root, int k,
                      INT sum)
```

```
{  
    // Base Case  
    if (root == null) return null;  
  
    // Initialize left and right  
    // sums as sum from root to  
    // this node (including this node)  
    INT lsum = new INT(sum.v + (root.data));  
    INT rsum = new INT(lsum.v);  
  
    // Recursively prune left  
    // and right subtrees  
    root.left = pruneUtil(root.left, k, lsum);  
    root.right = pruneUtil(root.right, k, rsum);  
  
    // Get the maximum of  
    // left and right sums  
    sum.v = max(lsum.v, rsum.v);  
  
    // If maximum is smaller  
    // than k, then this node  
    // must be deleted  
    if (sum.v < k)  
    {  
  
        root = null;  
    }  
  
    return root;  
}  
  
// A wrapper over pruneUtil()  
static Node prune(Node root, int k)  
{  
    INT sum = new INT(0);  
    return pruneUtil(root, k, sum);  
}  
  
// Driver Code  
public static void main(String args[])  
{  
    int k = 45;  
    Node root = newNode(1);  
    root.left = newNode(2);  
    root.right = newNode(3);  
    root.left.left = newNode(4);  
    root.left.right = newNode(5);  
    root.right.left = newNode(6);
```

```
root.right.right = newNode(7);
root.left.left.left = newNode(8);
root.left.left.right = newNode(9);
root.left.right.left = newNode(12);
root.right.right.left = newNode(10);
root.right.right.left.right = newNode(11);
root.left.left.right.left = newNode(13);
root.left.left.right.right = newNode(14);
root.left.left.right.right.left = newNode(15);

System.out.println("Tree before truncation\n");
print(root);

root = prune(root, k); // k is 45

System.out.println("\n\nTree after truncation\n");
print(root);
}

}

// This code is contributed by Arnab Kundu
```

**Output:**

```
Tree before truncation
8 4 13 9 15 14 2 12 5 1 6 3 10 11 7
```

```
Tree after truncation
4 9 15 14 2 1
```

**Time Complexity:** O(n), the solution does a single traversal of given Binary Tree.

**A Simpler Solution:**

The above code can be simplified using the fact that nodes are deleted in bottom up manner. The idea is to keep reducing the sum when traversing down. When we reach a leaf and sum is greater than the leaf's data, then we delete the leaf. Note that deleting nodes may convert a non-leaf node to a leaf node and if the data for the converted leaf node is less than the current sum, then the converted leaf should also be deleted.

Thanks to vicky for suggesting this solution in below comments.

**C**

```
#include <stdio.h>
#include <stdlib.h>

// A Binary Tree Node
struct Node
```

```
{  
    int data;  
    struct Node *left, *right;  
};  
  
// A utility function to create a new Binary  
// Tree node with given data  
struct Node* newNode(int data)  
{  
    struct Node* node =  
        (struct Node*) malloc(sizeof(struct Node));  
    node->data = data;  
    node->left = node->right = NULL;  
    return node;  
}  
  
// print the tree in LVR (Inorder traversal) way.  
void print(struct Node *root)  
{  
    if (root != NULL)  
    {  
        print(root->left);  
        printf("%d ",root->data);  
        print(root->right);  
    }  
}  
  
/* Main function which truncates the binary tree. */  
struct Node *prune(struct Node *root, int sum)  
{  
    // Base Case  
    if (root == NULL) return NULL;  
  
    // Recur for left and right subtrees  
    root->left = prune(root->left, sum - root->data);  
    root->right = prune(root->right, sum - root->data);  
  
    // If we reach leaf whose data is smaller than sum,  
    // we delete the leaf. An important thing to note  
    // is a non-leaf node can become leaf when its  
    // children are deleted.  
    if (root->left==NULL && root->right==NULL)  
    {  
        if (root->data < sum)  
        {  
            free(root);  
            return NULL;  
        }  
    }  
}
```

```
    }

    return root;
}

// Driver program to test above function
int main()
{
    int k = 45;
    struct Node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->left = newNode(6);
    root->right->right = newNode(7);
    root->left->left->left = newNode(8);
    root->left->left->right = newNode(9);
    root->left->right->left = newNode(12);
    root->right->right->left = newNode(10);
    root->right->right->right = newNode(11);
    root->left->left->right->left = newNode(13);
    root->left->left->right->right = newNode(14);
    root->left->left->right->right->left = newNode(15);

    printf("Tree before truncation\n");
    print(root);

    root = prune(root, k); // k is 45

    printf("\n\nTree after truncation\n");
    print(root);

    return 0;
}
```

### Java

```
// Java program to remove all nodes which do not
// lie on path having sum $\geq k$ 

// Class representing binary tree node
class Node {
    int data;
    Node left;
    Node right;

    // Constructor to create a new node
```

```
public Node(int data) {
    this.data = data;
    left = null;
    right = null;
}

// class to truncate binary tree
class BinaryTree {
    Node root;

    // recursive method to truncate binary tree
    public Node prune(Node root, int sum) {

        // base case
        if (root == null)
            return null;

        // recur for left and right subtree
        root.left = prune(root.left, sum - root.data);
        root.right = prune(root.right, sum - root.data);

        // if node is a leaf node whose data is smaller
        // than the sum we delete the leaf. An important
        // thing to note is a non-leaf node can become
        // leaf when its children are deleted.
        if (isLeaf(root)) {
            if (sum > root.data)
                root = null;
        }

        return root;
    }

    // utility method to check if node is leaf
    public boolean isLeaf(Node root) {
        if (root == null)
            return false;
        if (root.left == null && root.right == null)
            return true;
        return false;
    }

    // for print traversal
    public void print(Node root) {

        // base case
        if (root == null)
```

```
        return;

        print(root.left);
        System.out.print(root.data + " ");
        print(root.right);
    }
}

// Driver class to test above function
public class GFG {
    public static void main(String args[]) {

        BinaryTree tree = new BinaryTree();

        tree.root = new Node(1);
        tree.root.left = new Node(2);
        tree.root.right = new Node(3);
        tree.root.left.left = new Node(4);
        tree.root.left.right = new Node(5);
        tree.root.right.left = new Node(6);
        tree.root.right.right = new Node(7);
        tree.root.left.left.left = new Node(8);
        tree.root.left.left.right = new Node(9);
        tree.root.left.right.left = new Node(12);
        tree.root.right.right.left = new Node(10);
        tree.root.right.right.right = new Node(11);
        tree.root.left.left.right.left = new Node(13);
        tree.root.left.left.right.right = new Node(14);
        tree.root.left.left.right.right.left = new Node(15);

        System.out.println("Tree before truncation");
        tree.print(tree.root);

        tree.prune(tree.root, 45);

        System.out.println("\nTree after truncation");
        tree.print(tree.root);
    }
}

// This code is contributed by Shweta Singh
```

### Python3

```
"""
Python program to remove all nodes which don't
lie in any path with sum $\geq k$ 
"""
```

```
# binary tree node contains data field , left
# and right pointer
class Node:

    # constructor to create tree node
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

    # Function to remove all nodes which do not
    # lie in th sum path
    def prune(self, sum):

        # Base case
        if root is None:
            return None

        # Recur for left and right subtree
        root.left = prune(root.left, sum - root.data)
        root.right = prune(root.right, sum - root.data)

        # if node is leaf and sum is found greater
        # than data than remove node An important
        # thing to remember is that a non-leaf node
        # can become a leaf when its children are
        # removed
        if root.left is None and root.right is None:
            if sum > root.data:
                return None

        return root

    # inorder traversal
    def inorder(self):
        if root is None:
            return
        inorder(root.left)
        print(root.data, "", end="")
        inorder(root.right)

    # Driver program to test above function
    root = Node(1)
    root.left = Node(2)
    root.right = Node(3)
    root.left.left = Node(4)
    root.left.right = Node(5)
```

```
root.right.left = Node(6)
root.right.right = Node(7)
root.left.left.left = Node(8)
root.left.left.right = Node(9)
root.left.right.left = Node(12)
root.right.right.left = Node(10)
root.right.right.left.right = Node(11)
root.left.left.right.left = Node(13)
root.left.left.right.right = Node(14)
root.left.left.right.right.left = Node(15)

print("Tree before truncation")
inorder(root)
prune(root, 45)
print("\nTree after truncation")
inorder(root)

# This code is contributed by Shweta Singh
```

Output:

```
Tree before truncation
8 4 13 9 15 14 2 12 5 1 6 3 10 11 7
```

```
Tree after truncation
4 9 15 14 2 1
```

This article is contributed by **Chandra Prakash**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Improved By : [shweta44](#), [andrew1234](#)

## Source

<https://www.geeksforgeeks.org/remove-all-nodes-which-lie-on-a-path-having-sum-less-than-k/>

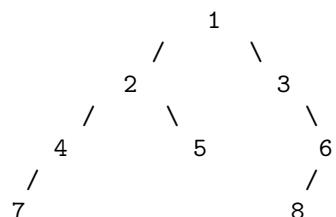
## Chapter 367

# Remove nodes on root to leaf paths of length < K

Remove nodes on root to leaf paths of length < K - GeeksforGeeks

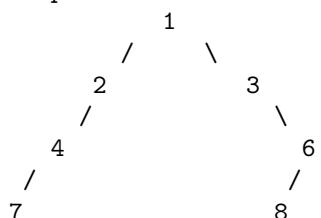
Given a Binary Tree and a number k, remove all nodes that lie only on root to leaf path(s) of length smaller than k. If a node X lies on multiple root-to-leaf paths and if any of the paths has path length  $\geq k$ , then X is not deleted from Binary Tree. In other words a node is deleted if all paths going through it have lengths smaller than k.

Consider the following example Binary Tree



Input: Root of above Binary Tree  
k = 4

Output: The tree should be changed to following



There are 3 paths  
i) 1->2->4->7      path length = 4

```
ii) 1->2->5      path length = 3
iii) 1->3->6->8   path length = 4
There is only one path " 1->2->5 " of length smaller than 4.
The node 5 is the only node that lies only on this path, so
node 5 is removed.
Nodes 2 and 1 are not removed as they are parts of other paths
of length 4 as well.
```

If k is 5 or greater than 5, then whole tree is deleted.

If k is 3 or less than 3, then nothing is deleted.

**We strongly recommend to minimize your browser and try this yourself first**

The idea here is to use post order traversal of the tree. Before removing a node we need to check that all the children of that node in the shorter path are already removed.

There are 2 cases:

- i) This node becomes a leaf node in which case it needs to be deleted.
- ii) This node has other child on a path with path length  $\geq k$ . In that case it needs not to be deleted.

The implementation of above approach is as below :

**C/C++**

```
// C++ program to remove nodes on root to leaf paths of length < K
#include<iostream>
using namespace std;

struct Node
{
    int data;
    Node *left, *right;
};

//New node of a tree
Node *newNode(int data)
{
    Node *node = new Node;
    node->data = data;
    node->left = node->right = NULL;
    return node;
}

// Utility method that actually removes the nodes which are not
// on the pathLen  $\geq k$ . This method can change the root as well.
Node *removeShortPathNodesUtil(Node *root, int level, int k)
{
```

```
//Base condition
if (root == NULL)
    return NULL;

// Traverse the tree in postorder fashion so that if a leaf
// node path length is shorter than k, then that node and
// all of its descendants till the node which are not
// on some other path are removed.
root->left = removeShortPathNodesUtil(root->left, level + 1, k);
root->right = removeShortPathNodesUtil(root->right, level + 1, k);

// If root is a leaf node and it's level is less than k then
// remove this node.
// This goes up and check for the ancestor nodes also for the
// same condition till it finds a node which is a part of other
// path(s) too.
if (root->left == NULL && root->right == NULL && level < k)
{
    delete root;
    return NULL;
}

// Return root;
return root;
}

// Method which calls the utility method to remove the short path
// nodes.
Node *removeShortPathNodes(Node *root, int k)
{
    int pathLen = 0;
    return removeShortPathNodesUtil(root, 1, k);
}

//Method to print the tree in inorder fashion.
void printInorder(Node *root)
{
    if (root)
    {
        printInorder(root->left);
        cout << root->data << " ";
        printInorder(root->right);
    }
}

// Driver method.
int main()
{
```

```
int k = 4;
Node *root = newNode(1);
root->left = newNode(2);
root->right = newNode(3);
root->left->left = newNode(4);
root->left->right = newNode(5);
root->left->left->left = newNode(7);
root->right->right = newNode(6);
root->right->right->left = newNode(8);
cout << "Inorder Traversal of Original tree" << endl;
printInorder(root);
cout << endl;
cout << "Inorder Traversal of Modified tree" << endl;
Node *res = removeShortPathNodes(root, k);
printInorder(res);
return 0;
}
```

### Java

```
// Java program to remove nodes on root to leaf paths of length < k

/* Class containing left and right child of current
   node and key value*/
class Node
{
    int data;
    Node left, right;

    public Node(int item)
    {
        data = item;
        left = right = null;
    }
}

class BinaryTree
{
    Node root;

    // Utility method that actually removes the nodes which are not
    // on the pathLen >= k. This method can change the root as well.
    Node removeShortPathNodesUtil(Node node, int level, int k)
    {
        //Base condition
        if (node == null)
            return null;
```

```
// Traverse the tree in postorder fashion so that if a leaf
// node path length is shorter than k, then that node and
// all of its descendants till the node which are not
// on some other path are removed.
node.left = removeShortPathNodesUtil(node.left, level + 1, k);
node.right = removeShortPathNodesUtil(node.right, level + 1, k);

// If root is a leaf node and it's level is less than k then
// remove this node.
// This goes up and check for the ancestor nodes also for the
// same condition till it finds a node which is a part of other
// path(s) too.
if (node.left == null && node.right == null && level < k)
    return null;

// Return root;
return node;
}

// Method which calls the utility method to remove the short path
// nodes.
Node removeShortPathNodes(Node node, int k)
{
    int pathLen = 0;
    return removeShortPathNodesUtil(node, 1, k);
}

//Method to print the tree in inorder fashion.
void printInorder(Node node)
{
    if (node != null)
    {
        printInorder(node.left);
        System.out.print(node.data + " ");
        printInorder(node.right);
    }
}

// Driver program to test for samples
public static void main(String args[])
{
    BinaryTree tree = new BinaryTree();
    int k = 4;
    tree.root = new Node(1);
    tree.root.left = new Node(2);
    tree.root.right = new Node(3);
    tree.root.left.left = new Node(4);
    tree.root.left.right = new Node(5);
```

```
tree.root.left.left.left = new Node(7);
tree.root.right.right = new Node(6);
tree.root.right.right.left = new Node(8);
System.out.println("The inorder traversal of original tree is : ");
tree.printInorder(tree.root);
Node res = tree.removeShortPathNodes(tree.root, k);
System.out.println("");
System.out.println("The inorder traversal of modified tree is : ");
tree.printInorder(res);
}
}

// This code has been contributed by Mayank Jaiswal
```

Output:

```
Inorder Traversal of Original tree
7 4 2 5 1 3 8 6
Inorder Traversal of Modified tree
7 4 2 1 3 8 6
```

Time complexity of the above solution is  $O(n)$  where  $n$  is number of nodes in given Binary Tree.

This article is contributed by **Kumar Gautam**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## Source

<https://www.geeksforgeeks.org/remove-nodes-root-leaf-paths-length-k/>

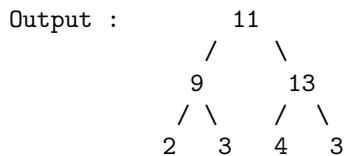
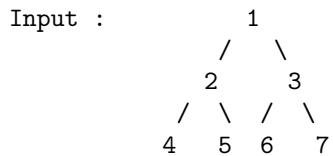
## Chapter 368

# Replace each node in binary tree with the sum of its inorder predecessor and successor

Replace each node in binary tree with the sum of its inorder predecessor and successor - GeeksforGeeks

Given a binary tree containing  $n$  nodes. The problem is to replace each node in the binary tree with the sum of its inorder predecessor and inorder successor.

Examples:



For 1:  
Inorder predecessor = 5  
Inorder successor = 6  
Sum = 11

For 4:  
Inorder predecessor = 0

```
(as inorder predecessor is not present)
Inorder successor = 2
Sum = 2

For 7:
Inorder predecessor = 3
Inorder successor = 0
(as inorder successor is not present)
Sum = 3
```

**Approach:** Create an array **arr**. Store **0** at index 0. Now, store the inorder traversal of tree in the array **arr**. Then, store **0** at last index. **0**'s are stored as inorder predecessor of leftmost leaf and inorder successor of rightmost leaf is not present. Now, perform inorder traversal and while traversing node replace node's value with **arr[i-1] + arr[i+1]** and then increment **i**. In the beginning initialize **i = 1**. For an element **arr[i]**, the values **arr[i-1]** and **arr[i+1]** are its inorder predecessor and inorder successor respectively.

```
// C++ implementation to replace each node
// in binary tree with the sum of its inorder
// predecessor and successor
#include <bits/stdc++.h>

using namespace std;

// node of a binary tree
struct Node {
    int data;
    struct Node* left, *right;
};

// function to get a new node of a binary tree
struct Node* getNode(int data)
{
    // allocate node
    struct Node* new_node =
        (struct Node*)malloc(sizeof(struct Node));

    // put in the data;
    new_node->data = data;
    new_node->left = new_node->right = NULL;

    return new_node;
}

// function to store the inorder traversal
// of the binary tree in 'arr'
void storeInorderTraversal(struct Node* root,
                           vector<int>& arr)
```

```
{  
    // if root is NULL  
    if (!root)  
        return;  
  
    // first recur on left child  
    storeInorderTraversal(root->left, arr);  
  
    // then store the root's data in 'arr'  
    arr.push_back(root->data);  
  
    // now recur on right child  
    storeInorderTraversal(root->right, arr);  
}  
  
// function to replace each node with the sum of its  
// inorder predecessor and successor  
void replaceNodeWithSum(struct Node* root,  
                      vector<int> arr, int* i)  
{  
    // if root is NULL  
    if (!root)  
        return;  
  
    // first recur on left child  
    replaceNodeWithSum(root->left, arr, i);  
  
    // replace node's data with the sum of its  
    // inorder predecessor and successor  
    root->data = arr[*i - 1] + arr[*i + 1];  
  
    // move 'i' to point to the next 'arr' element  
    ++*i;  
  
    // now recur on right child  
    replaceNodeWithSum(root->right, arr, i);  
}  
  
// Utility function to replace each node in binary  
// tree with the sum of its inorder predecessor  
// and successor  
void replaceNodeWithSumUtil(struct Node* root)  
{  
    // if tree is empty  
    if (!root)  
        return;  
  
    vector<int> arr;
```

```
// store the value of inorder predecessor
// for the leftmost leaf
arr.push_back(0);

// store the inoder traversal of the tree in 'arr'
storeInorderTraversal(root, arr);

// store the value of inorder successor
// for the rightmost leaf
arr.push_back(0);

// replace each node with the required sum
int i = 1;
replaceNodeWithSum(root, arr, &i);
}

// function to print the preorder traversal
// of a binary tree
void preorderTraversal(struct Node* root)
{
    // if root is NULL
    if (!root)
        return;

    // first print the data of node
    cout << root->data << " ";

    // then recur on left subtree
    preorderTraversal(root->left);

    // now recur on right subtree
    preorderTraversal(root->right);
}

// Driver program to test above
int main()
{
    // binary tree formation
    struct Node* root = getNode(1); /*      1      */
    root->left = getNode(2); /*      / \      */
    root->right = getNode(3); /*      2   3      */
    root->left->left = getNode(4); /*      / \ / \      */
    root->left->right = getNode(5); /*      4   5   6   7      */
    root->right->left = getNode(6);
    root->right->right = getNode(7);

    cout << "Preorder Traversal before tree modification:n";
}
```

```
preorderTraversal(root);

replaceNodeWithSumUtil(root);

cout << "\nPreorder Traversal after tree modification:n";
preorderTraversal(root);

return 0;
}
```

Output:

```
Preorder Traversal before tree modification:
1 2 4 5 3 6 7
Preorder Traversal after tree modification:
11 9 2 3 13 4 3
```

Time Complexity: O(n)  
Auxiliary Space: O(n)

## Source

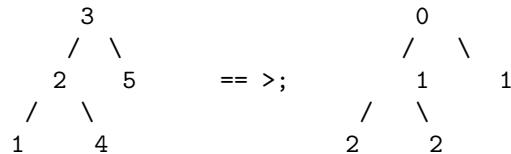
<https://www.geeksforgeeks.org/replace-node-binary-tree-sum-inorder-predecessor-successor/>

## Chapter 369

# Replace node with depth in a binary tree

Replace node with depth in a binary tree - GeeksforGeeks

Given a binary tree, replace each node with its depth value. For example, consider the following tree. Root is at depth 0, change its value to 0 and next level nodes are at depth 1 and so on.



The idea is to traverse tree starting from root. While traversing pass depth of node as parameter. We can track depth by passing it as 0 for root and one-plus-current-depth for children.

Below is C++ implementation of the idea.

```
// CPP program to replace every key value
// with its depth.
#include<bits/stdc++.h>
using namespace std;

/* A tree node structure */
struct Node
{
    int data;
```

```
    struct Node *left, *right;
};

/* Utility function to create a
   new Binary Tree node */
struct Node* newNode(int data)
{
    Node *temp = new Node;
    temp->data = data;
    temp->left = temp->right = NULL;
    return temp;
}

// Helper function replaces the data with depth
// Note : Default value of level is 0 for root.
void replaceNode(struct Node *node, int level=0)
{
    // Base Case
    if (node == NULL)
        return;

    // Replace data with current depth
    node->data = level;

    replaceNode(node->left, level+1);
    replaceNode(node->right, level+1);
}

// A utility function to print inorder
// traversal of a Binary Tree
void printInorder(struct Node* node)
{
    if (node == NULL)
        return;
    printInorder(node->left);
    cout << node->data << " ";
    printInorder(node->right);
}

/* Driver function to test above functions */
int main()
{
    struct Node *root = new struct Node;

    /* Constructing tree given in
       the above figure */
    root = newNode(3);
    root->left = newNode(2);
```

```
root->right = newNode(5);
root->left->left = newNode(1);
root->left->right = newNode(4);

cout << "Before Replacing Nodes\n";
printInorder(root);
replaceNode(root);
cout << endl;

cout << "After Replacing Nodes\n";
printInorder(root);

return 0;
}
```

Output:

```
Before Replacing Nodes
1 2 4 3 5
```

```
After Replacing Nodes
2 1 2 0 1
```

## Source

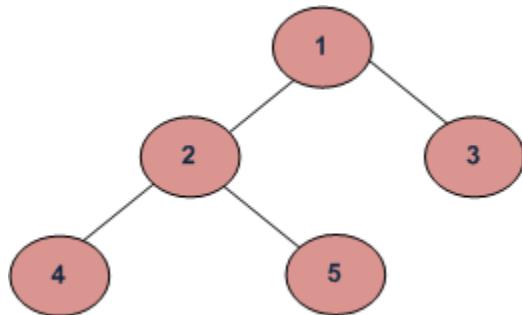
<https://www.geeksforgeeks.org/replace-node-with-depth-in-a-binary-tree/>

## Chapter 370

# Reverse Level Order Traversal

Reverse Level Order Traversal - GeeksforGeeks

We have discussed [level order traversal](#) of a post in previous post. The idea is to print last level first, then second last level, and so on. Like Level order traversal, every level is printed from left to right.



Example Tree

Reverse Level order traversal of the above tree is “4 5 2 3 1”.

Both methods for [normal level order traversal](#) can be easily modified to do reverse level order traversal.

### METHOD 1 (Recursive function to print a given level)

We can easily modify the method 1 of the normal [level order traversal](#). In method 1, we have a method `printGivenLevel()` which prints a given level number. The only thing we need to change is, instead of calling `printGivenLevel()` from first level to last level, we call it from last level to first level.

C

```
// A recursive C program to print REVERSE level order traversal
#include <stdio.h>
#include <stdlib.h>
```

```

/* A binary tree node has data, pointer to left and right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

/*Function prototypes*/
void printGivenLevel(struct node* root, int level);
int height(struct node* node);
struct node* newNode(int data);

/* Function to print REVERSE level order traversal a tree*/
void reverseLevelOrder(struct node* root)
{
    int h = height(root);
    int i;
    for (i=h; i>=1; i--) //THE ONLY LINE DIFFERENT FROM NORMAL LEVEL ORDER
        printGivenLevel(root, i);
}

/* Print nodes at a given level */
void printGivenLevel(struct node* root, int level)
{
    if (root == NULL)
        return;
    if (level == 1)
        printf("%d ", root->data);
    else if (level > 1)
    {
        printGivenLevel(root->left, level-1);
        printGivenLevel(root->right, level-1);
    }
}

/* Compute the "height" of a tree -- the number of
   nodes along the longest path from the root node
   down to the farthest leaf node.*/
int height(struct node* node)
{
    if (node==NULL)
        return 0;
    else
    {
        /* compute the height of each subtree */
        int lheight = height(node->left);

```

```

        int rheight = height(node->right);

        /* use the larger one */
        if (lheight > rheight)
            return(lheight+1);
        else return(rheight+1);
    }
}

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* node = (struct node*)
                           malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;

    return(node);
}

/* Driver program to test above functions*/
int main()
{
    struct node *root = newNode(1);
    root->left      = newNode(2);
    root->right     = newNode(3);
    root->left->left  = newNode(4);
    root->left->right = newNode(5);

    printf("Level Order traversal of binary tree is \n");
    reverseLevelOrder(root);

    return 0;
}

```

**Java**

```

// A recursive java program to print reverse level order traversal

// A binary tree node
class Node
{
    int data;
    Node left, right;

    Node(int item)

```

```

    {
        data = item;
        left = right;
    }
}

class BinaryTree
{
    Node root;

    /* Function to print REVERSE level order traversal a tree*/
    void reverseLevelOrder(Node node)
    {
        int h = height(node);
        int i;
        for (i = h; i >= 1; i--)
        //THE ONLY LINE DIFFERENT FROM NORMAL LEVEL ORDER
        {
            printGivenLevel(node, i);
        }
    }

    /* Print nodes at a given level */
    void printGivenLevel(Node node, int level)
    {
        if (node == null)
            return;
        if (level == 1)
            System.out.print(node.data + " ");
        else if (level > 1)
        {
            printGivenLevel(node.left, level - 1);
            printGivenLevel(node.right, level - 1);
        }
    }

    /* Compute the "height" of a tree -- the number of
       nodes along the longest path from the root node
       down to the farthest leaf node.*/
    int height(Node node)
    {
        if (node == null)
            return 0;
        else
        {
            /* compute the height of each subtree */
            int lheight = height(node.left);
            int rheight = height(node.right);

```

```
/* use the larger one */
if (lheight > rheight)
    return (lheight + 1);
else
    return (rheight + 1);
}

}

// Driver program to test above functions
public static void main(String args[])
{
    BinaryTree tree = new BinaryTree();

    // Let us create trees shown in above diagram
    tree.root = new Node(1);
    tree.root.left = new Node(2);
    tree.root.right = new Node(3);
    tree.root.left.left = new Node(4);
    tree.root.left.right = new Node(5);

    System.out.println("Level Order traversal of binary tree is : ");
    tree.reverseLevelOrder(tree.root);
}
}

// This code has been contributed by Mayank Jaiswal
```

### Python

```
# A recursive Python program to print REVERSE level order traversal

# A binary tree node
class Node:

    # Constructor to create a new node
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

    # Function to print reverse level order traversal
    def reverseLevelOrder(self):
        h = height(self)
        for i in reversed(range(1, h+1)):
            printGivenLevel(self,i)

    # Print nodes at a given level
```

```
def printGivenLevel(root, level):  
  
    if root is None:  
        return  
    if level == 1 :  
        print root.data,  
  
    elif level>1:  
        printGivenLevel(root.left, level-1)  
        printGivenLevel(root.right, level-1)  
  
# Compute the height of a tree-- the number of  
# nodes along the longest path from the root node  
# down to the farthest leaf node  
def height(node):  
    if node is None:  
        return 0  
    else:  
  
        # Compute the height of each subtree  
        lheight = height(node.left)  
        rheight = height(node.right)  
  
        # Use the larger one  
        if lheight > rheight :  
            return lheight + 1  
        else:  
            return rheight + 1  
  
# Driver program to test above function  
root = Node(1)  
root.left = Node(2)  
root.right = Node(3)  
root.left.left = Node(4)  
root.left.right = Node(5)  
  
print "Level Order traversal of binary tree is"  
reverseLevelOrder(root)  
  
# This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

*Output:*

```
Level Order traversal of binary tree is  
4 5 2 3 1
```

*Time Complexity:* The worst case time complexity of this method is  $O(n^2)$ . For a skewed tree, printGivenLevel() takes  $O(n)$  time where n is the number of nodes in the skewed tree.

So time complexity of printLevelOrder() is  $O(n) + O(n-1) + O(n-2) + \dots + O(1)$  which is  $O(n^2)$ .

### METHOD 2 (Using Queue and Stack)

The method 2 of [normal level order traversal](#) can also be easily modified to print level order traversal in reverse order. The idea is to use a stack to get the reverse level order. If we do normal level order traversal and instead of printing a node, push the node to a stack and then print contents of stack, we get “5 4 3 2 1” for above example tree, but output should be “4 5 2 3 1”. So to get the correct sequence (left to right at every level), we process children of a node in reverse order, we first push the right subtree to stack, then left subtree.

C++

```
// A C++ program to print REVERSE level order traversal using stack and queue
// This approach is adopted from following link
// http://tech-queries.blogspot.in/2008/12/level-order-tree-traversal-in-reverse.html
#include <iostream>
#include <stack>
#include <queue>
using namespace std;

/* A binary tree node has data, pointer to left and right children */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

/* Given a binary tree, print its nodes in reverse level order */
void reverseLevelOrder(node* root)
{
    stack <node *> S;
    queue <node *> Q;
    Q.push(root);

    // Do something like normal level order traversal order. Following are the
    // differences with normal level order traversal
    // 1) Instead of printing a node, we push the node to stack
    // 2) Right subtree is visited before left subtree
    while (Q.empty() == false)
    {
        /* Dequeue node and make it root */
        root = Q.front();
        Q.pop();
        S.push(root);

        /* Enqueue right child */
        if (root->right)
```

```

Q.push(root->right); // NOTE: RIGHT CHILD IS ENQUEUED BEFORE LEFT

/* Enqueue left child */
if (root->left)
    Q.push(root->left);
}

// Now pop all items from stack one by one and print them
while (S.empty() == false)
{
    root = S.top();
    cout << root->data << " ";
    S.pop();
}
}

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
node* newNode(int data)
{
    node* temp = new node;
    temp->data = data;
    temp->left = NULL;
    temp->right = NULL;

    return (temp);
}

/* Driver program to test above functions*/
int main()
{
    struct node *root = newNode(1);
    root->left      = newNode(2);
    root->right     = newNode(3);
    root->left->left  = newNode(4);
    root->left->right = newNode(5);
    root->right->left  = newNode(6);
    root->right->right = newNode(7);

    cout << "Level Order traversal of binary tree is \n";
    reverseLevelOrder(root);

    return 0;
}

```

**Java**

```
// A recursive java program to print reverse level order traversal
```

```

// using stack and queue

import java.util.LinkedList;
import java.util.Queue;
import java.util.Stack;

/* A binary tree node has data, pointer to left and right children */
class Node
{
    int data;
    Node left, right;

    Node(int item)
    {
        data = item;
        left = right;
    }
}

class BinaryTree
{
    Node root;

    /* Given a binary tree, print its nodes in reverse level order */
    void reverseLevelOrder(Node node)
    {
        Stack<Node> S = new Stack();
        Queue<Node> Q = new LinkedList();
        Q.add(node);

        // Do something like normal level order traversal order.Following
        // are the differences with normal level order traversal
        // 1) Instead of printing a node, we push the node to stack
        // 2) Right subtree is visited before left subtree
        while (Q.isEmpty() == false)
        {
            /* Dequeue node and make it root */
            node = Q.peek();
            Q.remove();
            S.push(node);

            /* Enqueue right child */
            if (node.right != null)
                // NOTE: RIGHT CHILD IS ENQUEUED BEFORE LEFT
                Q.add(node.right);

            /* Enqueue left child */
            if (node.left != null)

```

```
        Q.add(node.left);
    }

    // Now pop all items from stack one by one and print them
    while (S.empty() == false)
    {
        node = S.peek();
        System.out.print(node.data + " ");
        S.pop();
    }
}

// Driver program to test above functions
public static void main(String args[])
{
    BinaryTree tree = new BinaryTree();

    // Let us create trees shown in above diagram
    tree.root = new Node(1);
    tree.root.left = new Node(2);
    tree.root.right = new Node(3);
    tree.root.left.left = new Node(4);
    tree.root.left.right = new Node(5);
    tree.root.right.left = new Node(6);
    tree.root.right.right = new Node(7);

    System.out.println("Level Order traversal of binary tree is :");
    tree.reverseLevelOrder(tree.root);

}
}

// This code has been contributed by Mayank Jaiswal
```

### Python

```
# Python program to print REVERSE level order traversal using
# stack and queue

# A binary tree node
class Node:

    # Constructor to create a new node
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None
```

```
# Given a binary tree, print its nodes in reverse level order
def reverseLevelOrder(root):
    S = []
    Q = []
    Q.append(root)

    # Do something like normal level order traversal order.
    # Following are the differences with normal level order
    # traversal:
    # 1) Instead of printing a node, we push the node to stack
    # 2) Right subtree is visited before left subtree
    while(len(Q) > 0):

        # Dequeue node and make it root
        root = Q.pop(0)
        S.append(root)

        # Enqueue right child
        if (root.right):
            Q.append(root.right)

        # Enqueue left child
        if (root.left):
            Q.append(root.left)

    # Now pop all items from stack one by one and print them
    while(len(S) > 0):
        root = S.pop()
        print root.data,

# Driver program to test above function
root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)
root.right.left = Node(6)
root.right.right = Node(7)

print "Level Order traversal of binary tree is"
reverseLevelOrder(root)

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

*Output:*

```
Level Order traversal of binary tree is
```

4 5 6 7 2 3 1

*Time Complexity:* O(n) where n is number of nodes in the binary tree.

### Source

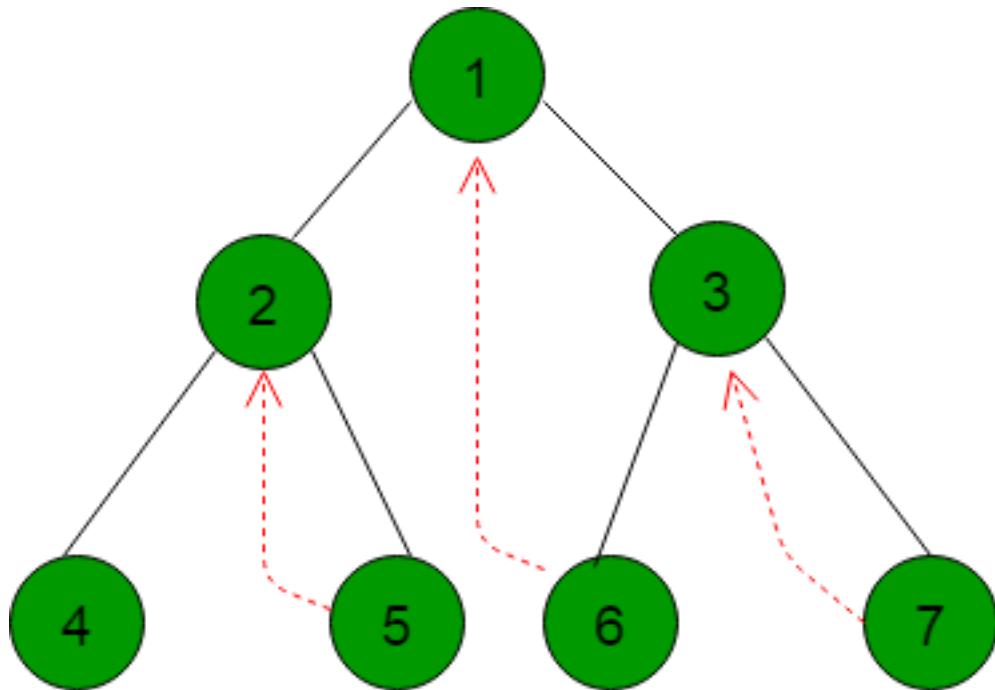
<https://www.geeksforgeeks.org/reverse-level-order-traversal/>

## Chapter 371

# Reverse Morris traversal using Threaded Binary Tree

Reverse Morris traversal using Threaded Binary Tree - GeeksforGeeks

Given a binary tree, task is to do reverse inorder traversal using Morris Traversal.



**Prerequisites :**  
[Morris Traversals](#)  
[Threaded Binary Trees](#)

In a binary tree with  $n$  nodes, there are  $n + 1$  NULL pointers which waste memory. So,

threaded binary trees makes use of these NULL pointers to save lots of Memory. So, in **Threaded Binary trees** these NULL pointers will store some useful information.

- 1) Storing predecessor information in NULL left pointers only, called as left threaded binary trees.
- 2) Storing successor information in NULL right pointers only, called as right threaded binary trees.
- 3) Storing predecessor information in NULL left pointers and successor information in NULL right pointers, called as fully threaded binary trees or simply threaded binary trees.

**Morris traversal** can be used to do Inorder traversal, reverse Inorder traversal, Pre-order traversal with constant extra memory consumed  $O(1)$ .

**Reverse Morris Traversal** : It is simply the reverse form of Morris Traversal. In reverse Morris traversal, first create links to the inorder successor of the current node and print the data using these links, and finally revert the changes to restore original tree, which will give a reverse inorder traversal.

**Algorithm :**

- 1) Initialize Current as root.
- 2) While current is not NULL :
  - 2.1) If current has no right child
    - a) Visit the current node.
    - b) Move to the left child of current.
  - 2.2) Else, here we have 2 cases:
    - a) Find the inorder successor of current node.  
Inorder successor is the left most node  
in the right subtree or right child itself.
    - b) If the left child of the inorder successor is NULL:
      - 1) Set current as the left child of its inorder successor.
      - 2) Move current node to its right.
    - c) Else, if the threaded link between the current node  
and it's inorder successor already exists :
      - 1) Set left pointer of the inorder successor as NULL.
      - 2) Visit Current node.
      - 3) Move current to it's left child.

```
// CPP code for reverse Morris Traversal
#include<bits/stdc++.h>

using namespace std;

// Node structure
struct Node {
```

```
int data;
Node *left, *right;
};

// helper function to create a new node
Node *newNode(int data){
    Node *temp = new Node;

    temp->data = data;
    temp->right = temp->left = NULL;

    return temp;
}

// function for reverse inorder traversal
void MorrisReverseInorder(Node *root)
{
    if(!root)
        return ;

    // Auxiliary node pointers
    Node *curr, *successor;

    // initialize current as root
    curr = root;

    while(curr)
    {
        // case-1, if curr has no right child then
        // visit current and move to left child
        if(curr -> right == NULL)
        {
            cout << curr->data << " ";
            curr = curr->left;
        }

        // case-2
        else
        {
            // find the inorder successor of
            // current node i.e left most node in
            // right subtree or right child itself
            successor = curr->right;

            // finding left most in right subtree
            while(successor->left != NULL &&
                  successor->left != curr)
```

```
successor = successor->left;

// if the left of inorder successor is NULL
if(successor->left == NULL)
{
    // then connect left link to current node
    successor->left = curr;

    // move current to right child
    curr = curr->right;
}

// otherwise inorder successor's left is
// not NULL and already left is linked
// with current node
else
{
    successor->left = NULL;

    // visiting the current node
    cout << curr->data << " ";

    // move current to its left child
    curr = curr->left;
}
}

}

// Driver code
int main()
{

/* Constructed binary tree is
      1
     /   \
    2     3
   / \   / \
  4   5  6   7
*/
Node *root = newNode(1);
root->left = newNode(2);
root->right = newNode(3);
root->left->left = newNode(4);
root->left->right = newNode(5);
root->right->left = newNode(6);
root->right->right = newNode(7);
```

```
//reverse inorder traversal.  
MorrisReverseInorder(root);  
  
return 0;  
}
```

**Output:**

7 3 6 1 5 2 4

**Time Complexity :**  $O(n)$

**Auxiliary Space :**  $O(1)$

**Source**

<https://www.geeksforgeeks.org/reverse-morris-traversal-using-threaded-binary-tree/>

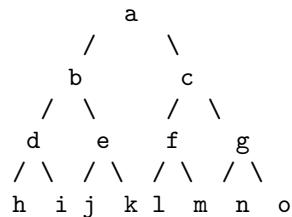
## Chapter 372

# Reverse alternate levels of a perfect binary tree

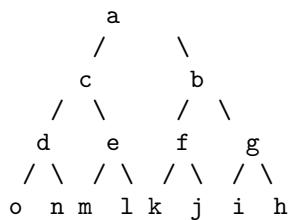
Reverse alternate levels of a perfect binary tree - GeeksforGeeks

Given a [Perfect Binary Tree](#), reverse the alternate level nodes of the binary tree.

Given tree:



Modified tree:



Method 1 (Simple)

A **simple solution** is to do following steps.

- 1) Access nodes level by level.
- 2) If current level is odd, then store nodes of this level in an array.
- 3) Reverse the array and store elements back in tree.

### Method 2 (Using Two Traversals)

Another is to do two inorder traversals. Following are steps to be followed.

1) Traverse the given tree in inorder fashion and store all odd level nodes in an auxiliary array. For the above example given tree, contents of array become {h, i, b, j, k, l, m, c, n, o}

2) Reverse the array. The array now becomes {o, n, c, m, l, k, j, b, i, h}

3) Traverse the tree again inorder fashion. While traversing the tree, one by one take elements from array and store elements from array to every odd level traversed node. For the above example, we traverse 'h' first in above array and replace 'h' with 'o'. Then we traverse 'i' and replace it with n.

Following is the implementation of the above algorithm.

C++

```
// C++ program to reverse alternate levels of a binary tree
#include<iostream>
#define MAX 100
using namespace std;

// A Binary Tree node
struct Node
{
    char data;
    struct Node *left, *right;
};

// A utility function to create a new Binary Tree Node
struct Node *newNode(char item)
{
    struct Node *temp = new Node;
    temp->data = item;
    temp->left = temp->right = NULL;
    return temp;
}

// Function to store nodes of alternate levels in an array
void storeAlternate(Node *root, char arr[], int *index, int l)
{
    // Base case
    if (root == NULL) return;

    // Store elements of left subtree
    storeAlternate(root->left, arr, index, l+1);
```

```
// Store this node only if this is a odd level node
if (l%2 != 0)
{
    arr[*index] = root->data;
    (*index)++;
}

// Store elements of right subtree
storeAlternate(root->right, arr, index, l+1);
}

// Function to modify Binary Tree (All odd level nodes are
// updated by taking elements from array in inorder fashion)
void modifyTree(Node *root, char arr[], int *index, int l)
{
    // Base case
    if (root == NULL) return;

    // Update nodes in left subtree
    modifyTree(root->left, arr, index, l+1);

    // Update this node only if this is an odd level node
    if (l%2 != 0)
    {
        root->data = arr[*index];
        (*index)++;
    }

    // Update nodes in right subtree
    modifyTree(root->right, arr, index, l+1);
}

// A utility function to reverse an array from index
// 0 to n-1
void reverse(char arr[], int n)
{
    int l = 0, r = n-1;
    while (l < r)
    {
        int temp = arr[l];
        arr[l] = arr[r];
        arr[r] = temp;
        l++; r--;
    }
}

// The main function to reverse alternate nodes of a binary tree
void reverseAlternate(struct Node *root)
```

```
{  
    // Create an auxiliary array to store nodes of alternate levels  
    char *arr = new char[MAX];  
    int index = 0;  
  
    // First store nodes of alternate levels  
    storeAlternate(root, arr, &index, 0);  
  
    // Reverse the array  
    reverse(arr, index);  
  
    // Update tree by taking elements from array  
    index = 0;  
    modifyTree(root, arr, &index, 0);  
}  
  
// A utility function to print inorder traversal of a  
// binary tree  
void printInorder(struct Node *root)  
{  
    if (root == NULL) return;  
    printInorder(root->left);  
    cout << root->data << " ";  
    printInorder(root->right);  
}  
  
// Driver Program to test above functions  
int main()  
{  
    struct Node *root = newNode('a');  
    root->left = newNode('b');  
    root->right = newNode('c');  
    root->left->left = newNode('d');  
    root->left->right = newNode('e');  
    root->right->left = newNode('f');  
    root->right->right = newNode('g');  
    root->left->left->left = newNode('h');  
    root->left->left->right = newNode('i');  
    root->left->right->left = newNode('j');  
    root->left->right->right = newNode('k');  
    root->right->left->left = newNode('l');  
    root->right->left->right = newNode('m');  
    root->right->right->left = newNode('n');  
    root->right->right->right = newNode('o');  
  
    cout << "Inorder Traversal of given tree\n";  
    printInorder(root);
```

```
    reverseAlternate(root);

    cout << "\n\nInorder Traversal of modified tree\n";
    printInorder(root);

    return 0;
}
```

**Java**

```
// Java program to reverse alternate levels of perfect binary tree
// A binary tree node
class Node {

    char data;
    Node left, right;

    Node(char item) {
        data = item;
        left = right = null;
    }
}

// class to access index value by reference
class Index {

    int index;
}

class BinaryTree {

    Node root;
    Index index_obj = new Index();

    // function to store alternate levels in a tree
    void storeAlternate(Node node, char arr[], Index index, int l) {
        // base case
        if (node == null) {
            return;
        }
        // store elements of left subtree
        storeAlternate(node.left, arr, index, l + 1);

        // store this node only if level is odd
        if (l % 2 != 0) {
            arr[index.index] = node.data;
            index.index++;
        }
    }
}
```

```
        storeAlternate(node.right, arr, index, l + 1);
    }

// Function to modify Binary Tree (All odd level nodes are
// updated by taking elements from array in inorder fashion)
void modifyTree(Node node, char arr[], Index index, int l) {

    // Base case
    if (node == null) {
        return;
    }

    // Update nodes in left subtree
    modifyTree(node.left, arr, index, l + 1);

    // Update this node only if this is an odd level node
    if (l % 2 != 0) {
        node.data = arr[index.index];
        (index.index)++;
    }

    // Update nodes in right subtree
    modifyTree(node.right, arr, index, l + 1);
}

// A utility function to reverse an array from index
// 0 to n-1
void reverse(char arr[], int n) {
    int l = 0, r = n - 1;
    while (l < r) {
        char temp = arr[l];
        arr[l] = arr[r];
        arr[r] = temp;
        l++;
        r--;
    }
}

void reverseAlternate() {
    reverseAlternate(root);
}

// The main function to reverse alternate nodes of a binary tree
void reverseAlternate(Node node) {

    // Create an auxiliary array to store nodes of alternate levels
    char[] arr = new char[100];
```

```
// First store nodes of alternate levels
storeAlternate(node, arr, index_obj, 0);

//index_obj.index = 0;

// Reverse the array
reverse(arr, index_obj.index);

// Update tree by taking elements from array
index_obj.index = 0;
modifyTree(node, arr, index_obj, 0);
}

void printInorder() {
    printInorder(root);
}

// A utility function to print inorder traversal of a
// binary tree
void printInorder(Node node) {
    if (node == null) {
        return;
    }
    printInorder(node.left);
    System.out.print(node.data + " ");
    printInorder(node.right);
}

// Driver program to test the above functions
public static void main(String args[]) {
    BinaryTree tree = new BinaryTree();
    tree.root = new Node('a');
    tree.root.left = new Node('b');
    tree.root.right = new Node('c');
    tree.root.left.left = new Node('d');
    tree.root.left.right = new Node('e');
    tree.root.right.left = new Node('f');
    tree.root.right.right = new Node('g');
    tree.root.left.left.left = new Node('h');
    tree.root.left.left.right = new Node('i');
    tree.root.left.right.left = new Node('j');
    tree.root.left.right.right = new Node('k');
    tree.root.right.left.left = new Node('l');
    tree.root.right.left.right = new Node('m');
    tree.root.right.right.left = new Node('n');
    tree.root.right.right.right = new Node('o');
    System.out.println("Inorder Traversal of given tree");
}
```

```
tree.printInorder();

tree.reverseAlternate();
System.out.println("");
System.out.println("");
System.out.println("Inorder Traversal of modified tree");
tree.printInorder();
}

}

// This code has been contributed by Mayank Jaiswal
```

Output:

```
Inorder Traversal of given tree
h d i b j e k a l f m c n g o
```

```
Inorder Traversal of modified tree
o d n c m e l a k f j b i g h
```

Time complexity of the above solution is O(n) as it does two inorder traversals of binary tree.

### Method 3 (Using One Traversal)

```
// C++ program to reverse alternate levels of a tree
#include <bits/stdc++.h>
using namespace std;

struct Node
{
    char key;
    Node *left, *right;
};

void preorder(struct Node *root1, struct Node* root2, int lvl)
{
    // Base cases
    if (root1 == NULL || root2==NULL)
        return;

    // Swap subtrees if level is even
    if (lvl%2 == 0)
```

```
swap(root1->key, root2->key);

// Recur for left and right subtrees (Note : left of root1
// is passed and right of root2 in first call and opposite
// in second call.
preorder(root1->left, root2->right, lvl+1);
preorder(root1->right, root2->left, lvl+1);
}

// This function calls preorder() for left and right children
// of root
void reverseAlternate(struct Node *root)
{
    preorder(root->left, root->right, 0);
}

// Inorder traversal (used to print initial and
// modified trees)
void printInorder(struct Node *root)
{
    if (root == NULL)
        return;
    printInorder(root->left);
    cout << root->key << " ";
    printInorder(root->right);
}

// A utility function to create a new node
Node *newNode(int key)
{
    Node *temp = new Node;
    temp->left = temp->right = NULL;
    temp->key = key;
    return temp;
}

// Driver program to test above functions
int main()
{
    struct Node *root = newNode('a');
    root->left = newNode('b');
    root->right = newNode('c');
    root->left->left = newNode('d');
    root->left->right = newNode('e');
    root->right->left = newNode('f');
    root->right->right = newNode('g');
    root->left->left->left = newNode('h');
    root->left->left->right = newNode('i');
```

```
root->left->right->left = newNode('j');
root->left->right->right = newNode('k');
root->right->left->left = newNode('l');
root->right->left->right = newNode('m');
root->right->right->left = newNode('n');
root->right->right->right = newNode('o');

cout << "Inorder Traversal of given tree\n";
printInorder(root);

reverseAlternate(root);

cout << "\n\nInorder Traversal of modified tree\n";
printInorder(root);
return 0;
}
```

Output :

```
Inorder Traversal of given tree
h d i b j e k a l f m c n g o
```

```
Inorder Traversal of modified tree
o d n c m e l a k f j b i g h
```

Thanks Soumyajit Bhattacharyay for suggesting above solution.

This article is contributed by **Kripal Gaurav**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## Source

<https://www.geeksforgeeks.org/reverse-alternate-levels-binary-tree/>

# Chapter 373

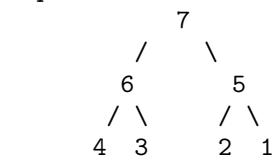
## Reverse tree path

Reverse tree path - GeeksforGeeks

Given a tree and a node data, your task to reverse the path till that particular Node.

Examples:

Input :

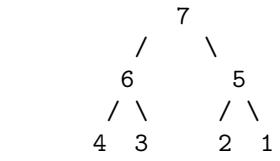


Data = 4

Output : Inorder of tree

7 6 3 4 2 5 1

Input :



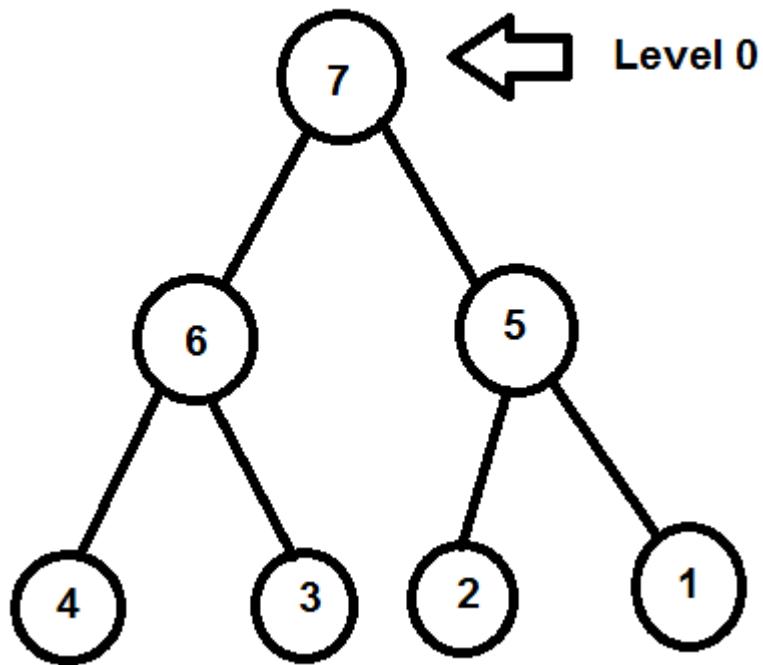
Data = 2

Output : Inorder of tree

4 6 3 2 7 5 1

The idea is to use a map to store path level wise.

**Given Data = 4**



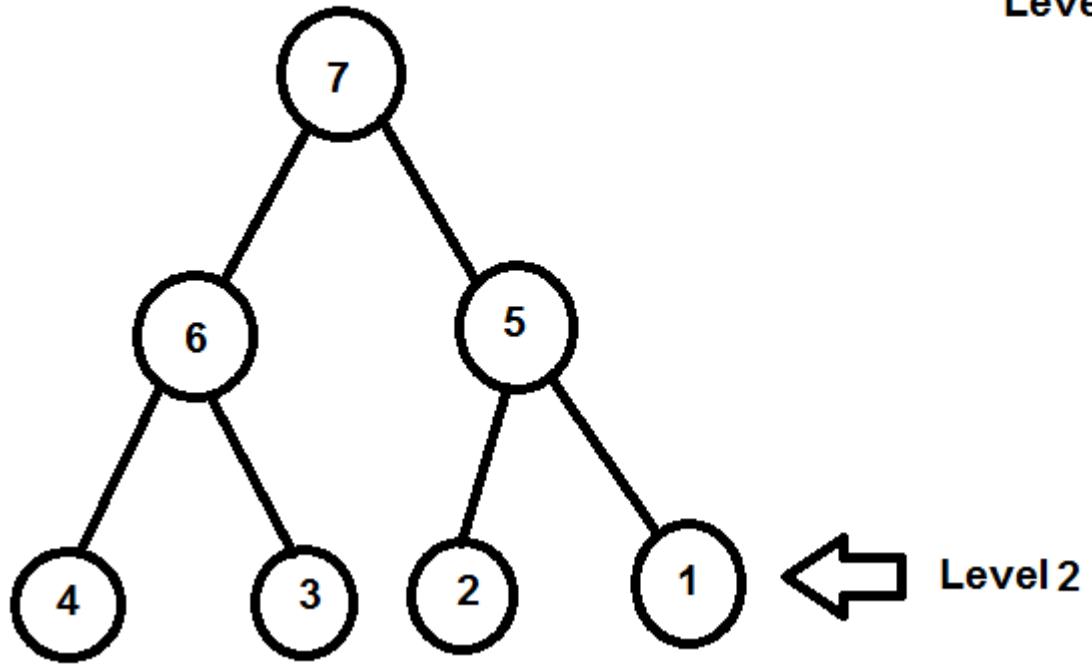
Find the Node path as well as store in the map

**Given Data = 4**

**mapTemp =**

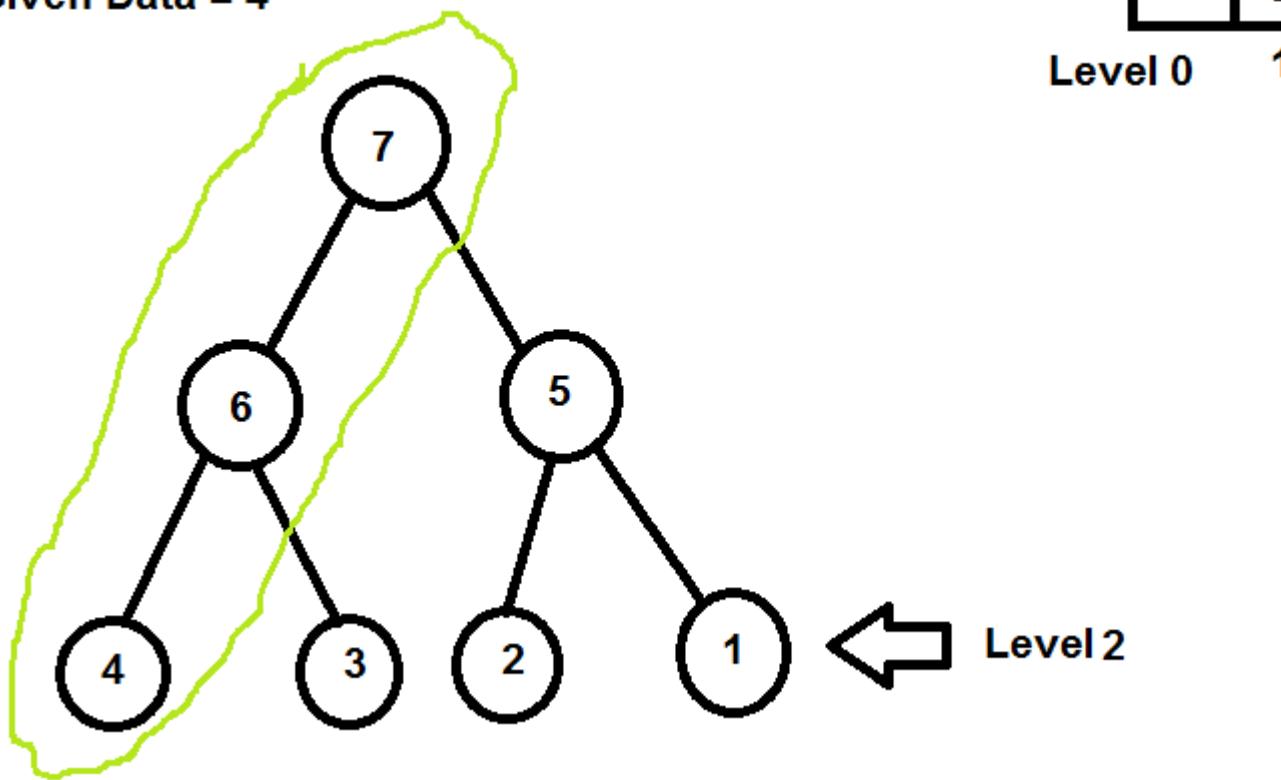
7	6
---	---

  
**Level 0**      1



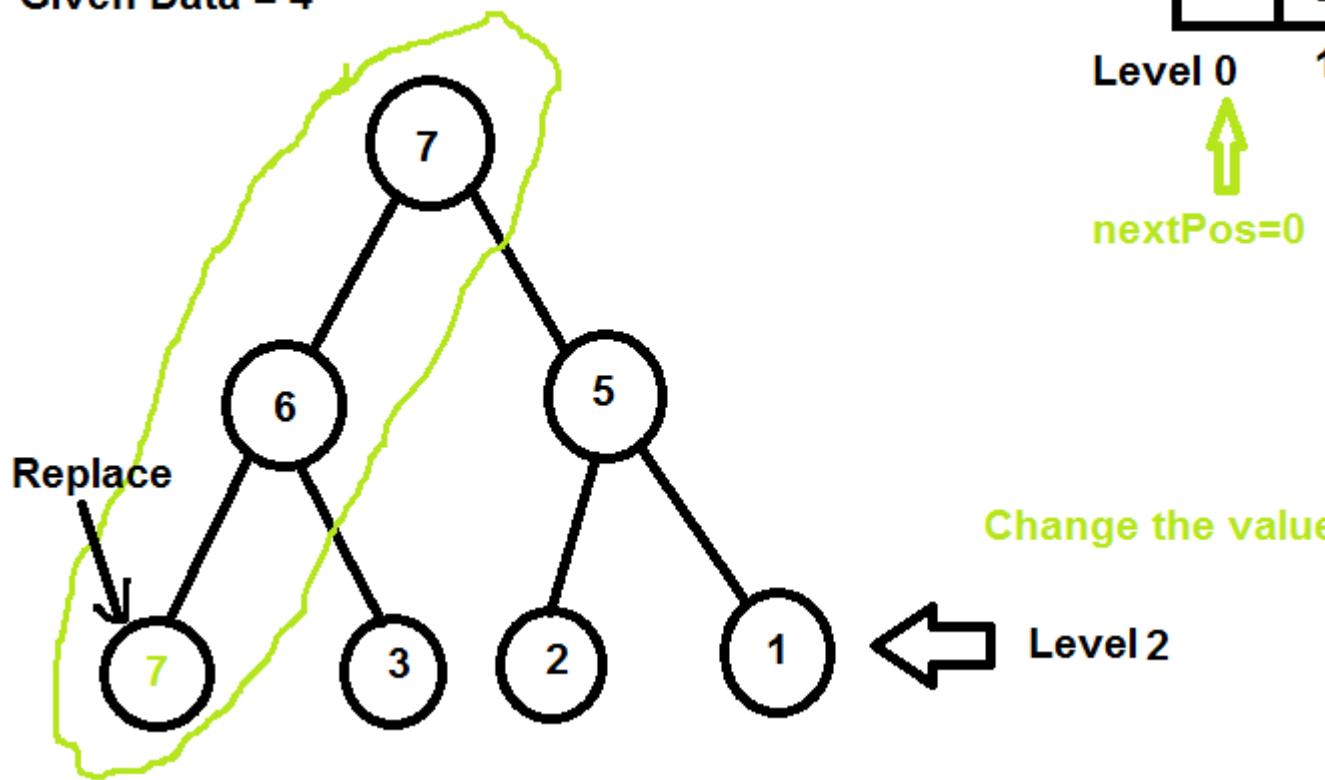
the path is

Given Data = 4



Replace the position with the map nextPos index value

Given Data = 4



increment the nextpos index and replace the next value

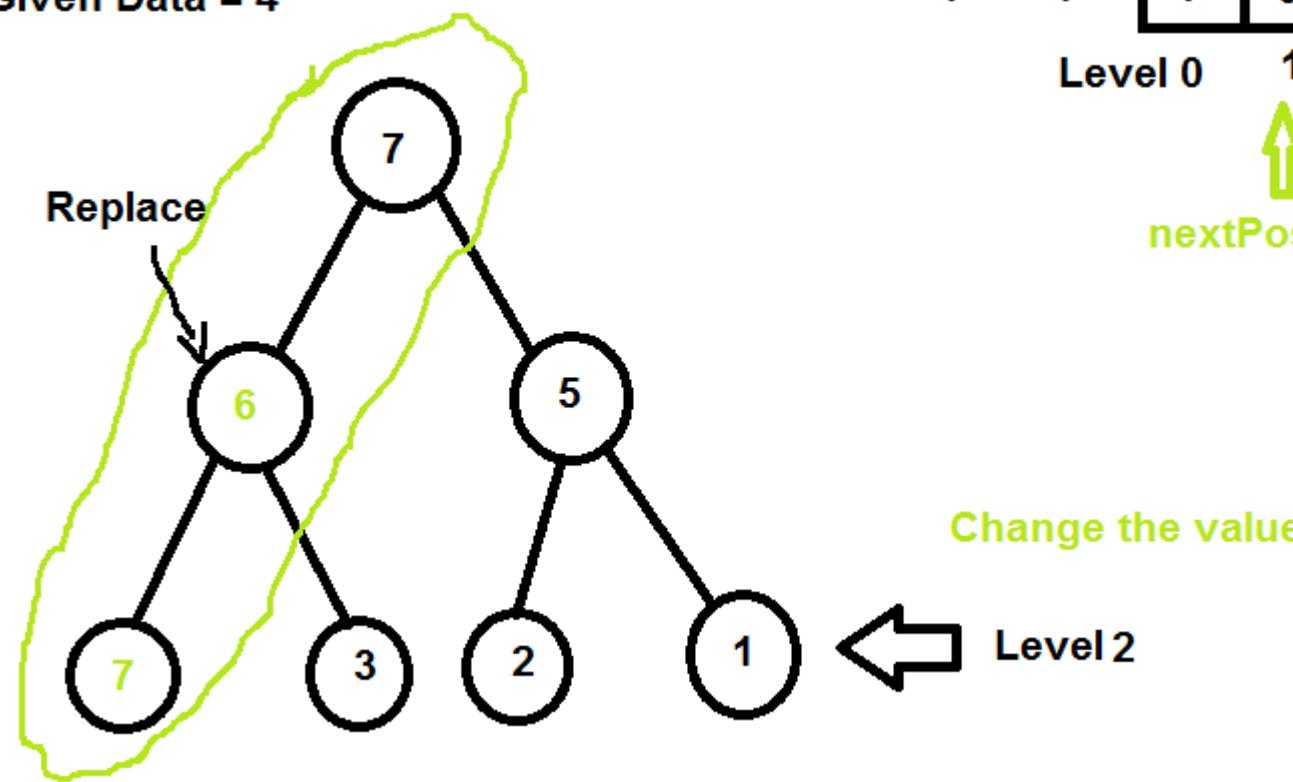
Given Data = 4

mapTemp = 

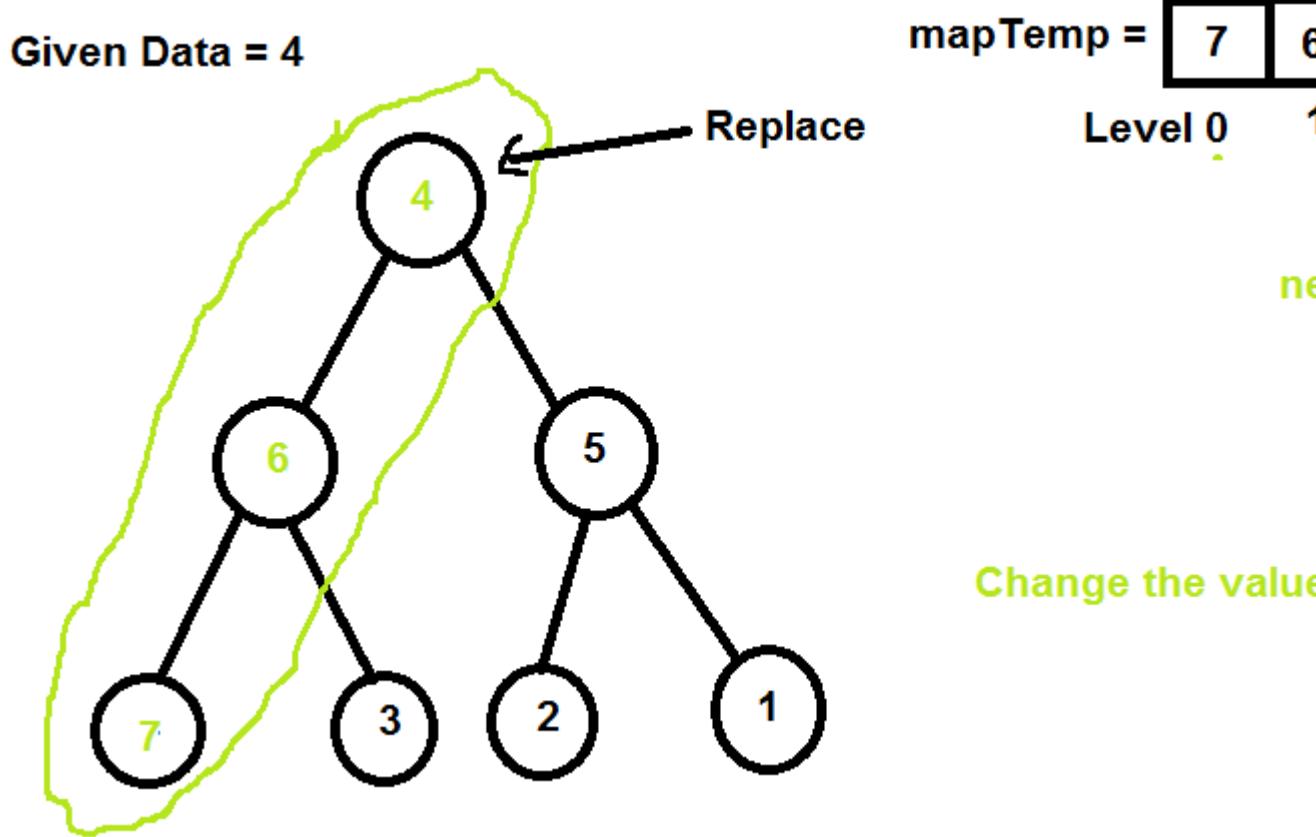
7	6
---	---

Level 0

nextPos



increment the nextpos index and replace the next value



Let's understand with the code.

```

// CPP program to Reverse Tree path
#include <bits/stdc++.h>
using namespace std;

// A Binary Tree Node
struct Node {
    int data;
    struct Node *left, *right;
};

// 'data' is input. We need to reverse path from
// root to data.
// 'level' is current level.
// 'temp' that stores path nodes.
// 'nextpos' used to pick next item for reversing.
Node* reverseTreePathUtil(Node* root, int data,
                           map<int, int>& temp, int level, int& nextpos)
{

```

```
// return NULL if root NULL
if (root == NULL)
    return NULL;

// Final condition
// if the node is found then
if (data == root->data) {

    // store the value in it's level
    temp[level] = root->data;

    // change the root value with the current
    // next element of the map
    root->data = temp[nextpos];

    // increment in k for the next element
    nextpos++;
    return root;
}

// store the data in perticular level
temp[level] = root->data;

// We go to right only when left does not
// contain given data. This way we make sure
// that correct path node is stored in temp[]
Node *left, *right;
left = reverseTreePathUtil(root->left, data, temp,
                           level + 1, nextpos);
if (left == NULL)
    right = reverseTreePathUtil(root->right, data,
                                temp, level + 1, nextpos);

// If current node is part of the path,
// then do reversing.
if (left || right) {
    root->data = temp[nextpos];
    nextpos++;
    return (left ? left : right);
}

// return NULL if not element found
return NULL;
}

// Reverse Tree path
void reverseTreePath(Node* root, int data)
{
```

```
// store per level data
map<int, int> temp;

// it is for replacing the data
int nextpos = 0;

// reverse tree path
reverseTreePathUtil(root, data, temp, 0, nextpos);
}

// INORDER
void inorder(Node* root)
{
    if (root != NULL) {
        inorder(root->left);
        cout << root->data << " ";
        inorder(root->right);
    }
}

// Utility function to create a new tree node
Node* newNode(int data)
{
    Node* temp = new Node;
    temp->data = data;
    temp->left = temp->right = NULL;
    return temp;
}

// Driver program to test above functions
int main()
{
    // Let us create binary tree shown in above diagram
    Node* root = newNode(7);
    root->left = newNode(6);
    root->right = newNode(5);
    root->left->left = newNode(4);
    root->left->right = newNode(3);
    root->right->left = newNode(2);
    root->right->right = newNode(1);

    /*      7
           /   \
          6     5
         / \   / \
        4  3  2  1           */
}

int data = 4;
```

```
// Reverse Tree Path
reverseTreePath(root, data);

// Traverse inorder
inorder(root);
return 0;
}
```

**Output:**

7 6 3 4 2 5 1

**Source**

<https://www.geeksforgeeks.org/reverse-tree-path/>

## Chapter 374

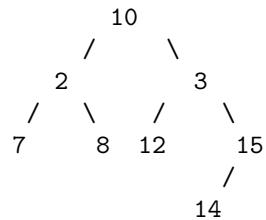
# Right view of Binary Tree using Queue

Right view of Binary Tree using Queue - GeeksforGeeks

Given a Binary Tree, print Right view of it. Right view of a Binary Tree is set of nodes visible when tree is visited from Right side.

Examples:

Input :



Output : 10 3 15 14

The output nodes are the rightmost nodes of their respective levels.

We have already discussed [recursive solution for right view](#). In this post, [level order traversal](#) based solution is discussed.

If we observe carefully, we will see that our main task is to print the right most node of every level. So, we will do a level order traversal on the tree and print the rightmost node at every level.

Below is the implementation of above approach:

C++

```
// C++ program to print right view of
// Binary Tree

#include<bits/stdc++.h>
using namespace std;

// A Binary Tree Node
struct Node
{
    int data;
    struct Node *left, *right;
};

// Utility function to create a new tree node
Node* newNode(int data)
{
    Node *temp = new Node;
    temp->data = data;
    temp->left = temp->right = NULL;
    return temp;
}

// function to print right view of
// binary tree
void printRightView(Node* root)
{
    if (!root)
        return;

    queue<Node*> q;
    q.push(root);

    while (!q.empty())
    {
        // number of nodes at current level
        int n = q.size();

        // Traverse all nodes of current level
        for(int i = 1; i <= n; i++)
        {
            Node* temp = q.front();
            q.pop();

            // Print the right most element
            // at the level
            if (i == n)
                cout<<temp->data<<" ";
        }
    }
}
```

```
// Add left node to queue
if (temp->left != NULL)
    q.push(temp->left);

// Add right node to queue
if (temp->right != NULL)
    q.push(temp->right);
}
}

// Driver program to test above functions
int main()
{
    // Let's construct the tree as
    // shown in example

    Node* root = newNode(10);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(7);
    root->left->right = newNode(8);
    root->right->right = newNode(15);
    root->right->left = newNode(12);
    root->right->right->left = newNode(14);

    printRightView(root);
}
```

### Java

```
// Java program to print right view of Binary
// Tree
import java.util.*;

public class PrintRightView
{
    // Binary tree node
    private static class Node
    {
        int data;
        Node left, right;

        public Node(int data) {
            this.data = data;
            this.left = null;
            this.right = null;
        }
}
```

```
}

// function to print right view of binary tree
private static void printRightView(Node root)
{
    if (root == null)
        return;

    Queue<Node> queue = new LinkedList<>();
    queue.add(root);

    while (!queue.isEmpty())
    {
        // number of nodes at current level
        int n = queue.size();

        // Traverse all nodes of current level
        for (int i = 1; i <= n; i++) {
            Node temp = queue.poll();

            // Print the right most element at
            // the level
            if (i == n)
                System.out.print(temp.data + " ");

            // Add left node to queue
            if (temp.left != null)
                queue.add(temp.left);

            // Add right node to queue
            if (temp.right != null)
                queue.add(temp.right);
        }
    }
}

// Driver code
public static void main(String[] args)
{
    // construct binary tree as shown in
    // above diagram
    Node root = new Node(10);
    root.left = new Node(2);
    root.right = new Node(3);
    root.left.left = new Node(7);
    root.left.right = new Node(8);
    root.right.right = new Node(15);
    root.right.left = new Node(12);
```

```
    root.right.right.left = new Node(14);  
    printRightView(root);  
}  
}
```

Output:

10 3 15 14

**Time Complexity:** O( n ), where n is the number of nodes in the binary tree.

## Source

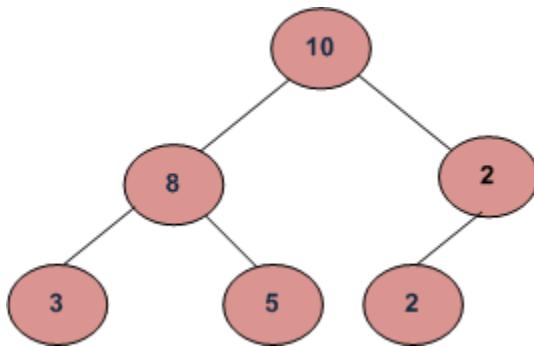
<https://www.geeksforgeeks.org/right-view-binary-tree-using-queue/>

## Chapter 375

# Root to leaf path sum equal to a given number

Root to leaf path sum equal to a given number - GeeksforGeeks

Given a binary tree and a number, return true if the tree has a root-to-leaf path such that adding up all the values along the path equals the given number. Return false if no such path can be found.



For example, in the above tree root to leaf paths exist with following sums.

21  $\rightarrow$  10 – 8 – 3  
23  $\rightarrow$  10 – 8 – 5  
14  $\rightarrow$  10 – 2 – 2

So the returned value should be true only for numbers 21, 23 and 14. For any other number, returned value should be false.

Algorithm:

Recursively check if left or right child has path sum equal to ( number – value at current node)

Implementation:

C

```
#include<stdio.h>
#include<stdlib.h>
#define bool int

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

/*
Given a tree and a sum, return true if there is a path from the root
down to a leaf, such that adding up all the values along the path
equals the given sum.

Strategy: subtract the node value from the sum when recurring down,
and check to see if the sum is 0 when you run out of tree.
*/
bool hasPathSum(struct node* node, int sum)
{
    /* return true if we run out of tree and sum==0 */
    if (node == NULL)
    {
        return (sum == 0);
    }

    else
    {
        bool ans = 0;

        /* otherwise check both subtrees */
        int subSum = sum - node->data;

        /* If we reach a leaf node and sum becomes 0 then return true*/
        if ( subSum == 0 && node->left == NULL && node->right == NULL )
            return 1;

        if(node->left)
            ans = ans || hasPathSum(node->left, subSum);
        if(node->right)
            ans = ans || hasPathSum(node->right, subSum);

        return ans;
    }
}
```

```
/* UTILITY FUNCTIONS */
/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newnode(int data)
{
    struct node* node = (struct node*)
                           malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;

    return(node);
}

/* Driver program to test above functions*/
int main()
{

    int sum = 21;

    /* Constructed binary tree is
           10
          /   \
         8     2
        / \   /
       3   5  2
    */
    struct node *root = newnode(10);
    root->left      = newnode(8);
    root->right     = newnode(2);
    root->left->left  = newnode(3);
    root->left->right = newnode(5);
    root->right->left = newnode(2);

    if(hasPathSum(root, sum))
        printf("There is a root-to-leaf path with sum %d", sum);
    else
        printf("There is no root-to-leaf path with sum %d", sum);

    getchar();
    return 0;
}
```

### Java

```
// Java program to print to print root to leaf path sum equal to
// a given number
```

```
/* A binary tree node has data, pointer to left child
   and a pointer to right child */
class Node
{
    int data;
    Node left, right;

    Node(int item)
    {
        data = item;
        left = right = null;
    }
}

class BinaryTree {

    Node root;

    /*
     Given a tree and a sum, return true if there is a path from the root
     down to a leaf, such that adding up all the values along the path
     equals the given sum.

     Strategy: subtract the node value from the sum when recurring down,
     and check to see if the sum is 0 when you run out of tree.
    */

    boolean haspathSum(Node node, int sum)
    {
        if (node == null)
            return (sum == 0);
        else
        {
            boolean ans = false;

            /* otherwise check both subtrees */
            int subsum = sum - node.data;
            if (subsum == 0 && node.left == null && node.right == null)
                return true;
            if (node.left != null)
                ans = ans || haspathSum(node.left, subsum);
            if (node.right != null)
                ans = ans || haspathSum(node.right, subsum);
            return ans;
        }
    }
}
```

```
/* Driver program to test the above functions */
public static void main(String args[])
{
    int sum = 21;

    /* Constructed binary tree is
        10
        / \
       8   2
      / \ /
     3   5 2
    */

    BinaryTree tree = new BinaryTree();
    tree.root = new Node(10);
    tree.root.left = new Node(8);
    tree.root.right = new Node(2);
    tree.root.left.left = new Node(3);
    tree.root.left.right = new Node(5);
    tree.root.right.left = new Node(2);

    if (tree.haspathSum(tree.root, sum))
        System.out.println("There is a root to leaf path with sum " + sum);
    else
        System.out.println("There is no root to leaf path with sum " + sum);
}
}

// This code has been contributed by Mayank Jaiswal(mayank_24)
```

### Python

```
# Python program to find if there is a root to sum path

# A binary tree node
class Node:

    # Constructor to create a new node
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

    """
    Given a tree and a sum, return true if there is a path from the root
    down to a leaf, such that adding up all the values along the path
    equals the given sum.
    """

Strategy: subtract the node value from the sum when recurring down,
```

```
    and check to see if the sum is 0 when you run out of tree.  
    """  
# s is the sum  
def hasPathSum(node, s):  
  
    # Return true if we run out of tree and s = 0  
    if node is None:  
        return (s == 0)  
  
    else:  
        ans = 0  
  
        # Otherwise check both subtrees  
        subSum = s - node.data  
  
        # If we reach a leaf node and sum becomes 0, then  
        # return True  
        if(subSum == 0 and node.left == None and node.right == None):  
            return True  
  
        if node.left is not None:  
            ans = ans or hasPathSum(node.left, subSum)  
        if node.right is not None:  
            ans = ans or hasPathSum(node.right, subSum)  
  
    return ans  
  
# Driver program to test above functions  
  
s = 21  
root = Node(10)  
root.left = Node(8)  
root.right = Node(2)  
root.left.right = Node(5)  
root.left.left = Node(3)  
root.right.left = Node(2)  
  
if hasPathSum(root, s):  
    print "There is a root-to-leaf path with sum %d" %(s)  
else:  
    print "There is no root-to-leaf path with sum %d" %(s)  
  
# This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

**Output :**

True

**Time Complexity:** O(n)

**References:** <http://cslibrary.stanford.edu/110/BinaryTrees.html>

Author: Tushar Roy

Please write comments if you find any bug in above code/algorithm, or find other ways to solve the same problem

**Improved By :** [akash1295](#)

## Source

<https://www.geeksforgeeks.org/root-to-leaf-path-sum-equal-to-a-given-number/>

## Chapter 376

# Root to leaf path with maximum distinct nodes

Root to leaf path with maximum distinct nodes - GeeksforGeeks

Given a Binary Tree, find count of distinct nodes in a root to leaf path with maximum distinct nodes.

Examples:

Input : 1  
      /    \  
     2    3  
    / \   / \  
   4   5   6   3  
      \   \  
      8   9

Output : 4  
The root to leaf path with maximum distinct  
nodes is 1-3-6-8.

A **simple solution** is to [explore all root to leaf paths](#). In every root to leaf path, count distinct nodes and finally return the maximum count.

An **efficient solution** is to use hashing. We recursively traverse the tree and maintain count of distinct nodes on path from root to current node. We recur for left and right subtrees and finally return maximum of two values.

Below c++ implementation of above idea

```
// C++ program to find count of distinct nodes
// on a path with maximum distinct nodes.
#include <bits/stdc++.h>
```

```
using namespace std;

// A node of binary tree
struct Node {
    int data;
    struct Node *left, *right;
};

// A utility function to create a new Binary
// Tree node
Node* newNode(int data)
{
    Node* temp = new Node;
    temp->data = data;
    temp->left = temp->right = NULL;
    return temp;
}

int largestUniquePathUtil(Node* node, unordered_map<int, int> m)
{
    if (!node)
        return m.size();

    // put this node into hash
    m[node->data]++;

    int max_path = max(largestUniquePathUtil(node->left, m),
                       largestUniquePathUtil(node->right, m));

    // remove current node from path "hash"
    m[node->data]--;

    // if we reached a condition where all duplicate value
    // of current node is deleted
    if (m[node->data] == 0)
        m.erase(node->data);

    return max_path;
}

// A utility function to find long unique value path
int largestUniquePath(Node* node)
{
    if (!node)
        return 0;

    // hash that store all node value
    unordered_map<int, int> hash;
```

```
// return max length unique value path
return largestUinquePathUtil(node, hash);
}

// Driver program to test above functions
int main()
{
    // Create binary tree shown in above figure
    Node* root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->left = newNode(6);
    root->right->right = newNode(7);
    root->right->left->right = newNode(8);
    root->right->right->right = newNode(9);

    cout << largestUinquePath(root) << endl;

    return 0;
}
```

4

Time Complexity :O(n)

## Source

<https://www.geeksforgeeks.org/root-leaf-path-maximum-distinct-nodes/>

## Chapter 377

# Root to leaf paths having equal lengths in a Binary Tree

Root to leaf paths having equal lengths in a Binary Tree - GeeksforGeeks

Given a binary tree, print the number of root to leaf paths having equal lengths.

Examples:

```
Input : Root of below tree
        10
        /   \
       8     2
      / \   / \
     3  5   2  4
Output : 4 paths are of length 3.
```

```
Input : Root of below tree
        10
        /   \
       8     2
      / \   / \
     3  5   2  4
      /           \
     9             1
Output : 2 paths are of length 3
         2 paths are of length 4
```

The idea is to traverse the tree and keep track of path length. Whenever we reach a leaf node, we increment path length count in a hash map.

Once we have traversed the tree, hash map has counts of distinct path lengths. Finally we print contents of hash map.

```
// C++ program to count root to leaf paths of different
// lengths.
#include<bits/stdc++.h>
using namespace std;

/* A binary tree node */
struct Node
{
    int data;
    struct Node* left, *right;
};

/* utility that allocates a new node with the
   given data and NULL left and right pointers. */
struct Node* newnode(int data)
{
    struct Node* node = new Node;
    node->data = data;
    node->left = node->right = NULL;
    return (node);
}

// Function to store counts of different root to leaf
// path lengths in hash map m.
void pathCountUtil(Node *node, unordered_map<int, int> &m,
                    int path_len)
{
    // Base condition
    if (node == NULL)
        return;

    // If leaf node reached, increment count of path
    // length of this root to leaf path.
    if (node->left == NULL && node->right == NULL)
    {
        m[path_len]++;
        return;
    }

    // Recursively call for left and right subtrees with
    // path lengths more than 1.
    pathCountUtil(node->left, m, path_len+1);
    pathCountUtil(node->right, m, path_len+1);
}

// A wrapper over pathCountUtil()
void pathCounts(Node *root)
{
```

```
// create an empty hash table
unordered_map<int, int> m;

// Recursively check in left and right subtrees.
pathCountUtil(root, m, 1);

// Print all path lengths and their counts.
for (auto itr=m.begin(); itr != m.end(); itr++)
    cout << itr->second << " paths have length "
        << itr->first << endl;
}

// Driver program to run the case
int main()
{
    struct Node *root = newnode(8);
    root->left      = newnode(5);
    root->right     = newnode(4);
    root->left->left = newnode(9);
    root->left->right = newnode(7);
    root->right->right = newnode(11);
    root->right->right->left = newnode(3);
    pathCounts(root);
    return 0;
}
```

Output:

```
4 paths have length 3
```

## Source

<https://www.geeksforgeeks.org/root-leaf-paths-equal-lengths-binary-tree/>

## Chapter 378

# ScapeGoat Tree | Set 1 (Introduction and Insertion)

ScapeGoat Tree | Set 1 (Introduction and Insertion) - GeeksforGeeks

A ScapeGoat tree is a self-balancing Binary Search Tree like [AVL Tree](#), [Red-Black Tree](#), [Splay Tree](#), ..etc.

- Search time is  $O(\log n)$  in worst case. Time taken by deletion and insertion is [amortized](#)  $O(\log n)$
- The balancing idea is to make sure that nodes are size balanced. A size balanced means sizes of left and right subtrees are at most \* (Size of node). The idea is based on the fact that *if a node is A weight balanced, then it is also height balanced:  $height \leq \log_{1/\alpha}(size) + 1$*
- Unlike other self-balancing BSTs, ScapeGoat tree doesn't require extra space per node. For example, Red Black Tree nodes are required to have color. In below implementation of ScapeGoat Tree, we only have left, right and parent pointers in Node class. Use of parent is done for simplicity of implementation and can be avoided.

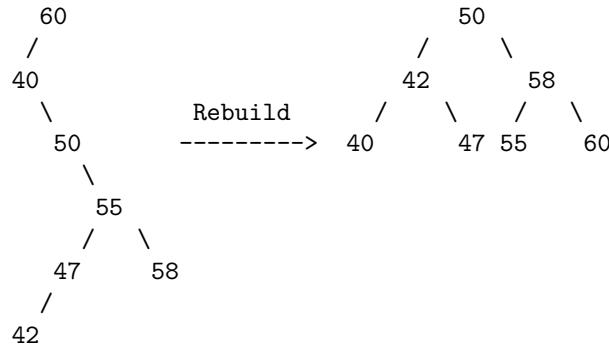
### Insertion (Assuming $\alpha = 2/3$ ):

To insert value  $x$  in a Scapegoat Tree:

- Create a new node  $u$  and insert  $x$  using the [BST insert](#) algorithm.
- If the depth of  $u$  is greater than  $\log_{3/2}n$  where  $n$  is number of nodes in tree then we need to make tree balanced. To make balanced, we use below step to find a scapegoat.
  - Walk up from  $u$  until we reach a node  $w$  with  $size(w) > (2/3)*size(w.parent)$ .  
This node is scapegoat
  - Rebuild the subtree rooted at  $w.parent$ .

### What does rebuilding the subtree mean?

In rebuilding, we simply convert the subtree to the most possible balanced BST. We first store inorder traversal of BST in an array, then we build a new BST from array by recursively dividing it into two halves.



Below is C++ implementation of insert operation on Scapegoat Tree.

```

// C++ program to implement insertion in
// ScapeGoat Tree
#include<bits/stdc++.h>
using namespace std;

// Utility function to get value of log32(n)
static int const log32(int n)
{
    double const log23 = 2.4663034623764317;
    return (int)ceil(log23 * log(n));
}

// A ScapeGoat Tree node
class Node
{
public:
    Node *left, *right, *parent;
    float value;
    Node()
    {
        value = 0;
        left = right = parent = NULL;
    }
    Node (float v)
    {
        value = v;
        left = right = parent = NULL;
    }
};

// This functions stores inorder traversal
// of tree rooted with ptr in an array arr[]
int storeInArray(Node *ptr, Node *arr[], int i)
  
```

```
{  
    if (ptr == NULL)  
        return i;  
  
    i = storeInArray(ptr->left, arr, i);  
    arr[i++] = ptr;  
    return storeInArray(ptr->right, arr, i);  
}  
  
// Class to represent a ScapeGoat Tree  
class SGTree  
{  
private:  
    Node *root;  
    int n; // Number of nodes in Tree  
public:  
    void preorder(Node *);  
    int size(Node *);  
    bool insert(float x);  
    void rebuildTree(Node *u);  
    SGTree() { root = NULL; n = 0; }  
    void preorder() { preorder(root); }  
  
    // Function to built tree with balanced nodes  
    Node *buildBalancedFromArray(Node **a, int i, int n);  
  
    // Height at which element is to be added  
    int BSTInsertAndFindDepth(Node *u);  
};  
  
// Preorder traversal of the tree  
void SGTree::preorder(Node *node)  
{  
    if (node != NULL)  
    {  
        cout << node->value << " ";  
        preorder(node -> left);  
        preorder(node -> right);  
    }  
}  
  
// To count number of nodes in the tree  
int SGTree::size(Node *node)  
{  
    if (node == NULL)  
        return 0;  
    return 1 + size(node->left) + size(node->right);  
}
```

```
// To insert new element in the tree
bool SGTree::insert(float x)
{
    // Create a new node
    Node *node = new Node(x);

    // Perform BST insertion and find depth of
    // the inserted node.
    int h = BSTInsertAndFindDepth(node);

    // If tree becomes unbalanced
    if (h > log32(n))
    {
        // Find Scapegoat
        Node *p = node->parent;
        while (3*size(p) <= 2*size(p->parent))
            p = p->parent;

        // Rebuild tree rooted under scapegoat
        rebuildTree(p->parent);
    }

    return h >= 0;
}

// Function to rebuilt tree from new node. This
// function basically uses storeInArray() to
// first store inorder traversal of BST rooted
// with u in an array.
// Then it converts array to the most possible
// balanced BST using buildBalancedFromArray()
void SGTree::rebuildTree(Node *u)
{
    int n = size(u);
    Node *p = u->parent;
    Node **a = new Node* [n];
    storeInArray(u, a, 0);
    if (p == NULL)
    {
        root = buildBalancedFromArray(a, 0, n);
        root->parent = NULL;
    }
    else if (p->right == u)
    {
        p->right = buildBalancedFromArray(a, 0, n);
        p->right->parent = p;
    }
}
```

```

    else
    {
        p->left = buildBalancedFromArray(a, 0, n);
        p->left->parent = p;
    }
}

// Function to built tree with balanced nodes
Node * SGTree::buildBalancedFromArray(Node **a,
                                      int i, int n)
{
    if (n== 0)
        return NULL;
    int m = n / 2;

    // Now a[m] becomes the root of the new
    // subtree a[0],.....,a[m-1]
    a[i+m]->left = buildBalancedFromArray(a, i, m);

    // elements a[0],...a[m-1] gets stored
    // in the left subtree
    if (a[i+m]->left != NULL)
        a[i+m]->left->parent = a[i+m];

    // elements a[m+1],....a[n-1] gets stored
    // in the right subtree
    a[i+m]->right =
        buildBalancedFromArray(a, i+m+1, n-m-1);
    if (a[i+m]->right != NULL)
        a[i+m]->right->parent = a[i+m];

    return a[i+m];
}

// Performs standard BST insert and returns
// depth of the inserted node.
int SGTree::BSTInsertAndFindDepth(Node *u)
{
    // If tree is empty
    Node *w = root;
    if (w == NULL)
    {
        root = u;
        n++;
        return 0;
    }

    // While the node is not inserted

```

```
// or a node with same key exists.
bool done = false;
int d = 0;
do
{
    if (u->value < w->value)
    {
        if (w->left == NULL)
        {
            w->left = u;
            u->parent = w;
            done = true;
        }
        else
            w = w->left;
    }
    else if (u->value > w->value)
    {
        if (w->right == NULL)
        {
            w->right = u;
            u->parent = w;
            done = true;
        }
        else
            w = w->right;
    }
    else
        return -1;
    d++;
}
while (!done);

n++;
return d;
}

// Driver code
int main()
{
    SGTree sgt;
    sgt.insert(7);
    sgt.insert(6);
    sgt.insert(3);
    sgt.insert(1);
    sgt.insert(0);
    sgt.insert(8);
    sgt.insert(9);
```

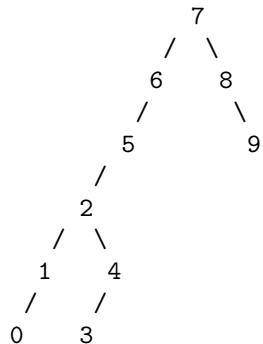
```
sgt.insert(4);
sgt.insert(5);
sgt.insert(2);
sgt.insert(3.5);
printf("Preorder traversal of the"
      " constructed ScapeGoat tree is \n");
sgt.preorder();
return 0;
}
```

Output:

```
Preorder traversal of the constructed ScapeGoat tree is
7 6 3 1 0 2 4 3.5 5 8 9
```

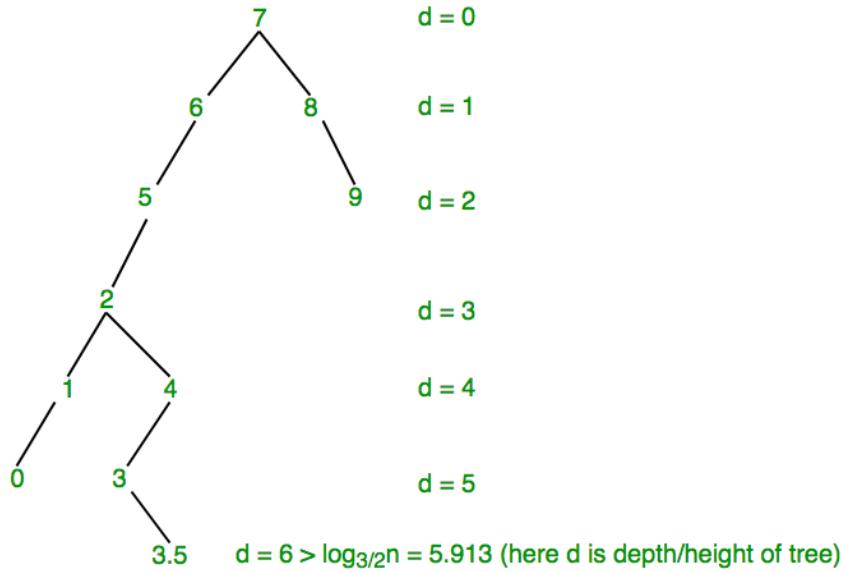
**Example to illustrate insertion:**

A scapegoat tree with 10 nodes and height 5.



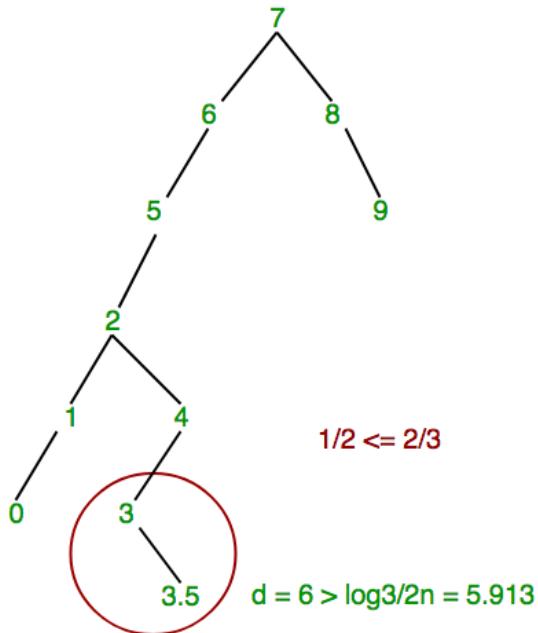
Let's insert 3.5 in the below scapegoat tree.

Initially  $d = 5 < \log_{3/2} n$  where  $n = 10$ ;

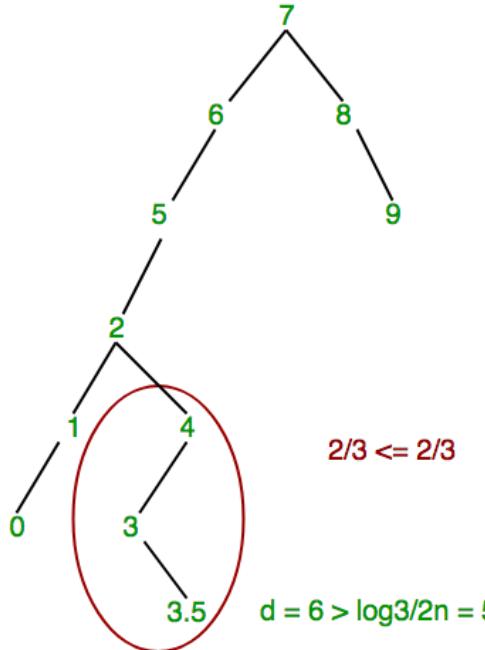


Since,  $d > \log_{3/2} n$  i.e.,  $6 > \log_{3/2} n$ , so we have to find the scapegoat in order to solve the problem of exceeding height.

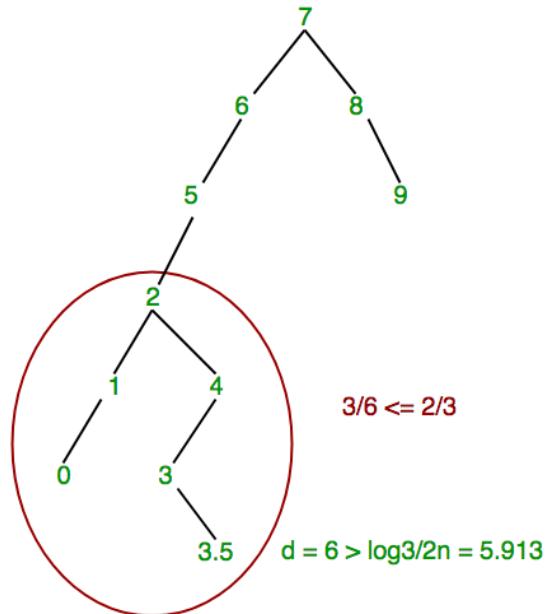
- Now we find a ScapeGoat. We start with newly added node 3.5 and check whether  $\text{size}(3.5)/\text{size}(3) > 2/3$ .
  - Since,  $\text{size}(3.5) = 1$  and  $\text{size}(3) = 2$ , so  $\text{size}(3.5)/\text{size}(3) = \frac{1}{2}$  which is less than  $2/3$ . So, this is not the scapegoat and we move up .



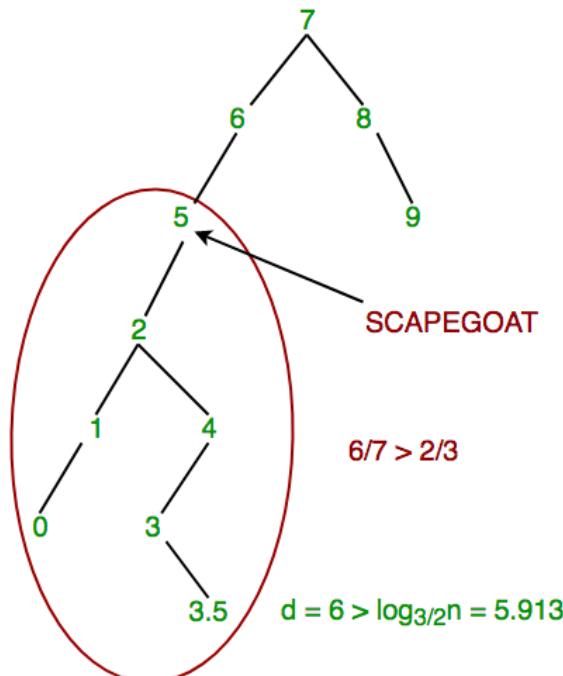
- Since 3 is not the scapegoat, we move and check the same condition for node 4. Since  $\text{size}(3) = 2$  and  $\text{size}(4) = 3$ , so  $\text{size}(3)/\text{size}(4) = 2/3$  which is not greater than  $2/3$ . So, this is not the scapegoat and we move up .



- Since 3 is not the scapegoat, we move and check the same condition for node 4. Since,  $\text{size}(3) = 2$  and  $\text{size}(4) = 3$ , so  $\text{size}(3)/\text{size}(4) = 2/3$  which is not greater than  $2/3$ . So, this is not the scapegoat and we move up .
- Now,  $\text{size}(4)/\text{size}(2) = 3/6$ . Since,  $\text{size}(4)= 3$  and  $\text{size}(2) = 6$  but  $3/6$  is still less than  $2/3$ , which does not fulfill the condition of scapegoat so we again move up.

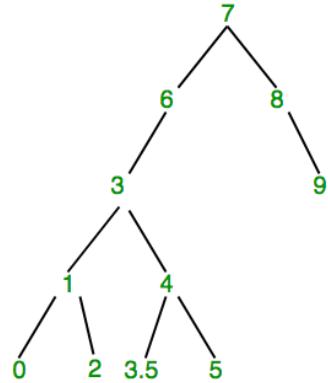


- Now, size(2)/size(5) = 6/7. Since, size(2) = 6 and size(5) = 7.  $6/7 > 2/3$  which fulfills the condition of scapegoat, so we stop here and hence node 5 is a scapegoat



Finally, after finding the scapegoat, rebuilding will be taken at the subtree rooted at scapegoat i.e., at 5.

Final tree:



#### Comparison with other self-balancing BSTs

Red-Black and AVL : Time complexity of search, insert and delete is  $O(\log n)$

Splay Tree : Worst case time complexities of search, insert and delete is  $O(n)$ . But amortized time complexity of these operations is  $O(\log n)$ .

ScapeGoat Tree: Like Splay Tree, it is easy to implement and has worst case time complexity of search as  $O(\log n)$ . Worst case and amortized time complexities of insert and delete are same as Splay Tree for Scapegoat tree.

#### References:

- [https://en.wikipedia.org/wiki/Scapegoat\\_tree](https://en.wikipedia.org/wiki/Scapegoat_tree)
- [http://opendatastructures.org/ods-java/8\\_Scapegoat\\_Trees.html](http://opendatastructures.org/ods-java/8_Scapegoat_Trees.html)

#### Source

<https://www.geeksforgeeks.org/scapegoat-tree-set-1-introduction-insertion/>

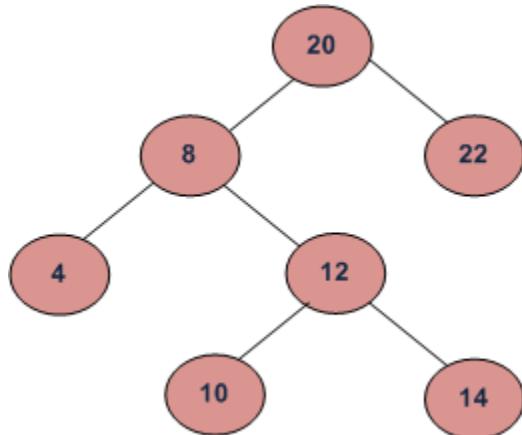
## Chapter 379

# Second Largest element in n-ary tree

Second Largest element in n-ary tree - GeeksforGeeks

Given an N-ary tree, find and return the node with second largest value in the given tree. Return NULL if no node with required value is present.

For example, in the given tree



Second largest node is 20.

A **simple** solution is to traverse the array twice. In the first traversal find the maximum value node. In the second traversal find the greatest element node less than the element obtained in first traversal. The time complexity of this solution is  $O(n)$ .

An **Efficient** Solution can be to find the second largest element in a single traversal. Below is the complete algorithm for doing this:

- 1) Initialize two nodes first and second to NULL as,  
first = second = NULL
- 2) Start traversing the tree,
  - a) If the current node data say root->key is greater than first->key then update first and second as,  
second = first  
first = root
  - b) If the current node data is in between first and second, then update second to store the value of current node as  
second = root
- 3) Return the node stored in second.

```
// CPP program to find second largest node
// in an n-ary tree.
#include <bits/stdc++.h>
using namespace std;

// Structure of a node of an n-ary tree
struct Node {
    int key;
    vector<Node*> child;
};

// Utility function to create a new tree node
Node* newNode(int key)
{
    Node* temp = new Node;
    temp->key = key;
    return temp;
}

void secondLargestUtil(Node* root, Node** first,
                      Node** second)
{
    if (root == NULL)
        return;

    // If first is NULL, make root equal to first
    if (!(*first))
        *first = root;

    // if root is greater than first then second
    // will become first and update first equal
    // to root
    else if (root->key > (*first)->key) {
        *second = *first;
        *first = root;
    }
}
```

```
}

// If root is less than first but greater than
// second
else if (!(*second) || root->key > (*second)->key)
    *second = root;

// number of children of root
int numChildren = root->child.size();

// recursively calling for every child
for (int i = 0; i < numChildren; i++)
    secondLargestUtil(root->child[i], first, second);
}

Node* secondLargest(Node* root)
{
    // second will store the second highest value
    Node* second = NULL;

    // first will store the largest value in the tree
    Node* first = NULL;

    // calling the helper function
    secondLargestUtil(root, &first, &second);

    // return the second largest element
    return second;
}

// Driver program
int main()
{
    /* Let us create below tree
     *      5
     *      /   |   \
     *      1   2   3
     *      /   / \   \
     *     15  4   5   6
     */
}

Node* root = newNode(5);
(root->child).push_back(newNode(1));
(root->child).push_back(newNode(2));
(root->child).push_back(newNode(3));
(root->child[0]->child).push_back(newNode(15));
(root->child[1]->child).push_back(newNode(4));
(root->child[1]->child).push_back(newNode(5));
```

```
(root->child[2]->child).push_back(newNode(6));  
  
cout << "Second largest element is : ";  
cout << secondLargest(root) - key << endl;  
  
return 0;  
}
```

Output:

```
Second largest element is : 6
```

## Source

<https://www.geeksforgeeks.org/second-largest-element-n-ary-tree/>

## Chapter 380

# Segment Tree | Set 1 (Sum of given range)

Segment Tree | Set 1 (Sum of given range) - GeeksforGeeks

Let us consider the following problem to understand Segment Trees.

We have an array  $\text{arr}[0 \dots n-1]$ . We should be able to

**1** Find the sum of elements from index l to r where  $0 \leq l \leq r \leq n-1$

**2** Change value of a specified element of the array to a new value x. We need to do  $\text{arr}[i] = x$  where  $0 \leq i \leq n-1$ .

A **simple solution** is to run a loop from l to r and calculate sum of elements in given range. To update a value, simply do  $\text{arr}[i] = x$ . The first operation takes  $O(n)$  time and second operation takes  $O(1)$  time.

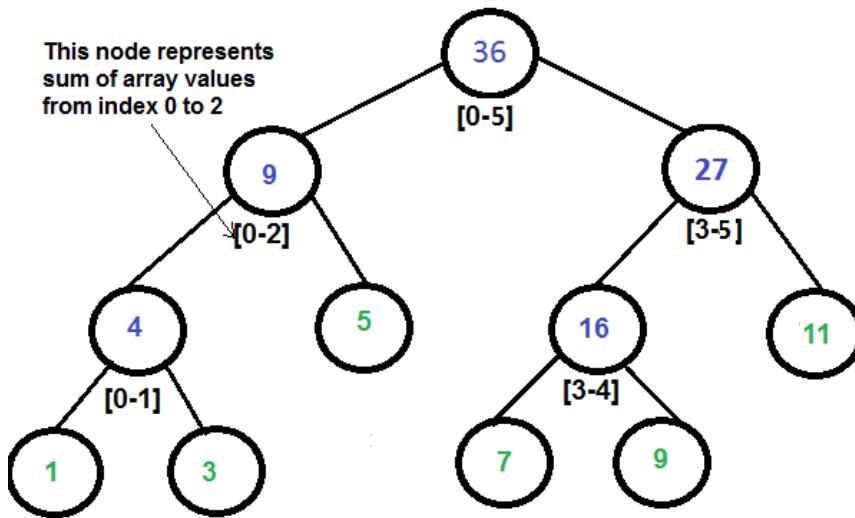
**Another solution** is to create another array and store sum from start to i at the ith index in this array. Sum of a given range can now be calculated in  $O(1)$  time, but update operation takes  $O(n)$  time now. This works well if the number of query operations are large and very few updates.

What if the number of query and updates are equal? **Can we perform both the operations in  $O(\log n)$  time once given the array?** We can use a Segment Tree to do both operations in  $O(\log n)$  time.

### Representation of Segment trees

1. Leaf Nodes are the elements of the input array.
2. Each internal node represents some merging of the leaf nodes. The merging may be different for different problems. For this problem, merging is sum of leaves under a node.

An array representation of tree is used to represent Segment Trees. For each node at index i, the left child is at index  $2*i+1$ , right child at  $2*i+2$  and the parent is at  $\lfloor (i-1)/2 \rfloor$ .



Segment Tree for input array {1, 3, 5, 7, 9, 11}

#### Construction of Segment Tree from given array

We start with a segment  $\text{arr}[0 \dots n-1]$ . and every time we divide the current segment into two halves(if it has not yet become a segment of length 1), and then call the same procedure on both halves, and for each such segment, we store the sum in the corresponding node. All levels of the constructed segment tree will be completely filled except the last level. Also, the tree will be a [Full Binary Tree](#) because we always divide segments in two halves at every level. Since the constructed tree is always a full binary tree with  $n$  leaves, there will be  $n-1$  internal nodes. So total number of nodes will be  $2*n - 1$ .

Height of the segment tree will be  $\lceil \log_2 n \rceil$ . Since the tree is represented using array and relation between parent and child indexes must be maintained, size of memory allocated for segment tree will be  $2 * 2^{\lceil \log_2 n \rceil} - 1$ .

#### Query for Sum of given range

Once the tree is constructed, how to get the sum using the constructed segment tree. Following is the algorithm to get the sum of elements.

```

int getSum(node, l, r)
{
    if the range of the node is within l and r
        return value in the node
    else if the range of the node is completely outside l and r
        return 0
    else
        return getSum(node's left child, l, r) +
               getSum(node's right child, l, r)
}
  
```

### Update a value

Like tree construction and query operations, the update can also be done recursively. We are given an index which needs to be updated. Let *diff* be the value to be added. We start from root of the segment tree and add *diff* to all nodes which have given index in their range. If a node doesn't have given index in its range, we don't make any changes to that node.

### Implementation:

Following is the implementation of segment tree. The program implements construction of segment tree for any given array. It also implements query and update operations.

### C

```
// C program to show segment tree operations like construction, query
// and update
#include <stdio.h>
#include <math.h>

// A utility function to get the middle index from corner indexes.
int getMid(int s, int e) { return s + (e - s)/2; }

/* A recursive function to get the sum of values in given range
of the array. The following are parameters for this function.

st    --> Pointer to segment tree
si    --> Index of current node in the segment tree. Initially
        0 is passed as root is always at index 0
ss & se  --> Starting and ending indexes of the segment represented
            by current node, i.e., st[si]
qs & qe  --> Starting and ending indexes of query range */
int getSumUtil(int *st, int ss, int se, int qs, int qe, int si)
{
    // If segment of this node is a part of given range, then return
    // the sum of the segment
    if (qs <= ss && qe >= se)
        return st[si];

    // If segment of this node is outside the given range
    if (se < qs || ss > qe)
        return 0;

    // If a part of this segment overlaps with the given range
    int mid = getMid(ss, se);
    return getSumUtil(st, ss, mid, qs, qe, 2*si+1) +
           getSumUtil(st, mid+1, se, qs, qe, 2*si+2);
}

/* A recursive function to update the nodes which have the given
index in their range. The following are parameters
```

```

st, si, ss and se are same as getSumUtil()
i    --> index of the element to be updated. This index is
        in the input array.
diff --> Value to be added to all nodes which have i in range */
void updateValueUtil(int *st, int ss, int se, int i, int diff, int si)
{
    // Base Case: If the input index lies outside the range of
    // this segment
    if (i < ss || i > se)
        return;

    // If the input index is in range of this node, then update
    // the value of the node and its children
    st[si] = st[si] + diff;
    if (se != ss)
    {
        int mid = getMid(ss, se);
        updateValueUtil(st, ss, mid, i, diff, 2*si + 1);
        updateValueUtil(st, mid+1, se, i, diff, 2*si + 2);
    }
}

// The function to update a value in input array and segment tree.
// It uses updateValueUtil() to update the value in segment tree
void updateValue(int arr[], int *st, int n, int i, int new_val)
{
    // Check for erroneous input index
    if (i < 0 || i > n-1)
    {
        printf("Invalid Input");
        return;
    }

    // Get the difference between new value and old value
    int diff = new_val - arr[i];

    // Update the value in array
    arr[i] = new_val;

    // Update the values of nodes in segment tree
    updateValueUtil(st, 0, n-1, i, diff, 0);
}

// Return sum of elements in range from index qs (query start)
// to qe (query end). It mainly uses getSumUtil()
int getSum(int *st, int n, int qs, int qe)
{
    // Check for erroneous input values

```

```
if (qs < 0 || qe > n-1 || qs > qe)
{
    printf("Invalid Input");
    return -1;
}

return getSumUtil(st, 0, n-1, qs, qe, 0);
}

// A recursive function that constructs Segment Tree for array[ss..se].
// si is index of current node in segment tree st
int constructSTUtil(int arr[], int ss, int se, int *st, int si)
{
    // If there is one element in array, store it in current node of
    // segment tree and return
    if (ss == se)
    {
        st[si] = arr[ss];
        return arr[ss];
    }

    // If there are more than one elements, then recur for left and
    // right subtrees and store the sum of values in this node
    int mid = getMid(ss, se);
    st[si] = constructSTUtil(arr, ss, mid, st, si*2+1) +
             constructSTUtil(arr, mid+1, se, st, si*2+2);
    return st[si];
}

/* Function to construct segment tree from given array. This function
   allocates memory for segment tree and calls constructSTUtil() to
   fill the allocated memory */
int *constructST(int arr[], int n)
{
    // Allocate memory for the segment tree

    //Height of segment tree
    int x = (int)(ceil(log2(n)));

    //Maximum size of segment tree
    int max_size = 2*(int)pow(2, x) - 1;

    // Allocate memory
    int *st = new int[max_size];

    // Fill the allocated memory st
    constructSTUtil(arr, 0, n-1, st, 0);
```

```
// Return the constructed segment tree
return st;
}

// Driver program to test above functions
int main()
{
    int arr[] = {1, 3, 5, 7, 9, 11};
    int n = sizeof(arr)/sizeof(arr[0]);

    // Build segment tree from given array
    int *st = constructST(arr, n);

    // Print sum of values in array from index 1 to 3
    printf("Sum of values in given range = %dn",
           getSum(st, n, 1, 3));

    // Update: set arr[1] = 10 and update corresponding
    // segment tree nodes
    updateValue(arr, st, n, 1, 10);

    // Find sum after the value is updated
    printf("Updated sum of values in given range = %dn",
           getSum(st, n, 1, 3));
    return 0;
}
```

### Java

```
// Java Program to show segment tree operations like construction,
// query and update
class SegmentTree
{
    int st[]; // The array that stores segment tree nodes

    /* Constructor to construct segment tree from given array. This
       constructor allocates memory for segment tree and calls
       constructSTUtil() to fill the allocated memory */
    SegmentTree(int arr[], int n)
    {
        // Allocate memory for segment tree
        //Height of segment tree
        int x = (int) (Math.ceil(Math.log(n) / Math.log(2)));

        //Maximum size of segment tree
        int max_size = 2 * (int) Math.pow(2, x) - 1;

        st = new int[max_size]; // Memory allocation
    }
}
```

```

        constructSTUtil(arr, 0, n - 1, 0);
    }

    // A utility function to get the middle index from corner indexes.
    int getMid(int s, int e) {
        return s + (e - s) / 2;
    }

    /* A recursive function to get the sum of values in given range
       of the array. The following are parameters for this function.

       st      --> Pointer to segment tree
       si      --> Index of current node in the segment tree. Initially
                   0 is passed as root is always at index 0
       ss & se  --> Starting and ending indexes of the segment represented
                     by current node, i.e., st[si]
       qs & qe  --> Starting and ending indexes of query range */
    int getSumUtil(int ss, int se, int qs, int qe, int si)
    {
        // If segment of this node is a part of given range, then return
        // the sum of the segment
        if (qs <= ss && qe >= se)
            return st[si];

        // If segment of this node is outside the given range
        if (se < qs || ss > qe)
            return 0;

        // If a part of this segment overlaps with the given range
        int mid = getMid(ss, se);
        return getSumUtil(ss, mid, qs, qe, 2 * si + 1) +
               getSumUtil(mid + 1, se, qs, qe, 2 * si + 2);
    }

    /* A recursive function to update the nodes which have the given
       index in their range. The following are parameters
       st, si, ss and se are same as getSumUtil()
       i      --> index of the element to be updated. This index is in
                   input array.
       diff --> Value to be added to all nodes which have i in range */
    void updateValueUtil(int ss, int se, int i, int diff, int si)
    {
        // Base Case: If the input index lies outside the range of
        // this segment
        if (i < ss || i > se)
            return;

```

```

// If the input index is in range of this node, then update the
// value of the node and its children
st[si] = st[si] + diff;
if (se != ss) {
    int mid = getMid(ss, se);
    updateValueUtil(ss, mid, i, diff, 2 * si + 1);
    updateValueUtil(mid + 1, se, i, diff, 2 * si + 2);
}
}

// The function to update a value in input array and segment tree.
// It uses updateValueUtil() to update the value in segment tree
void updateValue(int arr[], int n, int i, int new_val)
{
    // Check for erroneous input index
    if (i < 0 || i > n - 1) {
        System.out.println("Invalid Input");
        return;
    }

    // Get the difference between new value and old value
    int diff = new_val - arr[i];

    // Update the value in array
    arr[i] = new_val;

    // Update the values of nodes in segment tree
    updateValueUtil(0, n - 1, i, diff, 0);
}

// Return sum of elements in range from index qs (quey start) to
// qe (query end). It mainly uses getSumUtil()
int getSum(int n, int qs, int qe)
{
    // Check for erroneous input values
    if (qs < 0 || qe > n - 1 || qs > qe) {
        System.out.println("Invalid Input");
        return -1;
    }
    return getSumUtil(0, n - 1, qs, qe, 0);
}

// A recursive function that constructs Segment Tree for array[ss..se].
// si is index of current node in segment tree st
int constructSTUtil(int arr[], int ss, int se, int si)
{
    // If there is one element in array, store it in current node of
    // segment tree and return

```

```
if (ss == se) {
    st[si] = arr[ss];
    return arr[ss];
}

// If there are more than one elements, then recur for left and
// right subtrees and store the sum of values in this node
int mid = getMid(ss, se);
st[si] = constructSTUtil(arr, ss, mid, si * 2 + 1) +
         constructSTUtil(arr, mid + 1, se, si * 2 + 2);
return st[si];
}

// Driver program to test above functions
public static void main(String args[])
{
    int arr[] = {1, 3, 5, 7, 9, 11};
    int n = arr.length;
    SegmentTree tree = new SegmentTree(arr, n);

    // Build segment tree from given array

    // Print sum of values in array from index 1 to 3
    System.out.println("Sum of values in given range = " +
                       tree.getSum(n, 1, 3));

    // Update: set arr[1] = 10 and update corresponding segment
    // tree nodes
    tree.updateValue(arr, n, 1, 10);

    // Find sum after the value is updated
    System.out.println("Updated sum of values in given range = " +
                       tree.getSum(n, 1, 3));
}
}

//This code is contributed by Ankur Narain Verma
```

Output:

```
Sum of values in given range = 15
Updated sum of values in given range = 22
```

#### Time Complexity:

Time Complexity for tree construction is  $O(n)$ . There are total  $2n-1$  nodes, and value of every node is calculated only once in tree construction.

Time complexity to query is  $O(\log n)$ . To query a sum, we process at most four nodes at every level and number of levels is  $O(\log n)$ .

The time complexity of update is also  $O(\log n)$ . To update a leaf value, we process one node at every level and number of levels is  $O(\log n)$ .

[\*\*Segment Tree | Set 2 \(Range Minimum Query\)\*\*](#)

**References:**

IIT Kanpur paper.

**Source**

<https://www.geeksforgeeks.org/segment-tree-set-1-sum-of-given-range/>

## Chapter 381

# Segment Tree | Set 2 (Range Minimum Query)

Segment Tree | Set 2 (Range Minimum Query) - GeeksforGeeks

We have introduced [segment tree with a simple example](#) in the previous post. In this post, [Range Minimum Query](#) problem is discussed as another example where Segment Tree can be used. Following is problem statement.

We have an array  $\text{arr}[0 \dots n-1]$ . We should be able to efficiently find the minimum value from index  $qs$  (query start) to  $qe$  (query end) where  $0 \leq qs \leq qe \leq n-1$ .

A **simple solution** is to run a loop from  $qs$  to  $qe$  and find minimum element in given range. This solution takes  $O(n)$  time in worst case.

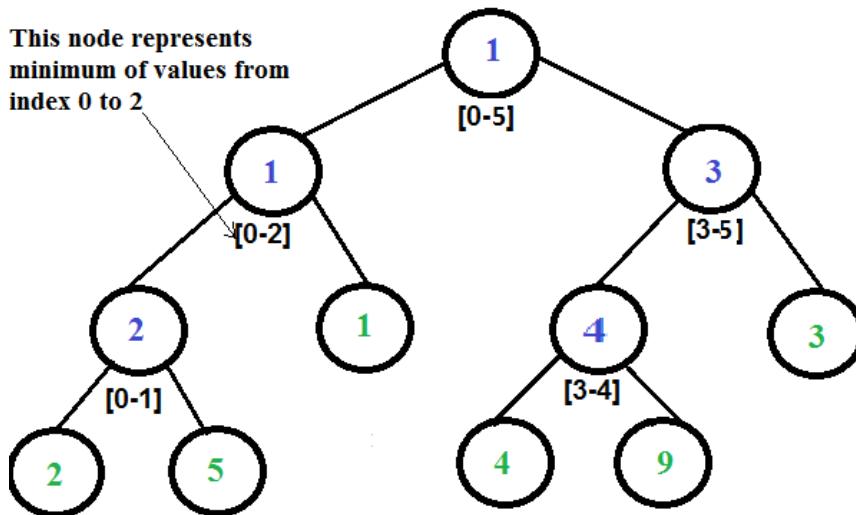
A **Another solution** is to create a 2D array where an entry  $[\text{i}, \text{j}]$  stores the minimum value in range  $\text{arr}[\text{i..j}]$ . Minimum of a given range can now be calculated in  $O(1)$  time, but preprocessing takes  $O(n^2)$  time. Also, this approach needs  $O(n^2)$  extra space which may become huge for large input arrays.

A **Segment tree** can be used to do preprocessing and query in moderate time. With segment tree, preprocessing time is  $O(n)$  and time to for range minimum query is  $O(\log n)$ . The extra space required is  $O(n)$  to store the segment tree.

Representation of Segment trees

1. Leaf Nodes are the elements of the input array.
2. Each internal node represents minimum of all leaves under it.

An array representation of tree is used to represent Segment Trees. For each node at index  $i$ , the left child is at index  $2*i+1$ , right child at  $2*i+2$  and the parent is at  $\lfloor (i-1)/2 \rfloor$



Segment Tree for input array {2, 5, 1, 4, 9, 3}

#### Construction of Segment Tree from given array

We start with a segment  $\text{arr}[0 \dots n-1]$ . and every time we divide the current segment into two halves(if it has not yet become a segment of length 1), and then call the same procedure on both halves, and for each such segment, we store the minimum value in a segment tree node.

All levels of the constructed segment tree will be completely filled except the last level. Also, the tree will be a [Full Binary Tree](#) because we always divide segments in two halves at every level. Since the constructed tree is always full binary tree with  $n$  leaves, there will be  $n-1$  internal nodes. So total number of nodes will be  $2*n - 1$ .

Height of the segment tree will be  $\lceil \log_2 n \rceil$ . Since the tree is represented using array and relation between parent and child indexes must be maintained, size of memory allocated for segment tree will be  $2 * 2^{\lceil \log_2 n \rceil} - 1$ .

#### Query for minimum value of given range

Once the tree is constructed, how to do range minimum query using the constructed segment tree. Following is algorithm to get the minimum.

```
// qs --> query start index, qe --> query end index
int RMQ(node, qs, qe)
{
    if range of node is within qs and qe
        return value in node
    else if range of node is completely outside qs and qe
        return INFINITE
    else
        return min( RMQ(node's left child, qs, qe), RMQ(node's right child, qs, qe) )
```

}

**Implementation:**

C

```
// C program for range minimum query using segment tree
#include <stdio.h>
#include <math.h>
#include <limits.h>

// A utility function to get minimum of two numbers
int minVal(int x, int y) { return (x < y)? x: y; }

// A utility function to get the middle index from corner indexes.
int getMid(int s, int e) { return s + (e - s)/2; }

/* A recursive function to get the minimum value in a given range
   of array indexes. The following are parameters for this function.

   st    --> Pointer to segment tree
   index --> Index of current node in the segment tree. Initially
              0 is passed as root is always at index 0
   ss & se --> Starting and ending indexes of the segment represented
              by current node, i.e., st[index]
   qs & qe --> Starting and ending indexes of query range */
int RMQUtil(int *st, int ss, int se, int qs, int qe, int index)
{
    // If segment of this node is a part of given range, then return
    // the min of the segment
    if (qs <= ss && qe >= se)
        return st[index];

    // If segment of this node is outside the given range
    if (se < qs || ss > qe)
        return INT_MAX;

    // If a part of this segment overlaps with the given range
    int mid = getMid(ss, se);
    return minVal(RMQUtil(st, ss, mid, qs, qe, 2*index+1),
                  RMQUtil(st, mid+1, se, qs, qe, 2*index+2));
}

// Return minimum of elements in range from index qs (query start) to
// qe (query end). It mainly uses RMQUtil()
int RMQ(int *st, int n, int qs, int qe)
{
    // Check for erroneous input values
```

```

if (qs < 0 || qe > n-1 || qs > qe)
{
    printf("Invalid Input");
    return -1;
}

return RMQUtil(st, 0, n-1, qs, qe, 0);
}

// A recursive function that constructs Segment Tree for array[ss..se].
// si is index of current node in segment tree st
int constructSTUtil(int arr[], int ss, int se, int *st, int si)
{
    // If there is one element in array, store it in current node of
    // segment tree and return
    if (ss == se)
    {
        st[si] = arr[ss];
        return arr[ss];
    }

    // If there are more than one elements, then recur for left and
    // right subtrees and store the minimum of two values in this node
    int mid = getMid(ss, se);
    st[si] = minVal(constructSTUtil(arr, ss, mid, st, si*2+1),
                    constructSTUtil(arr, mid+1, se, st, si*2+2));
    return st[si];
}

/* Function to construct segment tree from given array. This function
   allocates memory for segment tree and calls constructSTUtil() to
   fill the allocated memory */
int *constructST(int arr[], int n)
{
    // Allocate memory for segment tree

    //Height of segment tree
    int x = (int)(ceil(log2(n)));

    // Maximum size of segment tree
    int max_size = 2*(int)pow(2, x) - 1;

    int *st = new int[max_size];

    // Fill the allocated memory st
    constructSTUtil(arr, 0, n-1, st, 0);

    // Return the constructed segment tree
}

```

```
        return st;
    }

// Driver program to test above functions
int main()
{
    int arr[] = {1, 3, 2, 7, 9, 11};
    int n = sizeof(arr)/sizeof(arr[0]);

    // Build segment tree from given array
    int *st = constructST(arr, n);

    int qs = 1; // Starting index of query range
    int qe = 5; // Ending index of query range

    // Print minimum value in arr[qs..qe]
    printf("Minimum of values in range [%d, %d] is = %d\n",
           qs, qe, RMQ(st, n, qs, qe));

    return 0;
}
```

### Java

```
// Program for range minimum query using segment tree
class SegmentTreeRMQ
{
    int st[]; //array to store segment tree

    // A utility function to get minimum of two numbers
    int minVal(int x, int y) {
        return (x < y) ? x : y;
    }

    // A utility function to get the middle index from corner
    // indexes.
    int getMid(int s, int e) {
        return s + (e - s) / 2;
    }

    /* A recursive function to get the minimum value in a given
     * range of array indexes. The following are parameters for
     * this function.

    st    --> Pointer to segment tree
    index --> Index of current node in the segment tree. Initially
              0 is passed as root is always at index 0
    ss & se --> Starting and ending indexes of the segment
```

```

        represented by current node, i.e., st[index]
        qs & qe --> Starting and ending indexes of query range */
int RMQUtil(int ss, int se, int qs, int qe, int index)
{
    // If segment of this node is a part of given range, then
    // return the min of the segment
    if (qs <= ss && qe >= se)
        return st[index];

    // If segment of this node is outside the given range
    if (se < qs || ss > qe)
        return Integer.MAX_VALUE;

    // If a part of this segment overlaps with the given range
    int mid = getMid(ss, se);
    return minVal(RMQUtil(ss, mid, qs, qe, 2 * index + 1),
                  RMQUtil(mid + 1, se, qs, qe, 2 * index + 2));
}

// Return minimum of elements in range from index qs (quey
// start) to qe (query end). It mainly uses RMQUtil()
int RMQ(int n, int qs, int qe)
{
    // Check for erroneous input values
    if (qs < 0 || qe > n - 1 || qs > qe) {
        System.out.println("Invalid Input");
        return -1;
    }

    return RMQUtil(0, n - 1, qs, qe, 0);
}

// A recursive function that constructs Segment Tree for
// array[ss..se]. si is index of current node in segment tree st
int constructSTUtil(int arr[], int ss, int se, int si)
{
    // If there is one element in array, store it in current
    // node of segment tree and return
    if (ss == se) {
        st[si] = arr[ss];
        return arr[ss];
    }

    // If there are more than one elements, then recur for left and
    // right subtrees and store the minimum of two values in this node
    int mid = getMid(ss, se);
    st[si] = minVal(constructSTUtil(arr, ss, mid, si * 2 + 1),
                    constructSTUtil(arr, mid + 1, se, si * 2 + 2));
}

```

```
        return st[si];
    }

/* Function to construct segment tree from given array. This function
allocates memory for segment tree and calls constructSTUtil() to
fill the allocated memory */
void constructST(int arr[], int n)
{
    // Allocate memory for segment tree

    //Height of segment tree
    int x = (int) (Math.ceil(Math.log(n) / Math.log(2)));

    //Maximum size of segment tree
    int max_size = 2 * (int) Math.pow(2, x) - 1;
    st = new int[max_size]; // allocate memory

    // Fill the allocated memory st
    constructSTUtil(arr, 0, n - 1, 0);
}

// Driver program to test above functions
public static void main(String args[])
{
    int arr[] = {1, 3, 2, 7, 9, 11};
    int n = arr.length;
    SegmentTreeRMQ tree = new SegmentTreeRMQ();

    // Build segment tree from given array
    tree.constructST(arr, n);

    int qs = 1; // Starting index of query range
    int qe = 5; // Ending index of query range

    // Print minimum value in arr[qs..qe]
    System.out.println("Minimum of values in range [" + qs + ", "
                       + qe + "] is = " + tree.RMQ(n, qs, qe));
}
}

// This code is contributed by Ankur Narain Verma
```

Output:

```
Minimum of values in range [1, 5] is = 2
```

Time Complexity:

Time Complexity for tree construction is  $O(n)$ . There are total  $2n-1$  nodes, and value of every node is calculated only once in tree construction.

Time complexity to query is  $O(\log n)$ . To query a range minimum, we process at most two nodes at every level and number of levels is  $O(\log n)$ .

Please refer following links for more solutions to range minimum query problem.

<https://www.geeksforgeeks.org/range-minimum-query-for-static-array/>

[http://community.topcoder.com/tc?module=Static&d1=tutorials&d2=lowestCommonAncestor#Range\\_Minimum\\_Query\\_\(RMQ\)](http://community.topcoder.com/tc?module=Static&d1=tutorials&d2=lowestCommonAncestor#Range_Minimum_Query_(RMQ))

[http://wcipeg.com/wiki/Range\\_minimum\\_query](http://wcipeg.com/wiki/Range_minimum_query)

## Source

<https://www.geeksforgeeks.org/segment-tree-set-1-range-minimum-query/>

## Chapter 382

# Segment Tree | Set 3 (XOR of given range)

Segment Tree | Set 3 (XOR of given range) - GeeksforGeeks

We have an array  $\text{arr}[0 \dots n-1]$ . There are two type of queries

1. Find the XOR of elements from index l to r where  $0 \leq l \leq r \leq n-1$
2. Change value of a specified element of the array to a new value x. We need to do  $\text{arr}[i] = x$  where  $0 \leq i \leq n-1$ .

There will be total of q queries.

### Input Constraint

$n \leq 10^5$ ,  $q \leq 10^5$

### Solution 1

A simple solution is to run a loop from l to r and calculate xor of elements in given range. To update a value, simply do  $\text{arr}[i] = x$ . The first operation takes  $O(n)$  time and second operation takes  $O(1)$  time. Worst case time complexity is  $O(n*q)$  for q queries which will take huge time for  $n \sim 10^5$  and  $q \sim 10^5$ . Hence this solution will exceed time limit.

### Solution 2

Another solution is to store xor in all possible ranges but there are  $O(n^2)$  possible ranges hence with  $n \sim 10^5$  it wil exceed space complexity, hence without considering time complexity, we can state this solution will not work.

### Solution 3 (Segment Tree)

Prerequisite : [Segment Tree](#)

We build a segment tree of given array such that array elements are at leaves and internal nodes store XOR of leaves covered under them.

```

// C program to show segment tree operations like construction,
// query and update
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

// A utility function to get the middle index from corner indexes.
int getMid(int s, int e) { return s + (e - s)/2; }

/* A recursive function to get the xor of values in given range
   of the array. The following are parameters for this function.

   st    --> Pointer to segment tree
   si    --> Index of current node in the segment tree. Initially
             0 is passed as root is always at index 0
   ss & se  --> Starting and ending indexes of the segment
                 represented by current node, i.e., st[si]
   qs & qe  --> Starting and ending indexes of query range */
int getXorUtil(int *st, int ss, int se, int qs, int qe, int si)
{
    // If segment of this node is a part of given range, then return
    // the xor of the segment
    if (qs <= ss && qe >= se)
        return st[si];

    // If segment of this node is outside the given range
    if (se < qs || ss > qe)
        return 0;

    // If a part of this segment overlaps with the given range
    int mid = getMid(ss, se);
    return getXorUtil(st, ss, mid, qs, qe, 2*si+1) ^
           getXorUtil(st, mid+1, se, qs, qe, 2*si+2);
}

/* A recursive function to update the nodes which have the given
   index in their range. The following are parameters
   st, si, ss and se are same as getXorUtil()
   i    --> index of the element to be updated. This index is
           in input array.
   diff --> Value to be added to all nodes which have i in range */
void updateValueUtil(int *st, int ss, int se, int i, int diff, int si)
{
    // Base Case: If the input index lies outside the range of
    // this segment
    if (i < ss || i > se)
        return;

```

```
// If the input index is in range of this node, then update
// the value of the node and its children
st[si] = st[si] + diff;
if (se != ss)
{
    int mid = getMid(ss, se);
    updateValueUtil(st, ss, mid, i, diff, 2*si + 1);
    updateValueUtil(st, mid+1, se, i, diff, 2*si + 2);
}
}

// The function to update a value in input array and segment tree.
// It uses updateValueUtil() to update the value in segment tree
void updateValue(int arr[], int *st, int n, int i, int new_val)
{
    // Check for erroneous input index
    if (i < 0 || i > n-1)
    {
        printf("Invalid Input");
        return;
    }

    // Get the difference between new value and old value
    int diff = new_val - arr[i];

    // Update the value in array
    arr[i] = new_val;

    // Update the values of nodes in segment tree
    updateValueUtil(st, 0, n-1, i, diff, 0);
}

// Return xor of elements in range from index qs (quey start)
// to qe (query end). It mainly uses getXorUtil()
int getXor(int *st, int n, int qs, int qe)
{
    // Check for erroneous input values
    if (qs < 0 || qe > n-1 || qs > qe)
    {
        printf("Invalid Input");
        return -1;
    }

    return getXorUtil(st, 0, n-1, qs, qe, 0);
}

// A recursive function that constructs Segment Tree for array[ss..se].
// si is index of current node in segment tree st
```

```
int constructSTUtil(int arr[], int ss, int se, int *st, int si)
{
    // If there is one element in array, store it in current node of
    // segment tree and return
    if (ss == se)
    {
        st[si] = arr[ss];
        return arr[ss];
    }

    // If there are more than one elements, then recur for left and
    // right subtrees and store the xor of values in this node
    int mid = getMid(ss, se);
    st[si] = constructSTUtil(arr, ss, mid, st, si*2+1) ^
             constructSTUtil(arr, mid+1, se, st, si*2+2);
    return st[si];
}

/* Function to construct segment tree from given array. This function
   allocates memory for segment tree and calls constructSTUtil() to
   fill the allocated memory */
int *constructST(int arr[], int n)
{
    // Allocate memory for segment tree

    //Height of segment tree
    int x = (int)(ceil(log2(n)));

    //Maximum size of segment tree
    int max_size = 2*(int)pow(2, x) - 1;

    // Allocate memory
    int *st = (int *)malloc(sizeof(int)*max_size);

    // Fill the allocated memory st
    constructSTUtil(arr, 0, n-1, st, 0);

    // Return the constructed segment tree
    return st;
}

// Driver program to test above functions
int main()
{
    int arr[] = {1, 3, 5, 7, 9, 11};
    int n = sizeof(arr)/sizeof(arr[0]);

    // Build segment tree from given array
```

```
int *st = constructST(arr, n);

// Print xor of values in array from index 1 to 3
printf("Xor of values in given range = %d\n",
       getXor(st, n, 1, 3));

// Update: set arr[1] = 10 and update corresponding
// segment tree nodes
updateValue(arr, st, n, 1, 10);

// Find xor after the value is updated
printf("Updated xor of values in given range = %d\n",
       getXor(st, n, 1, 3));
return 0;
}
```

Output:

```
Xor of values in given range = 1
Updated xor of values in given range = 8
```

#### Time and Space Complexity:

Time Complexity for tree construction is  $O(n)$ . There are total  $2n-1$  nodes, and value of every node is calculated only once in tree construction.

Time complexity to query is  $O(\log n)$ .

The time complexity of update is also  $O(\log n)$ .

Total time Complexity is :  $O(n)$  for construction +  $O(\log n)$  for each query =  $O(n) + O(n * \log n) = O(n * \log n)$

Time Complexity  $O(n * \log n)$   
Auxiliary Space  $O(n)$

#### Source

<https://www.geeksforgeeks.org/segment-tree-set-3-xor-given-range/>

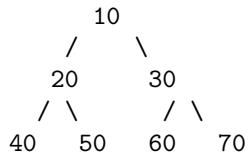
## Chapter 383

# Select a Random Node from a tree with equal probability

Select a Random Node from a tree with equal probability - GeeksforGeeks

Given a Binary Tree with children Nodes, Return a random Node with equal Probability of selecting any Node in tree.

Consider the given tree with root as 1.



Examples:

```
Input : getRandom(root);
Output : A Random Node From Tree : 3
```

```
Input : getRandom(root);
Output : A Random Node From Tree : 2
```

A **simple solution** is to store [Inorder traversal of tree](#) in an array. Let the count of nodes be n. To get a random node, we generate a random number from 0 to n-1, use this number as index in array and return the value at index.

An **alternate solution** is to modify tree structure. We store count of children in every node. Consider the above tree. We use inorder traversal here also. We generate a number smaller than or equal count of nodes. We traverse tree and go to the node at that index.

We use counts to quickly reach the desired node. With counts, we reach in  $O(h)$  time where  $h$  is height of tree.

```
    10,6
   /   \
  20,2   30,2
 / \   / \
40,0 50,0 60,0 70,0
The first value is node and second
value is count of children.
```

We start traversing the tree, on each node we either go to left subtree or right subtree considering whether the count of children is less than random count or not.

If the random count is less than the count of children then we go left else we go right.

Below is the implementation of above Algorithm. getElements will return count of children for root, InsertChildrenCount inserts children data to each node, RandomNode return the random node with the help of Utility Function RandomNodeUtil.

## Source

<https://www.geeksforgeeks.org/select-random-node-tree-equal-probability/>

C++

```
// CPP program to Select a Random Node from a tree
#include <bits/stdc++.h>
using namespace std;

struct Node {
    int data;
    int children;
    Node *left, *right;
};

Node* newNode(int data)
{
    Node *temp = new Node;
    temp->data = data;
    temp->left = temp->right = NULL;
    temp->children = 0;
    return temp;
}

// This is used to fill children counts.
int getElements(Node* root)
```

```
{  
    if (!root)  
        return 0;  
    return getElements(root->left) +  
        getElements(root->right) + 1;  
}  
  
// Inserts Children count for each node  
void insertChildrenCount(Node*& root)  
{  
    if (!root)  
        return;  
  
    root->children = getElements(root) - 1;  
    insertChildrenCount(root->left);  
    insertChildrenCount(root->right);  
}  
  
// returns number of children for root  
int children(Node* root)  
{  
    if (!root)  
        return 0;  
    return root->children + 1;  
}  
  
// Helper Function to return a random node  
int randomNodeUtil(Node* root, int count)  
{  
    if (!root)  
        return 0;  
  
    if (count == children(root->left))  
        return root->data;  
  
    if (count < children(root->left))  
        return randomNodeUtil(root->left, count);  
  
    return randomNodeUtil(root->right,  
        count - children(root->left) - 1);  
}  
  
// Returns Random node  
int randomNode(Node* root)  
{  
    srand(time(0));  
  
    int count = rand() % (root->children + 1);
```

```
    return randomNodeUtil(root, count);
}

int main()
{
    // Creating Above Tree
    Node* root = newNode(10);
    root->left = newNode(20);
    root->right = newNode(30);
    root->left->right = newNode(40);
    root->left->right = newNode(50);
    root->right->left = newNode(60);
    root->right->right = newNode(70);

    insertChildrenCount(root);

    cout << "A Random Node From Tree : "
        << randomNode(root) << endl;

    return 0;
}
```

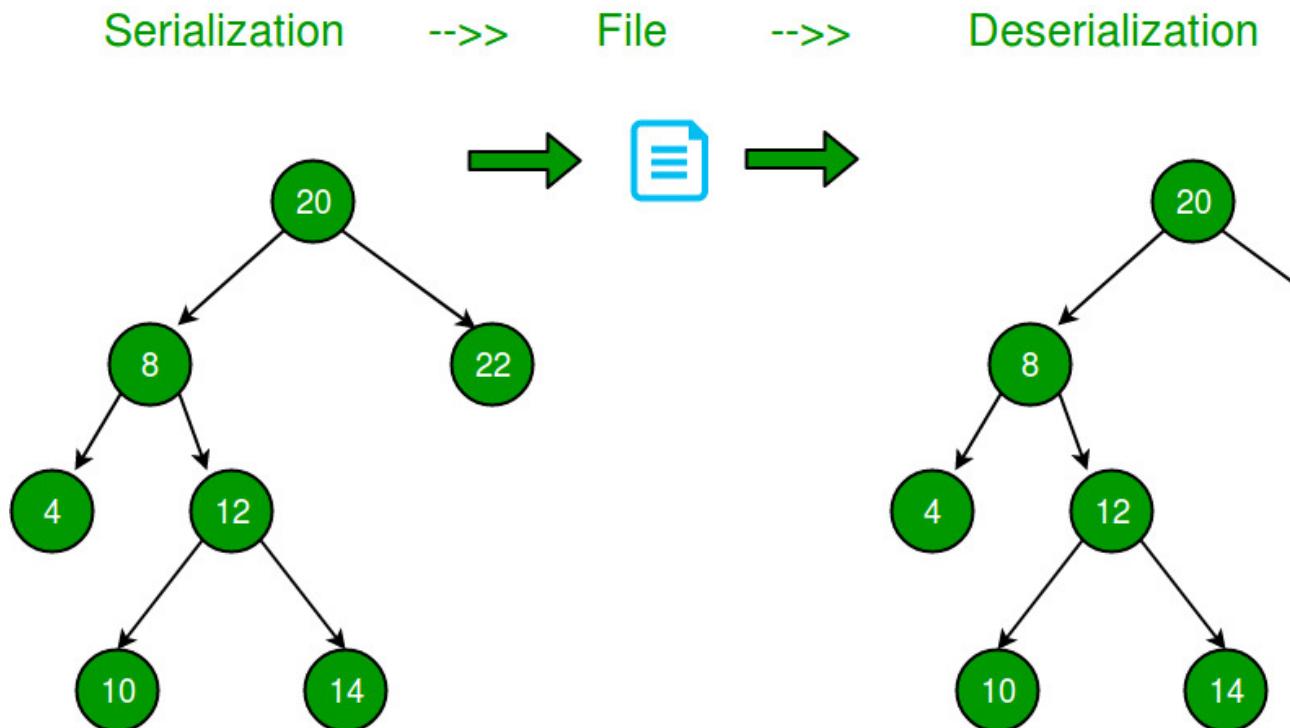
Time Complexity of randomNode is  $O(h)$  where  $h$  is height of tree. Note that we are either moving to right or to left at a time.

## Chapter 384

# Serialize and Deserialize a Binary Tree

Serialize and Deserialize a Binary Tree - GeeksforGeeks

Serialization is to store tree in a file so that it can be later restored. The structure of tree must be maintained. Deserialization is reading tree back from file.



Following are some simpler versions of the problem:

**If given Tree is Binary Search Tree?**

If the given Binary Tree is Binary Search Tree, we can store it by either storing preorder or postorder traversal. In case of Binary Search Trees, only [preorder or postorder traversal is sufficient to store structure information](#).

**If given Binary Tree is Complete Tree?**

A Binary Tree is complete if all levels are completely filled except possibly the last level and all nodes of last level are as left as possible (Binary Heaps are complete Binary Tree). For a complete Binary Tree, level order traversal is sufficient to store the tree. We know that the first node is root, next two nodes are nodes of next level, next four nodes are nodes of 2nd level and so on.

**If given Binary Tree is Full Tree?**

A full Binary is a Binary Tree where every node has either 0 or 2 children. It is easy to serialize such trees as every internal node has 2 children. We can simply store preorder traversal and store a bit with every node to indicate whether the node is an internal node or a leaf node.

**How to store a general Binary Tree?**

A simple solution is to store both Inorder and Preorder traversals. This solution requires space twice the size of Binary Tree.

We can save space by storing Preorder traversal and a marker for NULL pointers.

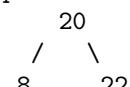
Let the marker for NULL pointers be '-1'

Input:



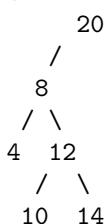
Output: 12 13 -1 -1 -1

Input:



Output: 20 8 -1 -1 22 -1 -1

Input:



Output: 20 8 4 -1 -1 12 10 -1 -1 14 -1 -1 -1

Input:

20

```
      /
     8
    /
   10
  /
 5
Output: 20 8 10 5 -1 -1 -1 -1 -1
```

Input:

```
 20
  \
   8
   \
    10
    \
     5
Output: 20 -1 8 -1 10 -1 5 -1 -1
```

Deserialization can be done by simply reading data from file one by one.

Following is C++ implementation of the above idea.

```
// A C++ program to demonstrate serialization and deserialization of
// Binary Tree
#include <stdio.h>
#define MARKER -1

/* A binary tree Node has key, pointer to left and right children */
struct Node
{
    int key;
    struct Node* left, *right;
};

/* Helper function that allocates a new Node with the
   given key and NULL left and right pointers. */
Node* newNode(int key)
{
    Node* temp = new Node;
    temp->key = key;
    temp->left = temp->right = NULL;
    return (temp);
}

// This function stores a tree in a file pointed by fp
void serialize(Node *root, FILE *fp)
{
    // If current node is NULL, store marker
```

```
if (root == NULL)
{
    fprintf(fp, "%d ", MARKER);
    return;
}

// Else, store current node and recur for its children
fprintf(fp, "%d ", root->key);
serialize(root->left, fp);
serialize(root->right, fp);
}

// This function constructs a tree from a file pointed by 'fp'
void deSerialize(Node *&root, FILE *fp)
{
    // Read next item from file. If there are no more items or next
    // item is marker, then return
    int val;
    if ( !fscanf(fp, "%d ", &val) || val == MARKER)
        return;

    // Else create node with this item and recur for children
    root = newNode(val);
    deSerialize(root->left, fp);
    deSerialize(root->right, fp);
}

// A simple inorder traversal used for testing the constructed tree
void inorder(Node *root)
{
    if (root)
    {
        inorder(root->left);
        printf("%d ", root->key);
        inorder(root->right);
    }
}

/* Driver program to test above functions*/
int main()
{
    // Let us construct a tree shown in the above figure
    struct Node *root      = newNode(20);
    root->left           = newNode(8);
    root->right          = newNode(22);
    root->left->left    = newNode(4);
    root->left->right   = newNode(12);
    root->left->right->left = newNode(10);
```

```
root->left->right->right = newNode(14);

// Let us open a file and serialize the tree into the file
FILE *fp = fopen("tree.txt", "w");
if (fp == NULL)
{
    puts("Could not open file");
    return 0;
}
serialize(root, fp);
fclose(fp);

// Let us deserialize the stored tree into root1
Node *root1 = NULL;
fp = fopen("tree.txt", "r");
deSerialize(root1, fp);

printf("Inorder Traversal of the tree constructed from file:\n");
inorder(root1);

return 0;
}
```

Output:

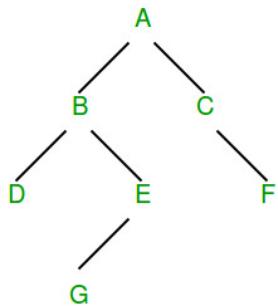
```
Inorder Traversal of the tree constructed from file:
4 8 10 12 14 20 22
```

#### How much extra space is required in above solution?

If there are  $n$  keys, then the above solution requires  $n+1$  markers which may be better than simple solution (storing keys twice) in situations where keys are big or keys have big data items associated with them.

#### Can we optimize it further?

The above solution can be optimized in many ways. If we take a closer look at above serialized trees, we can observe that all leaf nodes require two markers. One simple optimization is to store a separate bit with every node to indicate that the node is internal or external. This way we don't have to store two markers with every leaf node as leaves can be identified by extra bit. We still need marker for internal nodes with one child. For example in the following diagram ‘\*’ is used to indicate an internal node set bit, and ‘/’ is used as NULL marker. The diagram is taken from [here](#).

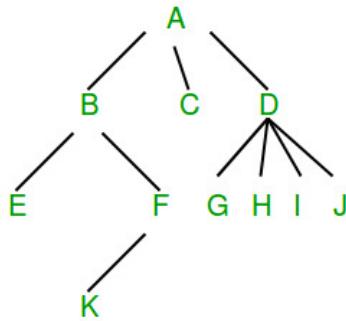


A'B'DE'G/C'/F

Please note that there are always more leaf nodes than internal nodes in a Binary Tree  
(Number of leaf nodes is number of internal nodes plus 1, so this optimization makes sense.)

#### How to serialize n-ary tree?

In an n-ary tree, there is no designated left or right child. We can store an ‘end of children’ marker with every node. The following diagram shows serialization where ‘)’ is used as end of children marker. We will soon be covering implementation for n-ary tree. The diagram is taken from [here](#).



ABE) FK) ) ) C) DG) H) I) J) ) )

#### References:

<http://www.cs.usfca.edu/~brooks/S04classes/cs245/lectures/lecture11.pdf>

This article is contributed by **Shivam Gupta**, Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Improved By : [sanjanab](#)

#### Source

<https://www.geeksforgeeks.org/serialize-deserialize-binary-tree/>

## Chapter 385

# Serialize and Deserialize an N-ary Tree

[Serialize and Deserialize an N-ary Tree - GeeksforGeeks](#)

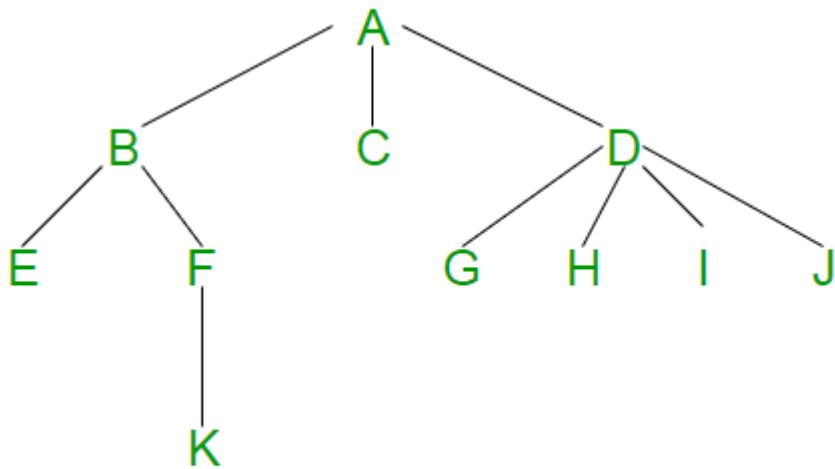
Given an N-ary tree where every node has at-most N children. How to serialize and deserialize it? Serialization is to store tree in a file so that it can be later restored. The structure of tree must be maintained. Deserialization is reading tree back from file.

This post is mainly an extension of below post.

[Serialize and Deserialize a Binary Tree](#)

In an N-ary tree, there are no designated left and right children. An N-ary tree is represented by storing an array or list of child pointers with every node.

The idea is to store an ‘end of children’ marker with every node. The following diagram shows serialization where ‘)’ is used as end of children marker.



**ABE)FK)))C)DG)H)I)J))))**

Following is C++ implementation of above idea.

```

// A C++ Program serialize and deserialize an N-ary tree
#include<cstdio>
#define MARKER ')'
#define N 5
using namespace std;

// A node of N-ary tree
struct Node {
    char key;
    Node *child[N]; // An array of pointers for N children
};

// A utility function to create a new N-ary tree node
Node *newNode(char key)
{
    Node *temp = new Node;
    temp->key = key;
    for (int i = 0; i < N; i++)
        temp->child[i] = NULL;
    return temp;
}

// This function stores the given N-ary tree in a file pointed by fp
void serialize(Node *root, FILE *fp)
{
  
```

```

// Base case
if (root == NULL) return;

// Else, store current node and recur for its children
fprintf(fp, "%c ", root->key);
for (int i = 0; i < N && root->child[i]; i++)
    serialize(root->child[i], fp);

// Store marker at the end of children
fprintf(fp, "%c ", MARKER);
}

// This function constructs N-ary tree from a file pointed by 'fp'.
// This function returns 0 to indicate that the next item is a valid
// tree key. Else returns 0
int deSerialize(Node *&root, FILE *fp)
{
    // Read next item from file. If there are no more items or next
    // item is marker, then return 1 to indicate same
    char val;
    if ( !fscanf(fp, "%c ", &val) || val == MARKER )
        return 1;

    // Else create node with this item and recur for children
    root = newNode(val);
    for (int i = 0; i < N; i++)
        if (deSerialize(root->child[i], fp))
            break;

    // Finally return 0 for successful finish
    return 0;
}

// A utility function to create a dummy tree shown in above diagram
Node *createDummyTree()
{
    Node *root = newNode('A');
    root->child[0] = newNode('B');
    root->child[1] = newNode('C');
    root->child[2] = newNode('D');
    root->child[0]->child[0] = newNode('E');
    root->child[0]->child[1] = newNode('F');
    root->child[2]->child[0] = newNode('G');
    root->child[2]->child[1] = newNode('H');
    root->child[2]->child[2] = newNode('I');
    root->child[2]->child[3] = newNode('J');
    root->child[0]->child[1]->child[0] = newNode('K');
    return root;
}

```

```
}

// A utility function to traverse the constructed N-ary tree
void traverse(Node *root)
{
    if (root)
    {
        printf("%c ", root->key);
        for (int i = 0; i < N; i++)
            traverse(root->child[i]);
    }
}

// Driver program to test above functions
int main()
{
    // Let us create an N-ary tree shown in above diagram
    Node *root = createDummyTree();

    // Let us open a file and serialize the tree into the file
    FILE *fp = fopen("tree.txt", "w");
    if (fp == NULL)
    {
        puts("Could not open file");
        return 0;
    }
    serialize(root, fp);
    fclose(fp);

    // Let us deserialize the stored tree into root1
    Node *root1 = NULL;
    fp = fopen("tree.txt", "r");
    deSerialize(root1, fp);

    printf("Constructed N-Ary Tree from file is \n");
    traverse(root1);

    return 0;
}
```

Output:

```
Constructed N-Ary Tree from file is
A B E F K C D G H I J
```

The above implementation can be optimized in many ways for example by using a vector in place of array of pointers. We have kept it this way to keep it simple to read and understand.

This article is contributed by **varun**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## Source

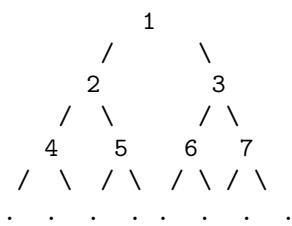
<https://www.geeksforgeeks.org/serialize-deserialize-n-ary-tree/>

## Chapter 386

# Shortest distance between two nodes in an infinite binary tree

Shortest distance between two nodes in an infinite binary tree - GeeksforGeeks

Consider you have an infinitely long binary tree having a pattern as below:



Given two nodes with values x and y. The task is to find the length of the shortest path between the two nodes.

**Examples:**

Input: x = 2, y = 3  
Output: 2

Input: x = 4, y = 6  
Output: 4

A **naive approach** is to store all the ancestors of both nodes in 2 Data-structures([vectors](#), [arrays](#), etc..) and do a [binary search](#) for the first element(let index i) in vector1, and check if it exists in the vector2 or not. If it does, return the index(let x) of the element in vector2. The answer will be thus

distance = v1.size() - 1 - i + v2.size() - 1 - x

Below is the implementation of the above approach.

```
// CPP program to find distance
// between two nodes
// in a infinite binary tree
#include <bits/stdc++.h>
using namespace std;

// to stores ancestors of first given node
vector<int> v1;
// to stores ancestors of first given node
vector<int> v2;

// normal binary search to find the element
int BinarySearch(int x)
{
    int low = 0;
    int high = v2.size() - 1;

    while (low <= high) {
        int mid = (low + high) / 2;

        if (v2[mid] == x)
            return mid;
        else if (v2[mid] > x)
            high = mid - 1;
        else
            low = mid + 1;
    }
    return -1;
}

// function to make ancestors of first node
void MakeAncestorNode1(int x)
{
    v1.clear();
    while (x) {
        v1.push_back(x);
        x /= 2;
    }
    reverse(v1.begin(), v1.end());
}

// function to make ancestors of second node
void MakeAncestorNode2(int x)
{
```

```
v2.clear();
while (x) {
    v2.push_back(x);
    x /= 2;
}
reverse(v2.begin(), v2.end());
}

// function to find distance bewteen two nodes
int Distance()
{
    for (int i = v1.size() - 1; i >= 0; i--) {
        int x = BinarySearch(v1[i]);
        if (x != -1) {
            return v1.size() - 1 - i + v2.size() - 1 - x;
        }
    }
}

// Driver code
int main()
{
    int node1 = 2, node2 = 3;

    // find ancestors
    MakeAncestorNode1(node1);
    MakeAncestorNode2(node2);

    cout << "Distance between " << node1 <<
    " and " << node2 << " is : " << Distance();

    return 0;
}
```

**Output:**

```
Distance between 2 and 3 is : 2
```

**Time Complexity:**  $O(\log(\max(x, y)) * \log(\max(x, y)))$   
**Auxiliary Space:**  $O(\log(\max(x, y)))$

An **efficient approach** is to use the property of  $2^x$  and  $2^x+1$  given. Keep dividing the larger of the two nodes by 2. If the larger becomes the smaller one, then divide the other one. At a stage, both the values will be the same, keep a count on the number of divisions done which will be the answer.

Below is the implementation of the above approach.

```
// C++ program to find the distance
// between two nodes in an infinite
// binary tree
#include <bits/stdc++.h>
using namespace std;

// function to find the distance
// between two nodes in an infinite
// binary tree
int Distance(int x, int y)
{
    // swap the smaller
    if (x < y) {
        swap(x, y);
    }
    int c = 0;

    // divide till x!=y
    while (x != y) {

        // keep a count
        ++c;

        // perform division
        if (x > y)
            x = x >> 1;

        // when the smaller
        // becomes the greater
        if (y > x) {
            y = y >> 1;
            ++c;
        }
    }
    return c;
}

// Driver code
int main()
{
    int x = 4, y = 6;
    cout << Distance(x, y);

    return 0;
}
```

**Output:**

4

**Time Complexity:**  $O(\log(\max(x, y)))$

**Auxiliary Space:**  $O(1)$

The efficient approach has been suggested by [Striver](#).

### Source

<https://www.geeksforgeeks.org/shortest-distance-between-two-nodes-in-an-infinite-binary-tree/>

## Chapter 387

# Shortest path between two nodes in array like representation of binary tree

Shortest path between two nodes in array like representation of binary tree - GeeksforGeeks

Consider a binary tree in which each node has two children except the leaf nodes. If a node is labeled as 'v' then its right children will be labeled as  $2v+1$  and left children as  $2v$ . Root is labelled as

Given two nodes labeled as  $i$  and  $j$ , the task is to find the shortest distance and the path from  $i$  to  $j$ . And print the path of node  $i$  and node  $j$  from root node.

Examples:

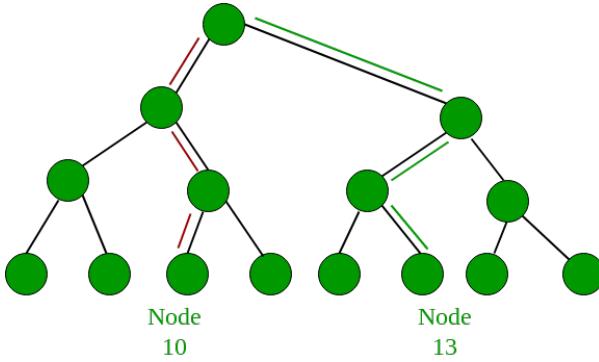
```
Input : i = 1, j = 2
Output : 1
         Path is 1 2
```

```
Input : i = 4, j = 3
Output : 3
         Path is 4 2 1 3
```

This problem is mainly an extension of [Find distance between two given keys of a Binary Tree](#). Here we not only find shortest distance, but also the path.

The between the two nodes  $i$  and  $j$  will be equal to  $\text{dist}(i, \text{LCA}(i, j)) + \text{dist}(j, \text{LCA}(i, j))$  where LCA means lowest common ancestor of nodes labeled as  $i$  and  $j$ . If a number  $x$  is represented in binary form then  $2^x$  can be represented by appending 0 to the binary representation of  $x$  and  $2x+1$  can be represented by appending 1 to the binary representation of  $x$ . This is because when we append 0 all the terms present in binary form

of  $x$  shift left, so it gets doubled similarly when we append 1, we get  $2x+1$ . Suppose the binary representation of a node is 1010 this tells us the path of this node from root. First term '1' represents root second term 0 represents left turn then third term 1 represents right turn from previous node and finally 0 represents left turn.



Node 10 in binary form is 1010 and 13 in binary form is 1101 secondly length of binary representation of any node also tells about its level in binary tree. Suppose binary representation of i is m length and is  $i_1 i_2 \dots i_m$  and binary representation of node j is n length  $j_1 j_2 \dots j_n$ .

Thus we know the path of i and j from root .Find out k such that for all  $p \leq k$   $i_p = j_p$ .This is the LCA of i and j in binary form .So  $\text{dist}(i, \text{LCA}(i, j))$  will be  $m - k$  and  $\text{dist}(j, \text{LCA}(i, j)) = n - k$ . so answer will be  $m + n - 2k$ . And printing the path is also not a big issue just store the path of i to LCA and path of j to LCA and concatenate them.

```
// c++ representation of finding shortest
// distance between node i and j
#include <bits/stdc++.h>
using namespace std;

// prints the path between node i and node j
void ShortestPath(int i, int j, int k, int m,
                  int n)
{
    // path1 stores path of node i to lca and
    // path2 stores path of node j to lca
    vector<int> path1, path2;
    int x = m - 1;

    // push node i in path1
    path1.push_back(i);

    // keep pushing parent of node labelled
```

```
// as i to path1 until lca is reached
while (x != k) {
    path1.push_back(i / 2);
    i = i / 2;
    x--;
}
int y = n - 1;

// push node j to path2
path2.push_back(j);

// keep pushing parent of node j till
// lca is reached
while (y != k) {
    path2.push_back(j / 2);
    j = j / 2;
    y--;
}

// printing path from node i to lca
for (int l = 0; l < path1.size(); l++)
    cout << path1[l] << " ";

// printing path from lca to node j
for (int l = path2.size() - 2; l >= 0; l--)
    cout << path2[l] << " ";
cout << endl;
}

// returns the shortest distance between
// nodes labelled as i and j
int ShortestDistance(int i, int j)
{
    // vector to store binary form of i and j
    vector<int> v1, v2;

    // finding binary form of i and j
    int p1 = i;
    int p2 = j;
    while (i != 0) {
        v1.push_back(i % 2);
        i = i / 2;
    }
    while (j != 0) {
        v2.push_back(j % 2);
        j = j / 2;
    }
}
```

```
// as binary form will be in reverse order
// reverse the vectors
reverse(v1.begin(), v1.end());
reverse(v2.begin(), v2.end());

// finding the k that is lca (i, j)
int m = v1.size(), n = v2.size(), k = 0;
if (m < n) {
    while (k < m && v1[k] == v2[k])
        k++;
} else {
    while (k < n && v1[k] == v2[k])
        k++;
}
ShortestPath(p1, p2, k - 1, m, n);
return m + n - 2 * k;
}

// Driver function
int main()
{
    cout << ShortestDistance(1, 2) << endl;
    cout << ShortestDistance(4, 3) << endl;
    return 0;
}
```

Output:

```
1 2
1
4 2 1 3
3
```

Time Complexity  $O(\log Z_i + \log Z_j)$

## Source

<https://www.geeksforgeeks.org/shortest-path-between-two-nodes-in-array-like-representation-of-binary-tree/>

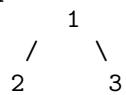
## Chapter 388

# Sink Odd nodes in Binary Tree

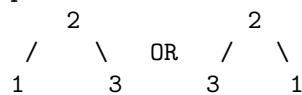
Sink Odd nodes in Binary Tree - GeeksforGeeks

Given a Binary Tree having odd and even elements, sink all its odd valued nodes such that no node with odd value could be parent of node with even value. There can be multiple outputs for a given tree, we need to print one of them. It is always possible to convert a tree (Note that a node with even nodes and all odd nodes follows the rule)

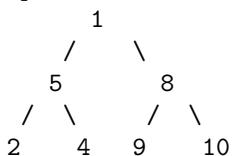
Input :



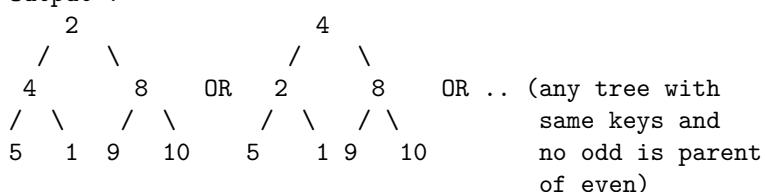
Output



Input :



Output :



We strongly recommend you to minimize your browser and try this yourself first.

Basically, we need to swap odd value of a node with even value of one of its descendants. The idea is to traverse the tree in postorder fashion. Since we process in postorder, for each odd node encountered, its left and right subtrees are already balanced (sank), we check if it's an odd node and its left or right child has an even value. If even value is found, we swap the node's data with that of even child node and call the procedure on the even child to balance the subtree. If both children have odd values, that means that all its descendants are odd.

Below is C++ implementation of the idea.

```
// Program to sink odd nodes to the bottom of
// binary tree
#include<bits/stdc++.h>
using namespace std;

// A binary tree node
struct Node
{
    int data;
    Node* left, *right;
};

// Helper function to allocates a new node
Node* newnode(int data)
{
    Node* node = new Node;
    node->data = data;
    node->left = node->right = NULL;
    return node;
}

// Helper function to check if node is leaf node
bool isLeaf(Node *root)
{
    return (root->left == NULL && root->right == NULL);
}

// A recursive method to sink a tree with odd root
// This method assumes that the subtrees are already
// sanked. This method is similar to Heapify of
// Heap-Sort
void sink(Node *&root)
{
    // If NULL or is a leaf, do nothing
    if (root == NULL || isLeaf(root))
        return;
```

```
// if left subtree exists and left child is even
if (root->left && !(root->left->data & 1))
{
    // swap root's data with left child and
    // fix left subtree
    swap(root->data, root->left->data);
    sink(root->left);
}

// if right subtree exists and right child is even
else if (root->right && !(root->right->data & 1))
{
    // swap root's data with right child and
    // fix right subtree
    swap(root->data, root->right->data);
    sink(root->right);
}

// Function to sink all odd nodes to the bottom of binary
// tree. It does a postorder traversal and calls sink()
// if any odd node is found
void sinkOddNodes(Node* &root)
{
    // If NULL or is a leaf, do nothing
    if (root == NULL || isLeaf(root))
        return;

    // Process left and right subtrees before this node
    sinkOddNodes(root->left);
    sinkOddNodes(root->right);

    // If root is odd, sink it
    if (root->data & 1)
        sink(root);
}

// Helper function to do Level Order Traversal of
// Binary Tree level by level. This function is used
// here only for showing modified tree.
void printLevelOrder(Node* root)
{
    queue<Node*> q;
    q.push(root);

    // Do Level order traversal
    while (!q.empty())
```

```

{
    int nodeCount = q.size();

    // Print one level at a time
    while (nodeCount)
    {
        Node *node = q.front();
        printf("%d ", node->data);
        q.pop();
        if (node->left != NULL)
            q.push(node->left);
        if (node->right != NULL)
            q.push(node->right);
        nodeCount--;
    }

    // Line separator for levels
    printf("\n");
}
}

// Driver program to test above functions
int main()
{
    /* Constructed binary tree is
       1
      /   \
     5     8
    / \   / \
   2   4 9   10 */
}

Node *root = newnode(1);
root->left = newnode(5);
root->right = newnode(8);
root->left->left = newnode(2);
root->left->right = newnode(4);
root->right->left = newnode(9);
root->right->right = newnode(10);

sinkOddNodes(root);

printf("Level order traversal of modified tree\n");
printLevelOrder(root);

return 0;
}

```

Output :

```
Level order traversal of modified tree
2
4 8
5 1 9 10
```

This article is contributed by **Aditya Goel**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## Source

<https://www.geeksforgeeks.org/sink-odd-nodes-binary-tree/>

## Chapter 389

# Smallest Subarray with given GCD

Smallest Subarray with given GCD - GeeksforGeeks

Given an array arr[] of n numbers and an integer k, find length of the minimum sub-array with gcd equals to k.

Example:

```
Input: arr[] = {6, 9, 7, 10, 12,
                24, 36, 27},
           K = 3
Output: 2
Explanation: GCD of subarray {6,9} is 3.
GCD of subarray {24,36,27} is also 3,
but {6,9} is the smallest
```

Note: Time complexity analysis of below approaches assume that numbers are fixed size and finding GCD of two elements take constant time.

### Method 1

Find GCD of all subarrays and keep track of the minimum length subarray with gcd k. Time Complexity of this is  $O(n^3)$ ,  $O(n^2)$  for each subarray and  $O(n)$  for finding gcd of a subarray.

### Method 2

Find GCD of all subarrays using segment tree based approach discussed [here](#). Time complexity of this solution is  $O(n^2 \log n)$ ,  $O(n^2)$  for each subarray and  $O(\log n)$  for finding GCD of subarray using segment tree.

### Method 3

The idea is to use [Segment Tree](#) and Binary Search to achieve time complexity  $O(n (\log n)^2)$ .

1. If we have any number equal to 'k' in the array then the answer is 1 as GCD of k is k. Return 1.
2. If there is no number which is divisible by k, then GCD doesn't exist. Return -1.
3. If none of the above cases is true, the length of minimum subarray is either greater than 1 or GCD doesn't exist. In this case, we follow following steps.
  - Build segment tree so that we can quickly find GCD of any subarray using the approach discussed [here](#)
  - After building Segment Tree, we consider every index as starting point and do binary search for ending point such that the subarray between these two points has GCD k

Following is C++ implementation of above idea.

```
// C++ Program to find GCD of a number in a given Range
// using segment Trees
#include <bits/stdc++.h>
using namespace std;

// To store segment tree
int *st;

// Function to find gcd of 2 numbers.
int gcd(int a, int b)
{
    if (a < b)
        swap(a, b);
    if (b==0)
        return a;
    return gcd(b, a%b);
}

/* A recursive function to get gcd of given
range of array indexes. The following are parameters for
this function.

st    --> Pointer to segment tree
si --> Index of current node in the segment tree. Initially
      0 is passed as root is always at index 0
ss & se  --> Starting and ending indexes of the segment
              represented by current node, i.e., st[index]
qs & qe  --> Starting and ending indexes of query range */
int findGcd(int ss, int se, int qs, int qe, int si)
{
    if (ss>qe || se < qs)
        return 0;
    if (qs<=ss && qe>=se)
        return st[si];
}
```

```

        int mid = ss+(se-ss)/2;
        return gcd(findGcd(ss, mid, qs, qe, si*2+1),
                    findGcd(mid+1, se, qs, qe, si*2+2));
    }

//Finding The gcd of given Range
int findRangeGcd(int ss, int se, int arr[], int n)
{
    if (ss<0 || se > n-1 || ss>se)
    {
        cout << "Invalid Arguments" << "\n";
        return -1;
    }
    return findGcd(0, n-1, ss, se, 0);
}

// A recursive function that constructs Segment Tree for
// array[ss..se]. si is index of current node in segment
// tree st
int constructST(int arr[], int ss, int se, int si)
{
    if (ss==se)
    {
        st[si] = arr[ss];
        return st[si];
    }
    int mid = ss+(se-ss)/2;
    st[si] = gcd(constructST(arr, ss, mid, si*2+1),
                  constructST(arr, mid+1, se, si*2+2));
    return st[si];
}

/* Function to construct segment tree from given array.
   This function allocates memory for segment tree and
   calls constructSTUtil() to fill the allocated memory */
int *constructSegmentTree(int arr[], int n)
{
    int height = (int)(ceil(log2(n)));
    int size = 2*(int)pow(2, height)-1;
    st = new int[size];
    constructST(arr, 0, n-1, 0);
    return st;
}

// Returns size of smallest subarray of arr[0..n-1]
// with GCD equal to k.
int findSmallestSubarr(int arr[], int n, int k)
{

```

```
// To check if a multiple of k exists.
bool found = false;

// Find if k or its multiple is present
for (int i=0; i<n; i++)
{
    // If k is present, then subarray size is 1.
    if (arr[i] == k)
        return 1;

    // Break the loop to indicate presence of a
    // multiple of k.
    if (arr[i] % k == 0)
        found = true;
}

// If there was no multiple of k in arr[], then
// we can't get k as GCD.
if (found == false)
    return -1;

// If there is a multiple of k in arr[], build
// segment tree from given array
constructSegmentTree(arr, n);

// Initialize result
int res = n+1;

// Now consider every element as starting point
// and search for ending point using Binary Search
for (int i=0; i<n-1; i++)
{
    // a[i] cannot be a starting point, if it is
    // not a multiple of k.
    if (arr[i] % k != 0)
        continue;

    // Initialize indexes for binary search of closest
    // ending point to i with GCD of subarray as k.
    int low = i+1;
    int high = n-1;

    // Initialize closest ending point for i.
    int closest = 0;

    // Binary Search for closest ending point
    // with GCD equal to k.
    while (true)
```

```

{
    // Find middle point and GCD of subarray
    // arr[i..mid]
    int mid = low + (high-low)/2;
    int gcd = findRangeGcd(i, mid, arr, n);

    // If GCD is more than k, look further
    if (gcd > k)
        low = mid;

    // If GCD is k, store this point and look for
    // a closer point
    else if (gcd == k)
    {
        high = mid;
        closest = mid;
        break;
    }

    // If GCD is less than, look closer
    else
        high = mid;

    // If termination condition reached, set
    // closest
    if (abs(high-low) <= 1)
    {
        if (findRangeGcd(i, low, arr, n) == k)
            closest = low;
        else if (findRangeGcd(i, high, arr, n)==k)
            closest = high;
        break;
    }
}

if (closest != 0)
    res = min(res, closest - i + 1);
}

// If res was not changed by loop, return -1,
// else return its value.
return (res == n+1) ? -1 : res;
}

// Driver program to test above functions
int main()
{
    int n = 8;
}

```

```
int k = 3;
int arr[] = {6, 9, 7, 10, 12, 24, 36, 27};
cout << "Size of smallest sub-array with given"
     << " size is " << findSmallestSubarr(arr, n, k);
return 0;
}
```

**Output:**

2

**Example:**

arr[] = {6, 9, 7, 10, 12, 24, 36, 27}, K = 3

```
// Initial value of minLen is equal to size
// of array
minLen = 8
```

No element is equal to k so result is either greater than 1 or doesn't exist.

### First index

- GCD of subarray from 1 to 5 is 1.
- GCD < k
- GCD of subarray from 1 to 3 is 1.
- GCD < k
- GCD of subarray from 1 to 2 is 3
- minLen = minimum(8, 2) = 2

### Second Index

- GCD of subarray from 2 to 5 is 1
- GCD < k
- GCD of subarray from 2 to 4 is 1
- GCD < k
- GCD of subarray from 6 to 8 is 3
- minLen = minimum(2, 3) = 2.

.

.

.

### Sixth Index

- GCD of subarray from 6 to 7 is 12
- GCD > k
- GCD of subarray from 6 to 8 is 3
- minLen = minimum(2, 3) = 2

**Time Complexity:**  $O(n (\log n)^2)$ ,  $O(n)$  for traversing to each index,  $O(\log n)$  for each subarray and  $O(\log n)$  for GCD of each subarray.

## Source

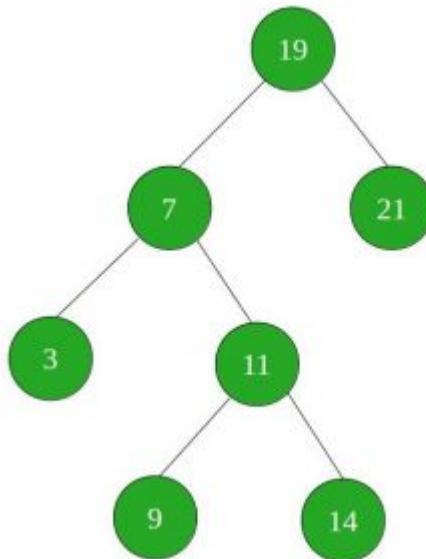
<https://www.geeksforgeeks.org/smallest-subarray-with-given-gcd/>

## Chapter 390

# Smallest number in BST which is greater than or equal to N

Smallest number in BST which is greater than or equal to N - GeeksforGeeks

Given a [Binary Search Tree](#) and a number N, the task is to find the smallest number in the binary search tree that is greater than or equal to N. Print the value of the element if it exists otherwise print -1.



Examples:

Input: N = 20

Output: 21

Explanation: 21 is the smallest element greater than 20.

Input: N = 18

Output: 19

Explanation: 19 is the smallest element greater than 18.

**Approach:**

The idea is to follow the recursive approach for solving the problem i.e. start searching for the element from the root.

- If there is a leaf node having a value less than N, then element doesn't exist and return -1.
- Otherwise, if node's value is greater than or equal to N and left child is NULL or less than N then return the node value.
- Else if node's value is less than N, then search for the element in the right subtree.
- Else search for the element in the left subtree by calling the function recursively according to the left or right value.

```
// C++ program to find the smallest value
// greater than or equal to N
#include <bits/stdc++.h>
using namespace std;

struct Node {
    int data;
    Node *left, *right;
};

// To create new BST Node
Node* createNode(int item)
{
    Node* temp = new Node;
    temp->data = item;
    temp->left = temp->right = NULL;

    return temp;
}

// To add a new node in BST
Node* add(Node* node, int key)
{
    // if tree is empty return new node
    if (node == NULL)
        return createNode(key);

    // if key is less then or grater then
    // node value then recur down the tree
    if (key < node->data)
        node->left = add(node->left, key);
```

```
else if (key > node->data)
    node->right = add(node->right, key);

// return the (unchanged) node pointer
return node;
}

// function to find min value less then N
int findMinforN(Node* root, int N)
{
    // If leaf node reached and is smaller than N
    if (root->left == NULL && root->right == NULL
        && root->data < N)
        return -1;

    // If node's value is greater than N and left value
    // is NULL or smaller then return the node value
    if ((root->data >= N && root->left == NULL)
        || (root->data >= N && root->left->data < N))
        return root->data;

    // if node value is smaller than N search in the
    // right subtree
    if (root->data <= N)
        return findMinforN(root->right, N);

    // if node value is greater than N search in the
    // left subtree
    else
        return findMinforN(root->left, N);
}

// Drivers code
int main()
{
    /*
        19
       /   \
      7    21
     /   \
    3    11
       /   \
      9    14
    */
}

Node* root = NULL;
root = add(root, 19);
root = add(root, 7);
root = add(root, 3);
```

```
root = add(root, 11);
root = add(root, 9);
root = add(root, 13);
root = add(root, 21);

int N = 18;
cout << findMinforN(root, N) << endl;

return 0;
}
```

**Output:**

19

**Source**

<https://www.geeksforgeeks.org/smallest-number-in-bst-which-is-greater-than-or-equal-to-n/>

## Chapter 391

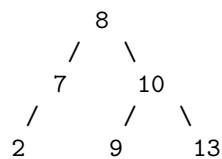
# Smallest number in BST which is greater than or equal to N ( Iterative Approach)

Smallest number in BST which is greater than or equal to N ( Iterative Approach) - Geeks-forGeeks

Given a Binary Search Tree and a number N, the task is to find the smallest number in the binary search tree that is greater than or equal to N.

**Examples:**

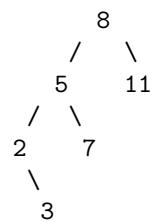
**Input:** N = 5



**Output:** 7

As 7 is the smallest number in BST which is greater than N = 5.

**Input:** N = 10



Output: 11

As 11 is the smallest number in BST which is greater than N = 10.

A recursive solution for this problem has been already been discussed in this [post](#). Below is an iterative approach for the problem:

Using [Morris Traversal](#) the above problem can be solved in constant space. Find the inorder successor of the target. Keep two pointers, one pointing to the current node and one for storing the answer.

Below is the implementation of the above approach:

```
// C++ code to find the smallest value greater
// than or equal to N
#include <bits/stdc++.h>
using namespace std;

struct Node {
    int key;
    Node *left, *right;
};

// To create new BST Node
Node* newNode(int item)
{
    Node* temp = new Node;
    temp->key = item;
    temp->left = temp->right = NULL;
    return temp;
}

// To insert a new node in BST
Node* insert(Node* node, int key)
{
    // if tree is empty return new node
    if (node == NULL)
        return newNode(key);

    // if key is less then or grater then
    // node value then recur down the tree
    if (key < node->key)
        node->left = insert(node->left, key);
    else if (key > node->key)
        node->right = insert(node->right, key);

    // return the (unchanged) node pointer
    return node;
}
```

```
// Returns smallest value greater than or
// equal to key.
int findFloor(Node* root, int key)
{
    Node *curr = root, *ans = NULL;

    // traverse in the tree
    while (curr) {

        // if the node is smaller than N,
        // move right.
        if (curr->key > key) {
            ans = curr;
            curr = curr->left;
        }

        // if it is equal to N, then it will be
        // the answer
        else if (curr->key == key) {
            ans = curr;
            break;
        }

        else // move to the right of the tree
            curr = curr->right;
    }

    if (ans != NULL)
        return ans->key;

    return -1;
}

// Driver code
int main()
{
    int N = 13;

    Node* root = insert(root, 19);
    insert(root, 2);
    insert(root, 1);
    insert(root, 3);
    insert(root, 12);
    insert(root, 9);
    insert(root, 21);
    insert(root, 25);
```

```
    printf("%d", findFloor(root, 15));  
  
    return 0;  
}
```

**Output:**

19

**Time complexity:** O(N)  
**Auxiliary Space:** O(1)

**Source**

<https://www.geeksforgeeks.org/smallest-number-in-bst-which-is-greater-than-or-equal-to-n-iterative-approach/>

## Chapter 392

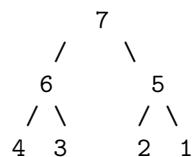
# Smallest value in each level of Binary Tree

Smallest value in each level of Binary Tree - GeeksforGeeks

Given a binary tree containing n nodes, the task is to print minimum element in each level of binary tree.

Examples:

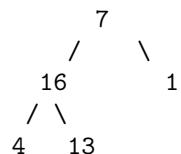
Input :



Output :

```
Every level minimum is
level 0 min is = 7
level 1 min is = 5
level 2 min is = 1
```

Input :



Output :

```
Every level minimum is
```

```
level 0 min is = 7
level 1 min is = 1
level 2 min is = 4
```

### Method 1: Using In-order traversal

**Approach:-** The idea is to recursively traverse tree in a in-order fashion. Root is considered to be at zeroth level. First find the height of tree and store it into res. res array store every smallest element in each level of binary tree.

Below is the implementation to find smallest value on each level of Binary Tree.

```
// CPP program to print smallest element
// in each level of binary tree.
#include <iostream>
#include <vector>
#define INT_MAX 10e6
using namespace std;

// A Binary Tree Node
struct Node {
    int data;
    struct Node *left, *right;
};

// return height of tree
int heightoftree(Node* root)
{
    if (root == NULL)
        return 0;

    int left = heightoftree(root->left);
    int right = heightoftree(root->right);

    return ((left > right ? left : right) + 1);
}

// Inorder Traversal
// Search minimum element in each level and
// store it into vector array.
void printPerLevelMinimum(Node* root,
                           vector<int>& res, int level)
{
    if (root != NULL) {

        printPerLevelMinimum(root->left,
                             res, level + 1);

        if (res.size() < level + 1)
            res.push_back(INT_MAX);

        res[level] = min(res[level], root->data);

        printPerLevelMinimum(root->right,
                             res, level + 1);
    }
}
```

```
if (root->data < res[level])
    res[level] = root->data;

    printPerLevelMinimum(root->right,
                          res, level + 1);
}
}

void perLevelMinimumUtility(Node* root)
{

// height of tree for the size of
// vector array
int n = heightoftree(root), i;

// vector for store all minimum of
// every level
vector<int> res(n, INT_MAX);

// save every level minimum using
// inorder traversal
printPerLevelMinimum(root, res, 0);

// print every level minimum
cout << "Every level minimum is\n";
for (i = 0; i < n; i++) {
    cout << "level " << i << " min is = "
        << res[i] << "\n";
}
}

// Utility function to create a new tree node
Node* newNode(int data)
{
    Node* temp = new Node;
    temp->data = data;
    temp->left = temp->right = NULL;

    return temp;
}

// Driver program to test above functions
int main()
{

// Let us create binary tree shown
// in above diagram
```

```
Node* root = newNode(7);
root->left = newNode(6);
root->right = newNode(5);
root->left->left = newNode(4);
root->left->right = newNode(3);
root->right->left = newNode(2);
root->right->right = newNode(1);

/*      7
   /   \
  6     5
 / \   / \
4   3 2   1      */
perLevelMinimumUtility(root);

return 0;
}
```

Output:-

```
Every level minimum is
level 0 min is = 7
level 1 min is = 5
level 2 min is = 1
```

### Method 2: Using level order Traversal

**Approach:-** The idea is to perform iterative level order traversal of the binary tree using queue. While traversing keep min variable which stores the minimum element of the current level of the tree being processed. When the level is completely traversed, print that min value.

```
// CPP program to print minimum element
// in each level of binary tree.
#include <iostream>
#include <queue>
#include <vector>
#define INT_MAX 10e6
using namespace std;

// A Binary Tree Node
struct Node {
    int data;
    struct Node *left, *right;
};

// return height of tree
```

```
int heightoftree(Node* root)
{
    if (root == NULL)
        return 0;

    int left = heightoftree(root->left);
    int right = heightoftree(root->right);

    return ((left > right ? left : right) + 1);
}

// Iterative method to find every level
// minimum element of Binary Tree
void printPerLevelMinimum(Node* root)
{
    // Base Case
    if (root == NULL)
        return ;

    // Create an empty queue for
    // level order traversal
    queue<Node*> q;

    // push the root for Change the level
    q.push(root);

    // for go level by level
    q.push(NULL);

    int min = INT_MAX;
    // for check the level
    int level = 0;

    while (q.empty() == false) {
        // Get top of queue
        Node* node = q.front();
        q.pop();

        // if node == NULL (Means this is
        // boundary between two levels)
        if (node == NULL) {

            cout << "level " << level <<
                " min is = " << min << "\n";
            level++;
        }
        else
            min = min << 1;
    }
}
```

```
// here queue is empty represent
// no element in the actual
// queue
if (q.empty())
    break;

q.push(NULL);

// increment level
level++;

// Reset min for next level
// minimum value
min = INT_MAX;

continue;
}

// get Minimum in every level
if (min > node->data)
    min = node->data;

/* Enqueue left child */
if (node->left != NULL) {
    q.push(node->left);
}

/*Enqueue right child */
if (node->right != NULL) {
    q.push(node->right);
}
}

// Utility function to create a
// new tree node
Node* newNode(int data)
{

    Node* temp = new Node;
    temp->data = data;
    temp->left = temp->right = NULL;

    return temp;
}

// Driver program to test above functions
int main()
```

```
{\n\n    // Let us create binary tree shown\n    // in above diagram\n    Node* root = newNode(7);\n    root->left = newNode(6);\n    root->right = newNode(5);\n    root->left->left = newNode(4);\n    root->left->right = newNode(3);\n    root->right->left = newNode(2);\n    root->right->right = newNode(1);\n\n    /*      7\n     /   \\\n    6     5\n   / \\  / \\\n  4   3 2   1          */\n\n    cout << "Every Level minimum is"\n        << "\n";\n\n    printPerLevelMinimum(root);\n\n    return 0;\n}
```

Output:-

```
Every level minimum is\nlevel 0 min is = 7\nlevel 1 min is = 5\nlevel 2 min is = 1
```

## Source

<https://www.geeksforgeeks.org/smallest-value-level-binary-tree/>

## Chapter 393

# Sorted Array to Balanced BST

Sorted Array to Balanced BST - GeeksforGeeks

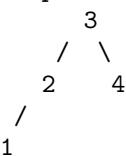
Given a sorted array. Write a function that creates a Balanced Binary Search Tree using array elements.

**Examples:**

Input: Array {1, 2, 3}  
Output: A Balanced BST



Input: Array {1, 2, 3, 4}  
Output: A Balanced BST



### Algorithm

In the [previous post](#), we discussed construction of BST from sorted Linked List. Constructing from sorted array in O(n) time is simpler as we can get the middle element in O(1) time. Following is a simple algorithm where we first find the middle node of list and make it root of the tree to be constructed.

- 1) Get the Middle of the array and make it root.
- 2) Recursively do same for left half and right half.
  - a) Get the middle of left half and make it left child of the root

- created in step 1.  
b) Get the middle of right half and make it right child of the root created in step 1.

Following is the implementation of the above algorithm. The main code which creates Balanced BST is highlighted.

C

```
#include<stdio.h>
#include<stdlib.h>

/* A Binary Tree node */
struct TNode
{
    int data;
    struct TNode* left;
    struct TNode* right;
};

struct TNode* newNode(int data);

/* A function that constructs Balanced Binary Search Tree from a sorted array */
struct TNode* sortedArrayToBST(int arr[], int start, int end)
{
    /* Base Case */
    if (start > end)
        return NULL;

    /* Get the middle element and make it root */
    int mid = (start + end)/2;
    struct TNode *root = newNode(arr[mid]);

    /* Recursively construct the left subtree and make it
       left child of root */
    root->left = sortedArrayToBST(arr, start, mid-1);

    /* Recursively construct the right subtree and make it
       right child of root */
    root->right = sortedArrayToBST(arr, mid+1, end);

    return root;
}

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct TNode* newNode(int data)
{
```

```
struct TNode* node = (struct TNode*)
    malloc(sizeof(struct TNode));
node->data = data;
node->left = NULL;
node->right = NULL;

return node;
}

/* A utility function to print preorder traversal of BST */
void preOrder(struct TNode* node)
{
    if (node == NULL)
        return;
    printf("%d ", node->data);
    preOrder(node->left);
    preOrder(node->right);
}

/* Driver program to test above functions */
int main()
{
    int arr[] = {1, 2, 3, 4, 5, 6, 7};
    int n = sizeof(arr)/sizeof(arr[0]);

    /* Convert List to BST */
    struct TNode *root = sortedArrayToBST(arr, 0, n-1);
    printf("n PreOrder Traversal of constructed BST ");
    preOrder(root);

    return 0;
}
```

### Java

```
// Java program to print BST in given range

// A binary tree node
class Node {

    int data;
    Node left, right;

    Node(int d) {
        data = d;
        left = right = null;
    }
}
```

```
class BinaryTree {  
  
    static Node root;  
  
    /* A function that constructs Balanced Binary Search Tree  
       from a sorted array */  
    Node sortedArrayToBST(int arr[], int start, int end) {  
  
        /* Base Case */  
        if (start > end) {  
            return null;  
        }  
  
        /* Get the middle element and make it root */  
        int mid = (start + end) / 2;  
        Node node = new Node(arr[mid]);  
  
        /* Recursively construct the left subtree and make it  
           left child of root */  
        node.left = sortedArrayToBST(arr, start, mid - 1);  
  
        /* Recursively construct the right subtree and make it  
           right child of root */  
        node.right = sortedArrayToBST(arr, mid + 1, end);  
  
        return node;  
    }  
  
    /* A utility function to print preorder traversal of BST */  
    void preOrder(Node node) {  
        if (node == null) {  
            return;  
        }  
        System.out.print(node.data + " ");  
        preOrder(node.left);  
        preOrder(node.right);  
    }  
  
    public static void main(String[] args) {  
        BinaryTree tree = new BinaryTree();  
        int arr[] = new int[]{1, 2, 3, 4, 5, 6, 7};  
        int n = arr.length;  
        root = tree.sortedArrayToBST(arr, 0, n - 1);  
        System.out.println("Preorder traversal of constructed BST");  
        tree.preOrder(root);  
    }  
}
```

```
// This code has been contributed by Mayank Jaiswal
```

### Python

```
# Python code to convert a sorted array
# to a balanced Binary Search Tree

# binary tree node
class Node:
    def __init__(self, d):
        self.data = d
        self.left = None
        self.right = None

# function to convert sorted array to a
# balanced BST
# input : sorted array of integers
# output: root node of balanced BST
def sortedArrayToBST(arr):

    if not arr:
        return None

    # find middle
    mid = (len(arr)) / 2

    # make the middle element the root
    root = Node(arr[mid])

    # left subtree of root has all
    # values <arr[mid]
    root.left = sortedArrayToBST(arr[:mid])

    # right subtree of root has all
    # values >arr[mid]
    root.right = sortedArrayToBST(arr[mid+1:])
    return root

# A utility function to print the preorder
# traversal of the BST
def preOrder(node):
    if not node:
        return

    print node.data,
    preOrder(node.left)
    preOrder(node.right)
```

```
# driver program to test above function
"""
Constructed balanced BST is
        4
       / \
      2 6
     / \ / \
    1 3 5 7
"""

arr = [1, 2, 3, 4, 5, 6, 7]
root = sortedArrayToBST(arr)
print "PreOrder Traversal of constructed BST ",
preOrder(root)

# This code is contributed by Ishita Tripathi
```

Time Complexity: O(n)

Following is the recurrence relation for sortedArrayToBST().

```
T(n) = 2T(n/2) + C
T(n) --> Time taken for an array of size n
C   --> Constant (Finding middle of array and linking root to left
           and right subtrees take constant time)
```

The above recurrence can be solved using [Master Theorem](#) as it falls in case 1.

**Improved By :** [IshitaTripathi](#)

## Source

<https://www.geeksforgeeks.org/sorted-array-to-balanced-bst/>

## Chapter 394

# Splay Tree | Set 3 (Delete)

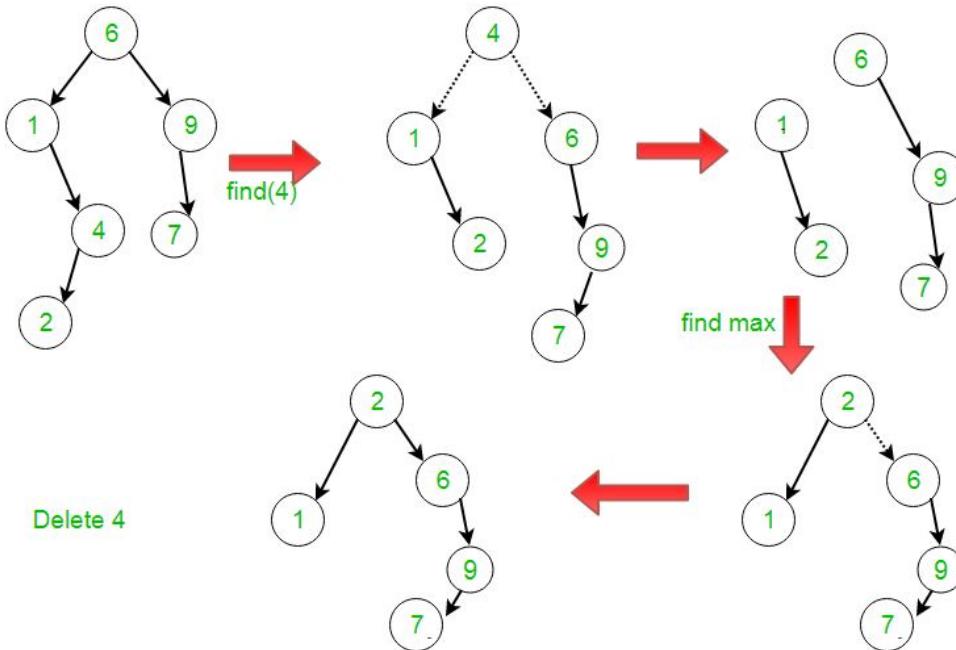
[Splay Tree | Set 3 \(Delete\) - GeeksforGeeks](#)

It is recommended to refer following post as prerequisite of this post.

[Splay Tree | Set 1 \(Search\)](#)

Following are the different cases to delete a key **k** from splay tree.

1. If **Root** is **NULL**: We simply return the root.
2. Else [Splay](#) the given key **k**. If **k** is present, then it becomes the new root. If not present, then last accessed leaf node becomes the new root.
3. If new root's key is not same as **k**, then return the root as **k** is not present.
4. Else the key **k** is present.
  - Split the tree into two trees **Tree1** = root's left subtree and **Tree2** = root's right subtree and delete the root node.
  - Let the root's of **Tree1** and **Tree2** be **Root1** and **Root2** respectively.
  - If **Root1** is **NULL**: Return **Root2**.
  - Else, Splay the maximum node (node having the maximum value) of **Tree1**.
  - After the Splay procedure, make **Root2** as the right child of **Root1** and return **Root1**.



```

// C implementation to delete a node from Splay Tree
#include<stdio.h>
#include<stdlib.h>

// An AVL tree node
struct node
{
    int key;
    struct node *left, *right;
};

/* Helper function that allocates a new node with the given key and
   NULL left and right pointers. */
struct node* newNode(int key)
{
    struct node* node = (struct node*)malloc(sizeof(struct node));
    node->key    = key;
    node->left   = node->right  = NULL;
    return (node);
}

// A utility function to right rotate subtree rooted with y
// See the diagram given above.
struct node *rightRotate(struct node **x)
{
    struct node *y = x->left;

```

```

x->left = y->right;
y->right = x;
return y;
}

// A utility function to left rotate subtree rooted with x
// See the diagram given above.
struct node *leftRotate(struct node *x)
{
    struct node *y = x->right;
    x->right = y->left;
    y->left = x;
    return y;
}

// This function brings the key at root if key is present in tree.
// If key is not present, then it brings the last accessed item at
// root. This function modifies the tree and returns the new root
struct node *splay(struct node *root, int key)
{
    // Base cases: root is NULL or key is present at root
    if (root == NULL || root->key == key)
        return root;

    // Key lies in left subtree
    if (root->key > key)
    {
        // Key is not in tree, we are done
        if (root->left == NULL) return root;

        // Zig-Zig (Left Left)
        if (root->left->key > key)
        {
            // First recursively bring the key as root of left-left
            root->left->left = splay(root->left->left, key);

            // Do first rotation for root, second rotation is
            // done after else
            root = rightRotate(root);
        }
        else if (root->left->key < key) // Zig-Zag (Left Right)
        {
            // First recursively bring the key as root of left-right
            root->left->right = splay(root->left->right, key);

            // Do first rotation for root->left
            if (root->left->right != NULL)
                root->left = leftRotate(root->left);
        }
    }
}

```

```

    }

    // Do second rotation for root
    return (root->left == NULL)? root: rightRotate(root);
}

else // Key lies in right subtree
{
    // Key is not in tree, we are done
    if (root->right == NULL) return root;

    // Zag-Zig (Right Left)
    if (root->right->key > key)
    {
        // Bring the key as root of right-left
        root->right->left = splay(root->right->left, key);

        // Do first rotation for root->right
        if (root->right->left != NULL)
            root->right = rightRotate(root->right);
    }
    else if (root->right->key < key)// Zag-Zag (Right Right)
    {
        // Bring the key as root of right-right and do
        // first rotation
        root->right->right = splay(root->right->right, key);
        root = leftRotate(root);
    }

    // Do second rotation for root
    return (root->right == NULL)? root: leftRotate(root);
}
}

// The delete function for Splay tree. Note that this function
// returns the new root of Splay Tree after removing the key
struct node* delete_key(struct node *root, int key)
{
    struct node *temp;
    if (!root)
        return NULL;

    // Splay the given key
    root = splay(root, key);

    // If key is not present, then
    // return root
    if (key != root->key)
        return root;
}

```

```
// If key is present
// If left child of root does not exist
// make root->right as root
if (!root->left)
{
    temp = root;
    root = root->right;
}

// Else if left child exists
else
{
    temp = root;

    /*Note: Since key == root->key,
    so after Splay(key, root->lchild),
    the tree we get will have no right child tree
    and maximum node in left subtree will get splayed*/
    // New root
    root = splay(root->left, key);

    // Make right child of previous root as
    // new root's right child
    root->right = temp->right;
}

// free the previous root node, that is,
// the node containing the key
free(temp);

// return root of the new Splay Tree
return root;

}

// A utility function to print preorder traversal of the tree.
// The function also prints height of every node
void preOrder(struct node *root)
{
    if (root != NULL)
    {
        printf("%d ", root->key);
        preOrder(root->left);
        preOrder(root->right);
    }
}
```

```
/* Drier program to test above function*/
int main()
{
    // Splay Tree Formation
    struct node *root = newNode(6);
    root->left = newNode(1);
    root->right = newNode(9);
    root->left->right = newNode(4);
    root->left->right->left = newNode(2);
    root->right->left = newNode(7);

    int key = 4;

    root = delete_key(root, key);
    printf("Preorder traversal of the modified Splay tree is \n");
    preOrder(root);
    return 0;
}
```

Output:

```
Preorder traversal of the modified Splay tree is
2 1 6 9 7
```

**References:**

<https://www.geeksforgeeks.org/splay-tree-set-1-insert/>  
<http://courses.cs.washington.edu/courses/cse326/01au/lectures/SplayTrees.ppt>

**Source**

<https://www.geeksforgeeks.org/splay-tree-set-3-delete/>

## Chapter 395

# Sqrt (or Square Root) Decomposition | Set 2 (LCA of Tree in O(sqrt(height)) time)

Sqrt (or Square Root) Decomposition | Set 2 (LCA of Tree in O(sqrt(height)) time) - GeeksforGeeks

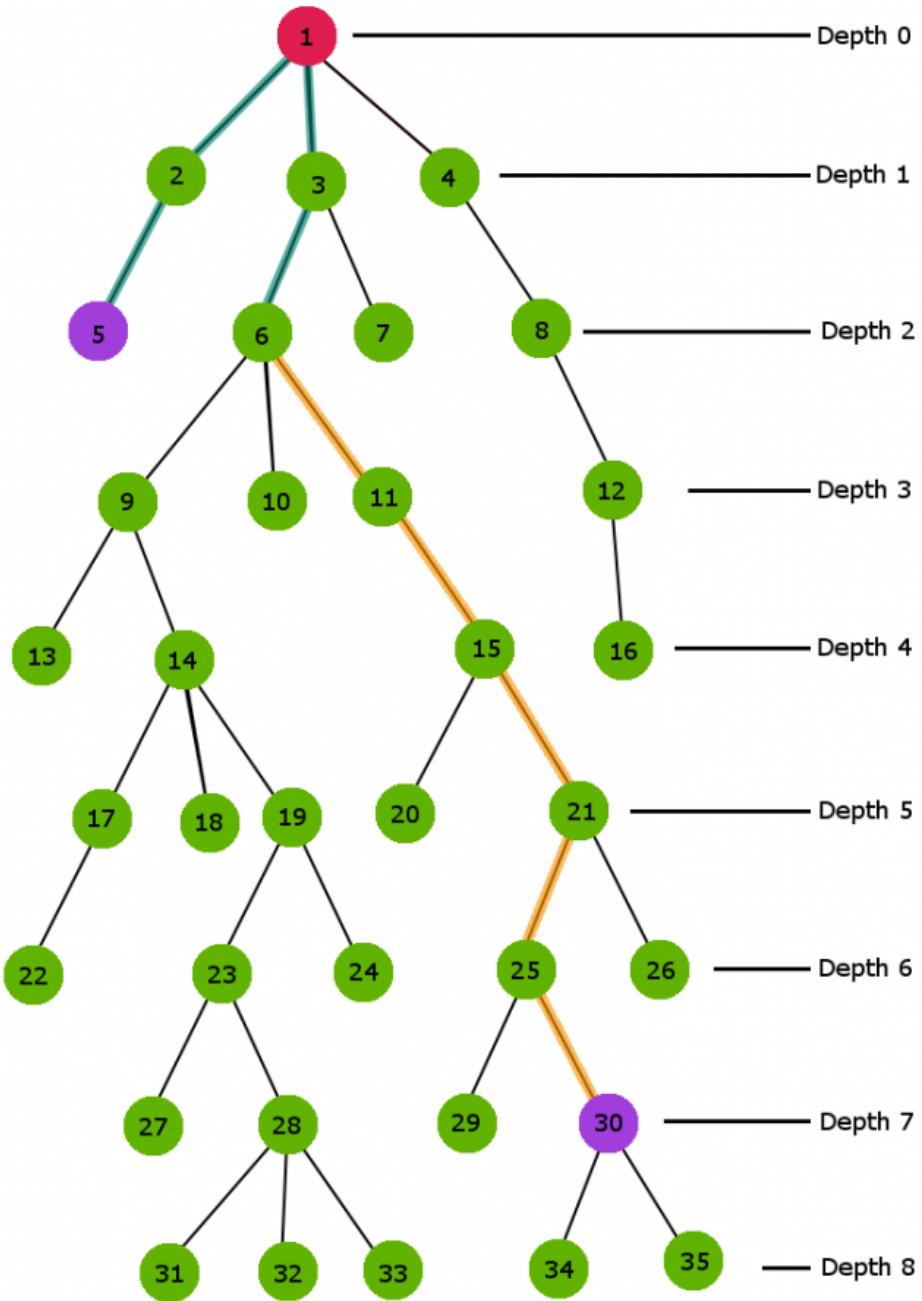
Prerequisite : [Introduction](#) and [DFS](#)

The task is to find LCA of two given nodes in a tree (not necessarily a Binary Tree). In previous posts, we have seen how to calculate [LCA using Sparse Matrix DP approach](#). In this post, we will see an optimization done on Naive method by sqrt decomposition technique that works well over the Naive Approach.

### Naive Approach

To calculate the LCA of two nodes first of all we will bring both the nodes to same height by making the node with greater depth jump one parent up the tree till both the nodes are at same height. Once, both the nodes are at same height we can then start jumping one parent up for both the nodes simultaneously till both the nodes become equal and that node will be the LCA of the two originally given nodes.

Consider the below n-ary Tree with depth 9 and lets examine how naive approach works for this sample tree.



**Here in the above Tree we need to calculate the LCA of node 6 and node 30**

Clearly node 30 has greater depth than node 6. So first of all we start jumping one parent above for node 30 till we reach the depth value of node 6 i.e at depth 2.

The **orange colored path** in the above figure demonstrates the jumping sequence to reach the depth 2. In this procedure we just simply jump one parent above the current node.

Now both nodes are at same depth 2. Therefore, now both the nodes will jump one parent up till both the nodes become equal. This end node at which both the nodes become equal for the first time is our LCA.

The **blue color path** in the above figure shows the jumping route for both the nodes

**C++ code for the above implementation:-**

```

// Naive C++ implementation to find LCA in a tree
#include <iostream>
#include <vector>
#include <math.h>
using namespace std;
#define MAXN 1001

int depth[MAXN];           // stores depth for each node
int parent[MAXN];          // stores first parent for each node

vector < int > adj[MAXN];

void addEdge(int u,int v)
{
    adj[u].push_back(v);
    adj[v].push_back(u);
}

void dfs(int cur, int prev)
{
    // marking parent for each node
    parent[cur] = prev;

    // marking depth for each node
    depth[cur] = depth[prev] + 1;

    // propagating marking down the tree
    for (int i=0; i<adj[cur].size(); i++)
        if (adj[cur][i] != prev)
            dfs(adj[cur][i],cur);
}

void preprocess()
{
    // a dummy node
}

```

```
depth[0] = -1;

// precalclating 1)depth. 2)parent.
// for each node
dfs(1,0);
}

// Time Complexity : O(Height of tree)
// recursively jumps one node above
// till both the nodes become equal
int LCANaive(int u,int v)
{
    if (u == v)  return u;
    if (depth[u] > depth[v])
        swap(u, v);
    v = parent[v];
    return LCANaive(u,v);
}

// Driver function to call the above functions
int main(int argc, char const *argv[])
{
    // adding edges to the tree
    addEdge(1,2);
    addEdge(1,3);
    addEdge(1,4);
    addEdge(2,5);
    addEdge(2,6);
    addEdge(3,7);
    addEdge(4,8);
    addEdge(4,9);
    addEdge(9,10);
    addEdge(9,11);
    addEdge(7,12);
    addEdge(7,13);

    preprocess();

    cout << "LCA(11,8) : " << LCANaive(11,8) << endl;
    cout << "LCA(3,13) : " << LCANaive(3,13) << endl;

    return 0;
}
```

Output:

```
LCA(11,8) : 4
```

LCA(3,13) : 3

**Time Complexity :** We pre-calculate the depth for each node using one **DFS traversal in O(n)**. Now in worst case, the two nodes will be two bottom most node on the tree in different child branches of the root node. Therefore, in this case the root will be the LCA of both the nodes. Hence, both the nodes will have to jump exactly  $h$  height above, where  $h$  is the height of the tree. So, to answer each **LCA query Time Complexity will be  $O(h)$** .

#### The Sqrt Decomposition Trick :

We categorize nodes of the tree into different groups according to their depth. Assuming the depth of the tree  $h$  is a perfect square. So once again like the [general sqrt decomposition approach](#) we will be having  $\sqrt{h}$  blocks or groups. Nodes from depth 0 to depth  $\sqrt{h} - 1$  lie in first group; then nodes having depth  $\sqrt{h}$  to  $2\sqrt{h} - 1$  lie in second group and so on till last node.

We keep track of the corresponding group number for every node and also depth of every node. This can be done by one single dfs on the tree (see the code for better understanding).

**Sqrt trick :-** In naive approach we were jumping one parent up the tree till both nodes aren't on the same depth. But here we perform group wise jump. To perform this group wise jump, we need two parameter associated with each node : 1) parent and 2) jump parent. Here **parent** for each node is defined as the first node above the current node that is directly connected to it, whereas **jump\_parent** for each node is the node that is the first ancestor of the current node in the group just above the current node.

So, now we need to maintain 3 parameters for each node :

- 1) **depth**
- 2) **parent**
- 3) **jump\_parent**

All these three parameters can be maintained in one dfs(refer to the code for better understanding)

#### Pseudo code for optimization process

```

LCAsqrt(u, v){

    // assuming v is at greater depth
    while (jump_parent[u] != jump_parent[v]){
        v = jump_parent[v];
    }

    // now both nodes are in same group
    // and have same jump_parent
    return LCAnaive(u,v);
}

```

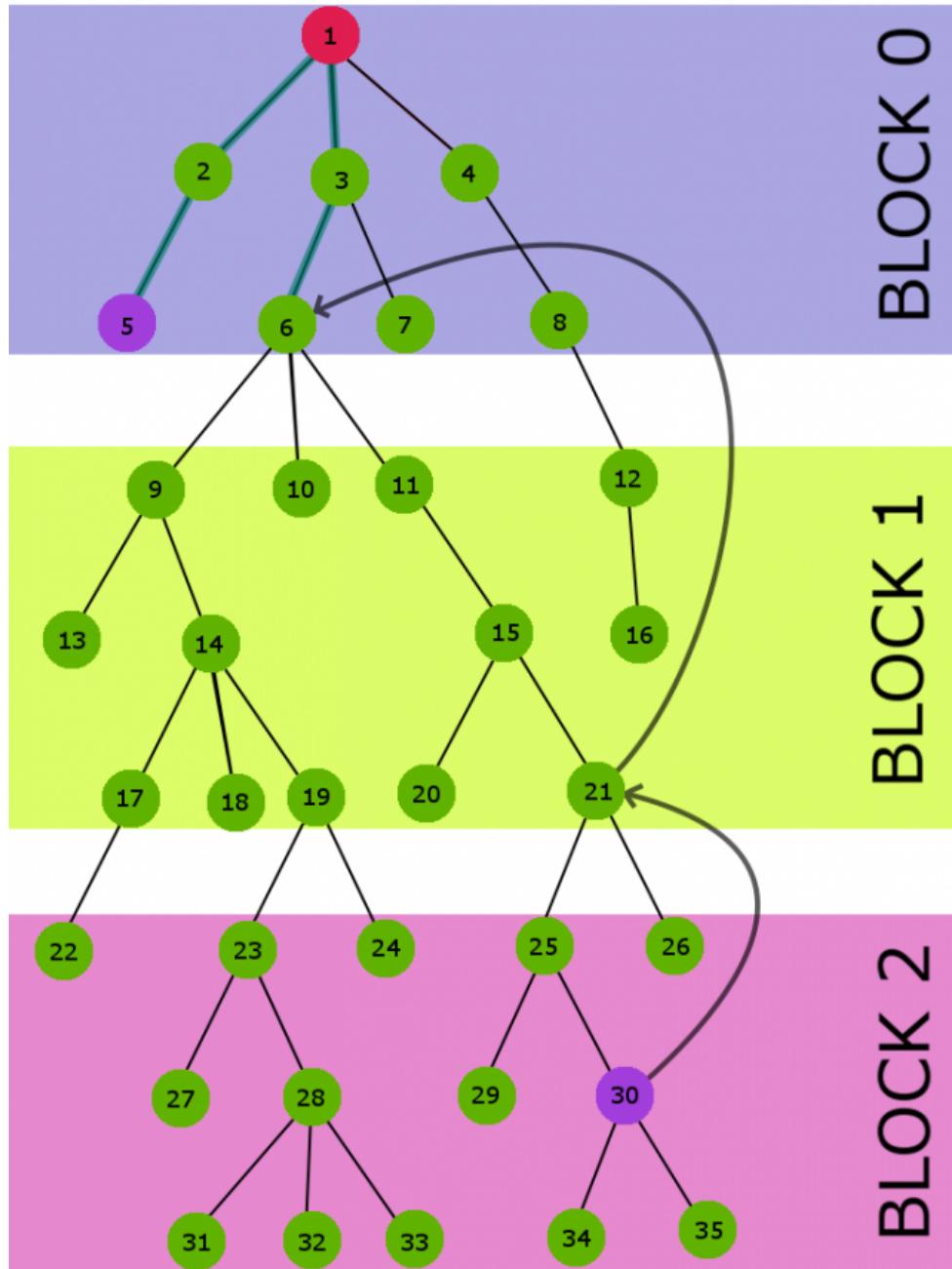
The key concept here is that first we bring both the nodes in same group and having same **jump\_parent** by climbing decomposed blocks above the tree one by one and then when both

the nodes are in same group and have same jump\_parent we use our naive approach to find LCA of the nodes.

This optimized group jumping technique reduces the iterating space by a factor of **sqrt(h)** and hence reduces the Time Complexity(refer below for better time complexity analysis)

Lets decompose the above tree in  $\text{sqrt}(h)$  groups ( $h = 9$ ) and calculate LCA for node 6 and 30.

### SQRT Decomposition of Tree



In the above decomposed tree

Jump_parent[6] = 0	parent[6] = 3
Jump_parent[5] = 0	parent[5] = 2
Jump_parent[1] = 0	parent[1] = 0
Jump_parent[11] = 6	parent[11] = 6
Jump_parent[15] = 6	parent[15] = 11
Jump_parent[21] = 6	parent[21] = 15
Jump_parent[25] = 21	parent[25] = 21
Jump_parent[26] = 21	parent[26] = 21
Jump_parent[30] = 21	parent[30] = 25

Now at this stage Jump\_parent for node 30 is 21 and Jump\_parent for node 5 is 0, So we will climb to jump\_parent[30] i.e to node 21

Now once again Jump\_parent of node 21 is not equal to Jump\_parent of node 5, So once again we will climb to jump\_parent[21] i.e node 6

At this stage jump\_parent[6] == jump\_parent[5], So now we will use our naive climbing approach and climb one parent above for both the nodes till it reach node 1 and that will be the required LCA .

**Blue path** in the above figure describes jumping path sequence for node 6 and node 5.

**The C++ code for the above description is given below:-**

```
// C++ program to find LCA using Sqrt decomposition
#include <iostream>
#include <vector>
#include <math.h>
using namespace std;
#define MAXN 1001

int block_sz;           // block size = sqrt(height)
int depth[MAXN];        // stores depth for each node
int parent[MAXN];       // stores first parent for
                        // each node
int jump_parent[MAXN]; // stores first ancestor in
                        // previous block

vector < int > adj[MAXN];

void addEdge(int u,int v)
{
    adj[u].push_back(v);
    adj[v].push_back(u);
}
```

```

int LCANaive(int u,int v)
{
    if (u == v)  return u;
    if (depth[u] > depth[v])
        swap(u,v);
    v = parent[v];
    return LCANaive(u,v);
}

// precalculating the required parameters
// associated with every node
void dfs(int cur, int prev)
{
    // marking depth of cur node
    depth[cur] = depth[prev] + 1;

    // marking parent of cur node
    parent[cur] = prev;

    // making jump_parent of cur node
    if (depth[cur] % block_sz == 0)

        /* if it is first node of the block
           then its jump_parent is its cur parent */
        jump_parent[cur] = parent[cur];

    else

        /* if it is not the first node of this block
           then its jump_parent is jump_parent of
           its parent */
        jump_parent[cur] = jump_parent[prev];

    // propogating the marking down the subtree
    for (int i = 0; i<adj[cur].size(); ++i)
        if (adj[cur][i] != prev)
            dfs(adj[cur][i], cur);
}

// using sqrt decomposition trick
int LCASQRT(int u, int v)
{
    while (jump_parent[u] != jump_parent[v])
    {
        if (depth[u] > depth[v])

```

```
// maintaining depth[v] > depth[u]
swap(u,v);

// climb to its jump parent
v = jump_parent[v];
}

// u and v have same jump_parent
return LCANaive(u,v);
}

void preprocess(int height)
{
    block_sz = sqrt(height);
    depth[0] = -1;

    // precalclating 1)depth. 2)parent. 3)jump_parent
    // for each node
    dfs(1, 0);
}

// Driver function to call the above functions
int main(int argc, char const *argv[])
{
    // adding edges to the tree
    addEdge(1,2);
    addEdge(1,3);
    addEdge(1,4);
    addEdge(2,5);
    addEdge(2,6);
    addEdge(3,7);
    addEdge(4,8);
    addEdge(4,9);
    addEdge(9,10);
    addEdge(9,11);
    addEdge(7,12);
    addEdge(7,13);

    // here we are directly taking height = 4
    // according to the given tree but we can
    // pre-calculate height = max depth
    // in one more dfs
    int height = 4;
    preprocess(height);

    cout << "LCA(11,8) : " << LCASQRT(11,8) << endl;
    cout << "LCA(3,13) : " << LCASQRT(3,13) << endl;
}
```

```
    return 0;
}
```

Output:

```
LCA(11,8) : 4
LCA(3,13) : 3
```

**Note :** The above code works even if height is not perfect square.

Now Lets see how the Time Complexity is changed by this simple grouping technique :

#### Time Complexity Analysis:

We have divided the tree into  $\sqrt{h}$  groups according to their depth and each group contain nodes having max difference in their depth equal to  $\sqrt{h}$ . Now once again take an example of worst case, let's say the first node 'u' is in first group and the node 'v' is in  $\sqrt{h}$ th group(last group). So, first we will make group jumps(single group jumps) till we reach group 1 from last group; This will take exactly  $\sqrt{h} - 1$  iterations or jumps. So, till this step the Time Complexity is  **$O(\sqrt{h})$** .

Now once we are in same group, we call the LCAnaive function. The Time complexity for LCA\_Naive is  $O(\sqrt{h})$ , where  $h'$  is the height of the tree. Now, in our case value of  $h'$  will be  $\sqrt{h}$ , because each group has a subtree of at max  $\sqrt{h}$  height. So the complexity for this step is also  $O(\sqrt{h})$ .

Hence, the total Time Complexity will be  **$O(\sqrt{h}) + \sqrt{h}) \sim O(\sqrt{h})$** .

#### Source

<https://www.geeksforgeeks.org/sqrt-square-root-decomposition-set-2-lca-tree-osqrth-time/>

## Chapter 396

# Sub-tree with minimum color difference in a 2-coloured tree

Sub-tree with minimum color difference in a 2-coloured tree - GeeksforGeeks

A tree with N nodes and N-1 edges is given with 2 different colours for its nodes. Find the sub-tree with minimum colour difference i.e.  $\text{abs}(\text{1-colour nodes} - \text{2-colour nodes})$  is minimum.

Examples:

```
Input :  
Edges : 1 2  
       1 3  
       2 4  
       3 5  
Colours : 1 1 2 2 1 [1-based indexing where  
                      index denotes the node]  
Output : 2  
Explanation : The sub-tree {1-2} and {1-2-3-5}  
have color difference of 2. Sub-tree {1-2} has two  
1-colour nodes and zero 2-colour nodes. So, color  
difference is 2. Sub-tree {1-2-3-5} has three 1-colour  
nodes and one 2-colour nodes. So color diff = 2.
```

**Method 1 :** The problem can be solved by checking every possible sub-tree from every node of the tree. This will take exponential time as we will check for sub-trees from every node.

**Method 2 : (Efficient)** If we observe, we are solving a portion of the tree several times. This produces recurring sub-problems. We can use **Dynamic Programming** approach to get the minimum color difference in one traversal. To make things simpler, we can have

color values as 1 and -1. Now, if we have a sub-tree with both colored nodes equal, our sum of colors will be 0. To get the minimum difference, we should have maximum negative sum or maximum positive sum.

- **Case 1** When we need to have a sub-tree with maximum sum : We take a node if its value  $> 0$ , i.e.  $\text{sum}(\text{parent}) += \max(0, \text{sum}(\text{child}))$
- **Case 2** When we need to have a sub-tree with minimum sum(or max negative sum) : We take a node if its value  $< 0$ , i.e.  $\text{sum}(\text{parent}) += \min(0, \text{sum}(\text{child}))$

To get the minimum sum, we can interchange the colors of nodes, i.e. -1 becomes 1 and vice-versa.

Below is the C++ implementation :

```
// CPP code to find the sub-tree with minimum color
// difference in a 2-coloured tree
#include <bits/stdc++.h>
using namespace std;

// Tree traversal to compute minimum difference
void dfs(int node, int parent, vector<int> tree[], 
         int colour[], int answer[])
{
    // Initial min difference is the color of node
    answer[node] = colour[node];

    // Traversing its children
    for (auto u : tree[node]) {

        // Not traversing the parent
        if (u == parent)
            continue;

        dfs(u, node, tree, colour, answer);

        // If the child is adding positively to
        // difference, we include it in the answer
        // Otherwise, we leave the sub-tree and
        // include 0 (nothing) in the answer
        answer[node] += max(answer[u], 0);
    }
}

int maxDiff(vector<int> tree[], int colour[], int N)
{
    int answer[N + 1];
    memset(answer, 0, sizeof(answer));
```

```
// DFS for colour difference : 1colour - 2colour
dfs(1, 0, tree, colour, answer);

// Minimum colour difference is maximum answer value
int high = 0;
for (int i = 1; i <= N; i++) {
    high = max(high, answer[i]);

    // Clearing the current value
    // to check for colour2 as well
    answer[i] = 0;
}

// Interchanging the colours
for (int i = 1; i <= N; i++) {
    if (colour[i] == -1)
        colour[i] = 1;
    else
        colour[i] = -1;
}

// DFS for colour difference : 2colour - 1colour
dfs(1, 0, tree, colour, answer);

// Checking if colour2 makes the minimum colour
// difference
for (int i = 1; i < N; i++)
    high = max(high, answer[i]);

return high;
}

// Driver code
int main()
{
    // Nodes
    int N = 5;

    // Adjacency list representation
    vector<int> tree[N + 1];

    // Edges
    tree[1].push_back(2);
    tree[2].push_back(1);

    tree[1].push_back(3);
    tree[3].push_back(1);
```

```
tree[2].push_back(4);
tree[4].push_back(2);

tree[3].push_back(5);
tree[5].push_back(3);

// Index represent the colour of that node
// There is no Node 0, so we start from
// index 1 to N
int colour[] = { 0, 1, 1, -1, -1, 1 };

// Printing the result
cout << maxDiff(tree, colour, N);

return 0;
}
```

Output:

2

## Source

<https://www.geeksforgeeks.org/sub-tree-minimum-color-difference-2-coloured-tree/>

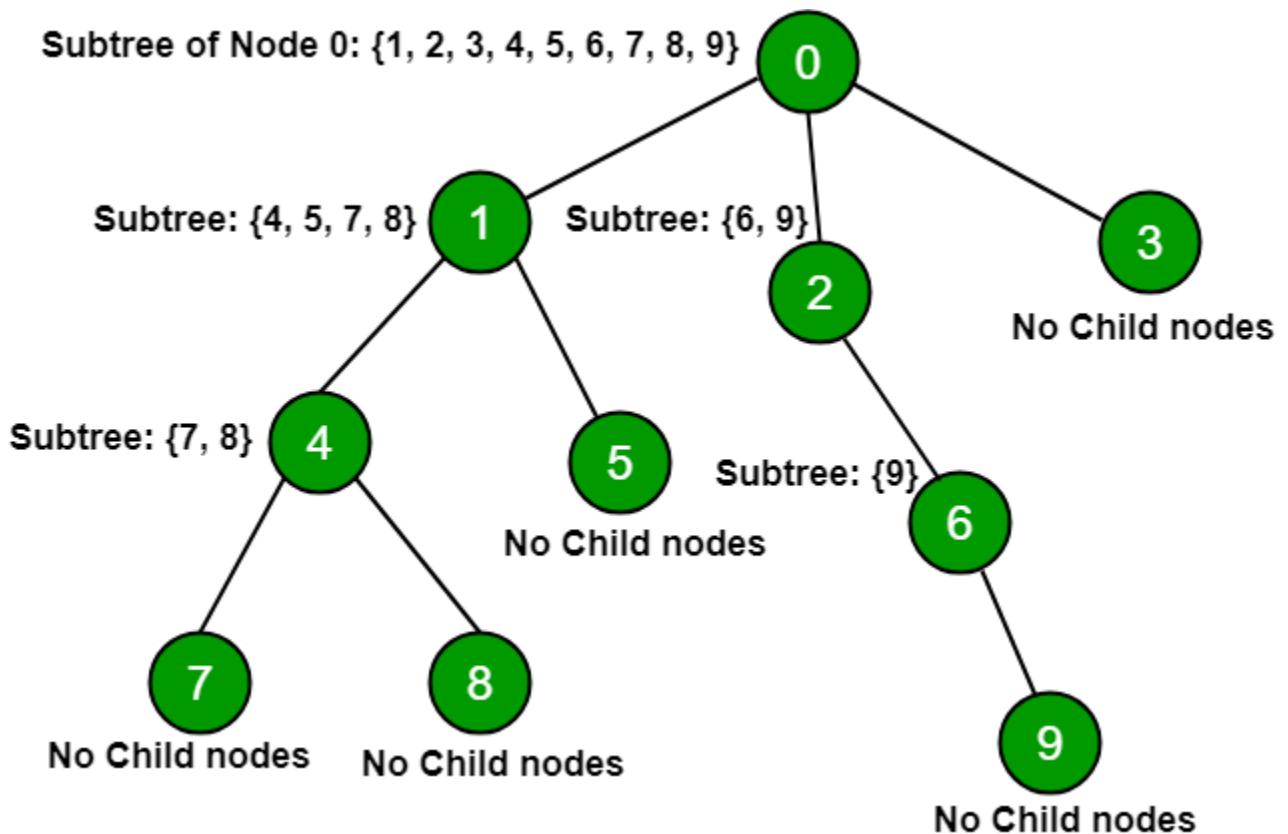
## Chapter 397

# Subtree of all nodes in a tree using DFS

Subtree of all nodes in a tree using DFS - GeeksforGeeks

Given n nodes of a tree and their connections, print Subtree nodes of every node.

**Subtree** of a node is defined as a tree which is a child of a node. The name emphasizes that everything which is a descendant of a tree node is a tree too, and is a subset of the larger tree.



Examples :

Input: N = 5

```

0 1
1 2
0 3
3 4

```

Output:

```

Subtree of node 0 is 1 2 3 4
Subtree of node 1 is 2
Subtree of node 3 is 4

```

Input: N = 7

```

0 1
1 2
2 3
0 4
4 5
4 6

```

Output:

```
Subtree of node 0 is 1 2 3 4 5 6
Subtree of node 1 is 2 3
Subtree of node 4 is 5 6
```

**Approach:** Do DFS traversal for every node and print all the nodes which are reachable from a particular node.

**Explanation of below code:**

1. When function dfs(0, 0) is called, start[0] = 0, dfs\_order.push\_back(0), visited[0] = 1 to keep track of dfs order.
2. Now, consider adjacency list (adj[100001]) as considering directional path elements connected to node 0 will be in adjacency list corresponding to node 0.
3. Now, recursively call dfs function till all elements traversed of adj[0].
4. Now, dfs(1, 2) is called, Now start[1] = 1, dfs\_order.push\_back(1), visited[1] = 1 after adj[1] elements is traversed.
5. Now adj [1] is traversed which contain only node 2 when adj[2] is traversed it contains no element, it will break and end[1]=2.
6. Similarly, all nodes traversed and store dfs\_order in array to find subtree of nodes.

```
// C++ code to print subtree of all nodes
#include<bits/stdc++.h>
using namespace std;

// arrays for keeping position
// at each dfs traversal for each node
int start[100001];
int endd[100001];

// Storing dfs order
vector<int>dfs_order;
vector<int>adj[100001];
int visited[100001];

// Recursive function for dfs
// traversal dfsUtil()
void dfs(int a,int &b)
{
    // keep track of node visited
    visited[a]=1;
    b++;
    start[a]=b;
    dfs_order.push_back(a);

    for(vector<int>:: iterator it=adj[a].begin();
        it!=adj[a].end();it++)
    {

```

```
        if(!visited[*it])
        {
            dfs(*it,b);
        }
    endd[a]=b;
}

// Function to print the subtree nodes
void Print(int n)
{
    for(int i=0;i<n;i++)
    {
        // if node is leaf node
        // start[i] is equals to endd[i]
        if(start[i]!=endd[i])
        {
            cout<<"subtree of node "<<i<<" is ";
            for(int j=start[i]+1;j<=endd[i];j++)
            {
                cout<<dfs_order[j-1]<<" ";
            }
            cout<<endl;
        }
    }
}

// Driver code
int main()
{
    // No of nodes n = 10
    int n =10, c = 0;

    adj[0].push_back(1);
    adj[0].push_back(2);
    adj[0].push_back(3);
    adj[1].push_back(4);
    adj[1].push_back(5);
    adj[4].push_back(7);
    adj[4].push_back(8);
    adj[2].push_back(6);
    adj[6].push_back(9);

    //Calling dfs for node 0
    //Considering root node at 0
    dfs(0, c);

    // Print child nodes
```

```
Print(n);  
return 0;  
}
```

### Source

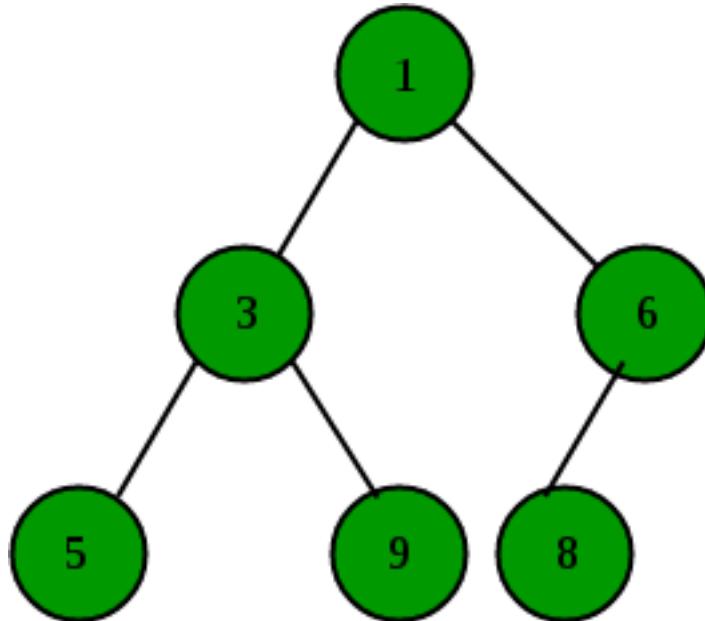
<https://www.geeksforgeeks.org/sub-tree-nodes-tree-using-dfs/>

## Chapter 398

# Subtree with given sum in a Binary Tree

Subtree with given sum in a Binary Tree - GeeksforGeeks

You are given a binary tree and a given sum. The task is to check if there exist a subtree whose sum of all nodes is equal to the given sum.



Examples :

```
// For above tree  
Input : sum = 17
```

```
Output: "Yes"
// sum of all nodes of subtree {3, 5, 9} = 17

Input : sum = 11
Output: "No"
// no subtree with given sum exist
```

The idea is to traverse tree in Postorder fashion because here we have to think bottom-up. First calculate the sum of left subtree then right subtree and check if **sum\_left + sum\_right + cur\_node = sum** is satisfying the condition that means any subtree with given sum exist. Below is the recursive implementation of algorithm.

C++

```
// C++ program to find if there is a subtree with
// given sum
#include<bits/stdc++.h>
using namespace std;

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct Node
{
    int data;
    struct Node* left, *right;
};

/* utility that allocates a new node with the
given data and NULL left and right pointers. */
struct Node* newnode(int data)
{
    struct Node* node = new Node;
    node->data = data;
    node->left = node->right = NULL;
    return (node);
}

// function to check if there exist any subtree with given sum
// cur_sum --> sum of current subtree from ptr as root
// sum_left --> sum of left subtree from ptr as root
// sum_right --> sum of right subtree from ptr as root
bool sumSubtreeUtil(struct Node *ptr, int *cur_sum, int sum)
{
    // base condition
    if (ptr == NULL)
    {
        *cur_sum = 0;
```

```
        return false;
    }

    // Here first we go to left sub-tree, then right subtree
    // then first we calculate sum of all nodes of subtree
    // having ptr as root and assign it as cur_sum
    // cur_sum = sum_left + sum_right + ptr->data
    // after that we check if cur_sum == sum
    int sum_left = 0, sum_right = 0;
    return ( sumSubtreeUtil(ptr->left, &sum_left, sum) ||
            sumSubtreeUtil(ptr->right, &sum_right, sum) ||
            (*cur_sum = sum_left + sum_right + ptr->data) == sum));
}

// Wrapper over sumSubtreeUtil()
bool sumSubtree(struct Node *root, int sum)
{
    // Initialize sum of subtree with root
    int cur_sum = 0;

    return sumSubtreeUtil(root, &cur_sum, sum);
}

// driver program to run the case
int main()
{
    struct Node *root = newnode(8);
    root->left      = newnode(5);
    root->right     = newnode(4);
    root->left->left = newnode(9);
    root->left->right = newnode(7);
    root->left->right->left = newnode(1);
    root->left->right->right = newnode(12);
    root->left->right->right->right = newnode(2);
    root->right->right = newnode(11);
    root->right->right->left = newnode(3);
    int sum = 22;

    if (sumSubtree(root, sum))
        cout << "Yes";
    else
        cout << "No";
    return 0;
}
```

### Java

```
// Java program to find if there
```

```
// is a subtree with given sum
import java.util.*;
class GFG
{
    /* A binary tree node has data,
    pointer to left child and a
    pointer to right child */
    static class Node
    {
        int data;
        Node left, right;
    }

    static class INT
    {
        int v;
        INT(int a)
        {
            v = a;
        }
    }

    /* utility that allocates a new
    node with the given data and
    null left and right pointers. */
    static Node newnode(int data)
    {
        Node node = new Node();
        node.data = data;
        node.left = node.right = null;
        return (node);
    }

    // function to check if there exist
    // any subtree with given sum
    // cur_sum -. sum of current subtree
    //           from ptr as root
    // sum_left -. sum of left subtree
    //           from ptr as root
    // sum_right -. sum of right subtree
    //           from ptr as root
    static boolean sumSubtreeUtil(Node ptr,
                                  INT cur_sum,
                                  int sum)
    {
        // base condition
        if (ptr == null)
```

```

{
    cur_sum = new INT(0);
    return false;
}

// Here first we go to left
// sub-tree, then right subtree
// then first we calculate sum
// of all nodes of subtree having
// ptr as root and assign it as
// cur_sum. (cur_sum = sum_left +
// sum_right + ptr.data) after that
// we check if cur_sum == sum
INT sum_left = new INT(0),
    sum_right = new INT(0);
return (sumSubtreeUtil(ptr.left, sum_left, sum) ||
        sumSubtreeUtil(ptr.right, sum_right, sum) ||
        ((cur_sum.v = sum_left.v +
           sum_right.v + ptr.data) == sum));
}

// Wrapper over sumSubtreeUtil()
static boolean sumSubtree(Node root, int sum)
{
    // Initialize sum of
    // subtree with root
    INT cur_sum = new INT( 0);

    return sumSubtreeUtil(root, cur_sum, sum);
}

// Driver Code
public static void main(String args[])
{
    Node root = newnode(8);
    root.left = newnode(5);
    root.right = newnode(4);
    root.left.left = newnode(9);
    root.left.right = newnode(7);
    root.left.right.left = newnode(1);
    root.left.right.right = newnode(12);
    root.left.right.right.right = newnode(2);
    root.right.right = newnode(11);
    root.right.right.left = newnode(3);
    int sum = 22;

    if (sumSubtree(root, sum))
        System.out.println( "Yes");
}

```

```
        else
            System.out.println( "No");
    }
}

// This code is contributed
// by Arnab Kundu
```

**Output:**

Yes

Improved By : [andrew1234](#)

**Source**

<https://www.geeksforgeeks.org/subtree-given-sum-binary-tree/>

## Chapter 399

# Subtrees formed after bursting nodes

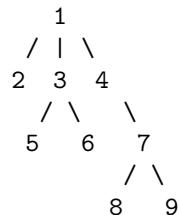
Subtrees formed after bursting nodes - GeeksforGeeks

You are given an **n-ary tree** with a special **property**: If we burst a random node of the tree, this node along with its immediate parents up to the root vanishes. The tree has N nodes and nodes are numbered from 1 to N. The root is always at 1. Given a sequence of **queries** denoting the number of the node we start bursting, the problem is to find the number of subtrees that would be formed in the end according to the above property, for each query independently.

Examples:

Input:

Consider the following tree:



q = 2

n = 1

n = 7

Output:

3

4

Explanation:

In the first query after bursting node 1, there

```
will be 3 subtrees formed rooted at 2, 3 and 4.  
In the second query after bursting node 7, nodes  
4 and 1 also get burst, thus there will  
be 4 subtrees formed rooted at 8, 9, 2 and 3.
```

Since we are dealing with n-ary tree we can use a representation similar to that of a **graph**, and add the bidirectional edges in an array of lists. Now if we burst a node, we can say for sure that all its children will become separate subtrees. Moreover all the children of its parents and others ancestors till the root that burst, will also become separate subtrees. So in our final answer we want to **exclude** the current node and all its ancestors in the path till the root. Thus we can form the **equation** to solve as:

```
answer[node] = degree[node] + allChild[parent[node]] - countPath[node]
```

where **allChild[]**: number of node's children + number of its parent's children + ..+ number of root's children

**parent[]**: parent of a node in the tree

**degree[]**: number of children for a node

**countPath[]**: number of nodes from root to parent of node

We can fill all the above arrays using **depth first search** over the adjacency list. We can start from the root 1, assuming its parent is 0 and recur depth first to propagate its values to its children. Thus we can pre-process and fill the above arrays initially and return the equation's value for each query accordingly.

Following is the C++ implementation of the above approach:

```
// CPP program to find number of subtrees after bursting nodes
#include <bits/stdc++.h>
using namespace std;

// do depth first search of node nod; par is its parent
void dfs(int nod, int par, list<int> adj[], int allChild[],
          int parent[], int degree[], int countPath[])
{
    // go through the adjacent nodes
    for (auto it = adj[nod].begin(); it != adj[nod].end(); it++) {
        int curr = *it;

        // avoid cycling
        if (curr == par)
            continue;

        degree[nod]++;
        countPath[curr] = countPath[nod] + 1;
        parent[curr] = nod;
    }

    // propagated from parent
    allChild[nod] = allChild[parent[nod]] + degree[nod];
}
```

```
// go through the adjacent nodes
for (auto it = adj[nod].begin(); it != adj[nod].end(); it++) {
    int curr = *it;

    // avoid cycling
    if (curr == par)
        continue;

    // recur and go depth first
    dfs(curr, nod, adj, allChild, parent, degree, countPath);
}
}

// Driver code
int main()
{
    int n = 9;

    // adjacency list for each node
    list<int> adj[n + 1];

    // allChild[]: number of node's children + number of its
    // parent's children + ..+ number of root's children
    // parent[]: parent of a node in the tree
    // degree[]: number of children for a node
    // countPath[]: number of nodes from root to parent of node
    int allChild[n + 1] = { 0 }, parent[n + 1] = { 0 },
        degree[n + 1] = { 0 }, countPath[n + 1] = { 0 };

    // construct tree
    adj[1].push_back(2);
    adj[2].push_back(1);
    adj[1].push_back(3);
    adj[3].push_back(1);
    adj[1].push_back(4);
    adj[4].push_back(1);
    adj[3].push_back(5);
    adj[5].push_back(3);
    adj[3].push_back(6);
    adj[6].push_back(3);
    adj[4].push_back(7);
    adj[7].push_back(4);
    adj[7].push_back(8);
    adj[8].push_back(7);
    adj[7].push_back(9);
    adj[9].push_back(7);
```

```
// assume 1 is root and 0 is its parent
dfs(1, 0, adj, allChild, parent, degree, countPath);

// 2 queries
int curr = 1;
cout << degree[curr] + allChild[parent[curr]] - countPath[curr] << endl;

curr = 7;
cout << degree[curr] + allChild[parent[curr]] - countPath[curr] << endl;

return 0;
}
```

Output:

```
3
4
```

The **time complexity** of the above algorithm is  $O(E * \lg(V))$  where E is the number of edges and V is the number of vertices.

## Source

<https://www.geeksforgeeks.org/subtrees-formed-bursting-nodes/>

## Chapter 400

# Succinct Encoding of Binary Tree

Succinct Encoding of Binary Tree - GeeksforGeeks

A succinct encoding of Binary Tree takes close to minimum possible space. The number of structurally different binary trees on  $n$  nodes is  $n^{\text{th}}$  Catalan number. For large  $n$ , this is about  $4^n$ ; thus we need at least about  $\log_2 4^n = 2n$  bits to encode it. A succinct binary tree therefore would occupy  $2n+o(n)$  bits.

One simple representation which meets this bound is to visit the nodes of the tree in preorder, outputting “1” for an internal node and “0” for a leaf. If the tree contains data, we can simply simultaneously store it in a consecutive array in preorder.

Below is algorithm for encoding:

```
function EncodeSuccinct(node n, bitstring structure, array data) {
    if n = nil then
        append 0 to structure;
    else
        append 1 to structure;
        append n.data to data;
        EncodeSuccinct(n.left, structure, data);
        EncodeSuccinct(n.right, structure, data);
}
```

And below is algorithm for decoding

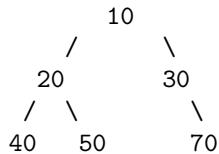
```
function DecodeSuccinct(bitstring structure, array data) {
    remove first bit of structure and put it in b
    if b = 1 then
```

```
create a new node n
remove first element of data and put it in n.data
n.left = DecodeSuccinct(structure, data)
n.right = DecodeSuccinct(structure, data)
return n
else
    return nil
}
```

Source: [https://en.wikipedia.org/wiki/Binary\\_tree#Succinct\\_encodings](https://en.wikipedia.org/wiki/Binary_tree#Succinct_encodings)

Example:

Input:



Data Array (Contains preorder traversal)

10 20 40 50 30 70

Structure Array

1 1 1 0 0 1 0 0 1 0 1 0 0

1 indicates data and 0 indicates NULL

Below is C++ implementation of above algorithms.

C++

```
// C++ program to demonstrate Succinct Tree Encoding and decoding
#include<bits/stdc++.h>
using namespace std;

// A Binary Tree Node
struct Node
{
    int key;
    struct Node* left, *right;
};

// Utility function to create new Node
Node *newNode(int key)
{
    Node *temp = new Node;
    temp->key = key;
```

```

temp->left = temp->right = NULL;
return (temp);
}

// This function fills lists 'struc' and 'data'. 'struc' list
// stores structure information. 'data' list stores tree data
void EncodeSuccinct(Node *root, list<bool> &struc, list<int> &data)
{
    // If root is NULL, put 0 in structure array and return
    if (root == NULL)
    {
        struc.push_back(0);
        return;
    }

    // Else place 1 in structure array, key in 'data' array
    // and recur for left and right children
    struc.push_back(1);
    data.push_back(root->key);
    EncodeSuccinct(root->left, struc, data);
    EncodeSuccinct(root->right, struc, data);
}

// Constructs tree from 'struc' and 'data'
Node *DecodeSuccinct(list<bool> &struc, list<int> &data)
{
    if (struc.size() <= 0)
        return NULL;

    // Remove one item from from structure list
    bool b = struc.front();
    struc.pop_front();

    // If removed bit is 1,
    if (b == 1)
    {
        // remove an item from data list
        int key = data.front();
        data.pop_front();

        // Create a tree node with the removed data
        Node *root = newNode(key);

        // And recur to create left and right subtrees
        root->left = DecodeSuccinct(struc, data);
        root->right = DecodeSuccinct(struc, data);
        return root;
    }
}

```

```

        return NULL;
    }

// A utility function to print tree
void preorder(Node* root)
{
    if (root)
    {
        cout << "key: " << root->key;
        if (root->left)
            cout << " | left child: " << root->left->key;
        if (root->right)
            cout << " | right child: " << root->right->key;
        cout << endl;
        preorder(root->left);
        preorder(root->right);
    }
}

// Driver program
int main()
{
    // Let us construct the Tree shown in the above figure
    Node *root          = newNode(10);
    root->left         = newNode(20);
    root->right        = newNode(30);
    root->left->left  = newNode(40);
    root->left->right = newNode(50);
    root->right->right = newNode(70);

    cout << "Given Tree\n";
    preorder(root);
    list<bool> struc;
    list<int> data;
    EncodeSuccinct(root, struc, data);

    cout << "\nEncoded Tree\n";
    cout << "Structure List\n";
    list<bool>::iterator si; // Structure iterator
    for (si = struc.begin(); si != struc.end(); ++si)
        cout << *si << " ";

    cout << "\nData List\n";
    list<int>::iterator di; // Data iterator
    for (di = data.begin(); di != data.end(); ++di)
        cout << *di << " ";
}

```

```
Node *newroot = DecodeSuccinct(struc, data);

cout << "\n\nPreorder traversal of decoded tree\n";
preorder(newroot);

return 0;
}
```

### Python

```
# Python program to demonstrate Succinct Tree Encoding and Decoding

# Node structure
class Node:
    # Utility function to create new Node
    def __init__(self , key):
        self.key = key
        self.left = None
        self.right = None

def EncodeSuccint(root , struc , data):

    # If root is None , put 0 in structure array and return
    if root is None :
        struc.append(0)
        return

    # Else place 1 in structure array, key in 'data' array
    # and recur for left and right children
    struc.append(1)
    data.append(root.key)
    EncodeSuccint(root.left , struc , data)
    EncodeSuccint(root.right , struc , data)

# Constructs tree from 'struc' and 'data'
def DecodeSuccinct(struc , data):
    if(len(struc) <= 0):
        return None

    # Remove one item from structure list
    b = struc[0]
    struc.pop(0)

    # If removed bit is 1:
    if b == 1:
        key = data[0]
        data.pop(0)
```

```
#Create a tree node with removed data
root = Node(key)

#And recur to create left and right subtrees
root.left = DecodeSuccinct(struc , data);
root.right = DecodeSuccinct(struc , data);
return root

return None

def preorder(root):
    if root is not None:
        print "key: %d" %(root.key),

        if root.left is not None:
            print "| left child: %d" %(root.left.key),
        if root.right is not None:
            print "| right child %d" %(root.right.key),
        print ""
        preorder(root.left)
        preorder(root.right)

# Driver Program
root = Node(10)
root.left = Node(20)
root.right = Node(30)
root.left.left = Node(40)
root.left.right = Node(50)
root.right.right = Node(70)

print "Given Tree"
preorder(root)
struc = []
data = []
EncodeSuccinct(root , struc , data)

print "\nEncoded Tree"
print "Structure List"

for i in struc:
    print i ,

print "\nDataList"
for value in data:
    print value,
```

```
newroot = DecodeSuccinct(struc , data)
print "\n\nPreorder Traversal of decoded tree"
preorder(newroot)

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

Output:

```
Given Tree
key: 10 | left child: 20 | right child: 30
key: 20 | left child: 40 | right child: 50
key: 40
key: 50
key: 30 | right child: 70
key: 70
```

```
Encoded Tree
Structure List
1 1 1 0 0 1 0 0 1 0 1 0 0
Data List
10 20 40 50 30 70
```

```
Preorder traversal of decoded tree
key: 10 | left child: 20 | right child: 30
key: 20 | left child: 40 | right child: 50
key: 40
key: 50
key: 30 | right child: 70
key: 70
```

This article is contribute by **Shivam**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## Source

<https://www.geeksforgeeks.org/succinct-encoding-of-binary-tree/>

## Chapter 401

# Sudo Placement[1.4] | BST Traversal

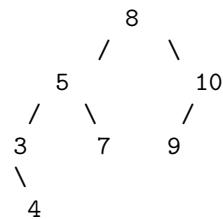
Sudo Placement[1.4] | BST Traversal - GeeksforGeeks

Given N elements to be inserted into Binary Search Tree. The task is to construct a binary search tree with only insert operation and finally print the elements in the postorder traversal. The BST is constructed according to the arrival order of elements.

**Examples:**

Input: N elements = {8, 5, 10, 3, 4, 9, 7}  
Output: 4 3 7 5 9 10 8

For the above input, the BST is:



The post-order traversal of the above BST is 4 3 7 5 9 10 8

**Approach:** The approach to solve this problem is to construct the BST using [insertion](#) method in BST. Once all the nodes are inserted, print the [postorder traversal](#) of the tree.

Below is the implementation of the above approach:

```
// C++ program to insert nodes
// and print the postorder traversal
#include <bits/stdc++.h>
```

```
using namespace std;

// structure to store the BST
struct Node {
    int data;
    Node* left = NULL;
    Node* right = NULL;
};

// locates the memory space
Node* newNode(int key)
{
    Node* temp = new Node;
    temp->data = key;
    temp->left = NULL;
    temp->right = NULL;
    return temp;
}

// inserts node in the BST
Node* insertNode(Node* head, int key)
{
    // if first node
    if (head == NULL)
        head = newNode(key);
    else {
        // move to left
        if (key < head->data)
            head->left = insertNode(head->left, key);
        // move to right
        else
            head->right = insertNode(head->right, key);
    }
    return head;
}

// print the postorder traversal
void posOrder(Node* head)
{
    // leaf node is null
    if (head == NULL)
        return;

    // left
    posOrder(head->left);

    // right
    posOrder(head->right);
```

```
// data
cout << head->data << " ";
}

// Driver Code
int main()
{
    Node* root = NULL;
    root = insertNode(root, 8);
    root = insertNode(root, 5);
    root = insertNode(root, 10);
    root = insertNode(root, 3);
    root = insertNode(root, 4);
    root = insertNode(root, 9);
    root = insertNode(root, 7);

    // prints the postorder traversal of
    // the tree
    posOrder(root);
    cout << endl;
}
```

## Source

<https://www.geeksforgeeks.org/sudo-placement1-4-bst-traversal/>

## Chapter 402

# Sudo Placement[1.4] | Jumping the Subtree

Sudo Placement[1.4] | Jumping the Subtree - GeeksforGeeks

Given a binary search tree of n nodes with distinct values. Also given are Q queries. Each query consists of a node value that has to be searched in the BST and skip the subtree that has given node as its root. If the provided node is the root itself then print “Empty” without quotes. After that print the preorder traversal of the BST.

**Examples:**

**Input:**

N = 7, Q = 2

BST elements: 8 4 10 15 14 88 64

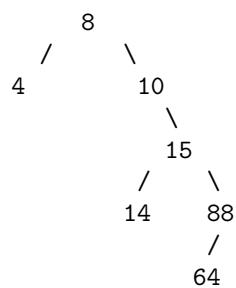
Query1: 15

Query2: 88

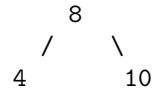
**Output:** 8 4 10

8 4 10 15 14

The tree below will be formed from the elements given

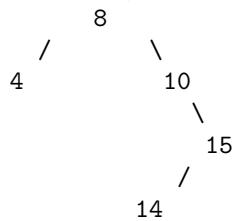


Query1 = 15. So, skip the subtree with 15 as root.  
The remaining tree is :



The preorder traversal of the above tree is: 8 4 10

Query2 = 88. So we skip the subtree with 88 as root.  
The remaining tree is :



The preorder traversal of the above tree is: 8 4 10 15 14

A naive approach is to traverse the entire tree and store its pre-order traversal. In every query, perform a pre-order traversal treating node as root. Print the entire tree's pre-order traversal except the elements that are in the pre-order traversal of the tree which treats node as the root.

An efficient approach is to store the entire pre-order traversal of the tree in a container. While finding the pre-order traversal of the tree, store the number of recursive calls from the node and store it in a hash-table(*mp*). This effectively stores the entire size of the subtree treating any node as the root. While performing every query, print the pre-order traversal of the tree, till the node is found, once it is found, perform a jump of *mp[node]* steps so that the subtree is skipped.

Below is the implementation of the above approach:

C++

```
// C++ program to insert nodes
// and print the preorder traversal
#include <bits/stdc++.h>
using namespace std;

// vector to store pre-order
vector<int> pre;

// map to store the height
// of every subtree
unordered_map<int, int> mp;
```

```
// structure to store the BST
struct Node {
    int data;
    Node* left = NULL;
    Node* right = NULL;
};

// locates the memory space
Node* newNode(int key)
{
    Node* temp = new Node;
    temp->data = key;
    temp->left = NULL;
    temp->right = NULL;
    return temp;
}

// inserts node in the BST
Node* insertNode(Node* head, int key)
{
    // if first node
    if (head == NULL)
        head = newNode(key);
    else {

        // move to left
        if (key < head->data)
            head->left = insertNode(head->left, key);
        // move to right
        else
            head->right = insertNode(head->right, key);
    }
    return head;
}

// Function to compute the pre-order
// and compute the height of every sub-tree
int preOrder(Node* head)
{
    // leaf node is null
    if (head == NULL)
        return 0;

    pre.push_back(head->data);

    mp[head->data] += preOrder(head->left);
    mp[head->data] += preOrder(head->right);
}
```

```

        mp[head->data] += 1;

        return mp[head->data];
    }

// Function to perform every queries
void performQueries(int node)
{
    // traverse in the pre-order
    // jump the subtree which has node
    for (int i = 0; i < pre.size(); ) {

        // jump the subtree which has the node
        if (pre[i] == node) {
            i += mp[pre[i]];
        }

        // print the pre-order
        else {
            cout << pre[i] << " ";
            i++;
        }
    }
    cout << endl;
}

// Driver Code
int main()
{
    Node* root = NULL;

/*
          8
         /   \
        4     10
           \
           15
          /   \
         14   88
            /
           64  */
    root = insertNode(root, 8);
    root = insertNode(root, 4);
    root = insertNode(root, 10);
    root = insertNode(root, 15);
    root = insertNode(root, 14);
}

```

```
root = insertNode(root, 88);
root = insertNode(root, 64);

// Pre-order traversal of tree
preOrder(root);

// Function call to perform queries
performQueries(15);
performQueries(88);

return 0;
}
```

**Java**

```
// Java program to insert nodes
// and print the preorder traversal
import java.util.*;

class Node
{
    int data;
    Node left, right;
    Node(int key)
    {
        data = key;
        left = right = null;
    }
}

class GFG
{
    // ArrayList to
    // store pre-order
    static ArrayList<Integer> pre =
        new ArrayList<Integer>();

    // map to store the height
    // of every subtree
    static HashMap<Integer, Integer> mp =
        new HashMap<Integer, Integer>();

    public static Node insertNode(Node head, int key)
    {
        // if first node
        if (head == null)
            head = new Node(key);
        else
```

```
{  
  
    // move to left  
    if (key < head.data)  
        head.left = insertNode(head.left, key);  
  
    // move to right  
    else  
        head.right = insertNode(head.right, key);  
}  
return head;  
}  
  
public static int preOrder(Node head)  
{  
    // leaf node is null  
    if (head == null)  
        return 0;  
  
    pre.add(head.data);  
  
    mp.put(head.data, head.data +  
            preOrder(head.left));  
    mp.put(head.data, head.data +  
            preOrder(head.right));  
    mp.put(head.data, head.data + 1);  
  
    return mp.get(head.data);  
}  
  
// Function to perform  
// every queries  
public static void performQueries(int node)  
{  
  
    // traverse in the pre-order  
    // jump the subtree which has node  
    for (int i = 0; i < pre.size();)  
    {  
  
        // jump the subtree  
        // which has the node  
        if (pre.get(i) == node)  
        {  
            i += mp.get(pre.get(i));  
        }  
  
        // print the pre-order
```

```
        else
        {
            System.out.print(pre.get(i) + " ");
            i++;
        }
    }
    System.out.println();
}

public static void main (String[] args)
{
    Node root = null;

    /*          8
     *         /   \
    4       10
           \
           15
          / \
        14  88
           /
         64 */

    root = insertNode(root, 8);
    root = insertNode(root, 4);
    root = insertNode(root, 10);
    root = insertNode(root, 15);
    root = insertNode(root, 14);
    root = insertNode(root, 88);
    root = insertNode(root, 64);

    // Pre-order traversal of tree
    preOrder(root);

    // Function call to
    // perform queries
    performQueries(15);
    performQueries(88);
}
}
```

**Output:**

```
8 4 10
8 4 10 15 14
```

## Source

<https://www.geeksforgeeks.org/sudo-placement1-4-jumping-the-subtree/>

## Chapter 403

# Sum of Interval and Update with Number of Divisors

Sum of Interval and Update with Number of Divisors - GeeksforGeeks

Given an array A of N integers. You have to answer two types of queries :

1. Update  $[l, r]$  – for every  $i$  in range from  $l$  to  $r$  update  $A_i$  with  $D(A_i)$ , where  $D(A_i)$  represents the number of divisors of  $A_i$
  2. Query  $[l, r]$  – calculate the sum of all numbers ranging between  $l$  and  $r$  in array A.
- Input is given as two integers  $N$  and  $Q$ , representing number of integers in array and number of queries respectively. Next line contains an array of  $n$  integers followed by  $Q$  queries where  $i$ th query is represented as  $\text{type}_i, l_i, r_i$ .

**Prerequisite :** [Binary Indexed Trees | Segment Trees](#)

Examples :

```
Input : 7 4
        6 4 1 10 3 2 4
        2 1 7
        2 4 5
        1 3 5
        2 4 4
Output : 30
        13
        4
```

**Explanation :** First query is to calculate the sum of numbers from  $A_1$  to  $A_7$  which is  $6 + 4 + 1 + 10 + 3 + 2 + 4 = 30$ . Similarly, second query results into 13. For third query, which is update operation, hence  $A_3$  will remain 1,  $A_4$  will become 4 and  $A_5$  will become 2. Fourth query will result into  $A_4 = 4$ .

**Naive Approach :**

A simple solution is to run a loop from l to r and calculate sum of elements in given range. To update a value, precompute the values of number of divisors of every number and simply do arr[i] = divisors[arr[i]].

**Efficient Approach :**

The idea is to reduce the time complexity for each query and update operation to O(logN). Use Binary Indexed Trees (BIT) or Segment Trees. Construct a BIT[] array and have two functions for query and update operation and precompute the number of divisors for each number. Now, for each update operation the key observation is that the numbers ‘1’ and ‘2’ will have ‘1’ and ‘2’ as their number of divisors respectively, so if it exists in the range of update query, they don’t need to be updated. We will use a set to store the index of only those numbers which are greater than 2 and use binary search to find the l index of the update query and increment the l index until every element is updated in range of that update query. If the arr[i] has only 2 divisors then after updating it, remove it from the set as it will always be 2 even after any next update query. For sum query operation, simply do query(r) – query(l – 1).

```
// CPP program to calculate sum
// in an interval and update with
// number of divisors
#include <bits/stdc++.h>
using namespace std;

int divisors[100], BIT[100];

// structure for queries with members type,
// leftIndex, rightIndex of the query
struct queries
{
    int type, l, r;
};

// function to calculate the number
// of divisors of each number
void calcDivisors()
{
    for (int i = 1; i < 100; i++) {
        for (int j = i; j < 100; j += i) {
            divisors[j]++;
        }
    }
}

// function for updating the value
void update(int x, int val, int n)
{
    for (x; x <= n; x += x&-x) {
        BIT[x] += val;
    }
}
```

```
        }
    }

// function for calculating the required
// sum between two indexes
int sum(int x)
{
    int s = 0;
    for (x; x > 0; x -= x&-x) {
        s += BIT[x];
    }
    return s;
}

// function to return answer to queries
void answerQueries(int arr[], queries que[], int n, int q)
{
    // Declaring a Set
    set<int> s;
    for (int i = 1; i < n; i++) {

        // inserting indexes of those numbers
        // which are greater than 2
        if(arr[i] > 2) s.insert(i);
        update(i, arr[i], n);
    }

    for (int i = 0; i < q; i++) {

        // update query
        if (que[i].type == 1) {
            while (true) {

                // find the left index of query in
                // the set using binary search
                auto it = s.lower_bound(que[i].l);

                // if it crosses the right index of
                // query or end of set, then break
                if(it == s.end() || *it > que[i].r) break;

                que[i].l = *it;

                // update the value of arr[i] to
                // its number of divisors
                update(*it, divisors[arr[*it]] - arr[*it], n);

                arr[*it] = divisors[arr[*it]];
            }
        }
    }
}
```

```
// if updated value becomes less than or
// equal to 2 remove it from the set
if(arr[*it] <= 2) s.erase(*it);

// increment the index
que[i].l++;
}

}

// sum query
else {
    cout << (sum(que[i].r) - sum(que[i].l - 1)) << endl;
}
}

// Driver Code
int main()
{
    // precompute the number of divisors for each number
    calcDivisors();

    int q = 4;

    // input array
    int arr[] = {0, 6, 4, 1, 10, 3, 2, 4};
    int n = sizeof(arr) / sizeof(arr[0]);

    // declaring array of structure of type queries
    queries que[q + 1];

    que[0].type = 2, que[0].l = 1, que[0].r = 7;
    que[1].type = 2, que[1].l = 4, que[1].r = 5;
    que[2].type = 1, que[2].l = 3, que[2].r = 5;
    que[3].type = 2, que[3].l = 4, que[3].r = 4;

    // answer the Queries
    answerQueries(arr, que, n, q);

    return 0;
}
```

**Output:**

4

Time Complexity for answering Q queries will be  $O(Q * \log(N))$ .

### Source

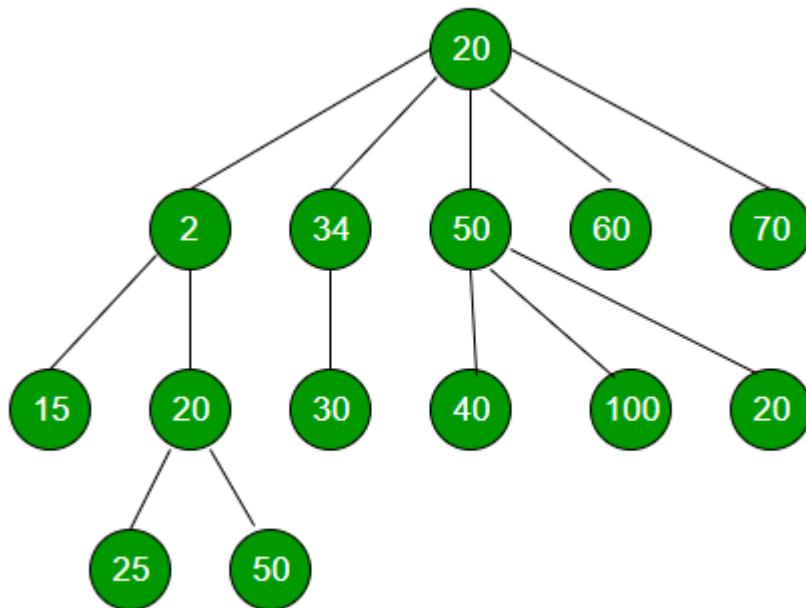
<https://www.geeksforgeeks.org/sum-interval-update-number-divisors/>

## Chapter 404

# Sum of all elements of N-ary Tree

Sum of all elements of N-ary Tree - GeeksforGeeks

Given an N-ary tree, find sum of all elements in it.



Example :

Input : Above tree

Output : Sum is 536

**Approach :** The approach used is similar to [Level Order traversal in a binary tree](#). Start by pushing the root node in the queue. And for each node, while popping it from queue, add the value of this node in the **sum** variable and push the children of the popped element in the queue. In case of a generic tree store child nodes in a vector. Thus, put all elements of the vector in the queue.

Below is the implementation of the above idea :

```
// C++ program to find sum of all
// elements in generic tree
#include <bits/stdc++.h>
using namespace std;

// Represents a node of an n-ary tree
struct Node {
    int key;
    vector<Node*> child;
};

// Utility function to create a new tree node
Node* newNode(int key)
{
    Node* temp = new Node;
    temp->key = key;
    return temp;
}

// Function to compute the sum
// of all elements in generic tree
int sumNodes(Node* root)
{
    // initialize the sum variable
    int sum = 0;

    if (root == NULL)
        return 0;

    // Creating a queue and pushing the root
    queue<Node*> q;
    q.push(root);

    while (!q.empty()) {
        int n = q.size();

        // If this node has children
        while (n > 0) {
```

```
// Dequeue an item from queue and
// add it to variable "sum"
Node* p = q.front();
q.pop();
sum += p->key;

// Enqueue all children of the dequeued item
for (int i = 0; i < p->child.size(); i++)
    q.push(p->child[i]);
n--;
}
}
return sum;
}

// Driver program
int main()
{
    // Creating a generic tree
    Node* root = newNode(20);
    (root->child).push_back(newNode(2));
    (root->child).push_back(newNode(34));
    (root->child).push_back(newNode(50));
    (root->child).push_back(newNode(60));
    (root->child).push_back(newNode(70));
    (root->child[0]->child).push_back(newNode(15));
    (root->child[0]->child).push_back(newNode(20));
    (root->child[1]->child).push_back(newNode(30));
    (root->child[2]->child).push_back(newNode(40));
    (root->child[2]->child).push_back(newNode(100));
    (root->child[2]->child).push_back(newNode(20));
    (root->child[0]->child[1]->child).push_back(newNode(25));
    (root->child[0]->child[1]->child).push_back(newNode(50));

    cout << sumNodes(root) << endl;

    return 0;
}
```

Output:

536

**Time Complexity :** O(N), where N is the number of nodes in tree.  
**Auxiliary Space :** O(N), where N is the number of nodes in tree.

**Source**

<https://www.geeksforgeeks.org/sum-elements-n-ary-tree/>

## Chapter 405

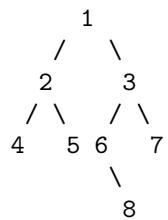
# Sum of all leaf nodes of binary tree

Sum of all leaf nodes of binary tree - GeeksforGeeks

Given a binary tree, find the sum of all the leaf nodes.

Examples:

Input :



Output :

Sum = 4 + 5 + 8 + 7 = 24

The idea is to traverse the tree in any fashion and check if the node is the leaf node or not. If the node is the leaf node, add node data to sum variable.

Following is the implementation of above approach.

C++

```
// CPP program to find sum of
// all leaf nodes of binary tree
#include<bits/stdc++.h>
using namespace std;
```

```
// struct binary tree node
struct Node{
    int data;
    Node *left, *right;
};

// return new node
Node *newNode(int data){
    Node *temp = new Node();
    temp->data = data;
    temp->left = temp->right = NULL;
}

// utility function which calculates
// sum of all leaf nodes
void leafSum(Node *root, int *sum){
    if (!root)
        return;

    // add root data to sum if
    // root is a leaf node
    if (!root->left && !root->right)
        *sum += root->data;

    // propagate recursively in left
    // and right subtree
    leafSum(root->left, sum);
    leafSum(root->right, sum);
}

// driver program
int main(){

    //construct binary tree
    Node *root = newNode(1);
    root->left = newNode(2);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right = newNode(3);
    root->right->right = newNode(7);
    root->right->left = newNode(6);
    root->right->left->right = newNode(8);

    // variable to store sum of leaf nodes
    int sum = 0;
    leafSum(root, &sum);
    cout << sum << endl;
    return 0;
}
```

}

**Java**

```
// Java program to find sum of
// all leaf nodes of binary tree
public class GFG {

    // user define class node
    static class Node{
        int data;
        Node left, right;

        // constructor
        Node(int data){
            this.data = data;
            left = null;
            right = null;
        }
    }

    static int sum;

    // utility function which calculates
    // sum of all leaf nodes
    static void leafSum(Node root){
        if (root == null)
            return;

        // add root data to sum if
        // root is a leaf node
        if (root.left == null && root.right == null)
            sum += root.data;

        // propagate recursively in left
        // and right subtree
        leafSum(root.left);
        leafSum(root.right);
    }

    // driver program
    public static void main(String args[])
    {
        //construct binary tree
        Node root = new Node(1);
        root.left = new Node(2);
        root.left.left = new Node(4);
        root.left.right = new Node(5);
```

```
root.right = new Node(3);
root.right.right = new Node(7);
root.right.left = new Node(6);
root.right.left.right = new Node(8);

// variable to store sum of leaf nodes
sum = 0;
leafSum(root);
System.out.println(sum);
}

}

// This code is contributed by Sumit Ghosh
```

Output:

24

**Time Complexity :**  $O(n)$

### Source

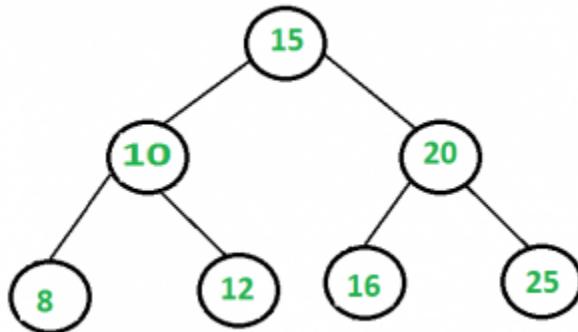
<https://www.geeksforgeeks.org/sum-leaf-nodes-binary-tree/>

## Chapter 406

### Sum of all nodes in a binary tree

Sum of all nodes in a binary tree - GeeksforGeeks

Give an algorithm for finding the sum of all elements in a binary tree.



In the above binary tree sum = 66.

The idea is to recursively, call left subtree sum, right subtree sum and add their values to current node's data.

C++

```
/* Program to print sum of all the elements of a binary tree */
#include <iostream>
using namespace std;

struct Node {
    int key;
    Node* left, *right;
};
```

```
/* utility that allocates a new Node with the given key */
Node* newNode(int key)
{
    Node* node = new Node;
    node->key = key;
    node->left = node->right = NULL;
    return (node);
}

/* Function to find sum of all the elements*/
int addBT(Node* root)
{
    if (root == NULL)
        return 0;
    return (root->key + addBT(root->left) + addBT(root->right));
}

/* Driver program to test above functions*/
int main()
{
    Node* root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->left = newNode(6);
    root->right->right = newNode(7);
    root->right->left->right = newNode(8);

    int sum = addBT(root);
    cout << "Sum of all the elements is: " << sum << endl;

    return 0;
}
```

### Java

```
// Java Program to print sum of
// all the elements of a binary tree
class GFG
{
    static class Node
    {
        int key;
        Node left, right;
    }

    /* utility that allocates a new
    Node with the given key */
```

```
static Node newNode(int key)
{
    Node node = new Node();
    node.key = key;
    node.left = node.right = null;
    return (node);
}

/* Function to find sum
of all the elements*/
static int addBT(Node root)
{
    if (root == null)
        return 0;
    return (root.key + addBT(root.left) +
            addBT(root.right));
}

// Driver Code
public static void main(String args[])
{
    Node root = newNode(1);
    root.left = newNode(2);
    root.right = newNode(3);
    root.left.left = newNode(4);
    root.left.right = newNode(5);
    root.right.left = newNode(6);
    root.right.right = newNode(7);
    root.right.left.right = newNode(8);

    int sum = addBT(root);
    System.out.println("Sum of all the elements is: " + sum);
}
}

// This code is contributed by Arnab Kundu
```

**Output:**

Sum of all the elements is: 36

Improved By : [andrew1234](#)

**Source**

<https://www.geeksforgeeks.org/sum-nodes-binary-tree/>

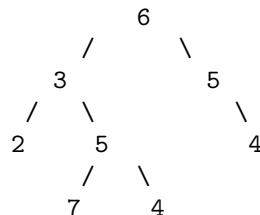
## Chapter 407

# Sum of all the numbers that are formed from root to leaf paths

Sum of all the numbers that are formed from root to leaf paths - GeeksforGeeks

Given a binary tree, where every node value is a Digit from 1-9 .Find the sum of all the numbers which are formed from root to leaf paths.

For example consider the following Binary Tree.



There are 4 leaves, hence 4 root to leaf paths:

Path	Number
6->3->2	632
6->3->5->7	6357
6->3->5->4	6354
6->5->4	654

Answer = 632 + 6357 + 6354 + 654 = 13997

The idea is to do a preorder traversal of the tree. In the preorder traversal, keep track of the value calculated till the current node, let this value be *val*. For every node, we update the *val* as *val*\*10 plus node's data.

C

```
// C program to find sum of all paths from root to leaves
```

```
#include <stdio.h>
#include <stdlib.h>

struct node
{
    int data;
    struct node *left, *right;
};

// function to allocate new node with given data
struct node* newNode(int data)
{
    struct node* node = (struct node*)malloc(sizeof(struct node));
    node->data = data;
    node->left = node->right = NULL;
    return (node);
}

// Returns sum of all root to leaf paths. The first parameter is root
// of current subtree, the second parameter is value of the number formed
// by nodes from root to this node
int treePathsSumUtil(struct node *root, int val)
{
    // Base case
    if (root == NULL)  return 0;

    // Update val
    val = (val*10 + root->data);

    // if current node is leaf, return the current value of val
    if (root->left==NULL && root->right==NULL)
        return val;

    // recur sum of values for left and right subtree
    return treePathsSumUtil(root->left, val) +
           treePathsSumUtil(root->right, val);
}

// A wrapper function over treePathsSumUtil()
int treePathsSum(struct node *root)
{
    // Pass the initial value as 0 as there is nothing above root
    return treePathsSumUtil(root, 0);
}

// Driver function to test the above functions
int main()
{
```

```
struct node *root = newNode(6);
root->left      = newNode(3);
root->right     = newNode(5);
root->left->left = newNode(2);
root->left->right = newNode(5);
root->right->right = newNode(4);
root->left->right->left = newNode(7);
root->left->right->right = newNode(4);
printf("Sum of all paths is", treePathsSum(root));
return 0;
}
```

**Java**

```
// Java program to find sum of all numbers that are formed from root
// to leaf paths

// A binary tree node
class Node
{
    int data;
    Node left, right;

    Node(int item)
    {
        data = item;
        left = right = null;
    }
}

class BinaryTree
{
    Node root;

    // Returns sum of all root to leaf paths. The first parameter is
    // root of current subtree, the second parameter is value of the
    // number formed by nodes from root to this node
    int treePathsSumUtil(Node node, int val)
    {
        // Base case
        if (node == null)
            return 0;

        // Update val
        val = (val * 10 + node.data);

        // if current node is leaf, return the current value of val
        if (node.left == null && node.right == null)
```

```
        return val;

    // recur sum of values for left and right subtree
    return treePathsSumUtil(node.left, val)
           + treePathsSumUtil(node.right, val);
}

// A wrapper function over treePathsSumUtil()
int treePathsSum(Node node)
{
    // Pass the initial value as 0 as there is nothing above root
    return treePathsSumUtil(node, 0);
}

// Driver program to test above functions
public static void main(String args[])
{
    BinaryTree tree = new BinaryTree();
    tree.root = new Node(6);
    tree.root.left = new Node(3);
    tree.root.right = new Node(5);
    tree.root.right.right = new Node(4);
    tree.root.left.left = new Node(2);
    tree.root.left.right = new Node(5);
    tree.root.left.right.right = new Node(4);
    tree.root.left.right.left = new Node(7);

    System.out.print("Sum of all paths is " +
                     tree.treePathsSum(tree.root));
}
}

// This code has been contributed by Mayank Jaiswal
```

### Python

```
# Python program to find sum of all paths from root to leaves

# A Binary tree node
class Node:

    # Constructor to create a new node
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

    # Returns sums of all root to leaf paths. The first parameter is root
```

```
# of current subtree, the second parameter is value of the number
# formed by nodes from root to this node
def treePathsSumUtil(root, val):

    # Base Case
    if root is None:
        return 0

    # Update val
    val = (val*10 + root.data)

    # If current node is leaf, return the current value of val
    if root.left is None and root.right is None:
        return val

    # Recur sum of values for left and right subtree
    return (treePathsSumUtil(root.left, val) +
            treePathsSumUtil(root.right, val))

# A wrapper function over treePathSumUtil()
def treePathsSum(root):

    # Pass the initial value as 0 as ther is nothing above root
    return treePathsSumUtil(root, 0)

# Driver function to test above function
root = Node(6)
root.left = Node(3)
root.right = Node(5)
root.left.left = Node(2)
root.left.right = Node(5)
root.right.right = Node(4)
root.left.right.left = Node(7)
root.left.right.right = Node(4)
print "Sum of all paths is", treePathsSum(root)

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

Output:

```
Sum of all paths is 13997
```

**Time Complexity:** The above code is a simple preorder traversal code which visits every exactly once. Therefore, the time complexity is  $O(n)$  where  $n$  is the number of nodes in the given binary tree.

This article is contributed by **Ramchand R.** Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

**Source**

<https://www.geeksforgeeks.org/sum-numbers-formed-root-leaf-paths/>

## Chapter 408

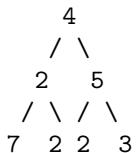
# Sum of all the parent nodes having child node x

Sum of all the parent nodes having child node x - GeeksforGeeks

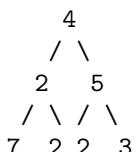
Given a binary tree containing **n** nodes. The problem is to find the sum of all the parent node's which have a child node with value **x**.

**Examples:**

Input : Binary tree with x = 2:



Output : 11



The highlighted nodes (4, 2, 5) above  
are the nodes having 2 as a child node.

**Algorithm:**

```
sumOfParentOfX(root,sum,x)
```

```
if root == NULL
    return

if (root->left && root->left->data == x) ||
(root->right && root->right->data == x)
    sum += root->data

sumOfParentOfX(root->left, sum, x)
sumOfParentOfX(root->right, sum, x)

sumOfParentOfXUtil(root,x)
Declare sum = 0
sumOfParentOfX(root, sum, x)
return sum
```

C++

```
// C++ implementation to find the sum of all
// the parent nodes having child node x
#include <bits/stdc++.h>

using namespace std;

// Node of a binary tree
struct Node
{
    int data;
    Node *left, *right;
};

// function to get a new node
Node* getNode(int data)
{
    // allocate memory for the node
    Node *newNode =
        (Node*)malloc(sizeof(Node));

    // put in the data
    newNode->data = data;
    newNode->left = newNode->right = NULL;
    return newNode;
}

// function to find the sum of all the
// parent nodes having child node x
void sumOfParentOfX(Node* root, int& sum, int x)
{
    // if root == NULL
```

```
if (!root)
    return;

// if left or right child of root is 'x', then
// add the root's data to 'sum'
if ((root->left && root->left->data == x) ||
    (root->right && root->right->data == x))
    sum += root->data;

// recursively find the required parent nodes
// in the left and right subtree
sumOfParentOfX(root->left, sum, x);
sumOfParentOfX(root->right, sum, x);

}

// utility function to find the sum of all
// the parent nodes having child node x
int sumOfParentOfXUtil(Node* root, int x)
{
    int sum = 0;
    sumOfParentOfX(root, sum, x);

    // required sum of parent nodes
    return sum;
}

// Driver program to test above
int main()
{
    // binary tree formation
    Node *root = getNode(4);          /*      4      */
    root->left = getNode(2);          /*      / \      */
    root->right = getNode(5);         /*      2 5      */
    root->left->left = getNode(7);   /*      / \ / \      */
    root->left->right = getNode(2); /*      7 2 2 3      */
    root->right->left = getNode(2);
    root->right->right = getNode(3);

    int x = 2;

    cout << "Sum = "
        << sumOfParentOfXUtil(root, x);

    return 0;
}
```

Java

```
// Java implementation to find
// the sum of all the parent
// nodes having child node x
class GFG
{
    // sum
    static int sum = 0;

    // Node of a binary tree
    static class Node
    {
        int data;
        Node left, right;
    };

    // function to get a new node
    static Node getNode(int data)
    {
        // allocate memory for the node
        Node newNode = new Node();

        // put in the data
        newNode.data = data;
        newNode.left = newNode.right = null;
        return newNode;
    }

    // function to find the sum of all the
    // parent nodes having child node x
    static void sumOfParentOfX(Node root, int x)
    {
        // if root == NULL
        if (root == null)
            return;

        // if left or right child
        // of root is 'x', then
        // add the root's data to 'sum'
        if ((root.left != null && root.left.data == x) ||
            (root.right != null && root.right.data == x))
            sum += root.data;

        // recursively find the required
        // parent nodes in the left and
        // right subtree
        sumOfParentOfX(root.left, x);
        sumOfParentOfX(root.right, x);
    }

    // utility function to find the
    // sum of all the parent nodes
```

```
// having child node x
static int sumOfParentOfXUtil(Node root,
int x)
{
sum = 0;
sumOfParentOfX(root, x);

// required sum of parent nodes
return sum;
}

// Driver Code
public static void main(String args[])
{
// binary tree formation
Node root = getNode(4); // 4
root.left = getNode(2); // / \
root.right = getNode(5); // 2 5
root.left.left = getNode(7); // / \ / \
root.left.right = getNode(2); // 7 2 2 3
root.right.left = getNode(2);
root.right.right = getNode(3);

int x = 2;

System.out.println("Sum = " +
sumOfParentOfXUtil(root, x));
}
}

// This code is contributed by Arnab Kundu
```

**Output:**

Sum = 11

**Time Complexity:** O(n).

**Improved By :** [gardhiwasam](#), [andrew1234](#)

**Source**

<https://www.geeksforgeeks.org/sum-parent-nodes-child-node-x/>

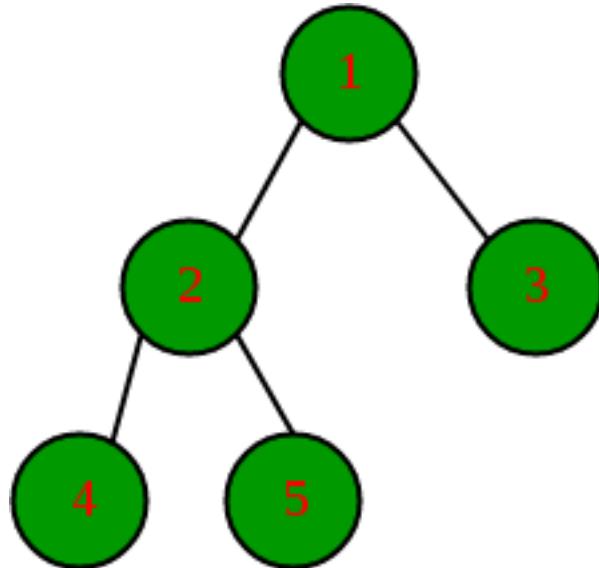
## Chapter 409

### Sum of heights of all individual nodes in a binary tree

Sum of heights of all individual nodes in a binary tree - GeeksforGeeks

Given a binary tree, find sum of heights all individual Nodes in the tree.

Example:



For this tree:

- 1). Height of Node 1 - 3
- 2). Height of Node 2 - 2
- 3). Height of Node 3 - 1
- 4). Height of Node 4 - 1

5). Height of Node 5 - 1

Adding all of them = 8

**Prerequisites :-** [Height of binary tree](#)

**Simple Solution :**

We get the height of all individual Nodes by parsing the tree in any of the following methods i.e. Inorder, postorder, preorder(I performed inorder tree traversal) and getting their heights using *getHeight* function which checks both left and right subtree and returns the maximum of them. Finally we add up all the individual heights.

```
// CPP program to find sum of heights of all
// nodes in a binary tree
#include <stdio.h>
#include <stdlib.h>

/* A binary tree Node has data, pointer to
   left child and a pointer to right child */
struct Node {
    int data;
    struct Node* left;
    struct Node* right;
};

/* Compute the "maxHeight" of a particular Node*/
int getHeight(struct Node* Node)
{
    if (Node == NULL)
        return 0;
    else {
        /* compute the height of each subtree */
        int lHeight = getHeight(Node->left);
        int rHeight = getHeight(Node->right);

        /* use the larger one */
        if (lHeight > rHeight)
            return (lHeight + 1);
        else
            return (rHeight + 1);
    }
}

/* Helper function that allocates a new Node with the
   given data and NULL left and right pointers. */
struct Node* newNode(int data)
{
```

```
struct Node* newNode (struct Node*)
{
    Node->data = data;
    Node->left = NULL;
    Node->right = NULL;

    return (Node);
}

/* Function to sum of heights of individual Nodes
   Uses Inorder traversal */
int getTotalHeight(struct Node* root)
{
    if (root == NULL)
        return 0;

    return getTotalHeight(root->left) +
           getHeight(root) +
           getTotalHeight(root->right);
}

// Driver code
int main()
{
    struct Node* root = newNode(1);

    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    printf("Sum of heights of all Nodes = %d",
          getTotalHeight(root));
    return 0;
}
```

**Output:**

```
Sum of heights of all Nodes = 8
```

Time Complexity : O(nh) where n is total number of nodes and h is height of binary tree.

**Efficient Solution :**

The idea is to compute heights and sum in same recursive call.

```
// CPP program to find sum of heights of all
```

```
// nodes in a binary tree
#include <bits/stdc++.h>
using namespace std;

/* A binary tree Node has data, pointer to
   left child and a pointer to right child */
struct Node {
    int data;
    struct Node* left;
    struct Node* right;
};

/* Helper function that allocates a new Node with the
   given data and NULL left and right pointers. */
struct Node* newNode(int data)
{
    struct Node* Node = (struct Node*)
        malloc(sizeof(struct Node));
    Node->data = data;
    Node->left = NULL;
    Node->right = NULL;

    return (Node);
}

/* Function to sum of heights of individual Nodes
   Uses Inorder traversal */
int getTotalHeightUtil(struct Node* root, int &sum)
{
    if (root == NULL)
        return 0;

    int lh = getTotalHeightUtil(root->left, sum);
    int rh = getTotalHeightUtil(root->right, sum);
    int h = max(lh, rh) + 1;

    sum = sum + h;
    return h;
}

int getTotalHeight(Node *root)
{
    int sum = 0;
    getTotalHeightUtil(root, sum);
    return sum;
}

// Driver code
```

```
int main()
{
    struct Node* root = newNode(1);

    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    printf("Sum of heights of all Nodes = %d",
           getTotalHeight(root));
    return 0;
}
```

**Output:**

Sum of heights of all Nodes = 8

Time Complexity :  $O(nh)$  where n is total number of nodes and h is height of binary tree.

**Improved By :** [akash1295](#)

**Source**

<https://www.geeksforgeeks.org/sum-heights-individual-nodes-binary-tree/>

## Chapter 410

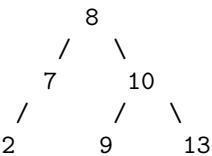
# Sum of k largest elements in BST

Sum of k largest elements in BST - GeeksforGeeks

Given a **BST**, the task is to find the sum of all elements greater than and equal to  $k$ th largest element.

Examples:

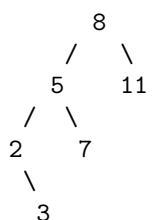
Input :  $K = 3$



Output : 32

Explanation: 3rd largest element is 9 so sum of all elements greater than or equal to 9 are  $9 + 10 + 13 = 32$ .

Input :  $K = 2$



Output : 19

Explanation: 2nd largest element is 8 so sum of all elements greater than or equal to 8 are  $8 + 11 = 19$ .

**Approach:**

The idea is to traverse BST in **Inorder traversal** in a **reverse** way (Right Root Left). Note that Inorder traversal of BST accesses elements in a sorted (or increasing) order, hence the reverse of inorder traversal will be in a sorted order(**decreasing**). While traversing, keep track of the count of visited Nodes and keep adding Nodes until the count becomes k.

```
// C++ program to find Sum Of All Elements larger
// than or equal to Kth Largest Element In BST
#include <bits/stdc++.h>
using namespace std;

struct Node {
    int data;
    Node *left, *right;
};

// utility function new Node of BST
struct Node* cNode(int data)
{
    Node* node = new Node;
    node->left = NULL;
    node->right = NULL;
    node->data = data;
    return node;
}

// A utility function to insert a new Node
// with given key in BST
struct Node* add(Node* root, int key)
{
    // If the tree is empty, return a new Node
    if (root == NULL)
        return cNode(key);

    // Otherwise, recur down the tree
    if (root->data > key)
        root->left = add(root->left, key);

    else if (root->data < key)
        root->right = add(root->right, key);

    // return the (unchanged) Node pointer
    return root;
}

// function to return sum of all elements larger than
// and equal to Kth largest element
int klargestElementSumUtil(Node* root, int k, int& c)
```

```
{  
    // Base cases  
    if (root == NULL)  
        return 0;  
    if (c > k)  
        return 0;  
  
    // Compute sum of elements in right subtree  
    int ans = klargestElementSumUtil(root->right, k, c);  
    if (c >= k)  
        return ans;  
  
    // Add root's data  
    ans += root->data;  
  
    // Add current Node  
    c++;  
    if (c >= k)  
        return ans;  
  
    // If c is less than k, return left subtree Nodes  
    return ans + klargestElementSumUtil(root->left, k, c);  
}  
  
// Wrapper over klargestElementSumRec()  
int klargestElementSum(struct Node* root, int k)  
{  
    int c = 0;  
    klargestElementSumUtil(root, k, c);  
}  
  
// Drivers code  
int main()  
{  
    /*      19  
         /   \\  
        7     21  
       /   \  
      3     11  
         /   \  
        9     13  
       */  
  
    Node* root = NULL;  
    root = add(root, 19);  
    root = add(root, 7);  
    root = add(root, 3);  
    root = add(root, 11);
```

```
root = add(root, 9);
root = add(root, 13);
root = add(root, 21);

int k = 2;
cout << klargestElementSum(root, k) << endl;
return 0;
}
```

Output:

40

## Source

<https://www.geeksforgeeks.org/sum-of-k-largest-elements-in-bst/>

## Chapter 411

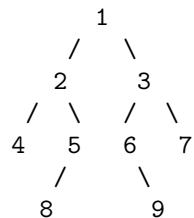
# Sum of leaf nodes at minimum level

Sum of leaf nodes at minimum level - GeeksforGeeks

Given a binary tree containing **n** nodes. The problem is to get the sum of all the leaf nodes which are at minimum level in the binary tree.

Examples:

Input :



Output : 11

Leaf nodes 4 and 7 are at minimum level.

Their sum = (4 + 7) = 11.

Source: [Microsoft IDC Interview Experience | Set 150.](#)

**Approach:** Perform [iterative level order traversal](#) using queue and find the first level containing a leaf node. Sum up all the leaf nodes at this level and then stop performing the traversal further.

```
// C++ implementation to find the sum of
// leaf nodes at minimum level
#include <bits/stdc++.h>
```

```
using namespace std;

// structure of a node of binary tree
struct Node {
    int data;
    Node *left, *right;
};

// function to get a new node
Node* getNode(int data)
{
    // allocate space
    Node* newNode = (Node*)malloc(sizeof(Node));

    // put in the data
    newNode->data = data;
    newNode->left = newNode->right = NULL;
    return newNode;
}

// function to find the sum of
// leaf nodes at minimum level
int sumOfLeafNodesAtMinLevel(Node* root)
{
    // if tree is empty
    if (!root)
        return 0;

    // if there is only one node
    if (!root->left && !root->right)
        return root->data;

    // queue used for level order traversal
    queue<Node*> q;
    int sum = 0;
    bool f = 0;

    // push root node in the queue 'q'
    q.push(root);

    while (f == 0) {

        // count number of nodes in the
        // current level
        int nc = q.size();

        // traverse the current level nodes
        while (nc--) {
```

```
// get front element from 'q'
Node* top = q.front();
q.pop();

// if it is a leaf node
if (!top->left && !top->right) {

    // accumulate data to 'sum'
    sum += top->data;

    // set flag 'f' to 1, to signify
    // minimum level for leaf nodes
    // has been encountered
    f = 1;
}
else {

    // if top's left and right child
    // exists, then push them to 'q'
    if (top->left)
        q.push(top->left);
    if (top->right)
        q.push(top->right);
}
}

// required sum
return sum;
}

// Driver program to test above
int main()
{
    // binary tree creation
    Node* root = getNode(1);
    root->left = getNode(2);
    root->right = getNode(3);
    root->left->left = getNode(4);
    root->left->right = getNode(5);
    root->right->left = getNode(6);
    root->right->right = getNode(7);
    root->left->right->left = getNode(8);
    root->right->left->right = getNode(9);

    cout << "Sum = "
        << sumOfLeafNodesAtMinLevel(root);
```

```
    return 0;  
}
```

Output:

```
Sum = 11
```

Time Complexity:  $O(n)$ .  
Auxiliary Space:  $O(n)$ .

## Source

<https://www.geeksforgeeks.org/sum-leaf-nodes-minimum-level/>

## Chapter 412

# Sum of nodes at k-th level in a tree represented as string

Sum of nodes at k-th level in a tree represented as string - GeeksforGeeks

Given an integer 'K' and a binary tree in string format. Every node of a tree has value in range from 0 to 9. We need to find sum of elements at K-th level from root. The root is at level 0.

Tree is given in the form: (node value(left subtree)(right subtree))

Examples:

```
Input : tree = "(0(5(6()())(4()(9()())))(7(1()())(3()())))"
         k = 2
```

```
Output : 14
```

Its tree representation is shown below

```
Elements at level k = 2 are 6, 4, 1, 3
sum of the digits of these elements = 6+4+1+3 = 14
```

```
Input : tree = "(8(3(2()())(6(5()())()))
                  (5(10()())(7(13()())())))
                  k = 3
```

```
Output : 9
```

```
Elements at level k = 3 are 5, 1 and 3
sum of digits of these elements = 5+1+3 = 9
```

1. Input 'tree' in string format and level k
2. Initialize level = -1 and sum = 0
3. for each character 'ch' in 'tree'
  - 3.1 if ch == '(' then

```
--> level++
3.2 else if ch == ')' then
    --> level--
3.3 else
    if level == k then
        sum = sum + (ch-'0')
4. Print sum

// C++ implementation to find sum of
// digits of elements at k-th level
#include <bits/stdc++.h>
using namespace std;

// Function to find sum of digits
// of elements at k-th level
int sumAtKthLevel(string tree, int k)
{
    int level = -1;
    int sum = 0; // Initialize result
    int n = tree.length();

    for (int i=0; i<n; i++)
    {
        // increasing level number
        if (tree[i] == '(')
            level++;

        // decreasing level number
        else if (tree[i] == ')')
            level--;

        else
        {
            // check if current level is
            // the desired level or not
            if (level == k)
                sum += (tree[i]-'0');
        }
    }

    // required sum
    return sum;
}

// Driver program to test above
int main()
{
    string tree = "(0(5(6())())(4()(9())())(7(1())(3())()))";
}
```

```
int k = 2;
cout << sumAtKthLevel(tree, k);
return 0;
}
```

Output:

14

Time Complexity: O(n)

## Source

<https://www.geeksforgeeks.org/sum-nodes-k-th-level-tree-represented-string/>

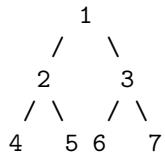
## Chapter 413

# Sum of nodes at maximum depth of a Binary Tree

Sum of nodes at maximum depth of a Binary Tree - GeeksforGeeks

Given a root node to a tree, find the sum of all the leaf nodes which are at maximum depth from root node.

Example:



Input : root(of above tree)  
Output : 22

Explanation:

Nodes at maximum depth are: 4, 5, 6, 7.  
So, sum of these nodes = 22

**Approach:** Calculate the max depth of the given tree. Now, start traversing the tree similarly as traversed during maximum depth calculation. But, this time with one more argument (i.e. maxdepth), and traverse recursively with decreasing depth by 1 for each left or right call. Wherever max == 1, means the node at max depth is reached. So add its data value to sum. Finally, return sum.

Below is the implementation for above approach:

```
// Java code for sum of nodes
```

```
// at maximum depth
import java.util.*;

class Node {
    int data;
    Node left, right;

    // Constructor
    public Node(int data)
    {
        this.data = data;
        this.left = null;
        this.right = null;
    }
}

class GfG {

    // function to find the sum of nodes at
    // maximum depth arguments are node and
    // max, where max is to match the depth
    // of node at every call to node, if
    // max will be equal to 1, means
    // we are at deepest node.
    public static int sumMaxLevelRec(Node node,
                                      int max)
    {
        // base case
        if (node == null)
            return 0;

        // max == 1 to track the node
        // at deepest level
        if (max == 1)
            return node.data;

        // recursive call to left and right nodes
        return sumMaxLevelRec(node.left, max - 1) +
               sumMaxLevelRec(node.right, max - 1);
    }

    public static int sumMaxLevel(Node root) {

        // call to function to calculate
        // max depth
        int MaxDepth = maxDepth(root);

        return sumMaxLevelRec(root, MaxDepth);
    }
}
```

```
}

// maxDepth function to find the
// max depth of the tree
public static int maxDepth(Node node)
{
    // base case
    if (node == null)
        return 0;

    // either leftDepth or rightDepth is
    // greater add 1 to include height
    // of node at which call is
    return 1 + Math.max(maxDepth(node.left),
                        maxDepth(node.right));
}

// Driver code
public static void main(String[] args)
{
    /*
        1
       / \
      2   3
     / \ / \
    4 5 6 7
    */

    // Constructing tree
    Node root = new Node(1);
    root.left = new Node(2);
    root.right = new Node(3);
    root.left.left = new Node(4);
    root.left.right = new Node(5);
    root.right.left = new Node(6);
    root.right.right = new Node(7);

    // call to calculate required sum
    System.out.println(sumMaxLevel(root));
}
}
```

Output :

22

**Time Complexity:** O(N), where N is the number of nodes in the tree.

**Source**

<https://www.geeksforgeeks.org/sum-nodes-maximum-depth-binary-tree/>

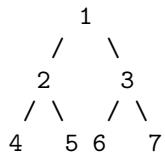
## Chapter 414

# Sum of nodes at maximum depth of a Binary Tree | Iterative Approach

Sum of nodes at maximum depth of a Binary Tree | Iterative Approach - GeeksforGeeks

Given a root node to a tree, find the sum of all the leaf nodes which are at maximum depth from root node.

Example:



Input : root(of above tree)  
Output : 22

Explanation:

Nodes at maximum depth are 4, 5, 6, 7.  
So, the sum of these nodes = 22

**Approach:** There exists a [recursive](#) approach to this problem. This can also be solved using level order traversal and map. The idea is to do a traversal using a queue and keep track of current level. A [map](#) has been used to store the sum of nodes at the current level. Once all nodes are visited and the traversal is done, the last element of the map will contain the sum at the maximum depth of the tree.

Below is the implementation of the above approach:

```
// C++ program to calculate the sum of
// nodes at the maximum depth of a binary tree
#include <bits/stdc++.h>
using namespace std;

struct node {
    int data;
    node *left, *right;
} * temp;

node* newNode(int data)
{
    temp = new node;
    temp->data = data;
    temp->left = temp->right = NULL;

    return temp;
}

// Function to return the sum
int SumAtMaxLevel(node* root)
{
    // Map to store level wise sum.
    map<int, int> mp;

    // Queue for performing Level Order Traversal.
    // First entry is the node and
    // second entry is the level of this node.
    queue<pair<node*, int>> q;

    // Root has level 0.
    q.push({root, 0});

    while (!q.empty()) {

        // Get the node from front of Queue.
        pair<node*, int> temp = q.front();
        q.pop();

        // Get the depth of current node.
        int depth = temp.second;

        // Add the value of this node in map.
        mp[depth] += (temp.first)->data;

        // Push children of this node,
        // with increasing the depth.
        if (temp.first->left)
```

```
q.push({ temp.first->left, depth + 1 });

if (temp.first->right)
    q.push({ temp.first->right, depth + 1 });
}

map<int, int>::iterator it;

// Get the max depth from map.
it = mp.end();

// last element
it--;

// return the max Depth sum.
return it->second;
}

// Driver Code
int main()
{
    node* root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->left = newNode(6);
    root->right->right = newNode(7);

    cout << SumAtMaxLevel(root) << endl;
    return 0;
}
```

**Output:**

22

**Source**

<https://www.geeksforgeeks.org/sum-of-nodes-at-maximum-depth-of-a-binary-tree-iterative-approach/>

## Chapter 415

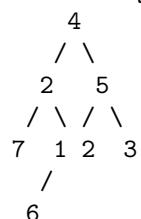
# Sum of nodes on the longest path from root to leaf node

Sum of nodes on the longest path from root to leaf node - GeeksforGeeks

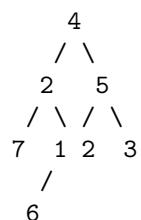
Given a binary tree containing  $n$  nodes. The problem is to find the sum of all nodes on the longest path from root to leaf node. If two or more paths compete for the longest path, then the path having maximum sum of nodes is being considered.

Examples:

Input : Binary tree:



Output : 13



The highlighted nodes (4, 2, 1, 6) above are part of the longest root to leaf path having sum =  $(4 + 2 + 1 + 6) = 13$

**Approach:** Recursively find the length and sum of nodes of each root to leaf path and accordingly update the maximum sum.

**Algorithm:**

```
sumOfLongRootToLeafPath(root, sum, len, maxLen, maxSum)
    if root == NULL
        if maxLen < len
            maxLen = len
            maxSum = sum
        else if maxLen == len && maxSum is less than sum
            maxSum = sum
        return
    sumOfLongRootToLeafPath(root-left, sum + root-data,
                           len + 1, maxLen, maxSum)
    sumOfLongRootToLeafPath(root-right, sum + root-data,
                           len + 1, maxLen, maxSum)

sumOfLongRootToLeafPathUtil(root)
    if (root == NULL)
        return 0

    Declare maxSum = Minimum Integer
    Declare maxLen = 0
    sumOfLongRootToLeafPath(root, 0, 0, maxLen, maxSum)
    return maxSum
```

C++

```
// C++ implementation to find the sum of nodes
// on the longest path from root to leaf node
#include <bits/stdc++.h>

using namespace std;

// Node of a binary tree
struct Node {
    int data;
    Node* left, *right;
};

// function to get a new node
Node* getNode(int data)
{
    // allocate memory for the node
    Node* newNode = (Node*)malloc(sizeof(Node));
```

```
// put in the data
newNode->data = data;
newNode->left = newNode->right = NULL;
return newNode;
}

// function to find the sum of nodes on the
// longest path from root to leaf node
void sumOfLongRootToLeafPath(Node* root, int sum,
                             int len, int& maxLen, int& maxSum)
{
    // if true, then we have traversed a
    // root to leaf path
    if (!root) {
        // update maximum length and maximum sum
        // according to the given conditions
        if (maxLen < len) {
            maxLen = len;
            maxSum = sum;
        } else if (maxLen == len && maxSum < sum)
            maxSum = sum;
        return;
    }

    // recur for left subtree
    sumOfLongRootToLeafPath(root->left, sum + root->data,
                           len + 1, maxLen, maxSum);

    // recur for right subtree
    sumOfLongRootToLeafPath(root->right, sum + root->data,
                           len + 1, maxLen, maxSum);
}

// utility function to find the sum of nodes on
// the longest path from root to leaf node
int sumOfLongRootToLeafPathUtil(Node* root)
{
    // if tree is NULL, then sum is 0
    if (!root)
        return 0;

    int maxSum = INT_MIN, maxLen = 0;

    // finding the maximum sum 'maxSum' for the
    // maximum length root to leaf path
    sumOfLongRootToLeafPath(root, 0, 0, maxLen, maxSum);
```

```
// required maximum sum
return maxSum;
}

// Driver program to test above
int main()
{
    // binary tree formation
    Node* root = getNode(4);          /*      4      */
    root->left = getNode(2);         /*      / \      */
    root->right = getNode(5);        /*      2 5      */
    root->left->left = getNode(7);  /*      / \ / \      */
    root->left->right = getNode(1); /*      7 1 2 3      */
    root->right->left = getNode(2); /*      /      */
    root->right->right = getNode(3);/*      6      */
    root->left->right->left = getNode(6);

    cout << "Sum = "
        << sumOfLongRootToLeafPathUtil(root);

    return 0;
}
```

### Java

```
// Java implementation to find the sum of nodes
// on the longest path from root to leaf node
public class GFG
{
    // Node of a binary tree
    static class Node {
        int data;
        Node left, right;

        Node(int data){
            this.data = data;
            left = null;
            right = null;
        }
    }
    static int maxLen;
    static int maxSum;

    // function to find the sum of nodes on the
    // longest path from root to leaf node
    static void sumOfLongRootToLeafPath(Node root, int sum,
                                         int len)
    {
```

```
// if true, then we have traversed a
// root to leaf path
if (root == null) {
    // update maximum length and maximum sum
    // according to the given conditions
    if (maxLen < len) {
        maxLen = len;
        maxSum = sum;
    } else if (maxLen == len && maxSum < sum)
        maxSum = sum;
    return;
}

// recur for left subtree
sumOfLongRootToLeafPath(root.left, sum + root.data,
                        len + 1);

sumOfLongRootToLeafPath(root.right, sum + root.data,
                        len + 1);

}

// utility function to find the sum of nodes on
// the longest path from root to leaf node
static int sumOfLongRootToLeafPathUtil(Node root)
{
    // if tree is NULL, then sum is 0
    if (root == null)
        return 0;

    maxSum = Integer.MIN_VALUE;
    maxLen = 0;

    // finding the maximum sum 'maxSum' for the
    // maximum length root to leaf path
    sumOfLongRootToLeafPath(root, 0, 0);

    // required maximum sum
    return maxSum;
}

// Driver program to test above
public static void main(String args[])
{
    // binary tree formation
    Node root = new Node(4);          /*      4      */
    root.left = new Node(2);          /*  / \ */
    /*
```

```
root.right = new Node(5);      /*      2      5      */
root.left.left = new Node(7);   /*      / \ / \      */
root.left.right = new Node(1);  /*      7  1 2  3      */
root.right.left = new Node(2);  /*      /      */
root.right.right = new Node(3); /*      6      */
root.left.right.left = new Node(6);

System.out.println( "Sum = "
+ sumOfLongRootToLeafPathUtil(root));
}

// This code is contributed by Sumit Ghosh
```

Output:

Sum = 13

Time Complexity: O(n)

### Source

<https://www.geeksforgeeks.org/sum-nodes-longest-path-root-leaf-node/>

## Chapter 416

# Swap Nodes in Binary tree of every k'th level

Swap Nodes in Binary tree of every k'th level - GeeksforGeeks

Given a binary tree and integer value k, the task is to swap sibling nodes of every k'th level where k >= 1.

Examples:

```
Input : k = 2 and Root of below tree
        1
          /   \
        2     3       Level 1
      /   /   \
    4   7   8       Level 2
  
```

Output : Root of the following modified tree

```
        1
        /   \
      3     2
    / \   /
  7   8   4
  
```

Explanation : We need to swap left and right sibling every second level. There is only one even level with nodes to be swapped are 2 and 3.

Input : k = 1 and Root of following tree

```
        1
          Level 1
  
```

```
      /   \
     2   3       Level 2
    / \
   4   5       Level 3
Output : Root of the following modified tree
      1
     / \
    3   2
   /   \
  5   4
```

Since k is 1, we need to swap sibling nodes of all levels.

A **simple solution** of this problem is that for each is to find sibling nodes for each multiple of k and swap them.

An **efficient solution** is to keep track of level number in recursive calls. And for every node being visited, check if level number of its children is a multiple of k. If yes, then swap the two children of the node. Else, recur for left and right children.

Below is C++ implementation of above idea

C++

```
// c++ program swap nodes
#include<bits/stdc++.h>
using namespace std;

// A Binary Tree Node
struct Node
{
    int data;
    struct Node *left, *right;
};

// function to create a new tree node
Node* newNode(int data)
{
    Node *temp = new Node;
    temp->data = data;
    temp->left = temp->right = NULL;
    return temp;
}

// swap two Node
void Swap( Node **a , Node **b)
{
    Node * temp = *a;
```

```

        *a = *b;
        *b = temp;
    }

// A utility function swap left- node & right node of tree
// of every k'th level
void swapEveryKLevelUtil( Node *root, int level, int k)
{
    // base case
    if (root== NULL ||
        (root->left==NULL && root->right==NULL) )
        return ;

    //if current level + 1  is present in swap vector
    //then we swap left & right node
    if ( (level + 1) % k == 0)
        Swap(&root->left, &root->right);

    // Recur for left and right subtrees
    swapEveryKLevelUtil(root->left, level+1, k);
    swapEveryKLevelUtil(root->right, level+1, k);
}

// This function mainly calls recursive function
// swapEveryKLevelUtil()
void swapEveryKLevel(Node *root, int k)
{
    // call swapEveryKLevelUtil function with
    // initial level as 1.
    swapEveryKLevelUtil(root, 1, k);
}

// Utility method for inorder tree traversal
void inorder(Node *root)
{
    if (root == NULL)
        return;
    inorder(root->left);
    cout << root->data << " ";
    inorder(root->right);
}

// Driver Code
int main()
{
    /*
        1
       /   \
      2     3

```

```
/      /  \
4      7   8  */
struct Node *root = newNode(1);
root->left = newNode(2);
root->right = newNode(3);
root->left->left = newNode(4);
root->right->right = newNode(8);
root->right->left = newNode(7);

int k = 2;
cout << "Before swap node :" << endl;
inorder(root);

swapEveryKLevel(root, k);

cout << "\nAfter swap Node :" << endl;
inorder(root);
return 0;
}
```

### Python

```
# Python program to swap nodes

# A binary tree node
class Node:

    # constructor to create a new node
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

    # A utility function swap left node and right node of tree
    # of every k'th level
    def swapEveryKLevelUtil(self, level, k):

        # Base Case
        if (self is None or (self.left is None and
                             self.right is None)):
            return

        # If current level+1 is present in swap vector
        # then we swap left and right node
        if (level+1)%k == 0:
            self.left, self.right = self.right, self.left

        # Recur for left and right subtree
```

```
swapEveryKLevelUtil(root.left, level+1, k)
swapEveryKLevelUtil(root.right, level+1, k)

# This function mainly calls recursive function
# swapEveryKLevelUtil
def swapEveryKLevel(root, k):

    # Call swapEveryKLevelUtil function with
    # initial level as 1
    swapEveryKLevelUtil(root, 1, k)

# Method to find the inorder tree traversal
def inorder(root):

    # Base Case
    if root is None:
        return
    inorder(root.left)
    print root.data,
    inorder(root.right)

# Driver code
"""
          1
         /   \
        2     3
       /   /   \
      4   7   8
"""

root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.right.right = Node(8)
root.right.left = Node(7)

k = 2
print "Before swap node :"
inorder(root)

swapEveryKLevel(root, k)

print "\nAfter swap Node : "
inorder(root)

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

Output:

```
Before swap node :  
4 2 1 7 3 8  
After swap Node :  
7 3 8 1 4 2
```

### Source

<https://www.geeksforgeeks.org/swap-nodes-binary-tree-every-kth-level/>

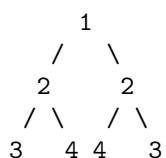
## Chapter 417

# Symmetric Tree (Mirror Image of itself)

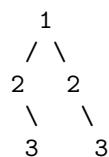
Symmetric Tree (Mirror Image of itself) - GeeksforGeeks

Given a binary tree, check whether it is a mirror of itself.

For example, this binary tree is symmetric:



But the following is not:



The idea is to write a recursive function `isMirror()` that takes two trees as argument and returns true if trees are mirror and false if trees are not mirror. The `isMirror()` function recursively checks two roots and subtrees under the root.

Below is implementation of above algorithm.

C++

```
// C++ program to check if a given Binary Tree is symmetric or not
#include<bits/stdc++.h>
using namespace std;

// A Binary Tree Node
struct Node
{
    int key;
    struct Node* left, *right;
};

// Utility function to create new Node
Node *newNode(int key)
{
    Node *temp = new Node;
    temp->key = key;
    temp->left = temp->right = NULL;
    return (temp);
}

// Returns true if trees with roots as root1 and root2 are mirror
bool isMirror(struct Node *root1, struct Node *root2)
{
    // If both trees are empty, then they are mirror images
    if (root1 == NULL && root2 == NULL)
        return true;

    // For two trees to be mirror images, the following three
    // conditions must be true
    // 1 - Their root node's key must be same
    // 2 - left subtree of left tree and right subtree
    //      of right tree have to be mirror images
    // 3 - right subtree of left tree and left subtree
    //      of right tree have to be mirror images
    if (root1 && root2 && root1->key == root2->key)
        return isMirror(root1->left, root2->right) &&
               isMirror(root1->right, root2->left);

    // if neither of above conditions is true then root1
    // and root2 are not mirror images
    return false;
}

// Returns true if a tree is symmetric i.e. mirror image of itself
bool isSymmetric(struct Node* root)
{
    // Check if tree is mirror of itself
    return isMirror(root, root);
```

```
}

// Driver program
int main()
{
    // Let us construct the Tree shown in the above figure
    Node *root      = newNode(1);
    root->left     = newNode(2);
    root->right    = newNode(2);
    root->left->left  = newNode(3);
    root->left->right = newNode(4);
    root->right->left = newNode(4);
    root->right->right = newNode(3);

    cout << isSymmetric(root);
    return 0;
}
```

### Java

```
// Java program to check is binary tree is symmetric or not
class Node
{
    int key;
    Node left, right;

    Node(int item)
    {
        key = item;
        left = right = null;
    }
}

class BinaryTree
{
    Node root;

    // returns true if trees with roots as root1 and root2 are mirror
    boolean isMirror(Node node1, Node node2)
    {
        // if both trees are empty, then they are mirror image
        if (node1 == null && node2 == null)
            return true;

        // For two trees to be mirror images, the following three
        // conditions must be true
        // 1 - Their root node's key must be same
        // 2 - left subtree of left tree and right subtree
```

```
//      of right tree have to be mirror images
// 3 - right subtree of left tree and left subtree
//      of right tree have to be mirror images
if (node1 != null && node2 != null && node1.key == node2.key)
    return (isMirror(node1.left, node2.right)
        && isMirror(node1.right, node2.left));

// if neither of the above conditions is true then
// root1 and root2 are mirror images
return false;
}

// returns true if the tree is symmetric i.e.
// mirror image of itself
boolean isSymmetric(Node node)
{
    // check if tree is mirror of itself
    return isMirror(root, root);
}

// Driver program
public static void main(String args[])
{
    BinaryTree tree = new BinaryTree();
    tree.root = new Node(1);
    tree.root.left = new Node(2);
    tree.root.right = new Node(2);
    tree.root.left.left = new Node(3);
    tree.root.left.right = new Node(4);
    tree.root.right.left = new Node(4);
    tree.root.right.right = new Node(3);
    boolean output = tree.isSymmetric(tree.root);
    if (output == true)
        System.out.println("1");
    else
        System.out.println("0");
}
}

// This code has been contributed by Mayank Jaiswal
```

### Python

```
# Python program to check if a given Binary Tree is
# symmetric or not

# Node structure
class Node:
```

```
# Utility function to create new node
def __init__(self, key):
    self.key = key
    self.left = None
    self.right = None

# Returns True if trees with roots as root1 and root 2
# are mirror
def isMirror(root1 , root2):
    # If both trees are empty, then they are mirror images
    if root1 is None and root2 is None:
        return True

    """ For two trees to be mirror images, the following three
    conditions must be true
    1 - Their root node's key must be same
    2 - left subtree of left tree and right subtree
        of right tree have to be mirror images
    3 - right subtree of left tree and left subtree
        of right tree have to be mirror images
    """
    if (root1 is not None and root2 is not None):
        if  root1.key == root2.key:
            return (isMirror(root1.left, root2.right)and
                    isMirror(root1.right, root2.left))

    # If neither of above conditions is true then root1
    # and root2 are not mirror images
    return False

def isSymmetric(root):

    # Check if tree is mirror of itself
    return isMirror(root, root)

# Driver Program
# Let's construct the tree show in the above figure
root = Node(1)
root.left = Node(2)
root.right = Node(2)
root.left.left = Node(3)
root.left.right = Node(4)
root.right.left = Node(4)
root.right.right = Node(3)
print "1" if isSymmetric(root) == True else "0"

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

Output:

1

[Check for Symmetric Binary Tree \(Iterative Approach\)](#)

This article is contributed by **Muneer Ahmed**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

**Source**

<https://www.geeksforgeeks.org/symmetric-tree-tree-which-is-mirror-image-of-itself/>

## Chapter 418

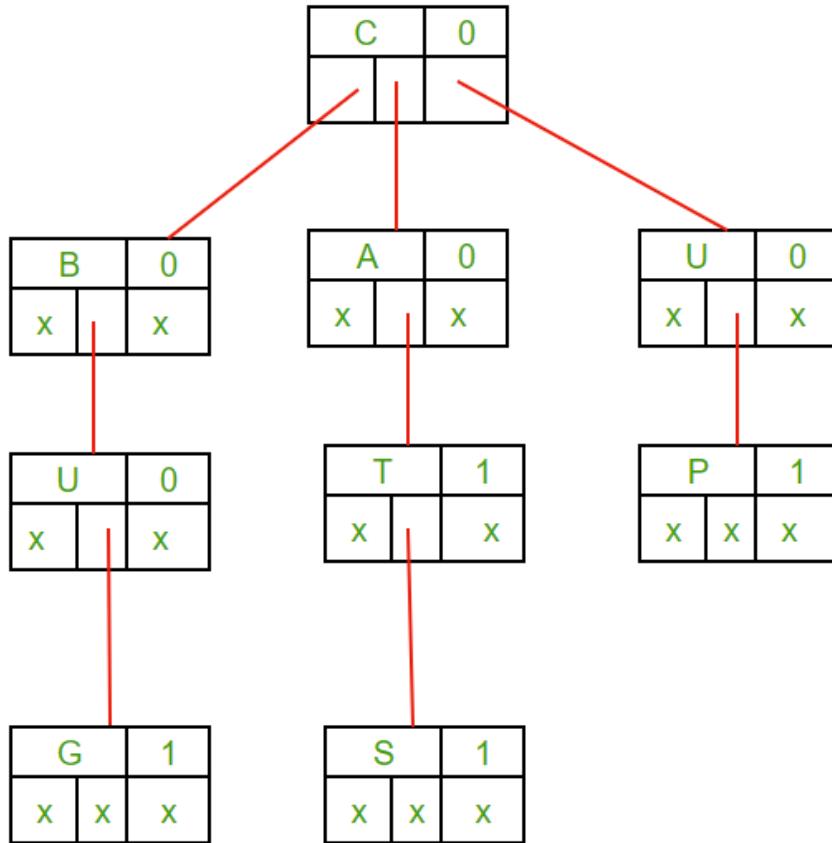
# Ternary Search Tree (Deletion)

Ternary Search Tree (Deletion) - GeeksforGeeks

In the [SET 1](#) post on TST we have described how to insert and search a node in TST. In this article we will discuss algorithm on how to delete a node from TST.

During delete operation we delete the key in bottom up manner using recursion. The following are possible cases when deleting a key from trie.

1. Key may not be there in TST.  
**Solution :** Delete operation should not modify TST.
2. Key present as unique key (no part of key contains another key (prefix), nor the key itself is prefix of another key in TST).  
**Solution :** Delete all the nodes.
3. Key is prefix key of another long key in TST.  
**Solution :** Unmark the leaf node.
4. Key present in TST, having atleast one other key as prefix key.  
**Solution :** Delete nodes from end of key until first leaf node of longest prefix key.



#### Explanation for delete\_node function

1. Let suppose we want to delete string “BIG”, since it is not present in TST so after matching with first character ‘B’, delete\_node function will return zero. Hence nothing is deleted.
2. Now we want to delete string “BUG”, it is Uniquely present in TST i.e neither it has part which is the prefix of other string nor it is prefix to any other string, so it will be deleted completely.
3. Now we want to delete string “CAT”, since it is prefix of string “CATS”, we cannot delete anything from the string “CAT” and we can only unmark the leaf node which will ensure that “CAT” is no longer a member string of TST.
4. Now we want to delete string “CATS”, since it has a prefix string “CAT” which also is a member string of TST so we can only delete last character of string “CATS” which will ensure that string “CAT” still remains the part of TST.

C

```
// C program to demonstrate deletion in
```

```
// Ternary Search Tree (TST). For insert
// and other functions, refer
// https://www.geeksforgeeks.org/ternary-search-tree/
#include<stdio.h>
#include<stdlib.h>

// structure of a node in TST
struct Node
{
    char key;
    int isleaf;
    struct Node *left;
    struct Node *eq;
    struct Node *right;
};

// function to create a Node in TST
struct Node *createNode(char key)
{
    struct Node *temp =
        (struct Node*)malloc(sizeof(struct Node));
    temp->key = key;
    temp->isleaf = 0;
    temp->left = NULL;
    temp->eq = NULL;
    temp->right = NULL;
    return temp;
};

// function to insert a Node in TST
void insert_node(struct Node **root ,char *s)
{
    if (!(*root))
        (*root) = createNode(*s);

    if ((*s)<(*root)->key)
        insert_node( &(*root)->left ,s);

    else if ((*s)>(*root)->key)
        insert_node( &(*root)->right ,s);

    else if ((*s) == (*root)->key)
    {
        if (*(s+1) == '\0')
        {
            (*root)->isleaf = 1;
            return;
        }
    }
}
```

```

        insert_node( &(*root)->eq ,s+1);
    }
}

// function to display the TST
void display(struct Node *root, char str[], int level)
{
    if (!root)
        return;

    display(root->left ,str ,level);
    str[level] = root->key;

    if (root->isleaf == 1)
    {
        str[level+1] = '\0';
        printf("%s\n",str);
    }

    display(root->eq ,str ,level+1);
    display(root->right ,str ,level);
}

// to check if current Node is leaf node or not
int isLeaf(struct Node *root)
{
    return root->isleaf == 1;
}

// to check if current node has any child node or not
int isFreeNode(struct Node *root)
{
    if (root->left ||root->eq ||root->right)
        return 0;
    return 1;
}

// function to delete a string in TST
int delete_node(struct Node *root, char str[],
                int level ,int n)
{
    if (root == NULL)
        return 0;

    // CASE 4 Key present in TST, having
    // atleast one other key as prefix key.
    if (str[level+1] == '\0')

```

```

{
    // Unmark leaf node if present
    if (isLeaf(root))
    {
        root->isleaf=0;
        return isFreeNode(root);
    }

    // else string is not present in TST and
    // return 0
    else
        return 0;
}
else
{
    // CASE 3 Key is prefix key of another
    // long key in TST.
    if (str[level] < root->key)
        delete_node(root->left ,str ,level ,n);
    else if (str[level] > root->key)
        delete_node(root->right ,str ,level ,n);

    // CASE 1 Key may not be there in TST.
    else if (str[level] == root->key)
    {
        // CASE 2 Key present as unique key
        if( delete_node(root->eq ,str ,level+1 ,n) )
        {
            // delete the last node, neither it
            // has any child
            // nor it is part of any other string
            free(root->eq);
            return !isLeaf(root) && isFreeNode(root);
        }
    }
}

return 0;
}

// Driver function
int main()
{
    struct Node *temp = NULL;

    insert_node(&temp , "CAT");
    insert_node(&temp , "BUGS");
    insert_node(&temp , "CATS");
}

```

```
insert_node(&temp , "UP");

int level = 0;
char str[20];
int p = 0;

printf( "1.Content of the TST before "
        "deletion:\n" );
display(temp ,str ,level);

level = 0;
delete_node(temp , "CAT" ,level ,5);

level = 0;
printf("\n2.Content of the TST after "
        "deletion:\n");
display(temp, str, level);
return 0;
}
```

C++

```
// C++ program to demonstrate deletion in
// Ternary Search Tree (TST)
// For insert and other functions, refer
// https://www.geeksforgeeks.org/ternary-search-tree

#include<iostream>
using namespace std;

// structure of a node in TST
struct Node
{
    char key;
    int isleaf;
    struct Node *left;
    struct Node *eq;
    struct Node *right;
};

// function to create a node in TST
struct Node *createNode(char key)
{
    struct Node *temp = new Node;
    temp->key = key;
    temp->isleaf = 0;
    temp->left = NULL;
    temp->eq = NULL;
```

```
temp->right = NULL;
return temp;
};

// function to insert a Node in TST
void insert_node(struct Node **root, char *s)
{
    if (!(*root))
    {
        (*root) = createNode(*s);
    }

    if ((*s)<(*root)->key)
        insert_node( &(*root)->left, s);

    else if ((*s)>(*root)->key)
        insert_node( &(*root)->right, s);

    else if ((*s) == (*root)->key)
    {
        if (*(s+1) == '\0')
        {
            (*root)->isleaf = 1;
            return;
        }
        insert_node( &(*root)->eq, s+1);
    }
}

// function to display the TST
void display(struct Node *root, char str[], int level)
{
    if (!root)
        return;

    display(root->left, str, level);
    str[level] = root->key;

    if (root->isleaf == 1)
    {
        str[level+1] = '\0';
        cout<< str << endl;
    }

    display(root->eq, str, level+1);
    display(root->right, str, level);
}
```

```

//to check if current node is leaf node or not
int isLeaf(struct Node *root)
{
    return root->isleaf == 1;
}

// to check if current node has any child
// node or not
int isFreeNode(struct Node *root)
{
    if (root->left || root->eq || root->right)
        return 0;
    return 1;
}

// function to delete a string in TST
int delete_node(struct Node *root, char str[],
                int level, int n)
{
    if (root == NULL)
        return 0;

    // CASE 4 Key present in TST, having atleast
    // one other key as prefix key.
    if (str[level+1] == '\0')
    {
        // Unmark leaf node if present
        if (isLeaf(root))
        {
            root->isleaf = 0;
            return isFreeNode(root);
        }
        // else string is not present in TST and
        // return 0
        else
            return 0;
    }

    // CASE 3 Key is prefix key of another long
    // key in TST.
    if (str[level] < root->key)
        return delete_node(root->left, str, level, n);
    if (str[level] > root->key)
        return delete_node(root->right, str, level, n);

    // CASE 1 Key may not be there in TST.

```

```
if (str[level] == root->key)
{
    // CASE 2 Key present as unique key
    if (delete_node(root->eq, str, level+1, n))
    {
        // delete the last node, neither it has
        // any child nor it is part of any other
        // string
        delete(root->eq);
        return !isLeaf(root) && isFreeNode(root);
    }
}

return 0;
}

// Driver function
int main()
{
    struct Node *temp = NULL;

    insert_node(&temp, "CAT");
    insert_node(&temp, "BUGS");
    insert_node(&temp, "CATS");
    insert_node(&temp, "UP");

    int level = 0;
    char str[20];
    int p = 0;

    cout << "1.Content of the TST before deletion:\n";
    display(temp, str, level);

    level = 0;
    delete_node(temp,"CAT", level, 5);

    level = 0;
    cout << "\n2.Content of the TST after deletion:\n";
    display(temp, str, level);
    return 0;
}
```

Output:

```
1.Content of the TST before deletion:
BUGS
CAT
```

CATS

UP

2.Content of the TST after deletion:

BUGS

CATS

UP

### **Source**

<https://www.geeksforgeeks.org/ternary-search-tree-deletion/>

## Chapter 419

# The Great Tree-List Recursion Problem.

The Great Tree-List Recursion Problem. - GeeksforGeeks

Asked by Varun Bhatia.

### Question:

Write a recursive function `treeToList(Node root)` that takes an ordered binary tree and rearranges the internal pointers to make a circular doubly linked list out of the tree nodes. The “previous” pointers should be stored in the “small” field and the “next” pointers should be stored in the “large” field. The list should be arranged so that the nodes are in increasing order. Return the head pointer to the new list.

This is very well explained and implemented at: [Convert a Binary Tree to a Circular Doubly Link List](#)

### References:

<http://cslibrary.stanford.edu/109/TreeListRecursion.html>

### Source

<https://www.geeksforgeeks.org/the-great-tree-list-recursion-problem/>

## Chapter 420

# Threaded Binary Tree

Threaded Binary Tree - GeeksforGeeks

Inorder traversal of a [Binary tree](#) can either be done using recursion or [with the use of a auxiliary stack](#). The idea of threaded binary trees is to make inorder traversal faster and do it without stack and without recursion. A binary tree is made threaded by making all right child pointers that would normally be NULL point to the inorder successor of the node (if it exists).

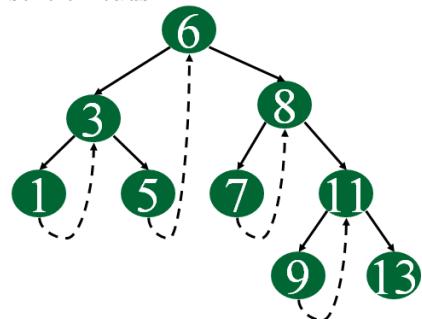
There are two types of threaded binary trees.

**Single Threaded:** Where a NULL right pointers is made to point to the inorder successor (if successor exists)

**Double Threaded:** Where both left and right NULL pointers are made to point to inorder predecessor and inorder successor respectively. The predecessor threads are useful for reverse inorder traversal and postorder traversal.

The threads are also useful for fast accessing ancestors of a node.

Following diagram shows an example Single Threaded Binary Tree. The dotted lines represent threads.



### C representation of a Threaded Node

Following is C representation of a single threaded node.

```
struct Node
```

```
{  
    int data;  
    Node *left, *right;  
    bool rightThread;  
}
```

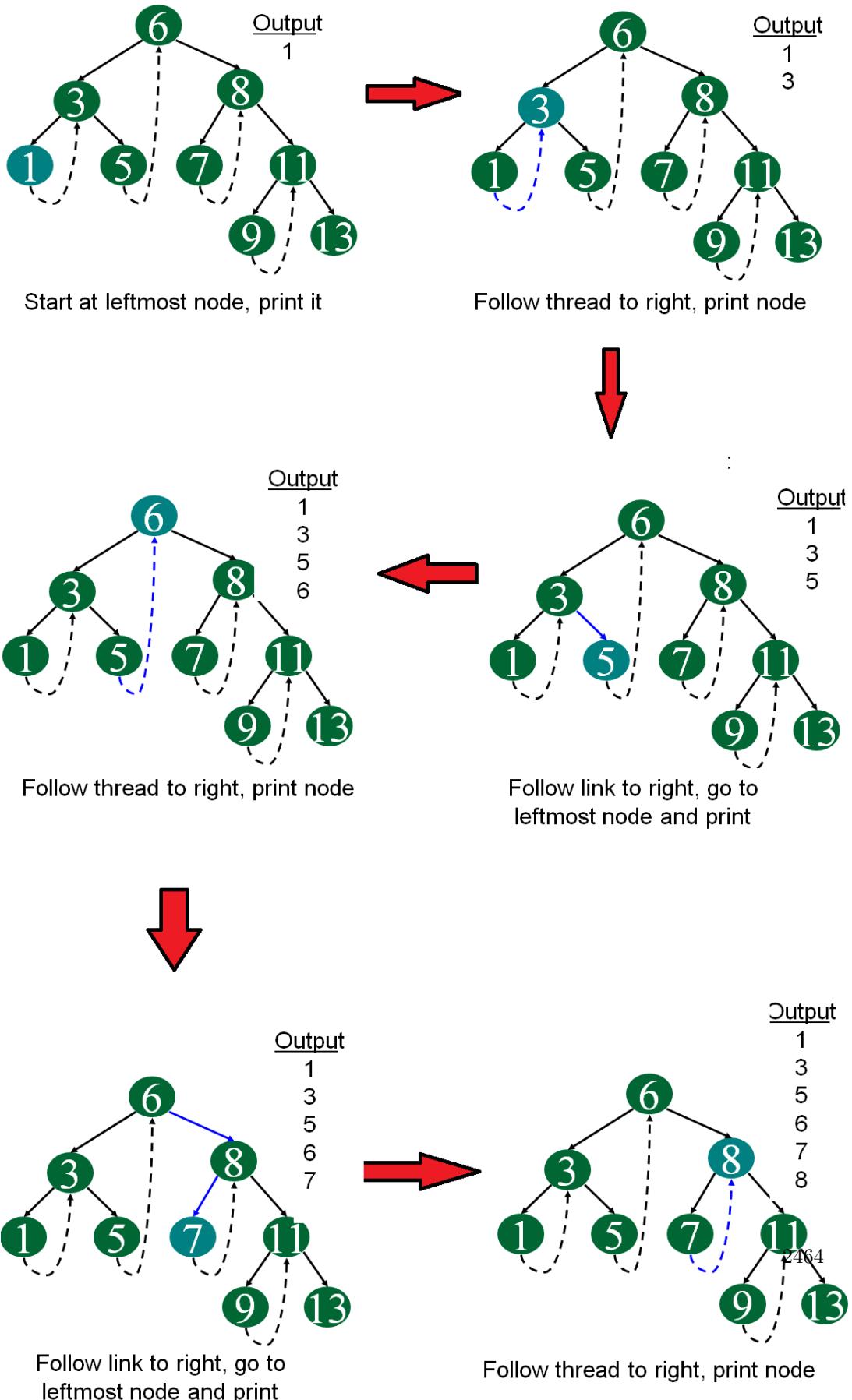
Since right pointer is used for two purposes, the boolean variable rightThread is used to indicate whether right pointer points to right child or inorder successor. Similarly, we can add leftThread for a double threaded binary tree.

### Inorder Traversal using Threads

Following is C code for inorder traversal in a threaded binary tree.

```
// Utility function to find leftmost node in a tree rooted with n  
struct Node* leftMost(struct Node *n)  
{  
    if (n == NULL)  
        return NULL;  
  
    while (n->left != NULL)  
        n = n->left;  
  
    return n;  
}  
  
// C code to do inorder traversal in a threaded binary tree  
void inOrder(struct Node *root)  
{  
    struct Node *cur = leftmost(root);  
    while (cur != NULL)  
    {  
        printf("%d ", cur->data);  
  
        // If this node is a thread node, then go to  
        // inorder successor  
        if (cur->rightThread)  
            cur = cur->rightThread;  
        else // Else go to the leftmost child in right subtree  
            cur = leftmost(cur->right);  
    }  
}
```

Following diagram demonstrates inorder order traversal using threads.



We will soon be discussing insertion and deletion in threaded binary trees.

**Sources:**

[http://en.wikipedia.org/wiki/Threaded\\_binary\\_tree](http://en.wikipedia.org/wiki/Threaded_binary_tree)  
[www.cs.berkeley.edu/~kamil/teaching/su02/080802.ppt](http://www.cs.berkeley.edu/~kamil/teaching/su02/080802.ppt)

**Source**

<https://www.geeksforgeeks.org/threaded-binary-tree/>

## Chapter 421

# Threaded Binary Tree | Insertion

Threaded Binary Tree | Insertion - GeeksforGeeks

We have already discuss the [Binary Threaded Binary Tree](#).

Insertion in Binary threaded tree is similar to insertion in binary tree but we will have to adjust the threads after insertion of each element.

C representation of Binary Threaded Node:

```
struct Node
{
    struct Node *left, *right;
    int info;

    // True if left pointer points to predecessor
    // in Inorder Traversal
    boolean lthread;

    // True if right pointer points to successor
    // in Inorder Traversal
    boolean rthread;
};
```

In the following explanation, we have considered [Binary Search Tree \(BST\)](#) for insertion as insertion is defined by some rules in BSTs.

Let **tmp** be the newly inserted node. There can be three cases during insertion:

**Case 1: Insertion in empty tree**

Both left and right pointers of tmp will be set to NULL and new node becomes the root.

```
root = tmp;
tmp -> left = NULL;
tmp -> right = NULL;
```

### Case 2: When new node inserted as the left child

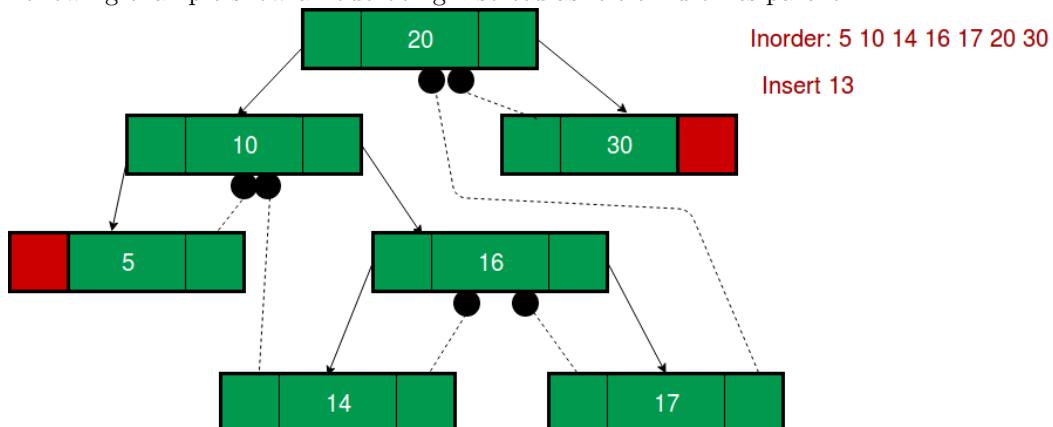
After inserting the node at its proper place we have to make its left and right threads points to inorder predecessor and successor respectively. The node which was [inorder successor](#). So the left and right threads of the new node will be-

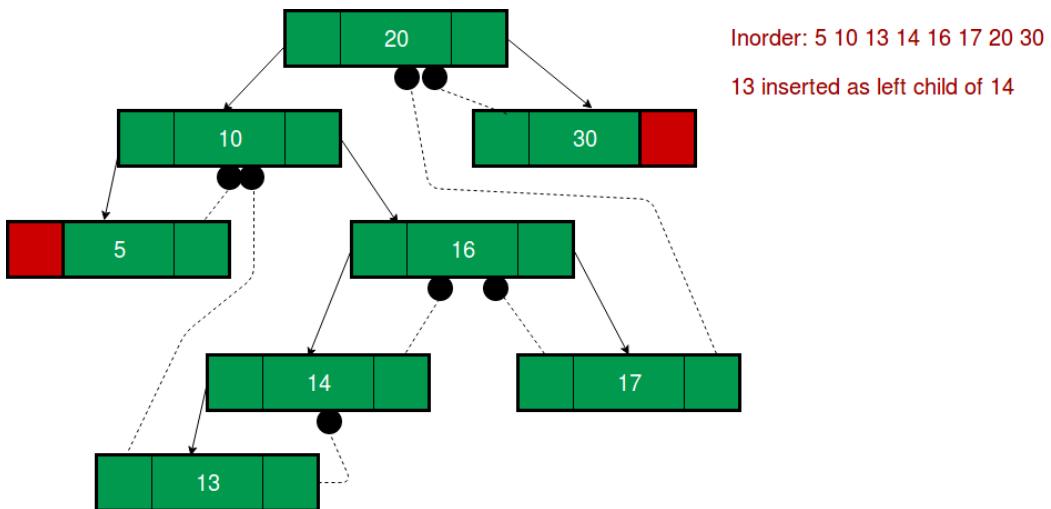
```
tmp -> left = par ->left;
tmp -> right = par;
```

Before insertion, the left pointer of parent was a thread, but after insertion it will be a link pointing to the new node.

```
par -> lthread = false;
par -> left = temp;
```

Following example show a node being inserted as left child of its parent.





Predecessor of 14 becomes the predecessor of 13, so left thread of 13 points to 10.  
Successor of 13 is 14, so right thread of 13 points to left child which is 13.  
Left pointer of 14 is not a thread now, it points to left child which is 13.

### Case 3: When new node is inserted as the right child

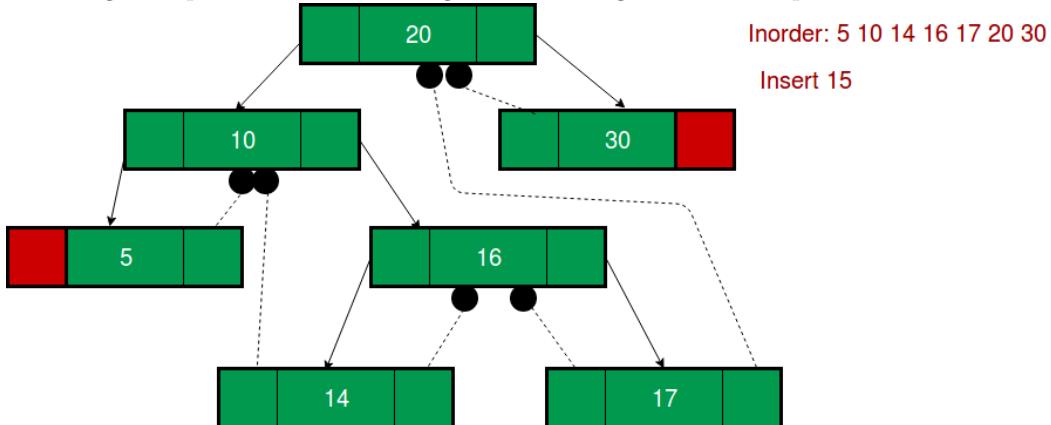
The parent of tmp is its inorder predecessor. The node which was inorder successor of the parent is now the inorder successor of this node tmp. So the left and right threads of the new node will be-

```
tmp -> left = par;
tmp -> right = par -> right;
```

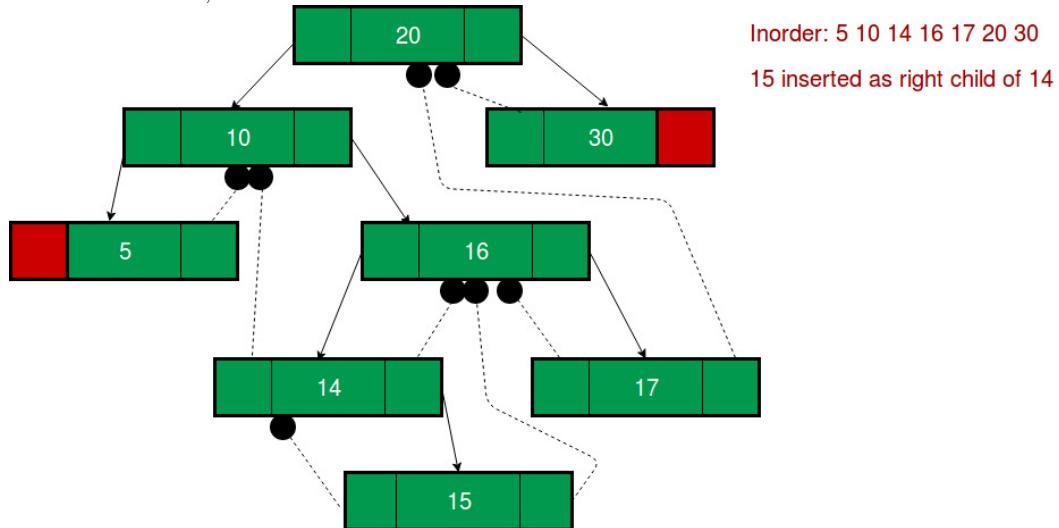
Before insertion, the right pointer of parent was a thread, but after insertion it will be a link pointing to the new node.

```
par -> rthread = false;
par -> right = tmp;
```

Following example shows a node being inserted as right child of its parent.



After 15 inserted,



Successor of 14 becomes the successor of 15, so right thread of 15 points to 16.  
Predecessor of 15 is 14, so left thread of 15 points to 14.

Right pointer of 14 is not a thread now, it points to right child which is 15.

**C++ implementation to insert a new node in Threaded Binary Search Tree:**  
Like standard BST insert, we search for the key value in the tree. If key is already present, then we return otherwise the new key is inserted at the point where search terminates. In BST, search terminates either when we find the key or when we reach a NULL left or right pointer. Here all left and right NULL pointers are replaced by threads except left pointer of first node and right pointer of last node. So here search will be unsuccessful when we reach a NULL pointer or a thread.

```
// Insertion in Threaded Binary Search Tree.
#include<bits/stdc++.h>
using namespace std;

struct Node
{
    struct Node *left, *right;
    int info;

    // True if left pointer points to predecessor
    // in Inorder Traversal
    bool lthread;

    // True if right pointer points to predecessor
    // in Inorder Traversal
    bool rthread;
};

// Insert a Node in Binary Threaded Tree
```

```

struct Node *insert(struct Node *root, int ikey)
{
    // Searching for a Node with given value
    Node *ptr = root;
    Node *par = NULL; // Parent of key to be inserted
    while (ptr != NULL)
    {
        // If key already exists, return
        if (ikey == (ptr->info))
        {
            printf("Duplicate Key !\n");
            return root;
        }

        par = ptr; // Update parent pointer

        // Moving on left subtree.
        if (ikey < ptr->info)
        {
            if (ptr -> lthread == false)
                ptr = ptr -> left;
            else
                break;
        }

        // Moving on right subtree.
        else
        {
            if (ptr->rthread == false)
                ptr = ptr -> right;
            else
                break;
        }
    }

    // Create a new node
    Node *tmp = new Node;
    tmp -> info = ikey;
    tmp -> lthread = true;
    tmp -> rthread = true;

    if (par == NULL)
    {
        root = tmp;
        tmp -> left = NULL;
        tmp -> right = NULL;
    }
    else if (ikey < (par -> info))

```

```
{  
    tmp -> left = par -> left;  
    tmp -> right = par;  
    par -> lthread = false;  
    par -> left = tmp;  
}  
else  
{  
    tmp -> left = par;  
    tmp -> right = par -> right;  
    par -> rthread = false;  
    par -> right = tmp;  
}  
  
return root;  
}  
  
// Returns inorder successor using rthread  
struct Node *inorderSuccessor(struct Node *ptr)  
{  
    // If rthread is set, we can quickly find  
    if (ptr -> rthread == true)  
        return ptr->right;  
  
    // Else return leftmost child of right subtree  
    ptr = ptr -> right;  
    while (ptr -> lthread == false)  
        ptr = ptr -> left;  
    return ptr;  
}  
  
// Printing the threaded tree  
void inorder(struct Node *root)  
{  
    if (root == NULL)  
        printf("Tree is empty");  
  
    // Reach leftmost node  
    struct Node *ptr = root;  
    while (ptr -> lthread == false)  
        ptr = ptr -> left;  
  
    // One by one print successors  
    while (ptr != NULL)  
    {  
        printf("%d ",ptr -> info);  
        ptr = inorderSuccessor(ptr);  
    }  
}
```

```
}
```

```
// Driver Program
int main()
{
    struct Node *root = NULL;

    root = insert(root, 20);
    root = insert(root, 10);
    root = insert(root, 30);
    root = insert(root, 5);
    root = insert(root, 16);
    root = insert(root, 14);
    root = insert(root, 17);
    root = insert(root, 13);

    inorder(root);

    return 0;
}
```

Output:

```
5 10 13 14 16 17 20 30
```

## Source

<https://www.geeksforgeeks.org/threaded-binary-tree-insertion/>

## Chapter 422

# Tilt of Binary Tree

Tilt of Binary Tree - GeeksforGeeks

Given a binary tree, return the tilt of the whole tree. The tilt of a tree node is defined as the absolute difference between the sum of all left subtree node values and the sum of all right subtree node values. Null nodes are assigned tilt to be zero. Therefore, tilt of the whole tree is defined as the sum of all nodes' tilt.

Examples:

Input :

```
    1
   / \
  2   3
```

Output : 1

Explanation:

Tilt of node 2 : 0

Tilt of node 3 : 0

Tilt of node 1 :  $|2-3| = 1$

Tilt of binary tree :  $0 + 0 + 1 = 1$

Input :

```
    4
   / \
  2   9
 / \   \
3   5   7
```

Output : 15

Explanation:

Tilt of node 3 : 0

Tilt of node 5 : 0

Tilt of node 7 : 0

Tilt of node 2 :  $|3-5| = 2$

```
Tilt of node 9 : |0-7| = 7
Tilt of node 4 : |(3+5+2)-(9+7)| = 6
Tilt of binary tree : 0 + 0 + 0 + 2 + 7 + 6 = 15
```

The idea is to recursively traverse tree. While traversing, we keep track of two things, sum of subtree rooted under current node, tilt of current node. Sum is needed to compute tilt of parent.

```
// CPP Program to find Tilt of Binary Tree
#include <bits/stdc++.h>
using namespace std;

/* A binary tree node has data, pointer to
left child and a pointer to right child */
struct Node {
    int val;
    struct Node *left, *right;
};

/* Recursive function to calculate Tilt of
whole tree */
int traverse(Node* root, int* tilt)
{
    if (!root)
        return 0;

    // Compute tilts of left and right subtrees
    // and find sums of left and right subtrees
    int left = traverse(root->left, tilt);
    int right = traverse(root->right, tilt);

    // Add current tilt to overall
    *tilt += abs(left - right);

    // Returns sum of nodes under current tree
    return left + right + root->val;
}

/* Driver function to print Tilt of whole tree */
int Tilt(Node* root)
{
    int tilt = 0;
    traverse(root, &tilt);
    return tilt;
}

/* Helper function that allocates a
new node with the given data and
```

```
NULL left and right pointers. */
Node* newNode(int data)
{
    Node* temp = new Node;
    temp->val = data;
    temp->left = temp->right = NULL;
    return temp;
}

// Driver code
int main()
{
    /* Let us construct a Binary Tree
        4
       / \
      2   9
     / \   \
    3   5   7 */

    Node* root = NULL;
    root = newNode(4);
    root->left = newNode(2);
    root->right = newNode(9);
    root->left->left = newNode(3);
    root->left->right = newNode(8);
    root->right->right = newNode(7);
    cout << "The Tilt of whole tree is " << Tilt(root);
    return 0;
}
```

Output:

The Tilt of whole tree is 15

#### Complexity Analysis:

- Time complexity :  $O(n)$ , where  $n$  is the number of nodes in binary tree.
- Auxiliary Space :  $O(n)$  as in worst case, depth of binary tree will be  $n$ .

#### Source

<https://www.geeksforgeeks.org/tilt-binary-tree/>

## Chapter 423

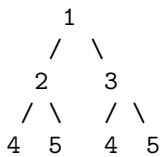
# Top three elements in binary tree

Top three elements in binary tree - GeeksforGeeks

We have a simple binary tree and we have to print the top 3 largest elements present in the binary tree.

Examples:

Input :



Output :Three largest elements are 5 4 3

**Approach** We can simply take three variables first, second, third to store the first largest, second largest, third largest respectively and perform preorder traversal and each time we will update the elements accordingly.

This approach will take  $O(n)$  time only.

Algorithm-

```
1- Take 3 variables first, second, third
2- Perform a preorder traversal
   if (root==NULL)
       return
   if root's data is larger then first
       update third with second
```

```
        second with first
        first with root's data
    else if root's data is larger then
        second and not equal to first
            update third with second
            second with root's data
    else if root's data is larger then
        third and not equal to first &
        second
            update third with root's data
3- call preorder for root->left
4- call preorder for root->right

// CPP program to find largest three elements in
// a binary tree.
#include <bits/stdc++.h>
using namespace std;

struct Node {
    int data;
    struct Node *left;
    struct Node *right;
};

/* Helper function that allocates a new Node with the
   given data and NULL left and right pointers. */
struct Node *newNode(int data) {
    struct Node *node = new Node;
    node->data = data;
    node->left = NULL;
    node->right = NULL;
    return (node);
}

// function to find three largest element
void threelargest(Node *root, int &first, int &second,
                  int &third) {
    if (root == NULL)
        return;

    // if data is greater than first large number
    // update the top three list
    if (root->data > first) {
        third = second;
        second = first;
        first = root->data;
    }
}
```

```
// if data is greater than second large number
// and not equal to first update the bottom
// two list
else if (root->data > second && root->data != first) {
    third = second;
    second = root->data;
}

// if data is greater than third large number
// and not equal to first & second update the
// third highest list
else if (root->data > third &&
         root->data != first &&
         root->data != second)
    third = root->data;

threelargest(root->left, first, second, third);
threelargest(root->right, first, second, third);
}

// driver function
int main() {
    struct Node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->left = newNode(4);
    root->right->right = newNode(5);

    int first = 0, second = 0, third = 0;
    threelargest(root, first, second, third);
    cout << "three largest elements are "
        << first << " " << second << " "
        << third;
    return 0;
}
```

**Output:**

```
three largest elements are 5 4 3
```

**Source**

<https://www.geeksforgeeks.org/top-three-elements-binary-tree/>

## Chapter 424

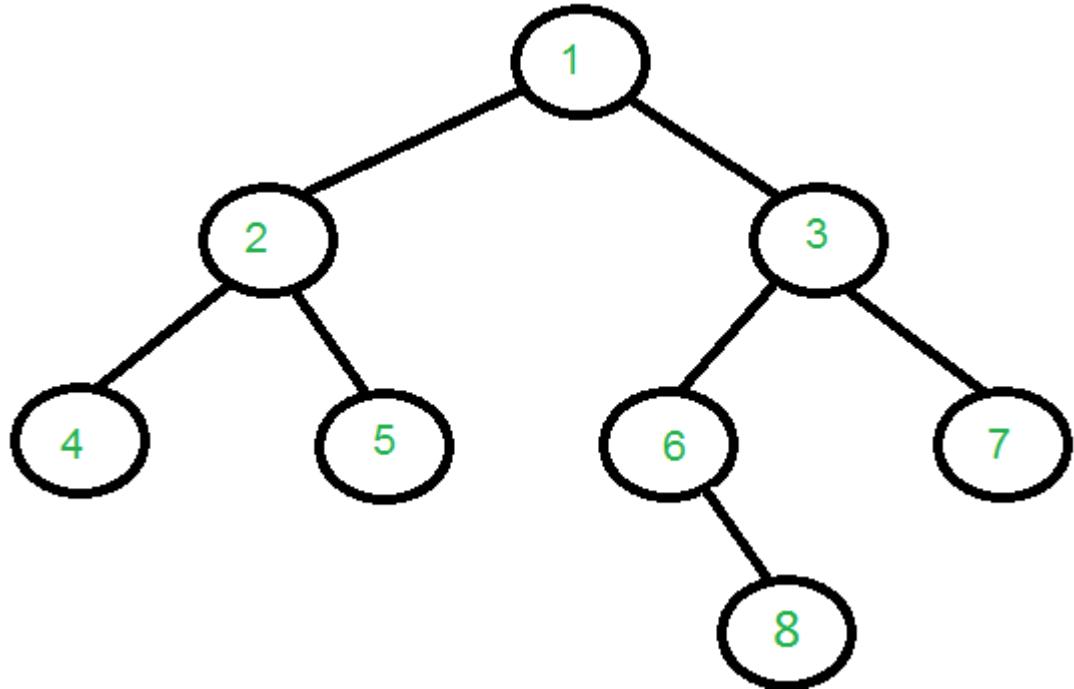
# Total nodes traversed in Euler Tour Tree

Total nodes traversed in Euler Tour Tree - GeeksforGeeks

*Euler tour of tree* has been already discussed which flattens the hierarchical structure of tree into array which contains exactly  $2*N-1$  values. In this post, the task is to prove that the degree of Euler Tour Tree is 2 times the number of nodes minus one. Here degree means the total number of nodes get traversed in Euler Tour.

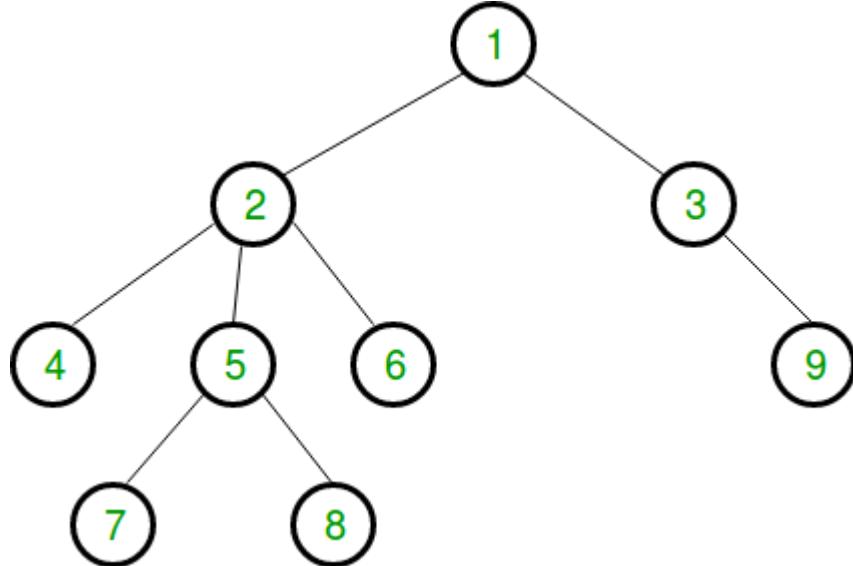
Examples:

Input:



Output: 15

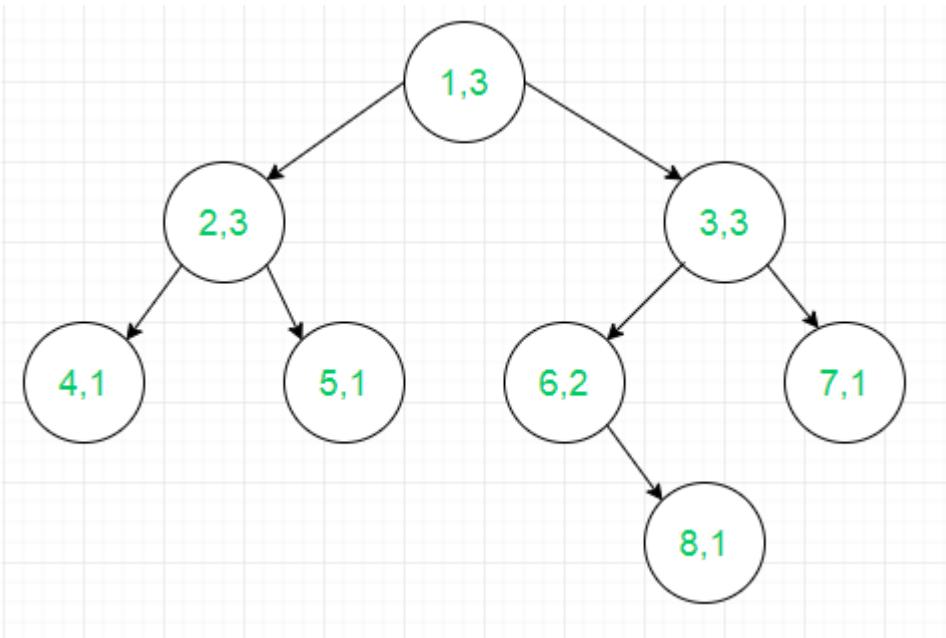
Input:



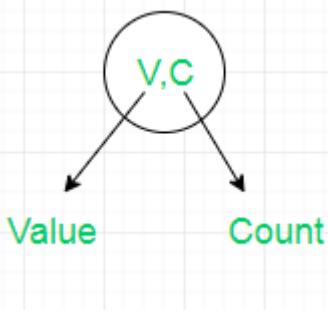
Output: 17

Explanation:

Using Example 1:



where



It can be seen that each node's count in Euler Tour is exactly equal to the out-degree of node plus 1.

Mathematically, it can be represented as:

$$\text{outdegree}_v = \text{count}_v + 1$$

$$\text{outdegree}_v = \text{count}_v + 1$$

Where

**Total** represents total number of nodes in Euler Tour Tree

**node<sub>i</sub>** represents ith node in given Tree

N represents the total number of node in given Tree

**size[adj[i]]** represents number of child of **node<sub>i</sub>**

```
// C++ program to check the number of nodes
// in Euler Tour tree.
#include <bits/stdc++.h>
using namespace std;

#define MAX 1001

// Adjacency list representation of tree
vector<int> adj[MAX];

// Function to add edges to tree
void add_edge(int u, int v)
{
    adj[u].push_back(v);
}

// Program to check if calculated Value is
// equal to 2*size-1
void checkTotalNumberofNodes(int actualAnswer,
                             int size)
{
    int calculatedAnswer = size;

    // Add out-degree of each node
    for (int i = 1; i <= size; i++)
        calculatedAnswer += adj[i].size();

    if (actualAnswer == calculatedAnswer)
        cout << "Calculated Answer is " << calculatedAnswer
            << " and is Equal to Actual Answer\n";
    else
        cout << "Calculated Answer is Incorrect\n";
}

int main()
{ // Constructing 1st tree from example
    int N = 8;
    add_edge(1, 2);
    add_edge(1, 3);
    add_edge(2, 4);
```

```
    add_edge(2, 5);
    add_edge(3, 6);
    add_edge(3, 7);
    add_edge(6, 8);

    // Out_deg[node[i]] is equal to adj[i].size()
    checkTotalNumberofNodes(2 * N - 1, N);

    // clear previous stored tree
    for (int i = 1; i <= N; i++)
        adj[i].clear();

    // Constructing 2nd tree from example
    N = 9;
    add_edge(1, 2);
    add_edge(1, 3);
    add_edge(2, 4);
    add_edge(2, 5);
    add_edge(2, 6);
    add_edge(3, 9);
    add_edge(5, 7);
    add_edge(5, 8);

    // Out_deg[node[i]] is equal to adj[i].size()
    checkTotalNumberofNodes(2 * N - 1, N);

    return 0;
}
```

Output:

```
Calculated Answer is 15 and is Equal to Actual Answer
Calculated Answer is 17 and is Equal to Actual Answer
```

## Source

<https://www.geeksforgeeks.org/total-nodes-traversed-in-euler-tour-tree/>

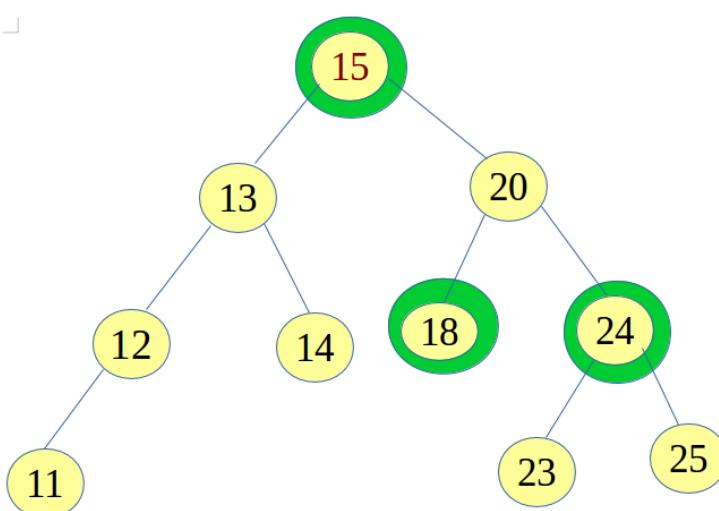
## Chapter 425

# Total sum except adjacent of a given node in a Binary Tree

Total sum except adjacent of a given node in a Binary Tree - GeeksforGeeks

Given a BT and a key Node, find the total sum in BT, except those Node which are adjacent to key Node.

Examples:



Assume that key node is 20, here big circle shows that they are adjacent to key node so except these node return the total sum.

Maximu sum without adding the adjecent element of a key node is 118.

1. Traverse the tree using pre-order.
2. If current node is adjacent to the key then do not add it to the final sum.

3. If current node is the key then do not add it's children to the final sum.
4. If key is not present then return sum of all nodes.

C++

```
// C++ program to find total sum except a given Node in BT
#include <bits/stdc++.h>
using namespace std;

struct Node {
    int data;
    struct Node *left, *right;
};

// insertion of Node in Tree
Node* getNode(int n)
{
    struct Node* root = new Node;
    root->data = n;
    root->left = NULL;
    root->right = NULL;
    return root;
}

// sum of all element except those which are adjacent to key Node
void find_sum(Node* root, int key, int& sum, bool incl)
{
    if (root) {
        if (incl) {
            sum += root->data;

            if (root->left && root->left->data == key) {
                sum -= root->data;
            }
            else if (root->right && root->right->data == key) {
                sum -= root->data;
            }
        }

        incl = root->data == key ? false : true;
        find_sum(root->left, key, sum, incl);
        find_sum(root->right, key, sum, incl);
    }
}

// Driver code
int main()
{
    struct Node* root = getNode(15);
```

```
root->left = getNode(13);
root->left->left = getNode(12);
root->left->left->left = getNode(11);
root->left->right = getNode(14);
root->right = getNode(20);
root->right->left = getNode(18);
root->right->right = getNode(24);
root->right->right->left = getNode(23);
root->right->right->right = getNode(25);
int key = 20;
int sum = 0;
find_sum(root, key, sum, true);
printf("%d ", sum);
return 0;
}
```

### Java

```
// Java program to find total sum except a given Node in BT
import java.util.*;
class Node
{
    int data;
    Node left, right;

    // insertion of Node in Tree
    Node(int key)
    {
        data = key;
        left = right = null;
    }
}
class GFG
{
    static class cal
    {
        int sum = 0;
    }

    // sum of all element except those which are adjacent to key Node
    public static void find_sum(Node root, int key, cal r, boolean incl)
    {
        if(root != null)
        {
            if(incl == true)
            {
                r.sum += root.data;
                if((root.left != null) && (root.left.data == key))

```

```
        {
            r.sum -= root.data;
        }
        else
            if((root.right != null) && (root.right.data == key))
            {
                r.sum -= root.data;
            }

    }

    if(root.data == key)
        incl = false;
    else
        incl = true;

    find_sum(root.left, key, r, incl);
    find_sum(root.right, key, r, incl);
}
}

// Driver code
public static void main (String[] args)
{
    Node root = new Node(15);
    root.left = new Node(13);
    root.left.left = new Node(12);
    root.left.left.left = new Node(11);
    root.left.right = new Node(14);
    root.right = new Node(20);
    root.right.left = new Node(18);
    root.right.right = new Node(24);
    root.right.right.right = new Node(25);
    root.right.right.left = new Node(23);
    int key = 20;
    cal obj = new cal();
    find_sum(root, key, obj, true);
    System.out.print(obj.sum);
}
```

}

**Output:**

118

Time Complexity : O(n) where n is number of nodes in the BT.

**Source**

<https://www.geeksforgeeks.org/total-sum-except-adjacent-of-a-given-node-in-a-binary-tree/>

## Chapter 426

# Traversal of tree with k jumps allowed between nodes of same height

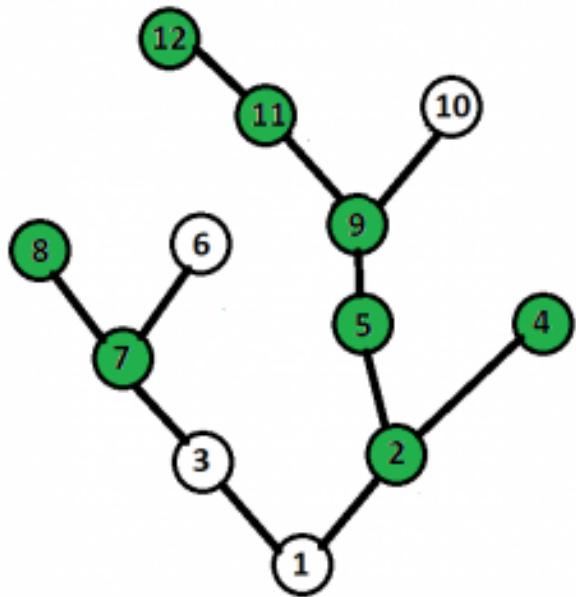
Traversal of tree with k jumps allowed between nodes of same height - GeeksforGeeks

Consider a tree with n nodes and root. You can jump from one node to any other node on the same height a maximum of k times on total jumps. Certain nodes of the tree contain a fruit, find out the maximum number of fruits he can collect.

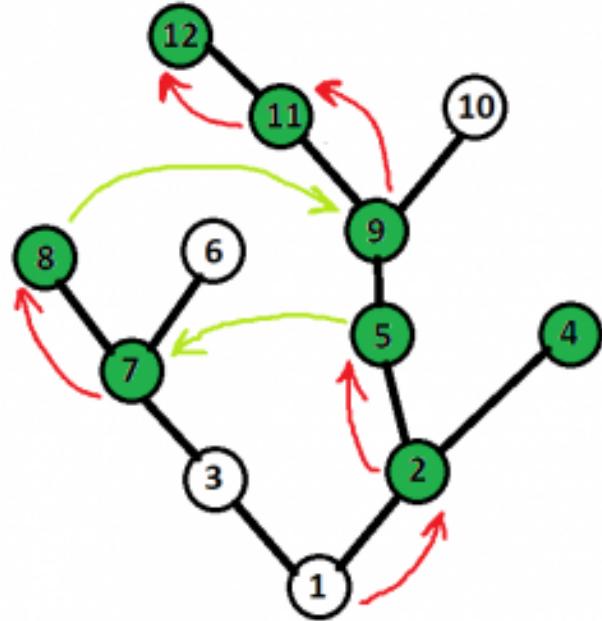
Example :

```
Input Tree :  
Number of Nodes = 12  
Number of jumps allowed : 2  
Edges:  
1 2  
1 3  
2 4  
2 5  
5 9  
9 10  
9 11  
11 12  
1 3  
3 7  
7 6  
7 8  
Nodes Containing Fruits : 2 4 5 7 8 9 11 12  
Output: 7
```

Tree for above testcase :



Explanation:



**Approach:** The idea is to use [DFS](#) to create a Height Adjacency List of the Nodes and to store the parents. Then use another dfs to compute the maximum no of special nodes that can be reached using the following dp state:

```
dp[current_node][j] = max( max{ dp[child_i][j], for all children of current_node },
                           max{ dp[node_at_same_height_i][j - 1],
                                 for all nodes at same height as current_node} )
```

Thus,  $\text{dp}[\text{Root\_Node}][\text{Total\_no\_of\_Jumps}]$  gives the answer to the problem.

Below is the implementation of above approach :

```
// Program to demonstrate tree traversal with
// ability to jump between nodes of same height
#include <bits/stdc++.h>
using namespace std;

#define N 1000

vector<int> H[N];

// Arrays declaration
```

```
int Fruit[N];
int Parent[N];
int dp[N][20];

// Function for DFS
void dfs1(vector<int> tree[], int s,
          int p, int h)
{
    Parent[s] = p;
    int i;
    H[h].push_back(s);
    for (i = 0; i < tree[s].size(); i++) {
        int v = tree[s][i];
        if (v != p)
            dfs1(tree, v, s, h + 1);
    }
}

// Function for DFS
int dfs2(vector<int> tree[], int s,
          int p, int h, int j)
{
    int i;
    int ans = 0;
    if (dp[s][j] != -1)
        return dp[s][j];

    // jump
    if (j > 0) {
        for (i = 0; i < H[h].size(); i++) {
            int v = H[h][i];
            if (v != s)
                ans = max(ans, dfs2(tree, v,
                                      Parent[v], h, j - 1));
        }
    }

    // climb
    for (i = 0; i < tree[s].size(); i++) {
        int v = tree[s][i];
        if (v != p)
            ans = max(ans, dfs2(tree, v, s, h + 1, j));
    }

    if (Fruit[s] == 1)
        ans++;
    dp[s][j] = ans;
}
```

```
    return ans;
}

// Function to calculate and
// return maximum number of fruits
int maxFruit(vector<int> tree[],
              int NodesWithFruits[],
              int n, int m, int k)
{
    // resetting dp table and Fruit array
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < 20; j++)
            dp[i][j] = -1;
        Fruit[i] = 0;
    }

    // This array is used to mark
    // which nodes contain Fruits
    for (int i = 0; i < m; i++)
        Fruit[NodesWithFruits[i]] = 1;

    dfs1(tree, 1, 0, 0);
    int ans = dfs2(tree, 1, 0, 0, k);

    return ans;
}

// Function to add Edge
void addEdge(vector<int> tree[], int u, int v)
{
    tree[u].push_back(v);
    tree[v].push_back(u);
}

// Driver Code
int main()
{
    int n = 12; // Number of nodes
    int k = 2; // Number of allowed jumps

    vector<int> tree[N];

    // Edges
    addEdge(tree, 1, 2);
    addEdge(tree, 1, 3);
    addEdge(tree, 2, 4);
    addEdge(tree, 2, 5);
    addEdge(tree, 5, 9);
```

```
addEdge(tree, 9, 10);
addEdge(tree, 9, 11);
addEdge(tree, 11, 12);
addEdge(tree, 1, 3);
addEdge(tree, 3, 7);
addEdge(tree, 7, 6);
addEdge(tree, 7, 8);

int NodesWithFruits[] = { 2, 4, 5, 7, 8, 9, 11, 12 };

// Number of nodes with fruits
int m = sizeof(NodesWithFruits) / sizeof(NodesWithFruits[0]);

int ans = maxFruit(tree, NodesWithFruits, n, m, k);

cout << ans << endl;

return 0;
}
```

**Output:**

7

**Time Complexity :**  $O(n \cdot n \cdot k)$  (worst case, eg: 2 level tree with the root having  $n-1$  child nodes)

**Source**

<https://www.geeksforgeeks.org/traversal-tree-ability-jump-nodes-height/>

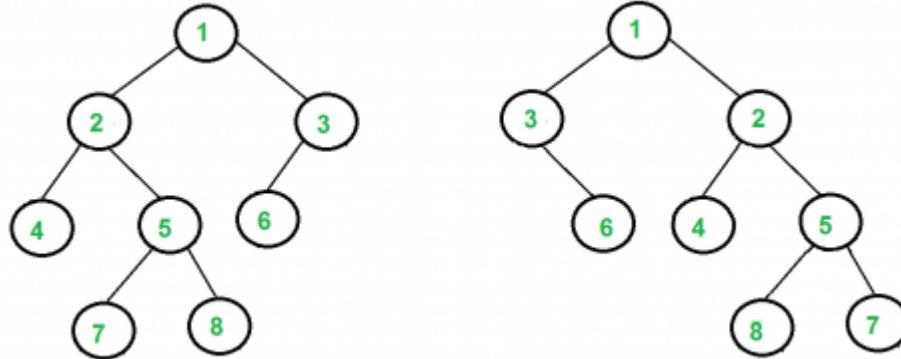
## Chapter 427

# Tree Isomorphism Problem

Tree Isomorphism Problem - GeeksforGeeks

Write a function to detect if two trees are isomorphic. Two trees are called isomorphic if one of them can be obtained from other by a series of flips, i.e. by swapping left and right children of a number of nodes. Any number of nodes at any level can have their children swapped. Two empty trees are isomorphic.

For example, following two trees are isomorphic with following sub-trees flipped: 2 and 3, NULL and 6, 7 and 8.



We simultaneously traverse both trees. Let the current internal nodes of two trees being traversed be **n1** and **n2** respectively. There are following two conditions for subtrees rooted with n1 and n2 to be isomorphic.

- 1) Data of n1 and n2 is same.
- 2) One of the following two is true for children of n1 and n2
  - .....a) Left child of n1 is isomorphic to left child of n2 and right child of n1 is isomorphic to right child of n2.
  - .....b) Left child of n1 is isomorphic to right child of n2 and right child of n1 is isomorphic to left child of n2.

C++

```

// A C++ program to check if two given trees are isomorphic
#include <iostream>
using namespace std;

/* A binary tree node has data, pointer to left and right children */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

/* Given a binary tree, print its nodes in reverse level order */
bool isIsomorphic(node* n1, node *n2)
{
    // Both roots are NULL, trees isomorphic by definition
    if (n1 == NULL && n2 == NULL)
        return true;

    // Exactly one of the n1 and n2 is NULL, trees not isomorphic
    if (n1 == NULL || n2 == NULL)
        return false;

    if (n1->data != n2->data)
        return false;

    // There are two possible cases for n1 and n2 to be isomorphic
    // Case 1: The subtrees rooted at these nodes have NOT been "Flipped".
    // Both of these subtrees have to be isomorphic, hence the &&
    // Case 2: The subtrees rooted at these nodes have been "Flipped"
    return
        (isIsomorphic(n1->left,n2->left) && isIsomorphic(n1->right,n2->right)) ||
        (isIsomorphic(n1->left,n2->right) && isIsomorphic(n1->right,n2->left));
}

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
node* newNode(int data)
{
    node* temp = new node;
    temp->data = data;
    temp->left = NULL;
    temp->right = NULL;

    return (temp);
}

/* Driver program to test above functions*/

```

```
int main()
{
    // Let us create trees shown in above diagram
    struct node *n1 = newNode(1);
    n1->left      = newNode(2);
    n1->right     = newNode(3);
    n1->left->left  = newNode(4);
    n1->left->right = newNode(5);
    n1->right->left  = newNode(6);
    n1->left->right->left = newNode(7);
    n1->left->right->right = newNode(8);

    struct node *n2 = newNode(1);
    n2->left      = newNode(3);
    n2->right     = newNode(2);
    n2->right->left  = newNode(4);
    n2->right->right = newNode(5);
    n2->left->right = newNode(6);
    n2->right->right->left = newNode(8);
    n2->right->right->right = newNode(7);

    if (isIsomorphic(n1, n2) == true)
        cout << "Yes";
    else
        cout << "No";
}
```

### Java

```
// An iterative java program to solve tree isomorphism problem

/* A binary tree node has data, pointer to left and right children */
class Node
{
    int data;
    Node left, right;

    Node(int item)
    {
        data = item;
        left = right;
    }
}

class BinaryTree
{
```

```
Node root1, root2;

/* Given a binary tree, print its nodes in reverse level order */
boolean isIsomorphic(Node n1, Node n2)
{
    // Both roots are NULL, trees isomorphic by definition
    if (n1 == null && n2 == null)
        return true;

    // Exactly one of the n1 and n2 is NULL, trees not isomorphic
    if (n1 == null || n2 == null)
        return false;

    if (n1.data != n2.data)
        return false;

    // There are two possible cases for n1 and n2 to be isomorphic
    // Case 1: The subtrees rooted at these nodes have NOT been
    // "Flipped".
    // Both of these subtrees have to be isomorphic.
    // Case 2: The subtrees rooted at these nodes have been "Flipped"
    return (isIsomorphic(n1.left, n2.left) &&
            isIsomorphic(n1.right, n2.right))
        || (isIsomorphic(n1.left, n2.right) &&
            isIsomorphic(n1.right, n2.left));
}

// Driver program to test above functions
public static void main(String args[])
{
    BinaryTree tree = new BinaryTree();

    // Let us create trees shown in above diagram
    tree.root1 = new Node(1);
    tree.root1.left = new Node(2);
    tree.root1.right = new Node(3);
    tree.root1.left.left = new Node(4);
    tree.root1.left.right = new Node(5);
    tree.root1.right.left = new Node(6);
    tree.root1.left.right.left = new Node(7);
    tree.root1.left.right.right = new Node(8);

    tree.root2 = new Node(1);
    tree.root2.left = new Node(3);
    tree.root2.right = new Node(2);
    tree.root2.right.left = new Node(4);
    tree.root2.right.right = new Node(5);
    tree.root2.left.right = new Node(6);
```

```
tree.root2.right.right.left = new Node(8);
tree.root2.right.right.right = new Node(7);

if (tree.isIsomorphic(tree.root1, tree.root2) == true)
    System.out.println("Yes");
else
    System.out.println("No");
}

}

// This code has been contributed by Mayank Jaiswal
```

### Python

```
# Python program to check if two given trees are isomorphic

# A Binary tree node
class Node:
    # Constructor to create the node of binary tree
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

# Check if the binary tree is isomorphic or not
def isIsomorphic(n1, n2):

    # Both roots are None, trees isomorphic by definition
    if n1 is None and n2 is None:
        return True

    # Exactly one of the n1 and n2 is None, trees are not
    # isomorphic
    if n1 is None or n2 is None:
        return False

    if n1.data != n2.data :
        return False
    # There are two possible cases for n1 and n2 to be isomorphic
    # Case 1: The subtrees rooted at these nodes have NOT
    # been "Flipped".
    # Both of these subtrees have to be isomorphic, hence the &&
    # Case 2: The subtrees rooted at these nodes have
    # been "Flipped"
    return ((isIsomorphic(n1.left, n2.left)and
            isIsomorphic(n1.right, n2.right)) or
            (isIsomorphic(n1.left, n2.right) and
            isIsomorphic(n1.right, n2.left)))
```

```
)  
  
# Driver program to test above function  
n1 = Node(1)  
n1.left = Node(2)  
n1.right = Node(3)  
n1.left.left = Node(4)  
n1.left.right = Node(5)  
n1.right.left = Node(6)  
n1.left.right.left = Node(7)  
n1.left.right.right = Node(8)  
  
n2 = Node(1)  
n2.left = Node(3)  
n2.right = Node(2)  
n2.right.left = Node(4)  
n2.right.right = Node(5)  
n2.left.right = Node(6)  
n2.right.right.left = Node(8)  
n2.right.right.right = Node(7)  
  
print "Yes" if (isIsomorphic(n1, n2) == True) else "No"  
  
# This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

Output:

Yes

**Time Complexity:** The above solution does a traversal of both trees. So time complexity is  $O(m + n)$  where m and n are number of nodes in given trees.

## Source

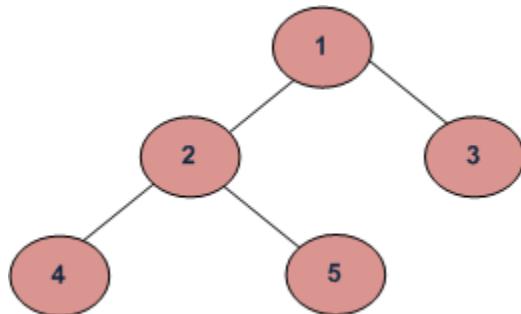
<https://www.geeksforgeeks.org/tree-isomorphism-problem/>

## Chapter 428

# Tree Traversals (Inorder, Preorder and Postorder)

Tree Traversals (Inorder, Preorder and Postorder) - GeeksforGeeks

Unlike linear data structures (Array, Linked List, Queues, Stacks, etc) which have only one logical way to traverse them, trees can be traversed in different ways. Following are the generally used ways for traversing trees.



Example Tree

Depth First Traversals:

- (a) Inorder (Left, Root, Right) : 4 2 5 1 3
- (b) Preorder (Root, Left, Right) : 1 2 4 5 3
- (c) Postorder (Left, Right, Root) : 4 5 2 3 1

Breadth First or Level Order Traversal : 1 2 3 4 5

Please see [this post](#) for Breadth First Traversal.

**Inorder Traversal (Practice):**

```
Algorithm Inorder(tree)
1. Traverse the left subtree, i.e., call Inorder(left-subtree)
```

2. Visit the root.
3. Traverse the right subtree, i.e., call Inorder(right-subtree)

Uses of Inorder

In case of binary search trees (BST), Inorder traversal gives nodes in non-decreasing order. To get nodes of BST in non-increasing order, a variation of Inorder traversal where Inorder traversal is reversed can be used.

Example: Inorder traversal for the above-given figure is 4 2 5 1 3.

#### **Preorder Traversal ([Practice](#)):**

```
Algorithm Preorder(tree)
    1. Visit the root.
    2. Traverse the left subtree, i.e., call Preorder(left-subtree)
    3. Traverse the right subtree, i.e., call Preorder(right-subtree)
```

Uses of Preorder

Preorder traversal is used to create a copy of the tree. Preorder traversal is also used to get prefix expression on of an expression tree. Please see [http://en.wikipedia.org/wiki/Polish\\_notation](http://en.wikipedia.org/wiki/Polish_notation) to know why prefix expressions are useful.

Example: Preorder traversal for the above given figure is 1 2 4 5 3.

#### **Postorder Traversal ([Practice](#)):**

```
Algorithm Postorder(tree)
    1. Traverse the left subtree, i.e., call Postorder(left-subtree)
    2. Traverse the right subtree, i.e., call Postorder(right-subtree)
    3. Visit the root.
```

Uses of Postorder

Postorder traversal is used to delete the tree. Please see [the question for deletion of tree](#) for details. Postorder traversal is also useful to get the postfix expression of an expression tree. Please see [http://en.wikipedia.org/wiki/Reverse\\_Polish\\_notation](http://en.wikipedia.org/wiki/Reverse_Polish_notation) to for the usage of postfix expression.

Example: Postorder traversal for the above given figure is 4 5 2 3 1.

**C++**

```
// C program for different tree traversals
#include <iostream>
using namespace std;

/* A binary tree node has data, pointer to left child
```

```
and a pointer to right child */
struct Node
{
    int data;
    struct Node* left, *right;
    Node(int data)
    {
        this->data = data;
        left = right = NULL;
    }
};

/* Given a binary tree, print its nodes according to the
"bottom-up" postorder traversal. */
void printPostorder(struct Node* node)
{
    if (node == NULL)
        return;

    // first recur on left subtree
    printPostorder(node->left);

    // then recur on right subtree
    printPostorder(node->right);

    // now deal with the node
    cout << node->data << " ";
}

/* Given a binary tree, print its nodes in inorder*/
void printInorder(struct Node* node)
{
    if (node == NULL)
        return;

    /* first recur on left child */
    printInorder(node->left);

    /* then print the data of node */
    cout << node->data << " ";

    /* now recur on right child */
    printInorder(node->right);
}

/* Given a binary tree, print its nodes in preorder*/
void printPreorder(struct Node* node)
{
```

```
if (node == NULL)
    return;

/* first print data of node */
cout << node->data << " ";

/* then recur on left subtree */
printPreorder(node->left);

/* now recur on right subtree */
printPreorder(node->right);
}

/* Driver program to test above functions*/
int main()
{
    struct Node *root = new Node(1);
    root->left           = new Node(2);
    root->right          = new Node(3);
    root->left->left     = new Node(4);
    root->left->right   = new Node(5);

    cout << "\nPreorder traversal of binary tree is \n";
    printPreorder(root);

    cout << "\nInorder traversal of binary tree is \n";
    printInorder(root);

    cout << "\nPostorder traversal of binary tree is \n";
    printPostorder(root);

    return 0;
}
```

C

```
// C program for different tree traversals
#include <stdio.h>
#include <stdlib.h>

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};
```

```
/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* node = (struct node*)
                           malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;

    return(node);
}

/* Given a binary tree, print its nodes according to the
   "bottom-up" postorder traversal. */
void printPostorder(struct node* node)
{
    if (node == NULL)
        return;

    // first recur on left subtree
    printPostorder(node->left);

    // then recur on right subtree
    printPostorder(node->right);

    // now deal with the node
    printf("%d ", node->data);
}

/* Given a binary tree, print its nodes in inorder*/
void printInorder(struct node* node)
{
    if (node == NULL)
        return;

    /* first recur on left child */
    printInorder(node->left);

    /* then print the data of node */
    printf("%d ", node->data);

    /* now recur on right child */
    printInorder(node->right);
}

/* Given a binary tree, print its nodes in preorder*/
```

```
void printPreorder(struct node* node)
{
    if (node == NULL)
        return;

    /* first print data of node */
    printf("%d ", node->data);

    /* then recur on left subtree */
    printPreorder(node->left);

    /* now recur on right subtree */
    printPreorder(node->right);
}

/* Driver program to test above functions*/
int main()
{
    struct node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);

    printf("\nPreorder traversal of binary tree is \n");
    printPreorder(root);

    printf("\nInorder traversal of binary tree is \n");
    printInorder(root);

    printf("\nPostorder traversal of binary tree is \n");
    printPostorder(root);

    getchar();
    return 0;
}
```

### Python

```
# Python program to for tree traversals

# A class that represents an individual node in a
# Binary Tree
class Node:
    def __init__(self, key):
        self.left = None
        self.right = None
        self.val = key
```

```
# A function to do inorder tree traversal
def printInorder(root):

    if root:

        # First recur on left child
        printInorder(root.left)

        # then print the data of node
        print(root.val),

        # now recur on right child
        printInorder(root.right)

# A function to do postorder tree traversal
def printPostorder(root):

    if root:

        # First recur on left child
        printPostorder(root.left)

        # then recur on right child
        printPostorder(root.right)

        # now print the data of node
        print(root.val),

# A function to do preorder tree traversal
def printPreorder(root):

    if root:

        # First print the data of node
        print(root.val),

        # Then recur on left child
        printPreorder(root.left)

        # Finally recur on right child
        printPreorder(root.right)
```

```
# Driver code
root = Node(1)
root.left      = Node(2)
root.right     = Node(3)
root.left.left = Node(4)
root.left.right= Node(5)
print "Preorder traversal of binary tree is"
printPreorder(root)

print "\nInorder traversal of binary tree is"
printInorder(root)

print "\nPostorder traversal of binary tree is"
printPostorder(root)
```

**Java**

```
// Java program for different tree traversals

/* Class containing left and right child of current
   node and key value*/
class Node
{
    int key;
    Node left, right;

    public Node(int item)
    {
        key = item;
        left = right = null;
    }
}

class BinaryTree
{
    // Root of Binary Tree
    Node root;

    BinaryTree()
    {
        root = null;
    }

    /* Given a binary tree, print its nodes according to the
       "bottom-up" postorder traversal. */
    void printPostorder(Node node)
    {
        if (node == null)
```

```
        return;

        // first recur on left subtree
        printPostorder(node.left);

        // then recur on right subtree
        printPostorder(node.right);

        // now deal with the node
        System.out.print(node.key + " ");
    }

/* Given a binary tree, print its nodes in inorder*/
void printInorder(Node node)
{
    if (node == null)
        return;

    /* first recur on left child */
    printInorder(node.left);

    /* then print the data of node */
    System.out.print(node.key + " ");

    /* now recur on right child */
    printInorder(node.right);
}

/* Given a binary tree, print its nodes in preorder*/
void printPreorder(Node node)
{
    if (node == null)
        return;

    /* first print data of node */
    System.out.print(node.key + " ");

    /* then recur on left subtree */
    printPreorder(node.left);

    /* now recur on right subtree */
    printPreorder(node.right);
}

// Wrappers over above recursive functions
void printPostorder() {      printPostorder(root);  }
void printInorder()   {      printInorder(root);   }
void printPreorder()  {      printPreorder(root); }
```

```
// Driver method
public static void main(String[] args)
{
    BinaryTree tree = new BinaryTree();
    tree.root = new Node(1);
    tree.root.left = new Node(2);
    tree.root.right = new Node(3);
    tree.root.left.left = new Node(4);
    tree.root.left.right = new Node(5);

    System.out.println("Preorder traversal of binary tree is ");
    tree.printPreorder();

    System.out.println("\nInorder traversal of binary tree is ");
    tree.printInorder();

    System.out.println("\nPostorder traversal of binary tree is ");
    tree.printPostorder();
}
```

Output:

```
Preorder traversal of binary tree is
1 2 4 5 3
Inorder traversal of binary tree is
4 2 5 1 3
Postorder traversal of binary tree is
4 5 2 3 1
```

**One more example:**

InOrder(root) visits nodes in the following order:

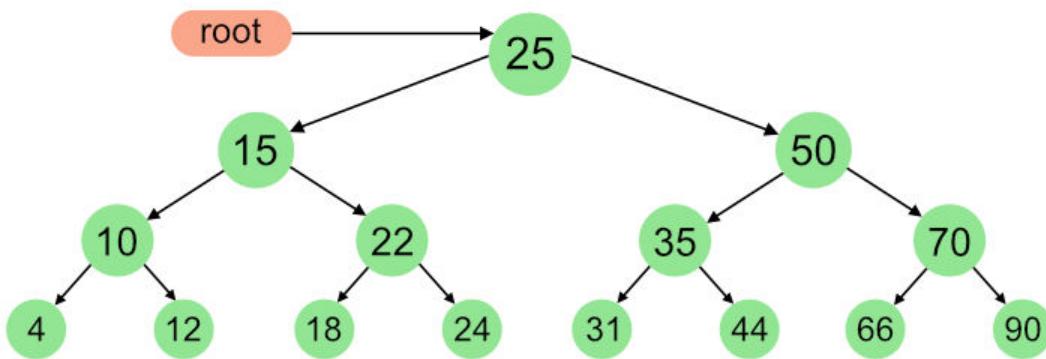
4, 10, 12, 15, 18, 22, 24, 25, 31, 35, 44, 50, 66, 70, 90

A Pre-order traversal visits nodes in the following order:

25, 15, 10, 4, 12, 22, 18, 24, 50, 35, 31, 44, 70, 66, 90

A Post-order traversal visits nodes in the following order:

4, 12, 10, 18, 24, 22, 15, 31, 44, 35, 66, 90, 70, 50, 25



**Time Complexity:**  $O(n)$

Let us see different corner cases.

Complexity function  $T(n)$  — for all problem where tree traversal is involved — can be defined as:

$$T(n) = T(k) + T(n - k - 1) + c$$

Where  $k$  is the number of nodes on one side of root and  $n-k-1$  on the other side.

Let's do an analysis of boundary conditions

Case 1: Skewed tree (One of the subtrees is empty and other subtree is non-empty )

$k$  is 0 in this case.

$$T(n) = T(0) + T(n-1) + c$$

$$T(n) = 2T(0) + T(n-2) + 2c$$

$$T(n) = 3T(0) + T(n-3) + 3c$$

$$T(n) = 4T(0) + T(n-4) + 4c$$

.....

$$T(n) = (n-1)T(0) + T(1) + (n-1)c$$

$$T(n) = nT(0) + (n)c$$

Value of  $T(0)$  will be some constant say  $d$ . (traversing an empty tree will take some constant time)

$$T(n) = n(c+d)$$

$$T(n) = \Theta(n)$$
 (Theta of  $n$ )

Case 2: Both left and right subtrees have equal number of nodes.

$$T(n) = 2T(\lfloor n/2 \rfloor) + c$$

This recursive function is in the standard form ( $T(n) = aT(n/b) + (-)(n)$ ) for master method [http://en.wikipedia.org/wiki/Master\\_theorem](http://en.wikipedia.org/wiki/Master_theorem). If we solve it by master method we get  $(-)(n)$

**Auxiliary Space :** If we don't consider size of stack for function calls then  $O(1)$  otherwise  $O(n)$ .

**Improved By :** [danielbritten](#)

## Source

<https://www.geeksforgeeks.org/tree-traversals-inorder-preorder-and-postorder/>

## Chapter 429

# Two Dimensional Segment Tree | Sub-Matrix Sum

Two Dimensional Segment Tree | Sub-Matrix Sum - GeeksforGeeks

Given a rectangular matrix  $M[0...n-1][0...m-1]$ , and queries are asked to find the sum / minimum / maximum on some sub-rectangles  $M[a...b][e...f]$ , as well as queries for modification of individual matrix elements (i.e  $M[x][y] = p$  ).

We can also answer sub-matrix queries using [Two Dimensional Binary Indexed Tree](#).

In this article, We will focus on solving sub-matrix queries using two dimensional segment tree.Two dimensional segment tree is nothing but segment tree of segment trees.

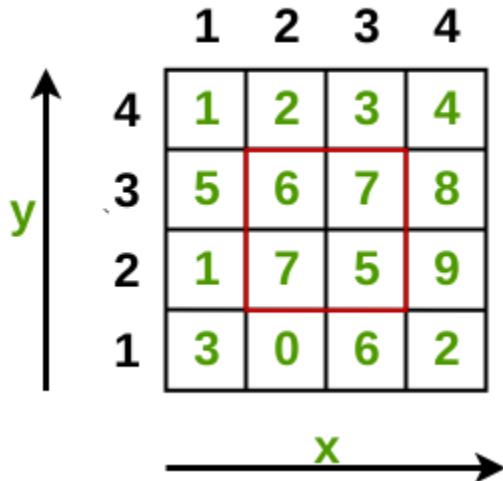
**Prerequisite :** [Segment Tree – Sum of given range](#)

**Algorithm :**

We will build a two-dimensional tree of segments by the following principle:

- 1 . In First step, We will construct an ordinary one-dimensional segment tree, working only with the first coordinate say ‘x’ and ‘y’ as constant. Here, we will not write number in inside the node as in the one-dimensional segment tree, but an entire tree of segments.
2. The second step is to combine the values of segmented trees. Assume that in second step instead of combining the elements we are combining the segment trees obtained from the step first.

**Consider the below example. Suppose we have to find the sum of all numbers inside the highlighted red area**



**Step 1 :** We will first create the segment tree of each strip of y- axis. We represent the segment tree here as an array where child node is  $2n$  and  $2n+1$  where  $n > 0$ .

Segment Tree for strip  $y=1$

10	3	7	1	2	3	4
----	---	---	---	---	---	---

Segment Tree for Strip  $y = 2$

26	11	15	5	6	7	8
----	----	----	---	---	---	---

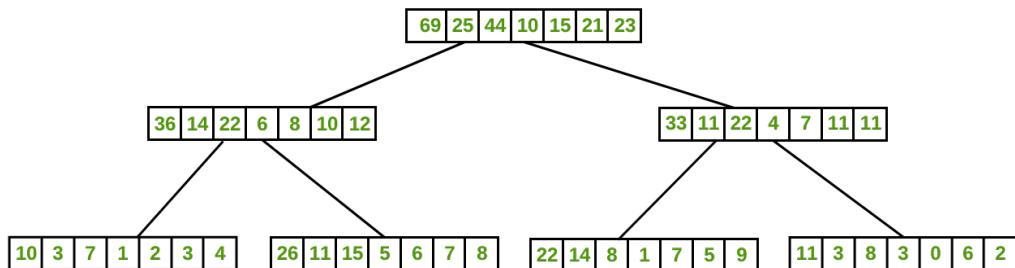
Segment Tree for Strip  $y = 3$

22	14	8	1	7	5	9
----	----	---	---	---	---	---

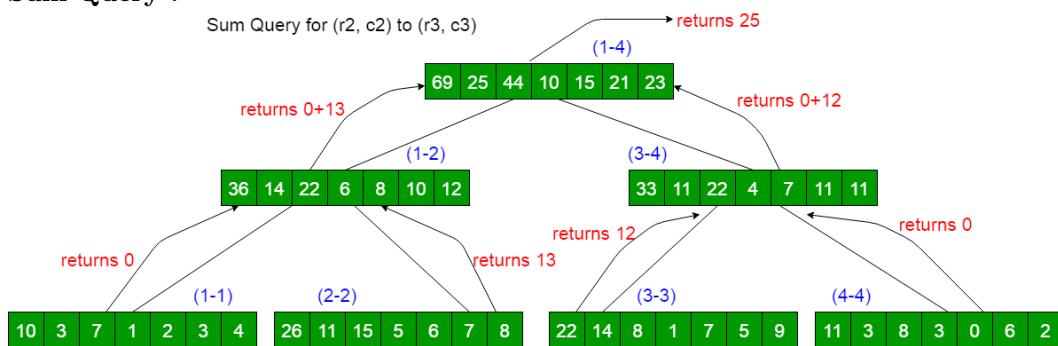
Segment Tree for Strip  $y = 4$

11	3	8	3	0	6	2
----	---	---	---	---	---	---

**Step 2:** In this step, we create the segment tree for the rectangular matrix where the base node are the strips of y-axis given above. The task is to merge above segment trees.



**Sum Query :**



Thanks to **Sahil Bansal** for contributing this image.

**Processing Query :**

We will respond to the two-dimensional query by the following principle: first to break the query on the first coordinate, and then, when we reached some vertex of the tree of segments with the first coordinate and then we call the corresponding tree of segments on the second coordinate.

This function works in time **O(log n \* log m)**, because it first descends the tree in the first coordinate, and for each traversed vertex of that tree, it makes a query from the usual tree of segments along the second coordinate.

**Modification Query :**

We want to learn how to modify the tree of segments in accordance with the change in the value of an element  $M[x][y] = p$ . It is clear that the changes will occur only in those vertices of the first tree of segments that cover the coordinate  $x$ , and for the trees of the segments corresponding to them, the changes will only occur in those vertices that cover the coordinate  $y$ . Therefore, the implementation of the modification request will not be very different from the one-dimensional case, only now we first descend the first coordinate, and then the second.

Output for the highlighted area will be 25.

Below is the implementation of above approach :

```

// C++ program for implementation
// of 2D segment tree.
#include <bits/stdc++.h>
using namespace std;

// Base node of segment tree.
int ini_seg[1000][1000] = { 0 };

// final 2d-segment tree.
int fin_seg[1000][1000] = { 0 };

// Rectangular matrix.
int rect[4][4] = {
    { 1, 2, 3, 4 },
    { 5, 6, 7, 8 },
    { 1, 7, 5, 9 },
    { 3, 0, 6, 2 },
};

// size of x coordinate.
int size = 4;

/*
 * A recursive function that constructs
 * Initial Segment Tree for array rect[][] = { }.
 * 'pos' is index of current node in segment
 * tree seg[]. 'strip' is the enumeration
 * for the y-axis.
 */
int segment(int low, int high,
            int pos, int strip)
{
    if (high == low) {
        ini_seg[strip][pos] = rect[strip][low];
    }
    else {
        int mid = (low + high) / 2;
        segment(low, mid, 2 * pos, strip);
        segment(mid + 1, high, 2 * pos + 1, strip);
        ini_seg[strip][pos] = ini_seg[strip][2 * pos] +
                            ini_seg[strip][2 * pos + 1];
    }
}

/*
 * A recursive function that constructs
 * Final Segment Tree for array ini_seg[][] = { }.

```

```

/*
int finalSegment(int low, int high, int pos)
{
    if (high == low) {

        for (int i = 1; i < 2 * size; i++)
            fin_seg[pos][i] = ini_seg[low][i];
    }
    else {
        int mid = (low + high) / 2;
        finalSegment(low, mid, 2 * pos);
        finalSegment(mid + 1, high, 2 * pos + 1);

        for (int i = 1; i < 2 * size; i++)
            fin_seg[pos][i] = fin_seg[2 * pos][i] +
                fin_seg[2 * pos + 1][i];
    }
}

/*
* Return sum of elements in range from index
* x1 to x2 . It uses the final_seg[][] array
* created using finalsegment() function.
* 'pos' is index of current node in
* segment tree fin_seg[][] .
*/
int finalQuery(int pos, int start, int end,
               int x1, int x2, int node)
{
    if (x2 < start || end < x1) {
        return 0;
    }

    if (x1 <= start && end <= x2) {
        return fin_seg[node][pos];
    }

    int mid = (start + end) / 2;
    int p1 = finalQuery(2 * pos, start, mid,
                        x1, x2, node);

    int p2 = finalQuery(2 * pos + 1, mid + 1,
                        end, x1, x2, node);

    return (p1 + p2);
}
*/

```

```

* This fuction calls the finalQuery fuction
* for elements in range from index x1 to x2 .
* This fuction queries the yth coordinate.
*/
int query(int pos, int start, int end,
          int y1, int y2, int x1, int x2)
{
    if (y2 < start || end < y1) {
        return 0;
    }

    if (y1 <= start && end <= y2) {
        return (finalQuery(1, 1, 4, x1, x2, pos));
    }

    int mid = (start + end) / 2;
    int p1 = query(2 * pos, start,
                   mid, y1, y2, x1, x2);
    int p2 = query(2 * pos + 1, mid + 1,
                   end, y1, y2, x1, x2);

    return (p1 + p2);
}

/* A recursive function to update the nodes
   which for the given index. The following
   are parameters : pos --> index of current
   node in segment tree fin_seg[][] . x ->
   index of the element to be updated. val -->
   Value to be change at node idx
*/
int finalUpdate(int pos, int low, int high,
                int x, int val, int node)
{
    if (low == high) {
        fin_seg[node][pos] = val;
    }
    else {
        int mid = (low + high) / 2;

        if (low <= x && x <= mid) {
            finalUpdate(2 * pos, low, mid, x, val, node);
        }
        else {
            finalUpdate(2 * pos + 1, mid + 1, high,
                        x, val, node);
        }
    }
}

```

```

        fin_seg[node][pos] = fin_seg[node][2 * pos] +
                            fin_seg[node][2 * pos + 1];
    }
}

/*
This function call the final update function after
visiting the yth coordinate in the segment tree fin_seg[][].
*/
int update(int pos, int low, int high, int x, int y, int val)
{
    if (low == high) {
        finalUpdate(1, 1, 4, x, val, pos);
    }
    else {
        int mid = (low + high) / 2;

        if (low <= y && y <= mid) {
            update(2 * pos, low, mid, x, y, val);
        }
        else {
            update(2 * pos + 1, mid + 1, high, x, y, val);
        }

        for (int i = 1; i < size; i++)
            fin_seg[pos][i] = fin_seg[2 * pos][i] +
                               fin_seg[2 * pos + 1][i];
    }
}

// Driver program to test above functions
int main()
{
    int pos = 1;
    int low = 0;
    int high = 3;

    // Call the ini_segment() to create the
    // initial segment tree on x- coordinate
    for (int strip = 0; strip < 4; strip++)
        segment(low, high, 1, strip);

    // Call the final function to built the 2d segment tree.
    finalSegment(low, high, 1);

    /*
Query:
* To request the query for sub-rectangle y1, y2=(2, 3) x1, x2=(2, 3)

```

```
* update the value of index (3, 3)=100;
* To request the query for sub-rectangle y1, y2=(2, 3) x1, x2=(2, 3)
*/
    cout << "The sum of the submatrix (y1, y2)->(2, 3), "
        << "(x1, x2)->(2, 3) is "
        << query(1, 1, 4, 2, 3, 2, 3) << endl;

// Function to update the value
update(1, 1, 4, 2, 3, 100);

cout << "The sum of the submatrix (y1, y2)->(2, 3), "
        << "(x1, x2)->(2, 3) is "
        << query(1, 1, 4, 2, 3, 2, 3) << endl;

return 0;
}
```

**Output:**

```
The sum of the submatrix (y1, y2)->(2, 3), (x1, x2)->(2, 3) is 25
The sum of the submatrix (y1, y2)->(2, 3), (x1, x2)->(2, 3) is 118
```

**Time complexity :**

Processing Query :  $O(\log n * \log m)$   
Modification Query:  $O(2 * n * \log n * \log m)$   
**Space Complexity :**  $O(4 * m * n)$

**Source**

<https://www.geeksforgeeks.org/two-dimensional-segment-tree-sub-matrix-sum/>

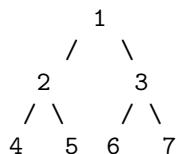
## Chapter 430

# Vertical Sum in Binary Tree | Set 2 (Space Optimized)

Vertical Sum in Binary Tree | Set 2 (Space Optimized) - GeeksforGeeks

Given a Binary Tree, find vertical sum of the nodes that are in same vertical line. Print all sums through different vertical lines.

Examples:



The tree has 5 vertical lines

Vertical-Line-1 has only one node 4 => vertical sum is 4

Vertical-Line-2: has only one node 2=> vertical sum is 2

Vertical-Line-3: has three nodes: 1,5,6 => vertical sum is  $1+5+6 = 12$

Vertical-Line-4: has only one node 3 => vertical sum is 3

Vertical-Line-5: has only one node 7 => vertical sum is 7

So expected output is 4, 2, 12, 3 and 7

We have discussed [Hashing](#) Based Solution in [Set 1](#). Hashing based solution requires a Hash Table to be maintained. We know that hashing requires more space than the number of entries in it. In this post, [Doubly Linked List](#) based solution is discussed. The solution discussed here requires only n nodes of linked list where n is total number of vertical lines in binary tree. Below is algorithm.

```
verticalSumDLL(root)
1) Create a node of doubly linked list node
   with value 0. Let the node be llnode.
2) verticalSumDLL(root, llnode)

verticalSumDLL(tnode, llnode)
1) Add current node's data to its vertical line
   llnode.data = llnode.data + tnode.data;
2) Recursively process left subtree
   // If left child is not empty
   if (tnode.left != null)
   {
      if (llnode.prev == null)
      {
         llnode.prev = new LLNode(0);
         llnode.prev.next = llnode;
      }
      verticalSumDLLUtil(tnode.left, llnode.prev);
   }
3) Recursively process right subtree
   if (tnode.right != null)
   {
      if (llnode.next == null)
      {
         llnode.next = new LLNode(0);
         llnode.next.prev = llnode;
      }
      verticalSumDLLUtil(tnode.right, llnode.next);
   }
```

C++

```
/// C++ program of space optimized solution
/// to find vertical sum of binary tree.
#include <bits/stdc++.h>

using namespace std;

/// Tree node structure
struct TNode{
    int key;
    struct TNode *left, *right;
};

/// Function to create new tree node
TNode* newTNode(int key)
{
    TNode* temp = new TNode;
```

```
temp->key = key;
temp->left = temp->right = NULL;
return temp;
}

/// Doubly linked list structure
struct LLNode{
    int key;
    struct LLNode *prev, *next;
};

/// Function to create new Linked List Node
LLNode* newLLNode(int key)
{
    LLNode* temp = new LLNode;
    temp->key = key;
    temp->prev = temp->next = NULL;
    return temp;
}

/// Function that creates Linked list and store
/// vertical sum in it.
void verticalSumDLLUtil(TNode *root, LLNode *sumNode)
{
    /// Update sum of current line by adding value
    /// of current tree node.
    sumNode->key = sumNode->key + root->key;

    /// Recursive call to left subtree.
    if(root->left)
    {
        if(sumNode->prev == NULL)
        {
            sumNode->prev = newLLNode(0);
            sumNode->prev->next = sumNode;
        }
        verticalSumDLLUtil(root->left, sumNode->prev);
    }

    /// Recursive call to right subtree.
    if(root->right)
    {
        if(sumNode->next == NULL)
        {
            sumNode->next = newLLNode(0);
            sumNode->next->prev = sumNode;
        }
        verticalSumDLLUtil(root->right, sumNode->next);
    }
}
```

```
        }
    }

/// Function to print vertical sum of Tree.
/// It uses verticalSumDLLUtil() to calculate sum.
void verticalSumDLL(TNode* root)
{
    /// Create Linked list node for
    /// line passing through root.
    LLNode* sumNode = newLLNode(0);

    /// Compute vertical sum of different lines.
    verticalSumDLLUtil(root, sumNode);

    /// Make doubly linked list pointer point
    /// to first node in list.
    while(sumNode->prev != NULL){
        sumNode = sumNode->prev;
    }

    /// Print vertical sum of different lines
    /// of binary tree.
    while(sumNode != NULL){
        cout << sumNode->key << " ";
        sumNode = sumNode->next;
    }
}

int main()
{
    /*
        1
       / \
      /   \
     2     3
    / \   / \
   /   \ /   \
  4     5   6   7
    */
    TNode *root = newTNode(1);
    root->left = newTNode(2);
    root->right = newTNode(3);
    root->left->left = newTNode(4);
    root->left->right = newTNode(5);
    root->right->left = newTNode(6);
    root->right->right = newTNode(7);

    cout << "Vertical Sums are\n";
```

```
    verticalSumDLL(root);
    return 0;
}

// This code is contributed by <b>Rahul Titare</b>

Java

// Java implementation of space optimized solution
// to find vertical sum.

public class VerticalSumBinaryTree
{
    // Prints vertical sum of different vertical
    // lines in tree. This method mainly uses
    // verticalSumDLLUtil().
    static void verticalSumDLL(TNode root)
    {
        // Create a doubly linked list node to
        // store sum of lines going through root.
        // Vertical sum is initialized as 0.
        LLNode llnode = new LLNode(0);

        // Compute vertical sum of different lines
        verticalSumDLLUtil(root, llnode);

        // llnode refers to sum of vertical line
        // going through root. Move llnode to the
        // leftmost line.
        while (llnode.prev != null)
            llnode = llnode.prev;

        // Prints vertical sum of all lines starting
        // from leftmost vertical line
        while (llnode != null)
        {
            System.out.print(llnode.data + " ");
            llnode = llnode.next;
        }
    }

    // Constructs linked list
    static void verticalSumDLLUtil(TNode tnode,
                                   LLNode llnode)
    {
        // Add current node's data to its vertical line
        llnode.data = llnode.data + tnode.data;
```

```
// Recursively process left subtree
if (tnode.left != null)
{
    if (llnode.prev == null)
    {
        llnode.prev = new LLNode(0);
        llnode.prev.next = llnode;
    }
    verticalSumDLLUtil(tnode.left, llnode.prev);
}

// Process right subtree
if (tnode.right != null)
{
    if (llnode.next == null)
    {
        llnode.next = new LLNode(0);
        llnode.next.prev = llnode;
    }
    verticalSumDLLUtil(tnode.right, llnode.next);
}
}

// Driver code
public static void main(String[] args)
{
    // Let us construct the tree shown above
    TNode root = new TNode(1);
    root.left = new TNode(2);
    root.right = new TNode(3);
    root.left.left = new TNode(4);
    root.left.right = new TNode(5);
    root.right.left = new TNode(6);
    root.right.right = new TNode(7);

    System.out.println("Vertical Sums are");
    verticalSumDLL(root);
}

// Doubly Linked List Node
static class LLNode
{
    int data;
    LLNode prev, next;
    public LLNode(int d) { data = d; }
}

// Binary Tree Node
```

```
static class TNode
{
    int data;
    TNode left, right;
    public TNode(int d) { data = d; }
}
```

Output :

```
Vertical Sums are
4 2 12 3 7
```

This article is contributed by **Rahul Titare**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Improved By : [nik1996](#)

## Source

<https://www.geeksforgeeks.org/vertical-sum-in-binary-tree-set-space-optimized/>

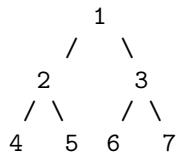
## Chapter 431

# Vertical Sum in a given Binary Tree | Set 1

Vertical Sum in a given Binary Tree | Set 1 - GeeksforGeeks

Given a Binary Tree, find vertical sum of the nodes that are in same vertical line. Print all sums through different vertical lines.

Examples:



The tree has 5 vertical lines

Vertical-Line-1 has only one node 4 => vertical sum is 4

Vertical-Line-2: has only one node 2=> vertical sum is 2

Vertical-Line-3: has three nodes: 1,5,6 => vertical sum is  $1+5+6 = 12$

Vertical-Line-4: has only one node 3 => vertical sum is 3

Vertical-Line-5: has only one node 7 => vertical sum is 7

So expected output is 4, 2, 12, 3 and 7

We need to check the Horizontal Distances from root for all nodes. If two nodes have the same Horizontal Distance (HD), then they are on same vertical line. The idea of HD is simple. HD for root is 0, a right edge (edge connecting to right subtree) is considered as +1 horizontal distance and a left edge is considered as -1 horizontal distance. For example, in the above tree, HD for Node 4 is at -2, HD for Node 2 is -1, HD for 5 and 6 is 0 and HD for node 7 is +2.

We can do inorder traversal of the given Binary Tree. While traversing the tree, we can

recursively calculate HDs. We initially pass the horizontal distance as 0 for root. For left subtree, we pass the Horizontal Distance as Horizontal distance of root minus 1. For right subtree, we pass the Horizontal Distance as Horizontal Distance of root plus 1.

Following is Java implementation for the same. HashMap is used to store the vertical sums for different horizontal distances. Thanks to Nages for suggesting this method.

### Java

```
import java.util.HashMap;

// Class for a tree node
class TreeNode {

    // data members
    private int key;
    private TreeNode left;
    private TreeNode right;

    // Accessor methods
    public int key()      { return key; }
    public TreeNode left() { return left; }
    public TreeNode right() { return right; }

    // Constructor
    public TreeNode(int key)
    { this.key = key; left = null; right = null; }

    // Methods to set left and right subtrees
    public void setLeft(TreeNode left)  { this.left = left; }
    public void setRight(TreeNode right) { this.right = right; }
}

// Class for a Binary Tree
class Tree {

    private TreeNode root;

    // Constructors
    public Tree() { root = null; }
    public Tree(TreeNode n) { root = n; }

    // Method to be called by the consumer classes
    // like Main class
    public void VerticalSumMain() { VerticalSum(root); }

    // A wrapper over VerticalSumUtil()
    private void VerticalSum(TreeNode root) {
```

```

// base case
if (root == null) { return; }

// Creates an empty hashMap hM
HashMap<Integer, Integer> hM =
    new HashMap<Integer, Integer>();

// Calls the VerticalSumUtil() to store the
// vertical sum values in hM
VerticalSumUtil(root, 0, hM);

// Prints the values stored by VerticalSumUtil()
if (hM != null) {
    System.out.println(hM.entrySet());
}
}

// Traverses the tree in Inorder form and builds
// a hashMap hM that contains the vertical sum
private void VerticalSumUtil(TreeNode root, int hD,
    HashMap<Integer, Integer> hM) {

    // base case
    if (root == null) { return; }

    // Store the values in hM for left subtree
    VerticalSumUtil(root.left(), hD - 1, hM);

    // Update vertical sum for hD of this node
    int prevSum = (hM.get(hD) == null) ? 0 : hM.get(hD);
    hM.put(hD, prevSum + root.key());

    // Store the values in hM for right subtree
    VerticalSumUtil(root.right(), hD + 1, hM);
}

}

// Driver class to test the verticalSum methods
public class Main {

    public static void main(String[] args) {
        /* Create following Binary Tree
           1
          /   \
         2     3
        / \   / \
       4   5   6   7
        */
    }
}

```

```
/*
TreeNode root = new TreeNode(1);
root.setLeft(new TreeNode(2));
root.setRight(new TreeNode(3));
root.left().setLeft(new TreeNode(4));
root.left().setRight(new TreeNode(5));
root.right().setLeft(new TreeNode(6));
root.right().setRight(new TreeNode(7));
Tree t = new Tree(root);

System.out.println("Following are the values of" +
                     " vertical sums with the positions" +
                     " of the columns with respect to root ");
t.VerticalSumMain();
}
}
```

C++

```
// C++ program to find Vertical Sum in
// a given Binary Tree
#include<bits/stdc++.h>
using namespace std;

struct Node
{
    int data;
    struct Node *left, *right;
};

// A utility function to create a new
// Binary Tree node
Node* newNode(int data)
{
    Node *temp = new Node;
    temp->data = data;
    temp->left = temp->right = NULL;
    return temp;
}

// Traverses the tree in Inorder form and
// populates a hashMap that contains the
// vertical sum
void verticalSumUtil(Node *node, int hd,
                     map<int, int> &Map)
{
    // Base case
```

```
if (node == NULL) return;

// Recur for left subtree
verticalSumUtil(node->left, hd-1, Map);

// Add val of current node to
// map entry of corresponding hd
Map[hd] += node->data;

// Recur for right subtree
verticalSumUtil(node->right, hd+1, Map);
}

// Function to find vertical sum
void verticalSum(Node *root)
{
    // a map to store sum of each horizontal
    // distance
    map < int, int> Map;
    map < int, int> :: iterator it;

    // populate the map
    verticalSumUtil(root, 0, Map);

    // Prints the values stored by VerticalSumUtil()
    for (it = Map.begin(); it != Map.end(); ++it)
    {
        cout << it->first << ":" 
            << it->second << endl;
    }
}

// Driver program to test above functions
int main()
{
    // Create binary tree shown in above figure
    Node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->left = newNode(6);
    root->right->right = newNode(7);
    root->right->left->right = newNode(8);
    root->right->right->right = newNode(9);

    cout << "Following are the values of vertical
          " sums with the positions of the "
```

```
    "columns with respect to root\n";
verticalSum(root);

return 0;
}
// This code is contributed by Aditi Sharma
```

[Vertical Sum in Binary Tree | Set 2 \(Space Optimized\)](#)

Time Complexity: O(n)

## Source

<https://www.geeksforgeeks.org/vertical-sum-in-a-given-binary-tree/>

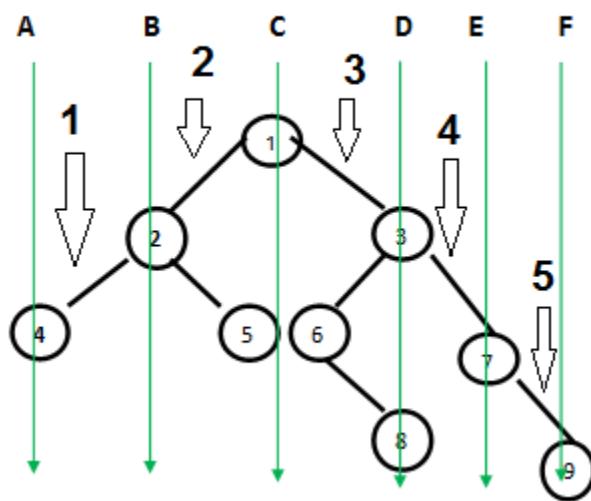
## Chapter 432

# Vertical width of Binary tree | Set 1

Vertical width of Binary tree | Set 1 - GeeksforGeeks

Given a binary tree, find the vertical width of the binary tree. Width of a binary tree is the number of vertical paths.

### Vertical Lines

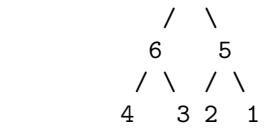


In this image, the tree contains 6 vertical lines which is the required width of tree.

Examples :

Input :

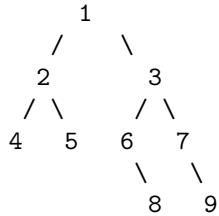
7



Output :

5

Input :



Output :

6

**Approach :** Take inorder traversal and then take a temporary variable, if we go left then temp value decreases and if go to right then temp value increases. Assert a condition in this, if minimum is greater than temp, then minimum = temp and if maximum less then temp then maximum = temp. At the end, print minimum + maximum which is the vertical width of the tree.

```

// CPP program to print vertical width
// of a tree
#include <bits/stdc++.h>
using namespace std;

// A Binary Tree Node
struct Node
{
    int data;
    struct Node *left, *right;
};

// get vertical width
void lengthUtil(Node* root, int &maximum,
                int &minimum, int curr=0)
{
    if (root == NULL)
        return;

    // traverse left
    lengthUtil(root->left, maximum,
               minimum, curr - 1);

    // if curr is decrease then get

```

```
// value in minimum
if (minimum > curr)
    minimum = curr;

// if curr is increase then get
// value in maximum
if (maximum < curr)
    maximum = curr;

// traverse right
lengthUtil(root->right, maximum,
           minimum, curr + 1);

}

int getLength(Node* root)
{
    int maximum = 0, minimum = 0;
    lengthUtil(root, maximum, minimum, 0);

    // 1 is added to include root in the width
    return (abs(minimum) + maximum) + 1;
}

// Utility function to create a new tree node
Node* newNode(int data)
{
    Node* curr = new Node;
    curr->data = data;
    curr->left = curr->right = NULL;
    return curr;
}

// Driver program to test above functions
int main()
{

    Node* root = newNode(7);
    root->left = newNode(6);
    root->right = newNode(5);
    root->left->left = newNode(4);
    root->left->right = newNode(3);
    root->right->left = newNode(2);
    root->right->right = newNode(1);

    cout << getLength(root) << "\n";
}
```

```
        return 0;  
    }
```

**Output:**

5

Time Complexity :  $O(n)$

Auxiliary Space :  $O(h)$  where  $h$  is height of binary tree. This much space is needed for recursive calls.

**Source**

<https://www.geeksforgeeks.org/width-binary-tree-set-1/>

## Chapter 433

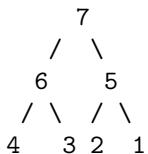
# Vertical width of Binary tree | Set 2

Vertical width of Binary tree | Set 2 - GeeksforGeeks

Given a binary tree, find the vertical width of the binary tree. Width of a binary tree is the number of vertical paths.

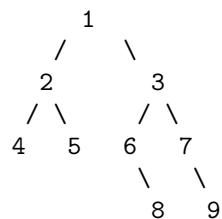
Examples:

Input :



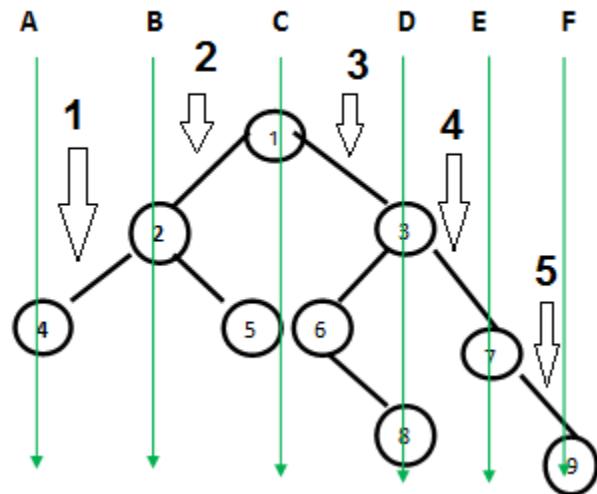
Output : 5

Input :



Output : 6

Prerequisite : [Print Binary Tree in Vertical order](#)

**Vertical Lines**


In this image, the tree contains 6 vertical lines which is the required width of tree.

**Approach :** In this Approach, we use the approach for printing vertical View of binary tree. Store the horizontal distances in a set and return **1 + highest horizontal distance – lowest horizontal distance**. 1 is added to consider horizontal distance 0 as well. While going left, do  $hd - 1$  and for right do  $hd + 1$ . We insert all possible distances in a hash table and finally return size of the hash table.

```

// CPP code to find vertical
// width of a binary tree
#include <bits/stdc++.h>
using namespace std;

// Tree class
class Node
{
public :
    int data;
    Node *left, *right;

    // Constructor
    Node(int data_new)
    {
        data = data_new;
        left = right = NULL;
    }
};

// Function to fill hd in set.
void fillSet(Node* root, unordered_set<int>& s,

```

```
int hd)
{
    if (!root)
        return;

    fillSet(root->left, s, hd - 1);
    s.insert(hd);
    fillSet(root->right, s, hd + 1);
}

int verticalWidth(Node* root)
{
    unordered_set<int> s;

    // Third parameter is horizontal
    // distance
    fillSet(root, s, 0);

    return s.size();
}

int main()
{
    Node* root = NULL;

    // Creating the above tree
    root = new Node(1);
    root->left = new Node(2);
    root->right = new Node(3);
    root->left->left = new Node(4);
    root->left->right = new Node(5);
    root->right->left = new Node(6);
    root->right->right = new Node(7);
    root->right->left->right = new Node(8);
    root->right->right->right = new Node(9);

    cout << verticalWidth(root) << "\n";

    return 0;
}
```

**Output:**

**Source**

<https://www.geeksforgeeks.org/vertical-width-binary-tree-set-2/>

## Chapter 434

# Ways to color a skewed tree such that parent and child have different colors

Ways to color a skewed tree such that parent and child have different colors - GeeksforGeeks

Given a skewed tree (Every node has at most one child) with N nodes and K colors. You have to assign a color from 1 to K to each node such that parent and child has different colors. Find the **maximum** number of **ways** of coloring the nodes.

**Examples –**

```
Input : N = 2, K = 2.  
Output :  
Let A1 and A2 be the two nodes.  
Let A1 is parent of A2.  
Colors are Red and Blue.  
Case 1: A1 is colored Red  
        and A2 is colored Blue.  
Case 2: A1 is colored Blue  
        and A2 is colored Red.  
No. of ways : 2
```

```
Input : N = 3, K = 3.  
Output :  
A1, A2, A3 are the nodes.  
A1 is parent of A2  
and A2 is parent of A3.  
Let colors be R, B, G.  
A1 can choose any three colors  
and A2 can choose
```

any other two colors  
and A3 can choose  
any other two colors  
than its parents.  
No. of ways: 12

Note that **only the root** and **children** (children, grand children, grand grand children .... and all) should have **different** colors. The root of the tree can choose any of the K colors so K ways. Every other node can choose other K-1 colors other than its parent. So every node has K-1 choices.

Here, we select the tree as every node as only one child. We can choose any of the K colors for the root of the tree so K ways. And we are left with K-1 colors for its child. So for every child we can assign a color other than its parent. Thus, for each of the N-1 nodes we are left with K-1 colors. Thus the answer is  $K^*(K-1)^{(N-1)}$ .

We can find the answer by using normal power function which takes  $O(N)$  time complexity. But for better time complexity we use Faster Exponentiation function which takes  $O(\log N)$  time complexity.

C++

```
// C++ program to count number of ways to color
// a N node skewed tree with k colors such that
// parent and children have different colors.
#include <bits/stdc++.h>
using namespace std;

// fast_way is recursive
// method to calculate power
int fastPow(int N, int K)
{
    if (K == 0)
        return 1;
    int temp = fastPow(N, K / 2);
    if (K % 2 == 0)
        return temp * temp;
    else
        return N * temp * temp;
}

int countWays(int N, int K)
{
    return K * fastPow(K - 1, N - 1);
}

// driver program
int main()
{
```

```
    int N = 3, K = 3;
    cout << countWays(N, K);
    return 0;
}
```

### Java

```
// Java program to count number of ways to color
// a N node skewed tree with k colors such that
// parent and children have different colors.
import java.io.*;

class GFG {
    // fast_way is recursive
    // method to calculate power
    static int fastPow(int N, int K)
    {
        if (K == 0)
            return 1;
        int temp = fastPow(N, K / 2);
        if (K % 2 == 0)
            return temp * temp;
        else
            return N * temp * temp;
    }

    static int countWays(int N, int K)
    {
        return K * fastPow(K - 1, N - 1);
    }

    // Driver program
    public static void main(String[] args)
    {
        int N = 3, K = 3;
        System.out.println(countWays(N, K));
    }
}

// This code is contributed by vt_m.
```

### Python3

```
# Python3 program to count
# number of ways to color
# a N node skewed tree with
# k colors such that parent
```

```
# and children have different
# colors.

# fast_way is recursive
# method to calculate power
def fastPow(N, K):
    if (K == 0):
        return 1;

    temp = fastPow(N, int(K / 2));
    if (K % 2 == 0):
        return temp * temp;
    else:
        return N * temp * temp;

def countWays(N, K):
    return K * fastPow(K - 1, N - 1);

# Driver Code
N = 3;
K = 3;
print(countWays(N, K));

# This code is contributed by mits
```

C#

```
// C# program to count number of ways
// to color a N node skewed tree with
// k colors such that parent and
// children have different colors
using System;

class GFG {

    // fast_way is recursive
    // method to calculate power
    static int fastPow(int N, int K)
    {
        if (K == 0)
            return 1;
        int temp = fastPow(N, K / 2);
        if (K % 2 == 0)
            return temp * temp;
        else
            return N * temp * temp;
    }
}
```

```
static int countWays(int N, int K)
{
    return K * fastPow(K - 1, N - 1);
}

// Driver code
public static void Main()
{
    int N = 3, K = 3;
    Console.WriteLine(countWays(N, K));
}
}

// This code is contributed by vt_m.
```

### PHP

```
<?php
// PHP program to count number
// of ways to color a N node
// skewed tree with k colors
// such that parent and children
// have different colors.

// fast_way is recursive
// method to calculate power
function fastPow($N, $K)
{
    if ($K == 0)
        return 1;

    $temp = fastPow($N, $K / 2);

    if ($K % 2 == 0)
        return $temp * $temp;
    else
        return $N * $temp * $temp;
}

function countWays($N, $K)
{
    return $K * fastPow($K - 1, $N - 1);
}

// Driver Code
$N = 3;
$K = 3;
echo countWays($N, $K);
```

```
// This code is contributed by ajit  
?>
```

**Output :**

12

**Improved By :** [vt\\_m](#), [jit\\_t](#), Mithun Kumar

**Source**

<https://www.geeksforgeeks.org/ways-to-color-a-skewed-tree-such-that-parent-and-child-have-different-colors/>

## Chapter 435

# Write Code to Determine if Two Trees are Identical

Write Code to Determine if Two Trees are Identical - GeeksforGeeks

Two trees are identical when they have same data and arrangement of data is also same.

To identify if two trees are identical, we need to traverse both trees simultaneously, and while traversing we need to compare data and children of the trees.

**Algorithm:**

```
sameTree(tree1, tree2)
1. If both trees are empty then return 1.
2. Else If both trees are non -empty
    (a) Check data of the root nodes (tree1->data == tree2->data)
    (b) Check left subtrees recursively i.e., call sameTree(
        tree1->left_subtree, tree2->left_subtree)
    (c) Check right subtrees recursively i.e., call sameTree(
        tree1->right_subtree, tree2->right_subtree)
    (d) If a,b and c are true then return 1.
3 Else return 0 (one is empty and other is not)
```

C/C++

```
#include <stdio.h>
#include <stdlib.h>

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
```

```
{  
    int data;  
    struct node* left;  
    struct node* right;  
};  
  
/* Helper function that allocates a new node with the  
   given data and NULL left and right pointers. */  
struct node* newNode(int data)  
{  
    struct node* node = (struct node*)  
                      malloc(sizeof(struct node));  
    node->data = data;  
    node->left = NULL;  
    node->right = NULL;  
  
    return(node);  
}  
  
/* Given two trees, return true if they are  
   structurally identical */  
int identicalTrees(struct node* a, struct node* b)  
{  
    /*1. both empty */  
    if (a==NULL && b==NULL)  
        return 1;  
  
    /* 2. both non-empty -> compare them */  
    if (a!=NULL && b!=NULL)  
    {  
        return  
        (  
            a->data == b->data &&  
            identicalTrees(a->left, b->left) &&  
            identicalTrees(a->right, b->right)  
        );  
    }  
  
    /* 3. one empty, one not -> false */  
    return 0;  
}  
  
/* Driver program to test identicalTrees function*/  
int main()  
{  
    struct node *root1 = newNode(1);  
    struct node *root2 = newNode(1);  
    root1->left = newNode(2);
```

```
root1->right = newNode(3);
root1->left->left  = newNode(4);
root1->left->right = newNode(5);

root2->left = newNode(2);
root2->right = newNode(3);
root2->left->left = newNode(4);
root2->left->right = newNode(5);

if(identicalTrees(root1, root2))
    printf("Both tree are identical.");
else
    printf("Trees are not identical.");

getchar();
return 0;
}
```

**Java**

```
// Java program to see if two trees are identical

// A binary tree node
class Node
{
    int data;
    Node left, right;

    Node(int item)
    {
        data = item;
        left = right = null;
    }
}

class BinaryTree
{
    Node root1, root2;

    /* Given two trees, return true if they are
       structurally identical */
    boolean identicalTrees(Node a, Node b)
    {
        /*1. both empty */
        if (a == null && b == null)
            return true;

        /* 2. both non-empty -> compare them */

```

```
if (a != null && b != null)
    return (a.data == b.data
            && identicalTrees(a.left, b.left)
            && identicalTrees(a.right, b.right));

/* 3. one empty, one not -> false */
return false;
}

/* Driver program to test identicalTrees() function */
public static void main(String[] args)
{
    BinaryTree tree = new BinaryTree();

    tree.root1 = new Node(1);
    tree.root1.left = new Node(2);
    tree.root1.right = new Node(3);
    tree.root1.left.left = new Node(4);
    tree.root1.left.right = new Node(5);

    tree.root2 = new Node(1);
    tree.root2.left = new Node(2);
    tree.root2.right = new Node(3);
    tree.root2.left.left = new Node(4);
    tree.root2.left.right = new Node(5);

    if (tree.identicalTrees(tree.root1, tree.root2))
        System.out.println("Both trees are identical");
    else
        System.out.println("Trees are not identical");
}
```

### Python

```
# Python program to determine if two trees are identical

# A binary tree node has data, pointer to left child
# and a pointer to right child
class Node:
    # Constructor to create a new node
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None
```

```
# Given two trees, return true if they are structurally
# identical
def identicalTrees(a, b):

    # 1. Both empty
    if a is None and b is None:
        return True

    # 2. Both non-empty -> Compare them
    if a is not None and b is not None:
        return ((a.data == b.data) and
                identicalTrees(a.left, b.left)and
                identicalTrees(a.right, b.right))

    # 3. one empty, one not -- false
    return False

# Driver program to test identicalTrees function
root1 = Node(1)
root2 = Node(1)
root1.left = Node(2)
root1.right = Node(3)
root1.left.left = Node(4)
root1.left.right = Node(5)

root2.left = Node(2)
root2.right = Node(3)
root2.left.left = Node(4)
root2.left.right = Node(5)

if identicalTrees(root1, root2):
    print "Both trees are identical"
else:
    print "Trees are not identical"

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

#### Time Complexity:

Complexity of the identicalTree() will be according to the tree with lesser number of nodes. Let number of nodes in two trees be m and n then complexity of sameTree() is O(m) where m < n. [Iterative function to check if two trees are identical.](#)

#### Source

<https://www.geeksforgeeks.org/write-c-code-to-determine-if-two-trees-are-identical/>

## Chapter 436

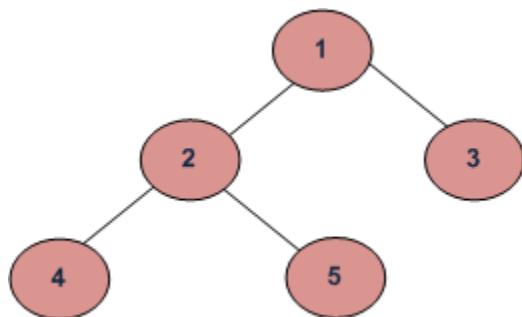
# Write a program to Delete a Tree

Write a program to Delete a Tree - GeeksforGeeks

To delete a tree we must traverse all the nodes of the tree and delete them one by one. So which traversal we should use – Inorder or Preorder or Postorder. Answer is simple – Postorder, because before deleting the parent node we should delete its children nodes first

We can delete tree with other traversals also with extra space complexity but why should we go for other traversals if we have Postorder available which does the work without storing anything in same time complexity.

For the following tree nodes are deleted in order – 4, 5, 2, 3, 1



Example Tree

**Note :** In Java automatic garbage collection happens, so we can simply make root null to delete the tree “root = null”;

C

```
#include<stdio.h>
#include<stdlib.h>
```

```
/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* node = (struct node*)
                           malloc(sizeof(struct node));

    node->data = data;
    node->left = NULL;
    node->right = NULL;
    return(node);
}

/* This function traverses tree in post order to
   to delete each and every node of the tree */
void deleteTree(struct node* node)
{
    if (node == NULL) return;

    /* first delete both subtrees */
    deleteTree(node->left);
    deleteTree(node->right);

    /* then delete the node */
    printf("\n Deleting node: %d", node->data);
    free(node);
}

/* Driver program to test deleteTree function*/
int main()
{
    struct node *root = newNode(1);
    root->left      = newNode(2);
    root->right     = newNode(3);
    root->left->left  = newNode(4);
    root->left->right = newNode(5);

    deleteTree(root);
```

```
    root = NULL;  
  
    printf("\n Tree deleted ");  
  
    return 0;  
}
```

**Java**

```
// Java program to delete a tree  
  
// A binary tree node  
class Node  
{  
    int data;  
    Node left, right;  
  
    Node(int item)  
    {  
        data = item;  
        left = right = null;  
    }  
}  
  
class BinaryTree  
{  
    Node root;  
  
    /* This function traverses tree in post order to  
       to delete each and every node of the tree */  
    void deleteTree(Node node)  
    {  
        // In Java automatic garbage collection  
        // happens, so we can simply make root  
        // null to delete the tree  
        root = null;  
    }  
  
    /* Driver program to test above functions */  
    public static void main(String[] args)  
    {  
        BinaryTree tree = new BinaryTree();  
  
        tree.root = new Node(1);  
        tree.root.left = new Node(2);  
        tree.root.right = new Node(3);  
        tree.root.left.left = new Node(4);  
        tree.root.left.right = new Node(5);
```

```
/* Print all root-to-leaf paths of the input tree */
tree.deleteTree(tree.root);
tree.root = null;
System.out.println("Tree deleted");

}
}
```

Output:

```
Deleting node: 4
Deleting node: 5
Deleting node: 2
Deleting node: 3
Deleting node: 1
Tree deleted
```

The above deleteTree() function deletes the tree, but doesn't change root to NULL which may cause problems if the user of deleteTree() doesn't change root to NULL and tries to access values using root pointer. We can modify the deleteTree() function to take reference to the root node so that this problem doesn't occur. See the following code.

## C

```
#include<stdio.h>
#include<stdlib.h>

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* node = (struct node*)
                           malloc(sizeof(struct node));

    node->data = data;
    node->left = NULL;
```

```
node->right = NULL;
return(node);
}

/* This function is same as deleteTree() in the previous program */
void _deleteTree(struct node* node)
{
    if (node == NULL) return;

    /* first delete both subtrees */
    _deleteTree(node->left);
    _deleteTree(node->right);

    /* then delete the node */
    printf("\n Deleting node: %d", node->data);
    free(node);
}

/* Deletes a tree and sets the root as NULL */
void deleteTree(struct node** node_ref)
{
    _deleteTree(*node_ref);
    *node_ref = NULL;
}

/* Driver program to test deleteTree function*/
int main()
{
    struct node *root = newNode(1);
    root->left      = newNode(2);
    root->right     = newNode(3);
    root->left->left   = newNode(4);
    root->left->right  = newNode(5);

    // Note that we pass the address of root here
    deleteTree(&root);
    printf("\n Tree deleted ");

    getchar();
    return 0;
}
```

### Java

```
// Java program to delete a tree

/* A binary tree node has data, pointer to left child
and pointer to right child */
```

```
class Node
{
    int data;
    Node left, right;

    Node(int d)
    {
        data = d;
        left = right = null;
    }
}

class BinaryTree
{

    static Node root;

    /* This function is same as deleteTree() in the previous program */
    void deleteTree(Node node)
    {
        // In Java automatic garbage collection
        // happens, so we can simply make root
        // null to delete the tree
        root = null;
    }

    /* Wrapper function that deletes the tree and
       sets root node as null */
    void deleteTreeRef(Node nodeRef)
    {
        deleteTree(nodeRef);
        nodeRef=null;
    }

    /* Driver program to test deleteTree function */
    public static void main(String[] args)
    {

        BinaryTree tree = new BinaryTree();

        tree.root = new Node(1);
        tree.root.left = new Node(2);
        tree.root.right = new Node(3);
        tree.root.left.left = new Node(4);
        tree.root.left.right = new Node(5);

        /* Note that we pass root node here */
        tree.deleteTreeRef(root);
    }
}
```

```
System.out.println("Tree deleted");

}

}

// This code has been contributed by Mayank Jaiswal(mayank_24)

Deleting node: 4
Deleting node: 5
Deleting node: 2
Deleting node: 3
Deleting node: 1
Tree deleted
```

**Time Complexity:**  $O(n)$

**Space Complexity:** If we don't consider size of stack for function calls then  $O(1)$  otherwise  $O(n)$

## Source

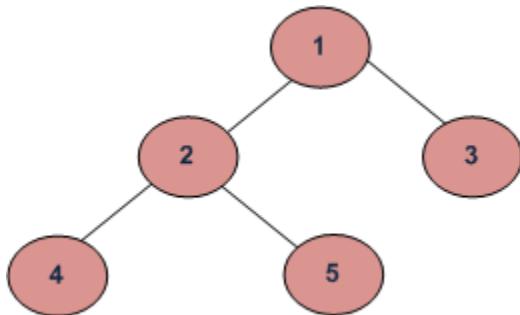
<https://www.geeksforgeeks.org/write-a-c-program-to-delete-a-tree/>

## Chapter 437

# Write a Program to Find the Maximum Depth or Height of a Tree

Write a Program to Find the Maximum Depth or Height of a Tree - GeeksforGeeks

Given a binary tree, find height of it. Height of empty tree is 0 and height of below tree is 3.



Example Tree

Recursively calculate height of left and right subtrees of a node and assign height to the node as max of the heights of two children plus 1. See below pseudo code and program for details.

### Algorithm:

```
maxDepth()  
1. If tree is empty then return 0  
2. Else
```

- (a) Get the max depth of left subtree recursively i.e.,
   
call maxDepth( tree->left-subtree)
- (a) Get the max depth of right subtree recursively i.e.,
   
call maxDepth( tree->right-subtree)
- (c) Get the max of max depths of left and right
   
subtrees and add 1 to it for the current node.
   
max\_depth = max(max dept of left subtree,
   
                  max depth of right subtree)
   
                  + 1
- (d) Return max\_depth

See the below diagram for more clarity about execution of the recursive function maxDepth() for above example tree.

```

maxDepth('1') = max(maxDepth('2'), maxDepth('3')) + 1
              = 2 + 1
                /   \
              /     \
            /       \
          /         \
        /           \
      maxDepth('2') = 1           maxDepth('3') = 1
= max(maxDepth('4'), maxDepth('5')) + 1
= 1 + 1    = 2
            /   \
          /     \
        /       \
      /         \
    maxDepth('4') = 1     maxDepth('5') = 1
  
```

#### Implementation:

C

```

#include<stdio.h>
#include<stdlib.h>

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};
  
```

```
/* Compute the "maxDepth" of a tree -- the number of
   nodes along the longest path from the root node
   down to the farthest leaf node.*/
int maxDepth(struct node* node)
{
    if (node==NULL)
        return 0;
    else
    {
        /* compute the depth of each subtree */
        int lDepth = maxDepth(node->left);
        int rDepth = maxDepth(node->right);

        /* use the larger one */
        if (lDepth > rDepth)
            return(lDepth+1);
        else return(rDepth+1);
    }
}

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* node = (struct node*)
                           malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;

    return(node);
}

int main()
{
    struct node *root = newNode(1);

    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);

    printf("Hight of tree is %d", maxDepth(root));

    getchar();
    return 0;
}
```

**Java**

```
// Java program to find height of tree

// A binary tree node
class Node
{
    int data;
    Node left, right;

    Node(int item)
    {
        data = item;
        left = right = null;
    }
}

class BinaryTree
{
    Node root;

    /* Compute the "maxDepth" of a tree -- the number of
       nodes along the longest path from the root node
       down to the farthest leaf node.*/
    int maxDepth(Node node)
    {
        if (node == null)
            return 0;
        else
        {
            /* compute the depth of each subtree */
            int lDepth = maxDepth(node.left);
            int rDepth = maxDepth(node.right);

            /* use the larger one */
            if (lDepth > rDepth)
                return (lDepth + 1);
            else
                return (rDepth + 1);
        }
    }

    /* Driver program to test above functions */
    public static void main(String[] args)
    {
        BinaryTree tree = new BinaryTree();

        tree.root = new Node(1);
```

```
tree.root.left = new Node(2);
tree.root.right = new Node(3);
tree.root.left.left = new Node(4);
tree.root.left.right = new Node(5);

System.out.println("Height of tree is : " +
                     tree.maxDepth(tree.root));
}

}

// This code has been contributed by Mayank Jaiswal(mayank_24)
```

### Python

```
# Python program to find the maximum depth of tree

# A binary tree node
class Node:

    # Constructor to create a new node
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

    # Compute the "maxDepth" of a tree -- the number of nodes
    # along the longest path from the root node down to the
    # farthest leaf node
    def maxDepth(node):
        if node is None:
            return 0;

        else :

            # Compute the depth of each subtree
            lDepth = maxDepth(node.left)
            rDepth = maxDepth(node.right)

            # Use the larger one
            if (lDepth > rDepth):
                return lDepth+1
            else:
                return rDepth+1

# Driver program to test above function
root = Node(1)
root.left = Node(2)
```

```
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)

print "Height of tree is %d" %(maxDepth(root))

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

**Time Complexity:**  $O(n)$  (Please see our post [Tree Traversal](#)for details)

**References:**

<http://cslibrary.stanford.edu/110/BinaryTrees.html>

**Improved By :** [coduitachi](#)

## Source

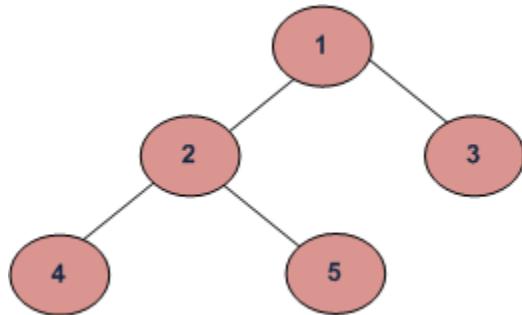
<https://www.geeksforgeeks.org/write-a-c-program-to-find-the-maximum-depth-or-height-of-a-tree/>

## Chapter 438

# Write a program to Calculate Size of a tree | Recursion

Write a program to Calculate Size of a tree | Recursion - GeeksforGeeks

Size of a tree is the number of elements present in the tree. Size of the below tree is 5.



Example Tree

Size() function recursively calculates the size of a tree. It works as follows:

Size of a tree = Size of left subtree + 1 + Size of right subtree.

**Algorithm:**

```
size(tree)
1. If tree is empty then return 0
2. Else
    (a) Get the size of left subtree recursively i.e., call
        size( tree->left-subtree)
    (b) Get the size of right subtree recursively i.e., call
        size( tree->right-subtree)
```

(c) Calculate size of the tree as following:  
tree\_size = size(left-subtree) + size(right-subtree) + 1  
(d) Return tree\_size

C

```
#include <stdio.h>
#include <stdlib.h>

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* node = (struct node*)
                           malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;

    return(node);
}

/* Computes the number of nodes in a tree. */
int size(struct node* node)
{
    if (node==NULL)
        return 0;
    else
        return(size(node->left) + 1 + size(node->right));
}

/* Driver program to test size function*/
int main()
{
    struct node *root = newNode(1);
    root->left      = newNode(2);
    root->right     = newNode(3);
    root->left->left = newNode(4);
```

```
root->left->right = newNode(5);

printf("Size of the tree is %d", size(root));
getchar();
return 0;
}
```

**Java**

```
// A recursive Java program to calculate the size of the tree

/* Class containing left and right child of current
   node and key value*/
class Node
{
    int data;
    Node left, right;

    public Node(int item)
    {
        data = item;
        left = right = null;
    }
}

/* Class to find size of Binary Tree */
class BinaryTree
{
    Node root;

    /* Given a binary tree. Print its nodes in level order
       using array for implementing queue */
    int size()
    {
        return size(root);
    }

    /* computes number of nodes in tree */
    int size(Node node)
    {
        if (node == null)
            return 0;
        else
            return(size(node.left) + 1 + size(node.right));
    }

    public static void main(String args[])
    {
```

```
/* creating a binary tree and entering the nodes */
BinaryTree tree = new BinaryTree();
tree.root = new Node(1);
tree.root.left = new Node(2);
tree.root.right = new Node(3);
tree.root.left.left = new Node(4);
tree.root.left.right = new Node(5);

System.out.println("The size of binary tree is : "
+ tree.size());
}
```

### Python

```
# Python Program to find the size of binary tree

# A binary tree node
class Node:

    # Constructor to create a new node
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

    # Computes the number of nodes in tree
    def size(node):
        if node is None:
            return 0
        else:
            return (size(node.left)+ 1 + size(node.right))

# Driver program to test above function
root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)

print "Size of the tree is %d" %(size(root))

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

Output:

Size of the tree is 5

**Time & Space Complexities:** Since this program is similar to traversal of tree, time and space complexities will be same as Tree traversal (Please see our [Tree Traversal](#) post for details)

## Source

<https://www.geeksforgeeks.org/write-a-c-program-to-calculate-size-of-a-tree/>

## Chapter 439

# XOR of numbers that appeared even number of times in given Range

XOR of numbers that appeared even number of times in given Range - GeeksforGeeks

Given an array of numbers of size N and Q queries. Each query or a range can be represented by L (LeftIndex) and R(RightIndex). Find the XOR-sum of the numbers that appeared even number of times in the given range.

**Prerequisite :** [Queries for number of distinct numbers in given range](#) | [Segment Tree for range query](#)

Examples :

```
Input : arr[] = { 1, 2, 1, 3, 3, 2, 3 }
        Q = 5
        L = 3,  R = 6
        L = 3,  R = 4
        L = 0,  R = 2
        L = 0,  R = 6
        L = 0,  R = 4
Output : 0
         3
         1
         3
         2
```

### Explanation of above example:

In Query 1, there are no numbers which appeared even number of times.  
Hence the XOR-sum is 0.

In Query 2, {3} appeared even number of times. XOR-sum is 3.

In Query 3, {1} appeared even number of times. XOR-sum is 1.

In Query 4, {1, 2} appeared even number of times. XOR-sum is 1 xor 2 = 3.

In Query 5, {1, 3} appeared even number of times. XOR-sum is 1 xor 3 = 2.

Segment Trees or Binary Indexed Trees can be used to solve this problem efficiently.

**Approach :**

Firstly, it is easy to note that the answer for the query is the XOR-sum of **all** elements in the query range xor-ed with XOR-sum of **distinct** elements in the query range (since taking XOR of an element with itself results into a null value). Find the XOR-sum of all numbers in query range using prefix XOR-sums.

**To find the XOR-sum of distinct elements in range : Number of distinct elements in a subarray of given range.**

Now, returning back to our main problem, just change the assignment  $\text{BIT}[i] = 1$  to  $\text{BIT}[i] = \text{arr}_i$  and count the XOR-sum instead of sum.

**Below is the implementation using Binary Indexed Trees in CPP**

```
// CPP Program to Find the XOR-sum
// of elements that appeared even
// number of times within a range
#include <bits/stdc++.h>
using namespace std;

/* structure to store queries
   L --> Left Bound of Query
   R --> Right Bound of Query
   idx --> Query Number */
struct que {
    int L, R, idx;
};

// cmp function to sort queries
// according to R
bool cmp(que a, que b)
{
    if (a.R != b.R)
        return a.R < b.R;
    else
        return a.L < b.L;
}

/* N --> Number of elements present in
   input array. BIT[0..N] --> Array that
   represents Binary Indexed Tree*/

// Returns XOR-sum of arr[0..index]. This
// function assumes that the array is
```

```
// preprocessed and partial sums of array
// elements are stored in BIT[] .
int getSum(int BIT[], int index)
{
    // Initialize result
    int xorSum = 0;

    // index in BITree[] is 1 more than
    // the index in arr[]
    index = index + 1;

    // Traverse ancestors of BIT[index]
    while (index > 0)
    {
        // Take XOR of current element
        // of BIT to xorSum
        xorSum ^= BIT[index];

        // Move index to parent node
        // in getSum View
        index -= index & (-index);
    }
    return xorSum;
}

// Updates a node in Binary Index Tree
// (BIT) at given index in BIT. The
// given value 'val' is xored to BIT[i]
// and all of its ancestors in tree.
void updateBIT(int BIT[], int N,
               int index, int val)
{
    // index in BITree[] is 1 more than
    // the index in arr[]
    index = index + 1;

    // Traverse all ancestors and
    // take xor with 'val'
    while (index <= N)
    {
        // Take xor with 'val' to
        // current node of BIT
        BIT[index] ^= val;

        // Update index to that of
        // parent in update View
        index += index & (-index);
    }
}
```

```
}  
  
// Constructs and returns a Binary Indexed  
// Tree for given array of size N.  
int* constructBITree(int arr[], int N)  
{  
    // Create and initialize BITree[] as 0  
    int* BIT = new int[N + 1];  
  
    for (int i = 1; i <= N; i++)  
        BIT[i] = 0;  
  
    return BIT;  
}  
  
// Function to answer the Queries  
void answeringQueries(int arr[], int N,  
                      que queries[], int Q, int BIT[])  
{  
    // Creating an array to calculate  
    // prefix XOR sums  
    int* prefixXOR = new int[N + 1];  
  
    // map for coordinate compression  
    // as numbers can be very large but we  
    // have limited space  
    map<int, int> mp;  
  
    for (int i = 0; i < N; i++) {  
  
        // If A[i] has not appeared yet  
        if (!mp[arr[i]])  
            mp[arr[i]] = i;  
  
        // calculate prefixXOR sums  
        if (i == 0)  
            prefixXOR[i] = arr[i];  
        else  
            prefixXOR[i] =  
                prefixXOR[i - 1] ^ arr[i];  
    }  
  
    // Creating an array to store the  
    // last occurrence of arr[i]  
    int lastOcc[10000001];  
    memset(lastOcc, -1, sizeof(lastOcc));  
  
    // sort the queries according to comparator
```

```

sort(queries, queries + Q, cmp);

// answer for each query
int res[Q];

// Query Counter
int j = 0;

for (int i = 0; i < Q; i++)
{
    while (j <= queries[i].R)
    {
        // If last visit is not -1 update
        // arr[j] to set null by taking
        // xor with itself at the idx
        // equal lastOcc[mp[arr[j]]]
        if (lastOcc[mp[arr[j]]] != -1)
            updateBIT(BIT, N,
                      lastOcc[mp[arr[j]]], arr[j]);

        // Setting lastOcc[mp[arr[j]]] as j and
        // updating the BIT array accordingly
        updateBIT(BIT, N, j, arr[j]);
        lastOcc[mp[arr[j]]] = j;
        j++;
    }

    // get the XOR-sum of all elements within
    // range using precomputed prefix XORsums
    int allXOR = prefixXOR[queries[i].R] ^
                  prefixXOR[queries[i].L - 1];

    // get the XOR-sum of distinct elements
    // within range using BIT query function
    int distinctXOR = getSum(BIT, queries[i].R) ^
                      getSum(BIT, queries[i].L - 1);

    // store the final answer at the numbered query
    res[queries[i].idx] = allXOR ^ distinctXOR;
}

// Output the result
for (int i = 0; i < Q; i++)
    cout << res[i] << endl;
}

// Driver program to test above functions
int main()

```

```
{  
    int arr[] = { 1, 2, 1, 3, 3, 2, 3 };  
    int N = sizeof(arr) / sizeof(arr[0]);  
  
    int* BIT = constructBITree(arr, N);  
  
    // structure of array for queries  
    que queries[5];  
  
    // Intializing values (L, R, idx) to queries  
    queries[0].L = 3;  
    queries[0].R = 6, queries[0].idx = 0;  
    queries[1].L = 3;  
    queries[1].R = 4, queries[1].idx = 1;  
    queries[2].L = 0;  
    queries[2].R = 2, queries[2].idx = 2;  
    queries[3].L = 0;  
    queries[3].R = 6, queries[3].idx = 3;  
    queries[4].L = 0;  
    queries[4].R = 4, queries[4].idx = 4;  
  
    int Q = sizeof(queries) / sizeof(queries[0]);  
  
    // answer Queries  
    answeringQueries(arr, N, queries, Q, BIT);  
  
    return 0;  
}
```

**Output:**

```
0  
3  
1  
3  
2
```

**Time Complexity:**  $O(Q * \log(N))$ , where  $N$  is the size of array,  $Q$  is the total number of queries.

## Source

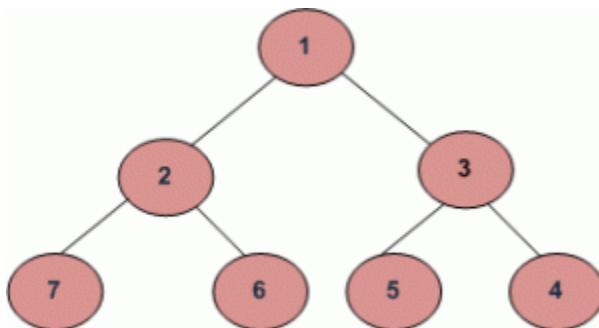
<https://www.geeksforgeeks.org/xor-numbers-appeared-even-number-times-given-range/>

## Chapter 440

# ZigZag Tree Traversal

ZigZag Tree Traversal - GeeksforGeeks

Write a function to print ZigZag order traversal of a binary tree. For the below binary tree the zigzag order traversal will be **1 3 2 7 6 5 4**



This problem can be solved using two stacks. Assume the two stacks are current: **currentlevel** and **nextlevel**. We would also need a variable to keep track of the current level order (whether it is left to right or right to left). We pop from the currentlevel stack and print the nodes value. Whenever the current level order is from left to right, push the nodes left child, then its right child to the stack nextlevel. Since a stack is a LIFO(Last-In-First\_out) structure, next time when nodes are popped off nextlevel, it will be in the reverse order. On the other hand, when the current level order is from right to left, we would push the nodes right child first, then its left child. Finally, do-not forget to swap those two stacks at the end of each level(i.e., when current level is empty)

*Below is the implementation of the above approach:*

C++

```
// C++ implementation of a O(n) time method for
// Zigzag order traversal
#include <iostream>
```

```
#include <stack>
using namespace std;

// Binary Tree node
struct Node {
    int data;
    struct Node *left, *right;
};

// function to print the zigzag traversal
void zigzagtraversal(struct Node* root)
{
    // if null then return
    if (!root)
        return;

    // declare two stacks
    stack<struct Node*> currentlevel;
    stack<struct Node*> nextlevel;

    // push the root
    currentlevel.push(root);

    // check if stack is empty
    bool lefttoright = true;
    while (!currentlevel.empty()) {

        // pop out of stack
        struct Node* temp = currentlevel.top();
        currentlevel.pop();

        // if not null
        if (temp) {

            // print the data in it
            cout << temp->data << " ";

            // store data according to current
            // order.
            if (lefttoright) {
                if (temp->left)
                    nextlevel.push(temp->left);
                if (temp->right)
                    nextlevel.push(temp->right);
            }
            else {
                if (temp->right)
                    nextlevel.push(temp->right);
            }
        }
    }
}
```

```
        if (temp->left)
            nextlevel.push(temp->left);
    }
}

if (currentlevel.empty()) {
    lefttoright = !lefttoright;
    swap(currentlevel, nextlevel);
}
}

// A utility function to create a new node
struct Node* newNode(int data)
{
    struct Node* node = new struct Node;
    node->data = data;
    node->left = node->right = NULL;
    return (node);
}

// driver program to test the above function
int main()
{
    // create tree
    struct Node* root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(7);
    root->left->right = newNode(6);
    root->right->left = newNode(5);
    root->right->right = newNode(4);
    cout << "ZigZag Order traversal of binary tree is \n";

    zigzagtraversal(root);

    return 0;
}
```

### Java

```
// Java implementation of a O(n) time
// method for Zigzag order traversal
import java.util.*;

// Binary Tree node
class Node
{
```

```
int data;
Node leftChild;
Node rightChild;
Node(int data)
{
    this.data = data;
}
}

class BinaryTree {
Node rootNode;

// function to print the
// zigzag traversal
void printZigZagTraversal() {

    // if null then return
    if (rootNode == null) {
        return;
    }

    // declare two stacks
    Stack<Node> currentLevel = new Stack<>();
    Stack<Node> nextLevel = new Stack<>();

    // push the root
    currentLevel.push(rootNode);
    boolean leftToRight = true;

    // check if stack is empty
    while (!currentLevel.isEmpty()) {

        // pop out of stack
        Node node = currentLevel.pop();

        // print the data in it
        System.out.print(node.data + " ");

        // store data according to current
        // order.
        if (leftToRight) {
            if (node.leftChild != null) {
                nextLevel.push(node.leftChild);
            }
        }

        if (node.rightChild != null) {
            nextLevel.push(node.rightChild);
        }
    }
}
```

```
        }
    else {
        if (node.rightChild != null) {
            nextLevel.push(node.rightChild);
        }

        if (node.leftChild != null) {
            nextLevel.push(node.leftChild);
        }
    }

    if (currentLevel.isEmpty()) {
        leftToRight = !leftToRight;
        Stack<Node> temp = currentLevel;
        currentLevel = nextLevel;
        nextLevel = temp;
    }
}
}

public class zigZagTreeTraversal {

// driver program to test the above function
public static void main(String[] args)
{
    BinaryTree tree = new BinaryTree();
    tree.rootNode = new Node(1);
    tree.rootNode.leftChild = new Node(2);
    tree.rootNode.rightChild = new Node(3);
    tree.rootNode.leftChild.leftChild = new Node(7);
    tree.rootNode.leftChild.rightChild = new Node(6);
    tree.rootNode.rightChild.leftChild = new Node(5);
    tree.rootNode.rightChild.rightChild = new Node(4);

    System.out.println("ZigZag Order traversal of binary tree is");
    tree.printZigZagTraversal();
}
}

// This Code is contributed by Harikrishnan Rajan.
```

### Python3

```
# Python Program to print zigzag traversal
# of binary tree

# Binary tree node
```

```
class Node:  
    # Constructor to create a new node  
    def __init__(self, data):  
        self.data = data  
        self.left = self.right = None  
  
# function to print zigzag traversal of  
# binary tree  
def zigzagtraversal(root):  
  
    # Base Case  
    if root is None:  
        return  
  
    # Create two stacks to store current  
    # and next level  
    currentLevel = []  
    nextLevel = []  
  
    # if ltr is true push nodes from  
    # left to right otherwise from  
    # right to left  
    ltr = True  
  
    # append root to currentlevel stack  
    currentLevel.append(root)  
  
    # Check if stack is empty  
    while len(currentLevel) > 0:  
        # pop from stack  
        temp = currentLevel.pop(-1)  
        # print the data  
        print(temp.data, " ", end="")  
  
        if ltr:  
            # if ltr is true push left  
            # before right  
            if temp.left:  
                nextLevel.append(temp.left)  
            if temp.right:  
                nextLevel.append(temp.right)  
        else:  
            # else push right before left  
            if temp.right:  
                nextLevel.append(temp.right)  
            if temp.left:  
                nextLevel.append(temp.left)
```

```
if len(currentLevel) == 0:  
    # reverse ltr to push node in  
    # opposite order  
    ltr = not ltr  
    # swapping of stacks  
    currentLevel, nextLevel = nextLevel, currentLevel  
  
# Driver program to check above function  
root = Node(1)  
root.left = Node(2)  
root.right = Node(3)  
root.left.left = Node(7)  
root.left.right = Node(6)  
root.right.left = Node(5)  
root.right.right = Node(4)  
print("Zigzag Order traversal of binary tree is")  
zigzagtraversal(root)  
  
# This code is contributed by Shweta Singh
```

**Output:**

```
ZigZag Order traversal of binary tree is  
1 3 2 7 6 5 4
```

**Time Complexity:**  $O(n)$   
**Space Complexity:**  $O(n)+(n)=O(n)$

**Improved By :** shweta44

**Source**

<https://www.geeksforgeeks.org/zigzag-tree-traversal/>

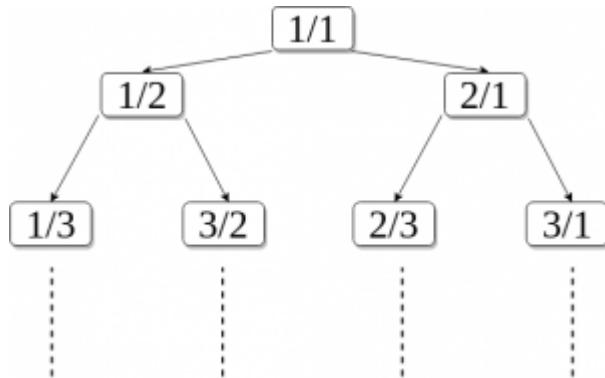
## Chapter 441

# nth Rational number in Calkin-Wilf sequence

nth Rational number in Calkin-Wilf sequence - GeeksforGeeks

### What is Calkin Wilf Sequence?

A Calkin-Wilf tree (or sequence) is a special binary tree which is obtained by starting with the fraction  $1/1$  and adding  $a/(a+b)$  and  $(a+b)/b$  iteratively below each fraction  $a/b$ . This tree generates every rational number. Writing out the terms in a sequence gives  $1/1, 1/2, 2/1, 1/3, 3/2, 2/3, 3/1, 1/4, 4/3, 3/5, 5/2, 2/5, 5/3, 3/4, 4/1, \dots$ . The sequence has the property that each denominator is the next numerator.



The image above is the Calkin-Wilf Tree where all the rational numbers are listed. The children of a node  $a/b$  is calculated as  $a/(a+b)$  and  $(a+b)/b$ .

**The task is to find the nth rational number in breadth first traversal of this tree.**  
**Examples:**

Input : 13  
Output : [5, 3]

```
Input : 5
Output : [3, 2]
```

Explanation: This tree is a Perfect Binary Search tree and we need  $\text{floor}(\log(n))$  steps to compute nth rational number. The concept is similar to searching in a binary search tree. Given n we keep dividing it by 2 until we get 0. We return fraction at each stage in the following manner:-

```
if n%2 == 0
    update frac[1]+=frac[0]
else
    update frac[0]+=frac[1]
```

**Below is the program to find the nth number in Calkin Wilf sequence:**

C++

```
// C++ program to find the
// nth number in Calkin
// Wilf sequence:
# include<bits/stdc++.h>
using namespace std;

int frac[] = {0, 1};

// returns 1x2 int array
// which contains the nth
// rational number
int nthRational(int n)
{
    if (n > 0)
        nthRational(n / 2);

    // ~n&1 is equivalent to
    // !n%2?1:0 and n&1 is
    // equivalent to n%2
    frac[~n & 1] += frac[n & 1];
}

// Driver Code
int main()
{
    int n = 13; // testing for n=13

    // converting array
    // to string format
```

```
    nthRational(n);
    cout << "[" << frac[0] << ","
        << frac[1] << "]" << endl;
    return 0;
}

// This code is contributed
// by Harshit Saini
```

**Java**

```
// Java program to find the nth number
// in Calkin Wilf sequence:
import java.util.*;

public class GFG {
    static int[] frac = { 0, 1 };

    public static void main(String args[])
    {
        int n = 13; // testing for n=13

        // converting array to string format
        System.out.println(Arrays.toString(nthRational(n)));
    }

    // returns 1x2 int array which
    // contains the nth rational number
    static int[] nthRational(int n)
    {
        if (n > 0)
            nthRational(n / 2);

        // ~n&1 is equivalent to !n%2?1:0
        // and n&1 is equivalent to n%2
        frac[~n & 1] += frac[n & 1];

        return frac;
    }
}
```

**Python3**

```
# Python program to find
# the nth number in Calkin
# Wilf sequence:
```

```
frac = [0, 1]

# returns 1x2 int array
# which contains the nth
# rational number
def nthRational(n):
    if n > 0:
        nthRational(int(n / 2))

        # ~n&1 is equivalent to
        # !n%2?1:0 and n&1 is
        # equivalent to n%2
        frac[~n & 1] += frac[n & 1]

    return frac

# Driver code
if __name__ == "__main__":
    n = 13 # testing for n=13

    # converting array
    # to string format
    print(nthRational(n))

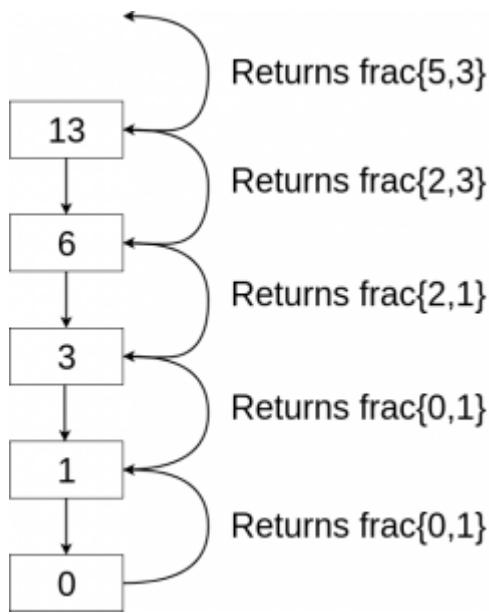
# This code is contributed
# by Harshit Saini
```

**Output:**

[5, 3]

**Explanation:**

For  $n = 13$ ,



Improved By : [Harshit Saini](#)

## Source

<https://www.geeksforgeeks.org/nth-rational-number-in-calkin-wilf-sequence/>