

# Contents

<b>1 A Programmer's approach of looking at Array vs. Linked List</b>	<b>17</b>
Source . . . . .	21
<b>2 Add 1 to a number represented as linked list</b>	<b>22</b>
Source . . . . .	27
<b>3 Add two numbers represented by linked lists   Set 1</b>	<b>28</b>
Source . . . . .	36
<b>4 Add two numbers represented by linked lists   Set 2</b>	<b>37</b>
Source . . . . .	46
<b>5 Adding two polynomials using Linked List</b>	<b>47</b>
Source . . . . .	51
<b>6 Alternate Odd and Even Nodes in a Singly Linked List</b>	<b>52</b>
Source . . . . .	59
<b>7 Alternate sorting of Linked list</b>	<b>60</b>
Source . . . . .	65
<b>8 Alternating split of a given Singly Linked List   Set 1</b>	<b>66</b>
Source . . . . .	70
<b>9 An interesting method to print reverse of a linked list</b>	<b>71</b>
Source . . . . .	74
<b>10 Applications of linked list data structure</b>	<b>75</b>
Source . . . . .	76
<b>11 Arrange consonants and vowels nodes in a linked list</b>	<b>77</b>
Source . . . . .	82
<b>12 Binary Search on Singly Linked List</b>	<b>83</b>
Source . . . . .	86
<b>13 Brent's Cycle Detection Algorithm</b>	<b>87</b>
Source . . . . .	93

<b>14 Bubble Sort On Doubly Linked List</b>	<b>94</b>
Source . . . . .	97
<b>15 C Program for Bubble Sort on Linked List</b>	<b>98</b>
Source . . . . .	101
<b>16 C Program to reverse each node value in Singly Linked List</b>	<b>102</b>
Source . . . . .	104
<b>17 Can we reverse a linked list in less than O(n)?</b>	<b>105</b>
Source . . . . .	105
<b>18 Check if a doubly linked list of characters is palindrome or not</b>	<b>106</b>
Source . . . . .	108
<b>19 Check if a linked list is Circular Linked List</b>	<b>109</b>
Source . . . . .	111
<b>20 Check if a linked list of strings forms a palindrome</b>	<b>112</b>
Source . . . . .	117
<b>21 Check if linked list is sorted (Iterative and Recursive)</b>	<b>118</b>
Source . . . . .	121
<b>22 Check linked list with a loop is palindrome or not</b>	<b>122</b>
Source . . . . .	126
<b>23 Check whether the length of given linked list is Even or Odd</b>	<b>127</b>
Source . . . . .	129
<b>24 Circular Linked List   Set 1 (Introduction and Applications)</b>	<b>130</b>
Source . . . . .	131
<b>25 Circular Linked List   Set 2 (Traversal)</b>	<b>132</b>
Source . . . . .	136
<b>26 Circular Queue   Set 2 (Circular Linked List Implementation)</b>	<b>137</b>
Source . . . . .	138
<b>27 Circular Singly Linked List   Insertion</b>	<b>139</b>
Source . . . . .	148
<b>28 Clone a linked list with next and random pointer in O(1) space</b>	<b>149</b>
Source . . . . .	153
<b>29 Clone a linked list with next and random pointer   Set 1</b>	<b>154</b>
Source . . . . .	156
<b>30 Clone a linked list with next and random pointer   Set 2</b>	<b>157</b>
Source . . . . .	163

<b>31 Compare two strings represented as linked lists</b>	<b>164</b>
Source . . . . .	169
<b>32 Construct a Maximum Sum Linked List out of two Sorted Linked Lists having some Common nodes</b>	<b>170</b>
Source . . . . .	180
<b>33 Construct a linked list from 2D matrix</b>	<b>181</b>
Source . . . . .	185
<b>34 Construct a linked list from 2D matrix (Iterative Approach)</b>	<b>186</b>
Source . . . . .	190
<b>35 Convert a Binary Tree to a Circular Doubly Link List</b>	<b>191</b>
Source . . . . .	198
<b>36 Convert a given Binary Tree to Doubly Linked List   Set 1</b>	<b>199</b>
Source . . . . .	207
<b>37 Convert a given Binary Tree to Doubly Linked List   Set 2</b>	<b>208</b>
Source . . . . .	216
<b>38 Convert a given Binary Tree to Doubly Linked List   Set 3</b>	<b>217</b>
Source . . . . .	222
<b>39 Convert a given Binary Tree to Doubly Linked List   Set 4</b>	<b>223</b>
Source . . . . .	228
<b>40 Convert singly linked list into circular linked list</b>	<b>229</b>
Source . . . . .	231
<b>41 Count nodes in Circular linked list</b>	<b>232</b>
Source . . . . .	234
<b>42 Count pairs from two linked lists whose sum is equal to a given value</b>	<b>235</b>
Source . . . . .	247
<b>43 Count pairs in a binary tree whose sum is equal to a given value x</b>	<b>248</b>
Source . . . . .	255
<b>44 Count rotations in sorted and rotated linked list</b>	<b>256</b>
Source . . . . .	258
<b>45 Count triplets in a sorted doubly linked list whose sum is equal to a given value x</b>	<b>259</b>
Source . . . . .	266
<b>46 Create a Doubly Linked List from a Ternary Tree</b>	<b>267</b>
Source . . . . .	273
<b>47 Data Structure design to perform required operations</b>	<b>274</b>

Source . . . . .	277
<b>48 Decimal Equivalent of Binary Linked List</b>	<b>278</b>
Source . . . . .	279
<b>49 Delete N nodes after M nodes of a linked list</b>	<b>280</b>
Source . . . . .	285
<b>50 Delete a Doubly Linked List node at a given position</b>	<b>286</b>
Source . . . . .	289
<b>51 Delete a Linked List node at a given position</b>	<b>290</b>
Source . . . . .	296
<b>52 Delete a given node in Linked List under given constraints</b>	<b>297</b>
Source . . . . .	302
<b>53 Delete a linked list using recursion</b>	<b>303</b>
Source . . . . .	304
<b>54 Delete a node in a Doubly Linked List</b>	<b>305</b>
Source . . . . .	313
<b>55 Delete all occurrences of a given key in a doubly linked list</b>	<b>314</b>
Source . . . . .	318
<b>56 Delete all occurrences of a given key in a linked list</b>	<b>319</b>
Source . . . . .	322
<b>57 Delete all the nodes from the doubly linked list that are greater than a given value</b>	<b>323</b>
Source . . . . .	326
<b>58 Delete all the nodes from the list that are greater than x</b>	<b>327</b>
Source . . . . .	330
<b>59 Delete alternate nodes of a Linked List</b>	<b>331</b>
Source . . . . .	336
<b>60 Delete last occurrence of an item from linked list</b>	<b>337</b>
Source . . . . .	341
<b>61 Delete middle of linked list</b>	<b>342</b>
Source . . . . .	344
<b>62 Delete multiple occurrences of key in Linked list using double pointer</b>	<b>345</b>
Source . . . . .	347
<b>63 Delete nodes which have a greater value on right side</b>	<b>348</b>
Source . . . . .	354

<b>64 Deletion from a Circular Linked List</b>	<b>355</b>
Source . . . . .	359
<b>65 Detect and Remove Loop in a Linked List</b>	<b>360</b>
Source . . . . .	383
<b>66 Detect loop in a linked list</b>	<b>384</b>
Source . . . . .	394
<b>67 Double elements and append zeros in linked list</b>	<b>395</b>
Source . . . . .	399
<b>68 Doubly Circular Linked List   Set 1 (Introduction and Insertion)</b>	<b>400</b>
Source . . . . .	408
<b>69 Doubly Circular Linked List   Set 2 (Deletion)</b>	<b>409</b>
Source . . . . .	415
<b>70 Doubly Linked List   Set 1 (Introduction and Insertion)</b>	<b>416</b>
Source . . . . .	437
<b>71 Exchange first and last nodes in Circular Linked List</b>	<b>438</b>
Source . . . . .	441
<b>72 Extract Leaves of a Binary Tree in a Doubly Linked List</b>	<b>442</b>
Source . . . . .	449
<b>73 Find Length of a Linked List (Iterative and Recursive)</b>	<b>450</b>
Source . . . . .	459
<b>74 Find a triplet from three linked lists with sum equal to a given number</b>	<b>460</b>
Source . . . . .	465
<b>75 Find common elements in three linked lists</b>	<b>466</b>
Source . . . . .	469
<b>76 Find first node of loop in a linked list</b>	<b>470</b>
Source . . . . .	474
<b>77 Find kth node from Middle towards Head of a Linked List</b>	<b>475</b>
Source . . . . .	478
<b>78 Find length of loop in linked list</b>	<b>479</b>
Source . . . . .	481
<b>79 Find middle of singly linked list Recursively</b>	<b>482</b>
Source . . . . .	484
<b>80 Find modular node in a linked list</b>	<b>485</b>
Source . . . . .	488

<b>81 Find pair for given sum in a sorted singly linked without extra space</b>	<b>489</b>
Source . . . . .	494
<b>82 Find pairs with given sum in doubly linked list</b>	<b>495</b>
Source . . . . .	498
<b>83 Find smallest and largest elements in singly linked list</b>	<b>499</b>
Source . . . . .	502
<b>84 Find the first non-repeating character from a stream of characters</b>	<b>503</b>
Source . . . . .	510
<b>85 Find the fractional (or n/k – th) node in linked list</b>	<b>511</b>
Source . . . . .	515
<b>86 Find the largest node in Doubly linked list</b>	<b>516</b>
Source . . . . .	518
<b>87 Find the middle of a given linked list in C and Java</b>	<b>519</b>
Source . . . . .	525
<b>88 Find the sum of last n nodes of the given Linked List</b>	<b>526</b>
Source . . . . .	537
<b>89 Find unique elements in linked list</b>	<b>538</b>
Source . . . . .	540
<b>90 First common element in two linked lists</b>	<b>541</b>
Source . . . . .	543
<b>91 First non-repeating in a linked list</b>	<b>544</b>
Source . . . . .	546
<b>92 Flatten a binary tree into linked list</b>	<b>547</b>
Source . . . . .	551
<b>93 Flatten a binary tree into linked list   Set-2</b>	<b>552</b>
Source . . . . .	555
<b>94 Flatten a multi-level linked list   Set 2 (Depth wise)</b>	<b>556</b>
Source . . . . .	559
<b>95 Flatten a multilevel linked list</b>	<b>560</b>
Source . . . . .	567
<b>96 Flattening a Linked List</b>	<b>568</b>
Source . . . . .	574
<b>97 Function to check if a singly linked list is palindrome</b>	<b>575</b>
Source . . . . .	588

<b>98 Generic Linked List in C</b>	<b>589</b>
Source . . . . .	591
<b>99 Given a linked list of line segments, remove middle points</b>	<b>592</b>
Source . . . . .	600
<b>100 Given a linked list which is sorted, how will you insert in sorted way</b>	<b>601</b>
Source . . . . .	607
<b>101 Given a linked list, reverse alternate nodes and append at the end</b>	<b>608</b>
Source . . . . .	615
<b>102 Given only a pointer to a node to be deleted in a singly linked list, how do you delete it?</b>	<b>616</b>
Source . . . . .	619
<b>103 Given only a pointer/reference to a node to be deleted in a singly linked list, how do you delete it?</b>	<b>620</b>
Source . . . . .	624
<b>104 Hashtables Chaining with Doubly Linked Lists</b>	<b>625</b>
Source . . . . .	629
<b>105 How does Floyd's slow and fast pointers approach work?</b>	<b>630</b>
Source . . . . .	631
<b>106 How to write C functions that modify head pointer of a Linked List?</b>	<b>632</b>
Source . . . . .	635
<b>107 Identical Linked Lists</b>	<b>636</b>
Source . . . . .	641
<b>108 Implementation of Deque using doubly linked list</b>	<b>642</b>
Source . . . . .	650
<b>109 Implementing Iterator pattern of a single Linked List</b>	<b>651</b>
Source . . . . .	656
<b>110 Implementing a Linked List in Java using Class</b>	<b>657</b>
Source . . . . .	679
<b>111 In-place Merge two linked lists without changing links of first list</b>	<b>680</b>
Source . . . . .	683
<b>112 In-place conversion of Sorted DLL to Balanced BST</b>	<b>684</b>
Source . . . . .	692
<b>113 Insert a node after the n-th node from the end</b>	<b>693</b>
Source . . . . .	698
<b>114 Insert a node at a specific position in a linked list</b>	<b>699</b>

Source . . . . .	702
<b>115 Insert a whole linked list into other at k-th position</b>	<b>703</b>
Source . . . . .	707
<b>116 Insert node into the middle of the linked list</b>	<b>708</b>
Source . . . . .	717
<b>117 Insert value in sorted way in a sorted doubly linked list</b>	<b>718</b>
Source . . . . .	721
<b>118 Insertion Sort for Singly Linked List</b>	<b>722</b>
Source . . . . .	727
<b>119 Insertion at Specific Position in a Circular Doubly Linked List</b>	<b>728</b>
Source . . . . .	733
<b>120 Insertion in Unrolled Linked List</b>	<b>734</b>
Source . . . . .	740
<b>121 Intersection of two Sorted Linked Lists</b>	<b>741</b>
Source . . . . .	748
<b>122 Iterative approach for removing middle points in a linked list of line segments</b>	<b>749</b>
Source . . . . .	752
<b>123 Iteratively Reverse a linked list using only 2 pointers (An Interesting Method)</b>	<b>753</b>
Source . . . . .	758
<b>124 Java Program for Reverse a linked list</b>	<b>759</b>
Source . . . . .	759
<b>125 Josephus Circle using circular linked list</b>	<b>764</b>
Source . . . . .	768
<b>126 Large number arithmetic using doubly linked list</b>	<b>769</b>
Source . . . . .	785
<b>127 Length of longest palindrome list in a linked list using O(1) extra space</b>	<b>786</b>
Source . . . . .	789
<b>128 Linked List Pair Sum</b>	<b>790</b>
Source . . . . .	794
<b>129 Linked List Sum of Nodes Between 0s</b>	<b>795</b>
Source . . . . .	797
<b>130 Linked List representation of Disjoint Set Data Structures</b>	<b>798</b>
Source . . . . .	803

<b>131 Linked List vs Array</b>	<b>804</b>
Source . . . . .	805
<b>132 Linked List   Set 1 (Introduction)</b>	<b>806</b>
Source . . . . .	816
<b>133 Linked List   Set 2 (Inserting a node)</b>	<b>817</b>
Source . . . . .	832
<b>134 Linked List   Set 3 (Deleting a node)</b>	<b>833</b>
Source . . . . .	839
<b>135 LinkedList in Java</b>	<b>840</b>
Source . . . . .	842
<b>136 Longest Common Prefix using Linked List</b>	<b>843</b>
Source . . . . .	846
<b>137 Longest common suffix of two linked lists</b>	<b>847</b>
Source . . . . .	850
<b>138 Longest increasing sublist in a linked list</b>	<b>851</b>
Source . . . . .	854
<b>139 Lucky alive person in a circle   Code Solution to sword puzzle</b>	<b>855</b>
Source . . . . .	857
<b>140 Majority element in a linked list</b>	<b>858</b>
Source . . . . .	862
<b>141 Make a loop at k-th position in a linked list</b>	<b>863</b>
Source . . . . .	865
<b>142 Make middle node head in a linked list</b>	<b>866</b>
Source . . . . .	871
<b>143 Maximum occurring character in a linked list</b>	<b>872</b>
Source . . . . .	875
<b>144 Memory efficient doubly linked list</b>	<b>876</b>
Source . . . . .	876
<b>145 Merge K sorted linked lists   Set 1</b>	<b>877</b>
Source . . . . .	881
<b>146 Merge Sort for Doubly Linked List</b>	<b>882</b>
Source . . . . .	890
<b>147 Merge Sort for Linked Lists</b>	<b>891</b>
Source . . . . .	898

<b>148 Merge Sort for Linked Lists in JavaScript</b>	<b>899</b>
Source . . . . .	903
<b>149 Merge a linked list into another linked list at alternate positions</b>	<b>904</b>
Source . . . . .	910
<b>150 Merge first half and reversed second half of the linked list alternatively</b>	<b>911</b>
Source . . . . .	918
<b>151 Merge k sorted linked lists   Set 2 (Using Min Heap)</b>	<b>919</b>
Source . . . . .	922
<b>152 Merge two sorted linked list without duplicates</b>	<b>923</b>
Source . . . . .	926
<b>153 Merge two sorted linked lists</b>	<b>927</b>
Source . . . . .	934
<b>154 Merge two sorted linked lists such that merged list is in reverse order</b>	<b>935</b>
Source . . . . .	941
<b>155 Merge two sorted lists (in-place)</b>	<b>942</b>
Source . . . . .	947
<b>156 Modify and Rearrange List</b>	<b>948</b>
Source . . . . .	952
<b>157 Modify contents of Linked List</b>	<b>953</b>
Source . . . . .	960
<b>158 Move all occurrences of an element to end in a linked list</b>	<b>961</b>
Source . . . . .	966
<b>159 Move all zeros to the front of the linked list</b>	<b>967</b>
Source . . . . .	970
<b>160 Move first element to end of a given Linked List</b>	<b>971</b>
Source . . . . .	973
<b>161 Move last element to front of a given Linked List</b>	<b>974</b>
Source . . . . .	978
<b>162 Multiply two numbers represented as linked lists into a third list</b>	<b>979</b>
Source . . . . .	984
<b>163 Multiply two numbers represented by Linked Lists</b>	<b>985</b>
Source . . . . .	990
<b>164 Multiply two polynomials</b>	<b>991</b>
Source . . . . .	995

<b>165 Pairwise swap elements of a given linked list</b>	<b>996</b>
Source . . . . .	.1002
<b>166 Pairwise swap elements of a given linked list by changing links</b>	<b>1003</b>
Source . . . . .	.1011
<b>167 Partitioning a linked list around a given value and If we don't care about making the elements of the list "stable"</b>	<b>1012</b>
Source . . . . .	.1014
<b>168 Partitioning a linked list around a given value and keeping the original order</b>	<b>1015</b>
Source . . . . .	.1019
<b>169 Point arbit pointer to greatest value right side node in a linked list</b>	<b>1020</b>
Source . . . . .	.1025
<b>170 Point to next higher value node in a linked list with an arbitrary pointer</b>	<b>1026</b>
Source . . . . .	.1031
<b>171 Practice questions for Linked List and Recursion</b>	<b>1032</b>
Source . . . . .	.1035
<b>172 Print Reverse a linked list using Stack</b>	<b>1036</b>
Source . . . . .	.1039
<b>173 Print alternate nodes of a linked list using recursion</b>	<b>1040</b>
Source . . . . .	.1042
<b>174 Print reverse of a Linked List without actually reversing</b>	<b>1043</b>
Source . . . . .	.1046
<b>175 Print reverse of a Linked List without extra space and modifications</b>	<b>1047</b>
Source . . . . .	.1049
<b>176 Print the alternate nodes of linked list (Iterative Method)</b>	<b>1050</b>
Source . . . . .	.1053
<b>177 Priority Queue using Linked List</b>	<b>1054</b>
Source . . . . .	.1057
<b>178 Priority Queue using doubly linked list</b>	<b>1058</b>
Source . . . . .	.1061
<b>179 Program for n'th node from the end of a Linked List</b>	<b>1062</b>
Source . . . . .	.1070
<b>180 Program to find size of Doubly Linked List</b>	<b>1071</b>
Source . . . . .	.1073
<b>181 Queue based approach for first non-repeating character in a stream</b>	<b>1074</b>

Source . . . . .	.1077
<b>182 Queue   Set 2 (Linked List Implementation)</b>	<b>1078</b>
Source . . . . .	.1083
<b>183 QuickSort on Doubly Linked List</b>	<b>1084</b>
Source . . . . .	.1091
<b>184 QuickSort on Singly Linked List</b>	<b>1092</b>
Source . . . . .	.1096
<b>185 Rearrange a Linked List in Zig-Zag fashion</b>	<b>1097</b>
Source . . . . .	.1100
<b>186 Rearrange a Linked List in Zig-Zag fashion   Set-2</b>	<b>1101</b>
Source . . . . .	.1105
<b>187 Rearrange a given linked list in-place.</b>	<b>1106</b>
Source . . . . .	.1114
<b>188 Rearrange a given list such that it consists of alternating minimum maximum elements</b>	<b>1115</b>
Source . . . . .	.1117
<b>189 Rearrange a linked list in to alternate first and last element</b>	<b>1118</b>
Source . . . . .	.1120
<b>190 Rearrange a linked list such that all even and odd positioned nodes are together</b>	<b>1121</b>
Source . . . . .	.1124
<b>191 Recursive Approach to find nth node from the end in the linked list</b>	<b>1125</b>
Source . . . . .	.1129
<b>192 Recursive approach for alternating split of Linked List</b>	<b>1130</b>
Source . . . . .	.1132
<b>193 Recursive function to delete k-th node from linked list</b>	<b>1133</b>
Source . . . . .	.1135
<b>194 Recursive insertion and traversal linked list</b>	<b>1136</b>
Source . . . . .	.1138
<b>195 Recursive selection sort for singly linked list   Swapping node links</b>	<b>1139</b>
Source . . . . .	.1143
<b>196 Recursively Reversing a linked list (A simple implementation)</b>	<b>1144</b>
Source . . . . .	.1146
<b>197 Remove all occurrences of duplicates from a sorted Linked List</b>	<b>1147</b>
Source . . . . .	.1153

<b>198 Remove duplicates from a sorted doubly linked list</b>	<b>1154</b>
Source . . . . .	.1158
<b>199 Remove duplicates from a sorted linked list</b>	<b>1159</b>
Source . . . . .	.1165
<b>200 Remove duplicates from a sorted linked list using recursion</b>	<b>1166</b>
Source . . . . .	.1168
<b>201 Remove duplicates from an unsorted doubly linked list</b>	<b>1169</b>
Source . . . . .	.1176
<b>202 Remove duplicates from an unsorted linked list</b>	<b>1177</b>
Source . . . . .	.1185
<b>203 Remove every k-th node of the linked list</b>	<b>1186</b>
Source . . . . .	.1189
<b>204 Replace nodes with duplicates in linked list</b>	<b>1190</b>
Source . . . . .	.1193
<b>205 Reverse a Doubly Linked List</b>	<b>1194</b>
Source . . . . .	.1200
<b>206 Reverse a Doubly Linked List   Set 4 (Swapping Data)</b>	<b>1201</b>
Source . . . . .	.1204
<b>207 Reverse a Doubly Linked List   Set-2</b>	<b>1205</b>
Source . . . . .	.1208
<b>208 Reverse a Doubly linked list using recursion</b>	<b>1209</b>
Source . . . . .	.1212
<b>209 Reverse a Linked List according to its Size</b>	<b>1213</b>
Source . . . . .	.1216
<b>210 Reverse a Linked List in groups of given size   Set 1</b>	<b>1217</b>
Source . . . . .	.1223
<b>211 Reverse a Linked List in groups of given size   Set 2</b>	<b>1224</b>
Source . . . . .	.1227
<b>212 Reverse a circular linked list</b>	<b>1228</b>
Source . . . . .	.1230
<b>213 Reverse a doubly circular linked list</b>	<b>1231</b>
Source . . . . .	.1235
<b>214 Reverse a doubly linked list in groups of given size</b>	<b>1236</b>
Source . . . . .	.1239

<b>215 Reverse a linked list</b>	<b>1240</b>
Source . . . . .	.1255
<b>216 Reverse a stack without using extra space in O(n)</b>	<b>1256</b>
Source . . . . .	.1260
<b>217 Reverse a sublist of linked list</b>	<b>1261</b>
Source . . . . .	.1264
<b>218 Reverse alternate K nodes in a Singly Linked List</b>	<b>1265</b>
Source . . . . .	.1270
<b>219 Reverse each word in a linked list node</b>	<b>1271</b>
Source . . . . .	.1273
<b>220 Reverse first K elements of given linked list</b>	<b>1274</b>
Source . . . . .	.1278
<b>221 Reverse nodes of a linked list without affecting the special characters</b>	<b>1279</b>
Source . . . . .	.1285
<b>222 Rotate Doubly linked list by N nodes</b>	<b>1286</b>
Source . . . . .	.1290
<b>223 Rotate Linked List block wise</b>	<b>1291</b>
Source . . . . .	.1294
<b>224 Rotate a Linked List</b>	<b>1295</b>
Source . . . . .	.1302
<b>225 Search an element in a Linked List (Iterative and Recursive)</b>	<b>1303</b>
Source . . . . .	.1313
<b>226 Segregate even and odd nodes in a Linked List</b>	<b>1314</b>
Source . . . . .	.1326
<b>227 Segregate even and odd nodes in a Linked List using Deque</b>	<b>1327</b>
Source . . . . .	.1330
<b>228 Select a Random Node from a Singly Linked List</b>	<b>1331</b>
Source . . . . .	.1337
<b>229 Self Organizing List : Count Method</b>	<b>1338</b>
Source . . . . .	.1343
<b>230 Self Organizing List : Move to Front Method</b>	<b>1344</b>
Source . . . . .	.1348
<b>231 Skip List   Set 2 (Insertion)</b>	<b>1349</b>
Source . . . . .	.1358

<b>232 Skip List   Set 3 (Searching and Deletion)</b>	<b>1359</b>
Source . . . . .	.1372
<b>233 Sort a k sorted doubly linked list</b>	<b>1373</b>
Source . . . . .	.1377
<b>234 Sort a linked list of 0s, 1s and 2s</b>	<b>1378</b>
Source . . . . .	.1385
<b>235 Sort a linked list of 0s, 1s and 2s by changing links</b>	<b>1386</b>
Source . . . . .	.1389
<b>236 Sort a linked list that is sorted alternating ascending and descending orders?</b>	<b>1390</b>
Source . . . . .	.1399
<b>237 Sort linked list which is already sorted on absolute values</b>	<b>1400</b>
Source . . . . .	.1405
<b>238 Sort the bitonic doubly linked list</b>	<b>1406</b>
Source . . . . .	.1410
<b>239 Sort the bitonic doubly linked list   Set-2</b>	<b>1411</b>
Source . . . . .	.1415
<b>240 Sort the linked list in the order of elements appearing in the array</b>	<b>1416</b>
Source . . . . .	.1419
<b>241 Sorted Linked List to Balanced BST</b>	<b>1420</b>
Source . . . . .	.1428
<b>242 Sorted insert for circular linked list</b>	<b>1429</b>
Source . . . . .	.1436
<b>243 Sorted insert in a doubly linked list with head and tail pointers</b>	<b>1437</b>
Source . . . . .	.1440
<b>244 Sorted merge of two sorted doubly circular linked lists</b>	<b>1441</b>
Source . . . . .	.1445
<b>245 Split a Circular Linked List into two halves</b>	<b>1446</b>
Source . . . . .	.1454
<b>246 Squareroot(n)-th node in a Linked List</b>	<b>1455</b>
Source . . . . .	.1458
<b>247 Stack Data Structure (Introduction and Program)</b>	<b>1459</b>
Source . . . . .	.1470
<b>248 Sublist Search (Search a linked list in another list)</b>	<b>1471</b>
Source . . . . .	.1474

<b>249 Subtract Two Numbers represented as Linked Lists</b>	<b>1475</b>
Source . . . . .	.1483
<b>250 Sudo Placement[1.4]   K Sum</b>	<b>1484</b>
Source . . . . .	.1490
<b>251 Sum of the nodes of a Singly Linked List</b>	<b>1491</b>
Source . . . . .	.1495
<b>252 Swap Kth node from beginning with Kth node from end in a Linked List</b>	<b>1496</b>
Source . . . . .	.1502
<b>253 Swap nodes in a linked list without swapping data</b>	<b>1503</b>
Source . . . . .	.1513
<b>254 The Great Tree-List Recursion Problem.</b>	<b>1514</b>
Source . . . . .	.1514
<b>255 Union and Intersection of two Linked Lists</b>	<b>1515</b>
Source . . . . .	.1527
<b>256 Union and Intersection of two linked lists   Set-2 (Using Merge Sort)</b>	<b>1528</b>
Source . . . . .	.1534
<b>257 Union and Intersection of two linked lists   Set-3 (Hashing)</b>	<b>1535</b>
Source . . . . .	.1539
<b>258 Unrolled Linked List   Set 1 (Introduction)</b>	<b>1540</b>
Source . . . . .	.1543
<b>259 Write a function that counts the number of times a given int occurs in a Linked List</b>	<b>1544</b>
Source . . . . .	.1551
<b>260 Write a function to delete a Linked List</b>	<b>1552</b>
Source . . . . .	.1556
<b>261 Write a function to get Nth node in a Linked List</b>	<b>1557</b>
Source . . . . .	.1564
<b>262 Write a function to get the intersection point of two Linked Lists.</b>	<b>1565</b>
Source . . . . .	.1572
<b>263 XOR Linked List – A Memory Efficient Doubly Linked List   Set 1</b>	<b>1573</b>
Source . . . . .	.1574
<b>264 XOR Linked List – A Memory Efficient Doubly Linked List   Set 2</b>	<b>1575</b>
Source . . . . .	.1578

# Chapter 1

## A Programmer's approach of looking at Array vs. Linked List

A Programmer's approach of looking at Array vs. Linked List - GeeksforGeeks

In general, array is considered a data structure for which size is fixed at the compile time and array memory is allocated either from Data section (e.g. global array) or Stack section (e.g. local array).

Similarly, linked list is considered a data structure for which size is not fixed and memory is allocated from Heap section (e.g. using malloc() etc.) as and when needed. In this sense, array is taken as a static data structure (residing in Data or Stack section) while linked list is taken as a dynamic data structure (residing in Heap section). Memory representation of array and linked list can be visualized as follows:

An array of 4 elements (integer type) which have been initialized with 1, 2, 3 and 4. Suppose, these elements are allocated at memory addresses 0x100, 0x104, 0x108 and 0x10C respectively.

[(1)]	[(2)]	[(3)]	[(4)]
0x100	0x104	0x108	0x10C

A linked list with 4 nodes where each node has integer as data and these data are initialized with 1, 2, 3 and 4. Suppose, these nodes are allocated via malloc() and memory allocated for them is 0x200, 0x308, 0x404 and 0x20B respectively.

[(1), 0x308]	[(2), 0x404]	[(3), 0x20B]	[(4), NULL]
0x200	0x308	0x404	0x20B

Anyone with even little understanding of array and linked-list might not be interested in the above explanation. I mean, it is well known that the array elements are allocated memory in

sequence i.e. contiguous memory while nodes of a linked list are non-contiguous in memory. Though it sounds trivial yet this is the most important difference between array and linked list. It should be noted that due to this contiguous versus non-contiguous memory, array and linked list are different. In fact, this difference is what makes array vs. linked list! In the following sections, we will try to explore on this very idea further.

Since elements of array are contiguous in memory, we can access any element randomly using index e.g. `intArr[3]` will access directly fourth element of the array. (For newbies, array indexing starts from 0 and that's why fourth element is indexed with 3). Also, due to contiguous memory for successive elements in array, no extra information is needed to be stored in individual elements i.e. no overhead of metadata in arrays. Contrary to this, linked list nodes are non-contiguous in memory. It means that we need some mechanism to traverse or access linked list nodes. To achieve this, each node stores the location of next node and this forms the basis of the link from one node to next node. Therefore, it's called Linked list. Though storing the location of next node is overhead in linked list but it's required. Typically, we see linked list node declaration as follows:

```
struct llNode
{
    int dataInt;

    /* nextNode is the pointer to next node in linked list*/
    struct llNode *nextNode;
};
```

So array elements are contiguous in memory and therefore not requiring any metadata. And linked list nodes are non-contiguous in memory thereby requiring metadata in the form of location of next node. Apart from this difference, we can see that array could have several unused elements because memory has already been allocated. But linked list will have only the required no. of data items. All the above information about array and linked list has been mentioned in several textbooks though in different ways.

What if we need to allocate array memory from Heap section (i.e. at run time) and linked list memory from Data/Stack section. First of all, is it possible? Before that, one might ask why would someone need to do this? Now, I hope that the remaining article would make you rethink about the idea of array vs. linked-list.

Now consider the case when we need to store certain data in array (because array has the property of random access due to contiguous memory) but we don't know the total size apriori. One possibility is to allocate memory of this array from Heap at run time. For example, as follows:

*/\*At run-time, suppose we know the required size for integer array (e.g. input size from user). Say, the array size is stored in variable arrSize. Allocate this array from Heap as follows\*/*

```
int * dynArr = (int *)malloc(sizeof(int)*arrSize);
```

Though the memory of this array is allocated from Heap, the elements can still be accessed via index mechanism e.g. `dynArr[i]`. Basically, based on the programming problem, we have

combined one benefit of array (i.e. random access of elements) and one benefit of linked list (i.e. delaying the memory allocation till run time and allocating memory from Heap). Another advantage of having this type of dynamic array is that, this method of allocating array from Heap at run time could reduce code-size (of course, it depends on certain other factors e.g. program format etc.)

Now consider the case when we need to store data in a linked list (because no. of nodes in linked list would be equal to actual data items stored i.e. no extra space like array) but we aren't allowed to get this memory from Heap again and again for each node. This might look hypothetical situation to some folks but it's not very uncommon requirement in embedded systems. Basically, in several embedded programs, allocating memory via malloc() etc. isn't allowed due to multiple reasons. One obvious reason is performance i.e. allocating memory via malloc() is costly in terms of time complexity because your embedded program is required to be deterministic most of the times. Another reason could be module specific memory management i.e. it's possible that each module in embedded system manages its own memory. In short, if we need to perform our own memory management, instead of relying on system provided APIs of malloc() and free(), we might choose the linked list which is simulated using array. I hope that you got some idea why we might need to simulate linked list using array. Now, let us first see how this can be done. Suppose, type of a node in linked list (i.e. underlying array) is declared as follows:

```
struct sllNode
{
    int dataInt;

    /*Here, note that nextIndex stores the location of next node in
     linked list*/
    int nextIndex;
};

struct sllNode arrayLL[5];
```

If we initialize this linked list (which is actually an array), it would look as follows in memory:

[(0), -1]	[(0), -1]	[(0), -1]	[(0), -1]	[(0), -1]
0x500	0x508	0x510	0x518	0x520

The important thing to notice is that all the nodes of the linked list are contiguous in memory (each one occupying 8 bytes) and nextIndex of each node is set to -1. This (i.e. -1) is done to denote that the each node of the linked list is empty as of now. This linked list is denoted by head index 0.

Now, if this linked list is updated with four elements of data part 4, 3, 2 and 1 successively, it would look as follows in memory. This linked list can be viewed as 0x500 -> 0x508 -> 0x510 -> 0x518.

[(1),1]	[(2),2]	[(3),3]	[(4),-2]	[(0),-1]
0x500	0x508	0x510	0x518	0x520

The important thing to notice is nextIndex of last node (i.e. fourth node) is set to -2. This (i.e. -2) is done to denote the end of linked list. Also, head node of the linked list is index 0. This concept of simulating linked list using array would look more interesting if we delete say second node from the above linked list. In that case, the linked list will look as follows in memory:

[(1),2]	[(0),-1]	[(3),3]	[(4),-2]	[(0),-1]
0x500	0x508	0x510	0x518	0x520

The resultant linked list is 0x500 -> 0x510 -> 0x518. Here, it should be noted that even though we have deleted second node from our linked list, the memory for this node is still there because underlying array is still there. But the nextIndex of first node now points to third node (for which index is 2).

Hopefully, the above examples would have given some idea that for the simulated linked list, we need to write our own API similar to malloc() and free() which would basically be used to insert and delete a node. Now this is what's called own memory management. Let us see how this can be done in algorithmic manner.

There are multiple ways to do so. If we take the simplistic approach of creating linked list using array, we can use the following logic. For inserting a node, traverse the underlying array and find a node whose nextIndex is -1. It means that this node is empty. Use this node as a new node. Update the data part in this new node and set the nextIndex of this node to current head node (i.e. head index) of the linked list. Finally, make the index of this new node as head index of the linked list. To visualize it, let us take an example. Suppose the linked list is as follows where head Index is 0 i.e. linked list is 0x500 -> 0x508 -> 0x518 -> 0x520

[(1),1]	[(2),3]	[(0),-1]	[(4),4]	[(5),-2]
0x500	0x508	0x510	0x518	0x520

After inserting a new node with data 8, the linked list would look as follows with head index as 2.

[(1),1]	[(2),3]	[(8),0]	[(4),4]	[(5),-2]
0x500	0x508	0x510	0x518	0x520

So the linked list nodes would be at addresses 0x510 -> 0x500 -> 0x508 -> 0x518 -> 0x520

For deleting a node, we need to set the nextIndex of the node as -1 so that the node is marked as empty node. But, before doing so, we need to make sure that the nextIndex of the previous node is updated correctly to index of next node of this node to be deleted. We

can see that we have done own memory management for creating a linked list out of the array memory. But, this is one way of inserting and deleting nodes in this linked list. It can be easily noticed that finding an empty node is not so efficient in terms of time complexity. Basically, we're searching the complete array linearly to find an empty node.

Let us see if we can optimize it further. Basically we can maintain a linked list of empty nodes as well in the same array. In that case, the linked list would be denoted by two indexes – one index would be for linked list which has the actual data values i.e. nodes which have been inserted so far and other index would for linked list of empty nodes. By doing so, whenever, we need to insert a new node in existing linked list, we can quickly find an empty node. Let us take an example:

[(4),2]	[(0),3]	[(5),5]	[(0),-1]	[(0),1]	[(9),-1]
0x500	0x508	0x510	0x518	0x520	0x528

The above linked list which is represented using two indexes (0 and 5) has two linked lists: one for actual values and another for empty nodes. The linked list with actual values has nodes at address 0x500 -> 0x510 -> 0x528 while the linked list with empty nodes has nodes at addresses 0x520 -> 0x508 -> 0x518. It can be seen that finding an empty node (i.e. writing own API similar to malloc()) should be relatively faster now because we can quickly find a free node. In real world embedded programs, a fixed chunk of memory (normally called memory pool) is allocated using malloc() only once by a module. And then the management of this memory pool (which is basically an array) is done by that module itself using techniques mentioned earlier. Sometimes, there are multiple memory pools each one having different size of node. Of course, there are several other aspects of own memory management but we'll leave it here itself. But it's worth mentioning that there are several methods by which the insertion (which requires our own memory allocation) and deletion (which requires our own memory freeing) can be improved further.

If we look carefully, it can be noticed that the Heap section of memory is basically a big array of bytes which is being managed by the underlying operating system (OS). And OS is providing this memory management service to programmers via malloc(), free() etc. Aha !!

The important take-aways from this article can be summed as follows:

- A) Array means contiguous memory. It can exist in any memory section be it Data or Stack or Heap.
- B) Linked List means non-contiguous linked memory. It can exist in any memory section be it Heap or Data or Stack.
- C) As a programmer, looking at a data structure from memory perspective could provide us better insight in choosing a particular data structure or even designing a new data structure. For example, we might create an array of linked lists etc.

## Source

<https://www.geeksforgeeks.org/programmers-approach-looking-array-vs-linked-list/>

## Chapter 2

# Add 1 to a number represented as linked list

Add 1 to a number represented as linked list - GeeksforGeeks

Number is represented in linked list such that each digit corresponds to a node in linked list. Add 1 to it. For example 1999 is represented as (1-> 9-> 9 -> 9) and adding 1 to it should change it to (2->0->0->0)

Below are the steps :

1. Reverse given linked list. For example, 1-> 9-> 9 -> 9 is converted to 9-> 9 -> 9 ->1.
2. Start traversing linked list from leftmost node and add 1 to it. If there is a carry, move to the next node. Keep moving to the next node while there is a carry.
3. Reverse modified linked list and return head.

Below is C++ implementation of above steps.

```
// C++ program to add 1 to a linked list
#include<bits/stdc++.h>

/* Linked list node */
struct Node
{
    int data;
    Node* next;
};

/* Function to create a new node with given data */
Node *newNode(int data)
{
    Node *new_node = new Node;
    new_node->data = data;
```

```
new_node->next = NULL;
return new_node;
}

/* Function to reverse the linked list */
Node *reverse(Node *head)
{
    Node * prev    = NULL;
    Node * current = head;
    Node * next;
    while (current != NULL)
    {
        next = current->next;
        current->next = prev;
        prev = current;
        current = next;
    }
    return prev;
}

/* Adds one to a linked lists and return the head
   node of resultant list */
Node *addOneUtil(Node *head)
{
    // res is head node of the resultant list
    Node* res = head;
    Node *temp, *prev = NULL;

    int carry = 1, sum;

    while (head != NULL) //while both lists exist
    {
        // Calculate value of next digit in resultant list.
        // The next digit is sum of following things
        // (i) Carry
        // (ii) Next digit of head list (if there is a
        //       next digit)
        sum = carry + head->data;

        // update carry for next calculation
        carry = (sum >= 10)? 1 : 0;

        // update sum if it is greater than 10
        sum = sum % 10;

        // Create a new node with sum as data
        head->data = sum;
    }
}
```

```
// Move head and second pointers to next nodes
temp = head;
head = head->next;
}

// if some carry is still there, add a new node to
// result list.
if (carry > 0)
    temp->next = newNode(carry);

// return head of the resultant list
return res;
}

// This function mainly uses addOneUtil().
Node* addOne(Node *head)
{
    // Reverse linked list
    head = reverse(head);

    // Add one from left to right of reversed
    // list
    head = addOneUtil(head);

    // Reverse the modified list
    return reverse(head);
}

// A utility function to print a linked list
void printList(Node *node)
{
    while (node != NULL)
    {
        printf("%d", node->data);
        node = node->next;
    }
    printf("\n");
}

/* Driver program to test above function */
int main(void)
{
    Node *head = newNode(1);
    head->next = newNode(9);
    head->next->next = newNode(9);
    head->next->next->next = newNode(9);

    printf("List is ");
}
```

```
    printList(head);

    head = addOne(head);

    printf("\nResultant list is ");
    printList(head);

    return 0;
}
```

Output:

```
List is 1999

Resultant list is 2000
```

#### **Recursive Implementation:**

We can recursively reach the last node and forward carry to previous nodes. Recursive solution doesn't require reversing of linked list. We can also use a stack in place of recursion to temporarily hold nodes.

Below is C++ implementation of recursive solution.

```
// Recursive C++ program to add 1 to a linked list
#include<bits/stdc++.h>

/* Linked list node */
struct Node
{
    int data;
    Node* next;
};

/* Function to create a new node with given data */
Node *newNode(int data)
{
    Node *new_node = new Node;
    new_node->data = data;
    new_node->next = NULL;
    return new_node;
}

// Recursively add 1 from end to beginning and returns
// carry after all nodes are processed.
int addWithCarry(Node *head)
{
    // If linked list is empty, then
```

```
// return carry
if (head == NULL)
    return 1;

// Add carry returned by next node call
int res = head->data + addWithCarry(head->next);

// Update data and return new carry
head->data = (res) % 10;
return (res) / 10;
}

// This function mainly uses addWithCarry().
Node* addOne(Node *head)
{
    // Add 1 to linked list from end to beginning
    int carry = addWithCarry(head);

    // If there is carry after processing all nodes,
    // then we need to add a new node to linked list
    if (carry)
    {
        Node *newNode = new Node;
        newNode->data = carry;
        newNode->next = head;
        return newNode; // New node becomes head now
    }

    return head;
}

// A utility function to print a linked list
void printList(Node *node)
{
    while (node != NULL)
    {
        printf("%d", node->data);
        node = node->next;
    }
    printf("\n");
}

/* Driver program to test above function */
int main(void)
{
    Node *head = newNode(1);
    head->next = newNode(9);
    head->next->next = newNode(9);
```

```
head->next->next->next = newNode(9);

printf("List is ");
printList(head);

head = addOne(head);

printf("\nResultant list is ");
printList(head);

return 0;
}
```

Output:

```
List is 1999

Resultant list is 2000
```

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/add-1-number-represented-linked-list/>

# Chapter 3

## Add two numbers represented by linked lists | Set 1

Add two numbers represented by linked lists | Set 1 - GeeksforGeeks

Given two numbers represented by two lists, write a function that returns sum list. The sum list is list representation of addition of two input numbers.

Example 1

```
Input:  
First List: 5->6->3 // represents number 365  
Second List: 8->4->2 // represents number 248  
Output  
Resultant list: 3->1->6 // represents number 613
```

Example 2

```
Input:  
First List: 7->5->9->4->6 // represents number 64957  
Second List: 8->4 // represents number 48  
Output  
Resultant list: 5->0->0->5->6 // represents number 65005
```

### Solution

Traverse both lists. One by one pick nodes of both lists and add the values. If sum is more than 10 then make carry as 1 and reduce sum. If one list has more elements than the other then consider remaining values of this list as 0. Following is the implementation of this approach.

C

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

/* Linked list node */
struct Node
{
    int data;
    struct Node* next;
};

/* Function to create a new node with given data */
struct Node *newNode(int data)
{
    struct Node *new_node = (struct Node *) malloc(sizeof(struct Node));
    new_node->data = data;
    new_node->next = NULL;
    return new_node;
}

/* Function to insert a node at the beginning of the Doubly Linked List */
void push(struct Node** head_ref, int new_data)
{
    /* allocate node */
    struct Node* new_node = newNode(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;
}

/* Adds contents of two linked lists and return the head node of resultant list */
struct Node* addTwoLists (struct Node* first, struct Node* second)
{
    struct Node* res = NULL; // res is head node of the resultant list
    struct Node *temp, *prev = NULL;
    int carry = 0, sum;

    while (first != NULL || second != NULL) //while both lists exist
    {
        // Calculate value of next digit in resultant list.
        // The next digit is sum of following things
        // (i) Carry
        // (ii) Next digit of first list (if there is a next digit)
        // (ii) Next digit of second list (if there is a next digit)
        sum = carry + (first? first->data: 0) + (second? second->data: 0);
```

```
// update carry for next calculation
carry = (sum >= 10)? 1 : 0;

// update sum if it is greater than 10
sum = sum % 10;

// Create a new node with sum as data
temp = newNode(sum);

// if this is the first node then set it as head of the resultant list
if(res == NULL)
    res = temp;
else // If this is not the first node then connect it to the rest.
    prev->next = temp;

// Set prev for next insertion
prev = temp;

// Move first and second pointers to next nodes
if (first) first = first->next;
if (second) second = second->next;
}

if (carry > 0)
    temp->next = newNode(carry);

// return head of the resultant list
return res;
}

// A utility function to print a linked list
void printList(struct Node *node)
{
    while(node != NULL)
    {
        printf("%d ", node->data);
        node = node->next;
    }
    printf("\n");
}

/* Driver program to test above function */
int main(void)
{
    struct Node* res = NULL;
    struct Node* first = NULL;
    struct Node* second = NULL;
```

```
// create first list 7->5->9->4->6
push(&first, 6);
push(&first, 4);
push(&first, 9);
push(&first, 5);
push(&first, 7);
printf("First List is ");
printList(first);

// create second list 8->4
push(&second, 4);
push(&second, 8);
printf("Second List is ");
printList(second);

// Add the two lists and see result
res = addTwoLists(first, second);
printf("Resultant list is ");
printList(res);

return 0;
}
```

**Java**

```
// Java program to add two numbers represented by linked list

class LinkedList {

    static Node head1, head2;

    static class Node {

        int data;
        Node next;

        Node(int d) {
            data = d;
            next = null;
        }
    }

    /* Adds contents of two linked lists and return the head node of resultant list */
    Node addTwoLists(Node first, Node second) {
        Node res = null; // res is head node of the resultant list
        Node prev = null;
        Node temp = null;
        int carry = 0, sum;
```

```

while (first != null || second != null) //while both lists exist
{
    // Calculate value of next digit in resultant list.
    // The next digit is sum of following things
    // (i) Carry
    // (ii) Next digit of first list (if there is a next digit)
    // (ii) Next digit of second list (if there is a next digit)
    sum = carry + (first != null ? first.data : 0)
        + (second != null ? second.data : 0);

    // update carry for next calculation
    carry = (sum >= 10) ? 1 : 0;

    // update sum if it is greater than 10
    sum = sum % 10;

    // Create a new node with sum as data
    temp = new Node(sum);

    // if this is the first node then set it as head of
    // the resultant list
    if (res == null) {
        res = temp;
    } else // If this is not the first node then connect it to the rest.
    {
        prev.next = temp;
    }

    // Set prev for next insertion
    prev = temp;

    // Move first and second pointers to next nodes
    if (first != null) {
        first = first.next;
    }
    if (second != null) {
        second = second.next;
    }
}

if (carry > 0) {
    temp.next = new Node(carry);
}

// return head of the resultant list
return res;
}

```

```
/* Utility function to print a linked list */

void printList(Node head) {
    while (head != null) {
        System.out.print(head.data + " ");
        head = head.next;
    }
    System.out.println("");
}

public static void main(String[] args) {
    LinkedList list = new LinkedList();

    // creating first list
    list.head1 = new Node(7);
    list.head1.next = new Node(5);
    list.head1.next.next = new Node(9);
    list.head1.next.next.next = new Node(4);
    list.head1.next.next.next.next = new Node(6);
    System.out.print("First List is ");
    list.printList(head1);

    // creating second list
    list.head2 = new Node(8);
    list.head2.next = new Node(4);
    System.out.print("Second List is ");
    list.printList(head2);

    // add the two lists and see the result
    Node rs = list.addTwoLists(head1, head2);
    System.out.print("Resultant List is ");
    list.printList(rs);
}
}

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

### Python

```
# Python program to add two numbers represented by linked list

# Node class
class Node:

    # Constructor to initialize the node object
    def __init__(self, data):
        self.data = data
        self.next = None
```

```
class LinkedList:

    # Function to initialize head
    def __init__(self):
        self.head = None

    # Function to insert a new node at the beginning
    def push(self, new_data):
        new_node = Node(new_data)
        new_node.next = self.head
        self.head = new_node

    # Add contents of two linked lists and return the head
    # node of resultant list
    def addTwoLists(self, first, second):
        prev = None
        temp = None
        carry = 0

        # While both list exists
        while(first is not None or second is not None):

            # Calculate the value of next digit in
            # resultant list
            # The next digit is sum of following things
            # (i) Carry
            # (ii) Next digit of first list (if ther is a
            # next digit)
            # (iii) Next digit of second list ( if there
            # is a next digit)
            fdata = 0 if first is None else first.data
            sdata = 0 if second is None else second.data
            Sum = carry + fdata + sdata

            # update carry for next calculation
            carry = 1 if Sum >= 10 else 0

            # update sum if it is greater than 10
            Sum = Sum if Sum < 10 else Sum % 10

            # Create a new node with sum as data
            temp = Node(Sum)

            # if this is the first node then set it as head
            # of resultant list
            if self.head is None:
                self.head = temp
```

```
        else :
            prev.next = temp

        # Set prev for next insertion
        prev = temp

        # Move first and second pointers to next nodes
        if first is not None:
            first = first.next
        if second is not None:
            second = second.next

        if carry > 0:
            temp.next = Node(carry)

    # Utility function to print the linked LinkedList
    def printList(self):
        temp = self.head
        while(temp):
            print temp.data,
            temp = temp.next

    # Driver program to test above function
    first = LinkedList()
    second = LinkedList()

    # Create first list
    first.push(6)
    first.push(4)
    first.push(9)
    first.push(5)
    first.push(7)
    print "First List is ",
    first.printList()

    # Create second list
    second.push(4)
    second.push(8)
    print "\nSecond List is ",
    second.printList()

    # Add the two lists and see result
    res = LinkedList()
    res.addTwoLists(first.head, second.head)
    print "\nResultant list is ",
    res.printList()

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

Output:

```
First List is 7 5 9 4 6
Second List is 8 4
Resultant list is 5 0 0 5 6
```

Time Complexity:  $O(m + n)$  where m and n are number of nodes in first and second lists respectively.

Related Article : [Add two numbers represented by linked lists | Set 2](#)

## Source

<https://www.geeksforgeeks.org/add-two-numbers-represented-by-linked-lists/>

## Chapter 4

# Add two numbers represented by linked lists | Set 2

Add two numbers represented by linked lists | Set 2 - GeeksforGeeks

Given two numbers represented by two linked lists, write a function that returns sum list. The sum list is linked list representation of addition of two input numbers. It is not allowed to modify the lists. Also, not allowed to use explicit extra space (Hint: Use Recursion).

Example

Input:

```
First List: 5->6->3 // represents number 563
Second List: 8->4->2 // represents number 842
```

Output

```
Resultant list: 1->4->0->5 // represents number 1405
```

We have discussed a solution [here](#) which is for linked lists where least significant digit is first node of lists and most significant digit is last node. In this problem, most significant node is first node and least significant digit is last node and we are not allowed to modify the lists. Recursion is used here to calculate sum from right to left.

Following are the steps.

- 1) Calculate sizes of given two linked lists.
- 2) If sizes are same, then calculate sum using recursion. Hold all nodes in recursion call stack till the rightmost node, calculate sum of rightmost nodes and forward carry to left side.
- 3) If size is not same, then follow below steps:
  - ....a) Calculate difference of sizes of two linked lists. Let the difference be *diff*
  - ....b) Move *diff* nodes ahead in the bigger linked list. Now use step 2 to calculate sum of smaller list and right sub-list (of same size) of larger list. Also, store the carry of this sum.
  - ....c) Calculate sum of the carry (calculated in previous step) with the remaining left sub-list

of larger list. Nodes of this sum are added at the beginning of sum list obtained previous step.

Following is implementation of the above approach.

## C

```
// A recursive program to add two linked lists

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

// A linked List Node
struct Node
{
    int data;
    struct Node* next;
};

typedef struct Node node;

/* A utility function to insert a node at the beginning of linked list */
void push(struct Node** head_ref, int new_data)
{
    /* allocate node */
    struct Node* new_node = (struct Node*) malloc(sizeof(struct Node));

    /* put in the data */
    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;
}

/* A utility function to print linked list */
void printList(struct Node *node)
{
    while (node != NULL)
    {
        printf("%d ", node->data);
        node = node->next;
    }
    printf("\n");
}

// A utility function to swap two pointers
```

```
void swapPointer( Node** a, Node** b )
{
    node* t = *a;
    *a = *b;
    *b = t;
}

/* A utility function to get size of linked list */
int getSize(struct Node *node)
{
    int size = 0;
    while (node != NULL)
    {
        node = node->next;
        size++;
    }
    return size;
}

// Adds two linked lists of same size represented by head1 and head2 and returns
// head of the resultant linked list. Carry is propagated while returning from
// the recursion
node* addSameSize(Node* head1, Node* head2, int* carry)
{
    // Since the function assumes linked lists are of same size,
    // check any of the two head pointers
    if (head1 == NULL)
        return NULL;

    int sum;

    // Allocate memory for sum node of current two nodes
    Node* result = (Node *)malloc(sizeof(Node));

    // Recursively add remaining nodes and get the carry
    result->next = addSameSize(head1->next, head2->next, carry);

    // add digits of current nodes and propagated carry
    sum = head1->data + head2->data + *carry;
    *carry = sum / 10;
    sum = sum % 10;

    // Assign the sum to current node of resultant list
    result->data = sum;

    return result;
}
```

```
// This function is called after the smaller list is added to the bigger
// lists's sublist of same size. Once the right sublist is added, the carry
// must be added toe left side of larger list to get the final result.
void addCarryToRemaining(Node* head1, Node* cur, int* carry, Node** result)
{
    int sum;

    // If diff. number of nodes are not traversed, add carry
    if (head1 != cur)
    {
        addCarryToRemaining(head1->next, cur, carry, result);

        sum = head1->data + *carry;
        *carry = sum/10;
        sum %= 10;

        // add this node to the front of the result
        push(result, sum);
    }
}

// The main function that adds two linked lists represented by head1 and head2.
// The sum of two lists is stored in a list referred by result
void addList(Node* head1, Node* head2, Node** result)
{
    Node *cur;

    // first list is empty
    if (head1 == NULL)
    {
        *result = head2;
        return;
    }

    // second list is empty
    else if (head2 == NULL)
    {
        *result = head1;
        return;
    }

    int size1 = getSize(head1);
    int size2 = getSize(head2) ;

    int carry = 0;

    // Add same size lists
    if (size1 == size2)
```

```
*result = addSameSize(head1, head2, &carry);

else
{
    int diff = abs(size1 - size2);

    // First list should always be larger than second list.
    // If not, swap pointers
    if (size1 < size2)
        swapPointer(&head1, &head2);

    // move diff. number of nodes in first list
    for (cur = head1; diff--; cur = cur->next);

    // get addition of same size lists
    *result = addSameSize(cur, head2, &carry);

    // get addition of remaining first list and carry
    addCarryToRemaining(head1, cur, &carry, result);
}

// if some carry is still there, add a new node to the front of
// the result list. e.g. 999 and 87
if (carry)
    push(result, carry);
}

// Driver program to test above functions
int main()
{
    Node *head1 = NULL, *head2 = NULL, *result = NULL;

    int arr1[] = {9, 9, 9};
    int arr2[] = {1, 8};

    int size1 = sizeof(arr1) / sizeof(arr1[0]);
    int size2 = sizeof(arr2) / sizeof(arr2[0]);

    // Create first list as 9->9->9
    int i;
    for (i = size1-1; i >= 0; --i)
        push(&head1, arr1[i]);

    // Create second list as 1->8
    for (i = size2-1; i >= 0; --i)
        push(&head2, arr2[i]);

    addList(head1, head2, &result);
}
```

```
    printList(result);

    return 0;
}

Java

// Java program to add two linked lists

public class linkedlistATN
{
    class node
    {
        int val;
        node next;

        public node(int val)
        {
            this.val = val;
        }
    }

    // Function to print linked list
    void printlist(node head)
    {
        while (head != null)
        {
            System.out.print(head.val + " ");
            head = head.next;
        }
    }

    node head1, head2, result;
    int carry;

    /* A utility function to push a value to linked list */
    void push(int val, int list)
    {
        node newnode = new node(val);
        if (list == 1)
        {
            newnode.next = head1;
            head1 = newnode;
        }
        else if (list == 2)
        {
            newnode.next = head2;
```

```
        head2 = newnode;
    }
else
{
    newnode.next = result;
    result = newnode;
}

}

// Adds two linked lists of same size represented by
// head1 and head2 and returns head of the resultant
// linked list. Carry is propagated while returning
// from the recursion
void addssamesize(node n, node m)
{
    // Since the function assumes linked lists are of
    // same size, check any of the two head pointers
    if (n == null)
        return;

    // Recursively add remaining nodes and get the carry
    addssamesize(n.next, m.next);

    // add digits of current nodes and propagated carry
    int sum = n.val + m.val + carry;
    carry = sum / 10;
    sum = sum % 10;

    // Push this to result list
    push(sum, 3);
}

node cur;

// This function is called after the smaller list is
// added to the bigger lists's sublist of same size.
// Once the right sublist is added, the carry must be
// added to the left side of larger list to get the
// final result.
void propogatecarry(node head1)
{
    // If diff. number of nodes are not traversed, add carry
    if (head1 != cur)
    {
        propogatecarry(head1.next);
        int sum = carry + head1.val;
```

```
    carry = sum / 10;
    sum %= 10;

    // add this node to the front of the result
    push(sum, 3);
}
}

int getsize(node head)
{
    int count = 0;
    while (head != null)
    {
        count++;
        head = head.next;
    }
    return count;
}

// The main function that adds two linked lists
// represented by head1 and head2. The sum of two
// lists is stored in a list referred by result
void addlists()
{
    // first list is empty
    if (head1 == null)
    {
        result = head2;
        return;
    }

    // first list is empty
    if (head2 == null)
    {
        result = head1;
        return;
    }

    int size1 = getsize(head1);
    int size2 = getsize(head2);

    // Add same size lists
    if (size1 == size2)
    {
        addsamesize(head1, head2);
    }
    else
    {
```

```
// First list should always be larger than second list.  
// If not, swap pointers  
if (size1 < size2)  
{  
    node temp = head1;  
    head1 = head2;  
    head2 = temp;  
}  
int diff = Math.abs(size1 - size2);  
  
// move diff. number of nodes in first list  
node temp = head1;  
while (diff-- >= 0)  
{  
    cur = temp;  
    temp = temp.next;  
}  
  
// get addition of same size lists  
addsmesame(cur, head2);  
  
// get addition of remaining first list and carry  
propogatecarry(head1);  
}  
// if some carry is still there, add a new node to  
// the front of the result list. e.g. 999 and 87  
if (carry > 0)  
    push(carry, 3);  
  
}  
  
// Driver program to test above functions  
public static void main(String args[])  
{  
    linkedlistATN list = new linkedlistATN();  
    list.head1 = null;  
    list.head2 = null;  
    list.result = null;  
    list.carry = 0;  
    int arr1[] = { 9, 9, 9 };  
    int arr2[] = { 1, 8 };  
  
    // Create first list as 9->9->9  
    for (int i = arr1.length - 1; i >= 0; --i)  
        list.push(arr1[i], 1);  
  
    // Create second list as 1->8  
    for (int i = arr2.length - 1; i >= 0; --i)
```

```
list.push(arr2[i], 2);

list.addlists();

list.printlist(list.result);
}

}

// This code is contributed by Rishabh Mahrsee
```

Output:

```
1 0 1 7
```

Time Complexity:  $O(m+n)$  where m and n are the sizes of given two linked lists.

Related Article : [Add two numbers represented by linked lists | Set 1](#)

## Source

<https://www.geeksforgeeks.org/sum-of-two-linked-lists/>

## Chapter 5

# Adding two polynomials using Linked List

Adding two polynomials using Linked List - GeeksforGeeks

Given two polynomial numbers represented by a linked list. Write a function that add these lists means add the coefficients who have same variable powers.

Example:

Input:

```
1st number = 5x^2 + 4x^1 + 2x^0
2nd number = 5x^1 + 5x^0
```

Output:

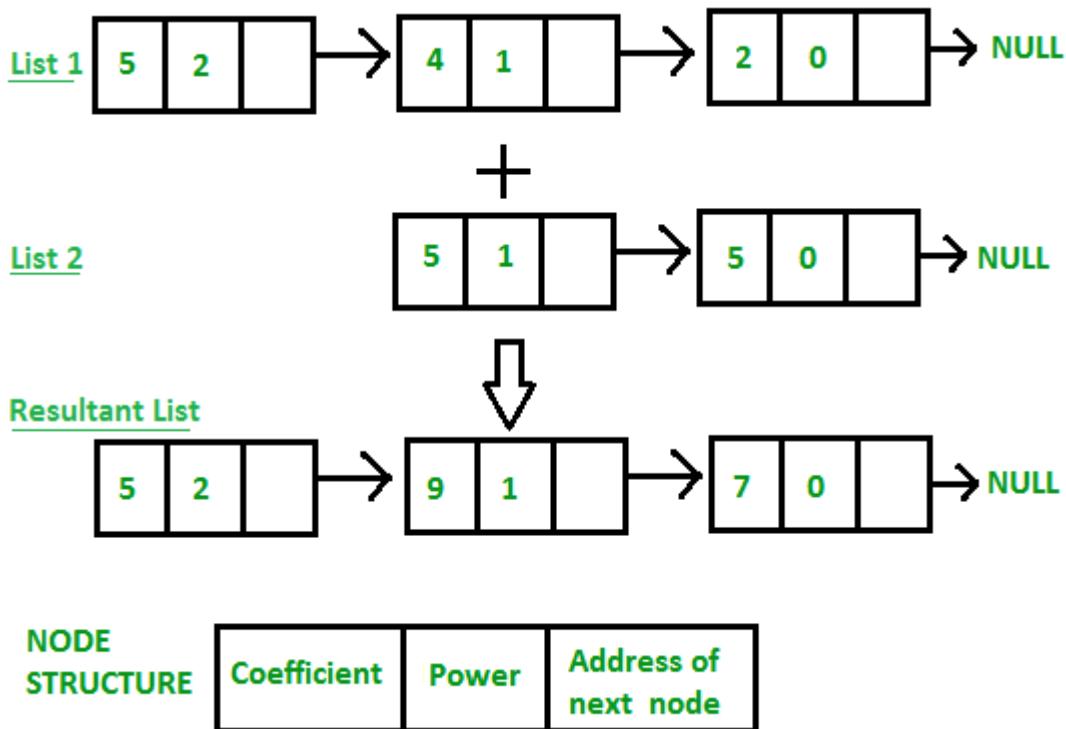
```
5x^2 + 9x^1 + 7x^0
```

Input:

```
1st number = 5x^3 + 4x^2 + 2x^0
2nd number = 5x^1 + 5x^0
```

Output:

```
5x^3 + 4x^2 + 5x^1 + 7x^0
```



```

// C++ program for addition of two polynomials
// using Linked Lists
#include<bits/stdc++.h>
using namespace std;

// Node structure containing power and coefficient of variable
struct Node
{
    int coeff;
    int pow;
    struct Node *next;
};

// Function to create new node
void create_node(int x, int y, struct Node **temp)
{
    struct Node *r, *z;
    z = *temp;
    if(z == NULL)
    {
        r =(struct Node*)malloc(sizeof(struct Node));
        r->coeff = x;

```

```

        r->pow = y;
        *temp = r;
        r->next = (struct Node*)malloc(sizeof(struct Node));
        r = r->next;
        r->next = NULL;
    }
else
{
    r->coeff = x;
    r->pow = y;
    r->next = (struct Node*)malloc(sizeof(struct Node));
    r = r->next;
    r->next = NULL;
}
}

// Function Adding two polynomial numbers
void polyadd(struct Node *poly1, struct Node *poly2, struct Node *poly)
{
while(poly1->next && poly2->next)
{
    // If power of 1st polynomial is greater then 2nd, then store 1st as it is
    // and move its pointer
    if(poly1->pow > poly2->pow)
    {
        poly->pow = poly1->pow;
        poly->coeff = poly1->coeff;
        poly1 = poly1->next;
    }

    // If power of 2nd polynomial is greater then 1st, then store 2nd as it is
    // and move its pointer
    else if(poly1->pow < poly2->pow)
    {
        poly->pow = poly2->pow;
        poly->coeff = poly2->coeff;
        poly2 = poly2->next;
    }

    // If power of both polynomial numbers is same then add their coefficients
    else
    {
        poly->pow = poly1->pow;
        poly->coeff = poly1->coeff+poly2->coeff;
        poly1 = poly1->next;
        poly2 = poly2->next;
    }
}
}

```

```

// Dynamically create new node
poly->next = (struct Node *)malloc(sizeof(struct Node));
poly = poly->next;
poly->next = NULL;
}
while(poly1->next || poly2->next)
{
    if(poly1->next)
    {
        poly->pow = poly1->pow;
        poly->coeff = poly1->coeff;
        poly1 = poly1->next;
    }
    if(poly2->next)
    {
        poly->pow = poly2->pow;
        poly->coeff = poly2->coeff;
        poly2 = poly2->next;
    }
    poly->next = (struct Node *)malloc(sizeof(struct Node));
    poly = poly->next;
    poly->next = NULL;
}
}

// Display Linked list
void show(struct Node *node)
{
while(node->next != NULL)
{
    printf("%dx^%d", node->coeff, node->pow);
    node = node->next;
    if(node->next != NULL)
        printf(" + ");
}
}

// Driver program
int main()
{
    struct Node *poly1 = NULL, *poly2 = NULL, *poly = NULL;

    // Create first list of 5x^2 + 4x^1 + 2x^0
    create_node(5,2,&poly1);
    create_node(4,1,&poly1);
    create_node(2,0,&poly1);

    // Create second list of 5x^1 + 5x^0
}

```

```
create_node(5,1,&poly2);
create_node(5,0,&poly2);

printf("1st Number: ");
show(poly1);

printf("\n2nd Number: ");
show(poly2);

poly = (struct Node *)malloc(sizeof(struct Node));

// Function add two polynomial numbers
polyadd(poly1, poly2, poly);

// Display resultant List
printf("\nAdded polynomial: ");
show(poly);

return 0;
}
```

Output:

```
1st Number: 5x^2 + 4x^1 + 2x^0
2nd Number: 5x^1 + 5x^0
Added polynomial: 5x^2 + 9x^1 + 7x^0
```

Time Complexity: O(m + n) where m and n are number of nodes in first and second lists respectively.

## Source

<https://www.geeksforgeeks.org/adding-two-polynomials-using-linked-list/>

## Chapter 6

# Alternate Odd and Even Nodes in a Singly Linked List

Alternate Odd and Even Nodes in a Singly Linked List - GeeksforGeeks

Given a singly linked list, rearrange the list so that even and odd nodes are alternate in the list.

There are two possible forms of this rearrangement. If the first data is odd, then the second node must be even. The third node must be odd and so on. Notice that another arrangement is possible where the first node is even, second odd, third even and so on.

**Examples:**

Input : 11 → 20 → 40 → 55 → 77 → 80 → NULL

Output : 11 → 20 → 55 → 40 → 77 → 80 → NULL

20, 40, 80 occur in even positions and 11, 55, 77 occur in odd positions.

Input : 10 → 1 → 2 → 3 → 5 → 6 → 7 → 8 → NULL

Output : 1 → 10 → 3 → 2 → 5 → 6 → 7 → 8 → NULL

1, 3, 5, 7 occur in odd positions and 10, 2, 6, 8 occur at even positions in the list

### Method 1 (Simple)

In this method, we create two stacks-Odd and Even. We traverse the list and when we encounter an even node in an odd position we push this node's address onto Even Stack. If we encounter an odd node in even position we push this node's address onto Odd Stack. After traversing the list, we simply pop the nodes at the top of the two stacks and exchange their data. We keep repeating this step until the stacks become empty.

Step 1: Create two stacks Odd and Even. These stacks will store the pointers to the nodes in the list

Step 2: Traverse list from start to end, using the variable current. Repeat following steps 3-4

Step 3: If current node is even and it occurs at an odd position, push this node's address to stack Even

Step 4: If current node is odd and it occurs at an even position, push this node's address to stack Odd.

[END OF TRAVERSAL]

Step 5: The size of both the stacks will be same. While both the stacks are not empty exchange the nodes at the top of the two stacks, and pop both nodes from their respective stacks.

Step 6: The List is now rearranged. STOP

C++

```
// CPP program to rearrange nodes
// as alternate odd even nodes in
// a Singly Linked List
#include <bits/stdc++.h>
using namespace std;

// Structure node
struct Node {
    int data;
    struct Node* next;
};

// A utility function to print
// linked list
void printList(struct Node* node)
{
    while (node != NULL) {
        cout << node->data << " ";
        node = node->next;
    }
    cout << endl;
}

// Function to create newNode
// in a linkedlist
Node* newNode(int key)
{
    Node* temp = new Node;
    temp->data = key;
    temp->next = NULL;
    return temp;
}

// Function to insert at beginning
```

```
Node* insertBeg(Node* head, int val)
{
    Node* temp = newNode(val);
    temp->next = head;
    head = temp;
    return head;
}

// Function to rearrange the
// odd and even nodes
void rearrangeOddEven(Node* head)
{
    stack<Node*> odd;
    stack<Node*> even;
    int i = 1;

    while (head != nullptr) {

        if (head->data % 2 != 0 && i % 2 == 0) {

            // Odd Value in Even Position
            // Add pointer to current node
            // in odd stack
            odd.push(head);
        }

        else if (head->data % 2 == 0 && i % 2 != 0) {

            // Even Value in Odd Postion
            // Add pointer to current node
            // in even stack
            even.push(head);
        }

        head = head->next;
        i++;
    }

    while (!odd.empty() && !even.empty()) {

        // Swap Data at the top of two stacks
        swap(odd.top()->data, even.top()->data);
        odd.pop();
        even.pop();
    }
}

// Driver code
```

```
int main()
{
    Node* head = newNode(8);
    head = insertBeg(head, 7);
    head = insertBeg(head, 6);
    head = insertBeg(head, 5);
    head = insertBeg(head, 3);
    head = insertBeg(head, 2);
    head = insertBeg(head, 1);

    cout << "Linked List:" << endl;
    printList(head);
    rearrangeOddEven(head);

    cout << "Linked List after "
        << "Rearranging:" << endl;
    printList(head);

    return 0;
}
```

**Output:**

```
Linked List:
1 2 3 5 6 7 8
Linked List after Rearranging:
1 2 3 6 5 8 7
```

Time Complexity : O(n)  
Auxiliary Space : O(n)

**Method 2 (Efficient)**

1. Segregate odd and even values in the list. After this, all odd values will occur together followed by all even values.
2. Split the list into two lists odd and even.
3. Merge the even list into odd list

**REARRANGE (HEAD)**

```
Step 1: Traverse the list using NODE TEMP.
    If TEMP is odd
        Add TEMP to the beginning of the List
    [END OF IF]
    [END OF TRAVERSAL]
Step 2: Set TEMP to 2nd element of LIST.
```

Step 3: Set PREV\_TEMP to 1st element of List  
Step 4: Traverse using node TEMP as long as an even node is not encountered.  
        PREV\_TEMP = TEMP, TEMP = TEMP->NEXT  
        [END OF TRAVERSAL]  
Step 5: Set EVEN to TEMP. Set PREV\_TEMP->NEXT to NULL  
Step 6: I = HEAD, J = EVEN  
Step 7: Repeat while I != NULL and J != NULL  
        Store next nodes of I and J in K and L  
        K = I->NEXT, L = J->NEXT  
        I->NEXT = J, J->NEXT = K, PTR = J  
        I = K and J = L  
        [END OF LOOP]  
Step 8: if I == NULL  
        PTR->NEXT = J  
        [END of IF]  
Step 8: Return HEAD.  
Step 9: End

C++

```
// Cpp program to rearrange nodes
// as alternate odd even nodes in
// a Singly Linked List
#include <iostream>
#include <stack>
using namespace std;

// Structure node
struct Node {
    int data;
    struct Node* next;
};

// A utility function to print
// linked list
void printList(struct Node* node)
{
    while (node != NULL) {
        cout << node->data << " ";
        node = node->next;
    }
    cout << endl;
}

// Function to create newNode
// in a linkedlist
Node* newNode(int key)
```

```
{  
    Node* temp = new Node;  
    temp->data = key;  
    temp->next = NULL;  
    return temp;  
}  
  
// Function to insert at beginning  
Node* insertBeg(Node* head, int val)  
{  
    Node* temp = newNode(val);  
    temp->next = head;  
    head = temp;  
    return head;  
}  
  
// Function to rearrange the  
// odd and even nodes  
void rearrange(Node** head)  
{  
    // Step 1: Segregate even and odd nodes  
    // Step 2: Split odd and even lists  
    // Step 3: Merge even list into odd list  
    Node* even;  
    Node *temp, *prev_temp;  
    Node *i, *j, *k, *l, *ptr;  
  
    // Step 1: Segregate Odd and Even Nodes  
    temp = (*head)->next;  
    prev_temp = *head;  
  
    while (temp != nullptr) {  
  
        // Backup next pointer of temp  
        Node* x = temp->next;  
  
        // If temp is odd move the node  
        // to beginning of list  
        if (temp->data % 2 != 0) {  
            prev_temp->next = x;  
            temp->next = (*head);  
            (*head) = temp;  
        }  
        else {  
            prev_temp = temp;  
        }  
  
        // Advance Temp Pointer
```

```
        temp = x;
    }

// Step 2
// Split the List into Odd and even
temp = (*head)->next;
prev_temp = (*head);

while (temp != nullptr && temp->data % 2 != 0) {
    prev_temp = temp;
    temp = temp->next;
}

even = temp;

// End the odd List (Make last node null)
prev_temp->next = nullptr;

// Step 3:
// Merge Even List into odd
i = *head;
j = even;

while (j != nullptr && i != nullptr) {

    // While both lists are not
    // exhausted Backup next
    // pointers of i and j
    k = i->next;
    l = j->next;

    i->next = j;
    j->next = k;

    // ptr points to the latest node added
    ptr = j;

    // Advance i and j pointers
    i = k;
    j = l;
}

if (i == nullptr) {

    // Odd list exhausts before even,
    // append remainder of even list to odd.
    ptr->next = j;
}
```

```
// The case where even list exhausts before
// odd list is automatically handled since we
// merge the even list into the odd list
}

// Driver Code
int main()
{
    Node* head = newNode(8);
    head = insertBeg(head, 7);
    head = insertBeg(head, 6);
    head = insertBeg(head, 3);
    head = insertBeg(head, 5);
    head = insertBeg(head, 1);
    head = insertBeg(head, 2);
    head = insertBeg(head, 10);

    cout << "Linked List:" << endl;
    printList(head);
    cout << "Rearranged List" << endl;
    rearrange(&head);
    printList(head);
}
```

**Output:**

```
Linked List:
10 2 1 5 3 6 7 8
Rearranged List
7 10 3 2 5 6 1 8
```

Time Complexity : O(n)  
Auxiliary Space : O(1)

**Source**

<https://www.geeksforgeeks.org/alternate-odd-even-nodes-singly-linked-list/>

## Chapter 7

# Alternate sorting of Linked list

Alternate sorting of Linked list - GeeksforGeeks

Given a linked list containing **n** nodes. The problem is to rearrange the nodes of the list in such a way that the data in first node is first minimum, second node is first maximum, third node is second minimum, fourth node is second maximum and so on.

Examples:

Input : list: 4->1->3->5->2  
Output : 1->5->2->4->3

Input : list: 10->9->8->7->6->5  
Output : 5->10->6->9->7->8

**Approach:** Following are the steps:

1. Sort the linked list using [Merge Sort](#) technique.
2. Split the list into **front** and **back** lists. Refer [FrontBackProcedure](#) of [this](#) post.
3. Now, reverse the **back** list. Refer [this](#) post.
4. Finally, merge the nodes of **first** and **back** lists in alternate order.

```
// C++ implementation of alternative sorting
// of the Linked list
#include <bits/stdc++.h>

using namespace std;

// node of a linked list
struct Node {
    int data;
    struct Node* next;
}
```

```
};

// function to get a new node
Node* getNode(int data)
{
    // allocate space for node
    Node* newNode = (Node*)malloc(sizeof(Node));

    // put in the data
    newNode->data = data;
    newNode->next = NULL;

    return newNode;
}

// Split the nodes of the given list into front and
// back halves, and return the two lists using the
// reference parameters. If the length is odd, the
// extra node should go in the front list. Uses the
// fast/slow pointer strategy.
void FrontBackSplit(Node* source,
                     Node** frontRef, Node** backRef)
{
    Node* fast;
    Node* slow;
    if (source == NULL || source->next == NULL) {
        // length < 2 cases
        *frontRef = source;
        *backRef = NULL;
    }
    else {
        slow = source;
        fast = source->next;

        // Advance 'fast' two nodes, and
        // advance 'slow' one node
        while (fast != NULL) {
            fast = fast->next;
            if (fast != NULL) {
                slow = slow->next;
                fast = fast->next;
            }
        }

        // 'slow' is before the midpoint in the list,
        // so split it in two at that point.
        *frontRef = source;
        *backRef = slow->next;
    }
}
```

```
    slow->next = NULL;
}
}

// function to merge two sorted lists in
// sorted order
Node* SortedMerge(Node* a, Node* b)
{
    Node* result = NULL;

    // Base cases
    if (a == NULL)
        return b;
    else if (b == NULL)
        return a;

    // Pick either a or b, and recur
    if (a->data <= b->data) {
        result = a;
        result->next = SortedMerge(a->next, b);
    }
    else {
        result = b;
        result->next = SortedMerge(a, b->next);
    }

    return result;
}

// sorts the linked list by changing
// next pointers (not data)
void MergeSort(Node** headRef)
{
    Node* head = *headRef;
    Node *a, *b;

    // Base case -- length 0 or 1
    if ((head == NULL) || (head->next == NULL))
        return;

    // Split head into 'a' and 'b' sublists
    FrontBackSplit(head, &a, &b);

    // Recursively sort the sublists
    MergeSort(&a);
    MergeSort(&b);

    // merge the two sorted lists together
```

```

        *headRef = SortedMerge(a, b);
    }

// function to reverse the linked list
static void reverse(Node** head_ref)
{
    Node* prev = NULL;
    Node* current = *head_ref;
    Node* next;

    while (current != NULL) {
        next = current->next;
        current->next = prev;
        prev = current;
        current = next;
    }

    *head_ref = prev;
}

// function to alternately merge two lists
void alternateMerge(Node* head1, Node* head2)
{
    Node *p, *q;
    while (head1 != NULL && head2 != NULL) {

        // merging nodes alternately by
        // making the desired links
        p = head1->next;
        head1->next = head2;
        head1 = p;
        q = head2->next;
        head2->next = head1;
        head2 = q;
    }
}

// function for alternative sort of the
// linked list
Node* altSortForLinkedList(Node* head)
{
    // sort the linked list
    MergeSort(&head);

    Node *front, *back;

    // Split head into 'front' and 'back' sublists
    FrontBackSplit(head, &front, &back);
}

```

```
// reversing the 'back' list
reverse(&back);

// merging the nodes of 'front' and 'back'
// lists alternately
alternateMerge(front, back);

// required head of final list
return front;
}

// Function to print nodes in
// a given linked list
void printList(Node* head)
{
    while (head != NULL) {
        cout << head->data << " ";
        head = head->next;
    }
}

// Driver program to test above
int main()
{
    // linked list: 10->9->8->7->6->5
    Node* head = getNode(10);
    head->next = getNode(9);
    head->next->next = getNode(8);
    head->next->next->next = getNode(7);
    head->next->next->next->next = getNode(6);
    head->next->next->next->next->next = getNode(5);

    cout << "Original list: ";
    printList(head);

    head = altSortForLinkedList(head);

    cout << "\nModified list: ";
    printList(head);

    return 0;
}
```

Output:

```
Original list: 10 9 8 7 6 5
```

Modified list: 5 10 6 9 7 8

Time Complexity:  $O(n * \log n)$ .

### Source

<https://www.geeksforgeeks.org/alternate-sorting-linked-list/>

## Chapter 8

# Alternating split of a given Singly Linked List | Set 1

Alternating split of a given Singly Linked List | Set 1 - GeeksforGeeks

Write a function AlternatingSplit() that takes one list and divides up its nodes to make two smaller lists ‘a’ and ‘b’. The sublists should be made from alternating elements in the original list. So if the original list is 0->1->0->1->0->1 then one sublist should be 0->0->0 and the other should be 1->1->1.

### Method 1(Simple)

The simplest approach iterates over the source list and pull nodes off the source and alternately put them at the front (or beginning) of ‘a’ and b’. The only strange part is that the nodes will be in the reverse order that they occurred in the source list. Method 2 inserts the node at the end by keeping track of last node in sublists.

```
/*Program to alternatively split a linked list into two halves */
#include<stdio.h>
#include<stdlib.h>
#include<assert.h>

/* Link list node */
struct Node
{
    int data;
    struct Node* next;
};

/* pull off the front node of the source and put it in dest */
void MoveNode(struct Node** destRef, struct Node** sourceRef) ;

/* Given the source list, split its nodes into two shorter lists.
 If we number the elements 0, 1, 2, ... then all the even elements
```

```

should go in the first list, and all the odd elements in the second.
The elements in the new lists may be in any order. */
void AlternatingSplit(struct Node* source, struct Node** aRef,
                      struct Node** bRef)
{
    /* split the nodes of source to these 'a' and 'b' lists */
    struct Node* a = NULL;
    struct Node* b = NULL;

    struct Node* current = source;
    while (current != NULL)
    {
        MoveNode(&a, &current); /* Move a node to list 'a' */
        if (current != NULL)
        {
            MoveNode(&b, &current); /* Move a node to list 'b' */
        }
    }
    *aRef = a;
    *bRef = b;
}

/* Take the node from the front of the source, and move it to the front of the dest.
   It is an error to call this with the source list empty.

   Before calling MoveNode():
   source == {1, 2, 3}
   dest == {1, 2, 3}

   After calling MoveNode():
   source == {2, 3}
   dest == {1, 1, 2, 3}
*/
void MoveNode(struct Node** destRef, struct Node** sourceRef)
{
    /* the front source node */
    struct Node* newNode = *sourceRef;
    assert(newNode != NULL);

    /* Advance the source pointer */
    *sourceRef = newNode->next;

    /* Link the old dest off the new node */
    newNode->next = *destRef;

    /* Move dest to point to the new node */
    *destRef = newNode;
}

```

```
/* UTILITY FUNCTIONS */
/* Function to insert a node at the beginning of the linked list */
void push(struct Node** head_ref, int new_data)
{
    /* allocate node */
    struct Node* new_node =
        (struct Node*) malloc(sizeof(struct Node));

    /* put in the data */
    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;
}

/* Function to print nodes in a given linked list */
void printList(struct Node *node)
{
    while(node!=NULL)
    {
        printf("%d ", node->data);
        node = node->next;
    }
}

/* Driver program to test above functions*/
int main()
{
    /* Start with the empty list */
    struct Node* head = NULL;
    struct Node* a = NULL;
    struct Node* b = NULL;

    /* Let us create a sorted linked list to test the functions
     * Created linked list will be 0->1->2->3->4->5 */
    push(&head, 5);
    push(&head, 4);
    push(&head, 3);
    push(&head, 2);
    push(&head, 1);
    push(&head, 0);

    printf("\n Original linked List: ");
    printList(head);
```

```

/* Remove duplicates from linked list */
AlternatingSplit(head, &a, &b);

printf("\n Resultant Linked List 'a' ");
printList(a);

printf("\n Resultant Linked List 'b' ");
printList(b);

getchar();
return 0;
}

```

Time Complexity: O(n) where n is number of node in the given linked list.

### Method 2(Using Dummy Nodes)

Here is an alternative approach which builds the sub-lists in the same order as the source list. The code uses a temporary dummy header nodes for the ‘a’ and ‘b’ lists as they are being built. Each sublist has a “tail” pointer which points to its current last node — that way new nodes can be appended to the end of each list easily. The dummy nodes give the tail pointers something to point to initially. The dummy nodes are efficient in this case because they are temporary and allocated in the stack. Alternately, local “reference pointers” (which always points to the last pointer in the list instead of to the last node) could be used to avoid Dummy nodes.

```

void AlternatingSplit(struct Node* source, struct Node** aRef,
                      struct Node** bRef)
{
    struct Node aDummy;
    struct Node* aTail = &aDummy; /* points to the last node in 'a' */
    struct Node bDummy;
    struct Node* bTail = &bDummy; /* points to the last node in 'b' */
    struct Node* current = source;
    aDummy.next = NULL;
    bDummy.next = NULL;
    while (current != NULL)
    {
        MoveNode(&(aTail->next), &current); /* add at 'a' tail */
        aTail = aTail->next; /* advance the 'a' tail */
        if (current != NULL)
        {
            MoveNode(&(bTail->next), &current);
            bTail = bTail->next;
        }
    }
    *aRef = aDummy.next;
    *bRef = bDummy.next;
}

```

Time Complexity:  $O(n)$  where  $n$  is number of node in the given linked list.

Source: <http://cslibrary.stanford.edu/105/LinkedListProblems.pdf>

### **Source**

<https://www.geeksforgeeks.org/alternating-split-of-a-given-singly-linked-list/>

## Chapter 9

# An interesting method to print reverse of a linked list

An interesting method to print reverse of a linked list - GeeksforGeeks

We are given a linked list, we need to print the linked list in reverse order.

Examples:

```
Input : list : 5-> 15-> 20-> 25
Output : Reversed Linked list : 25-> 20-> 15-> 5
```

```
Input : list : 85-> 15-> 4-> 20
Output : Reversed Linked list : 20-> 4-> 15-> 85
```

```
Input : list : 85
Output : Reversed Linked list : 85
```

For printing a list in reverse order, we have already discussed [Iterative and Recursive Methods to Reverse](#).

In this post, an interesting method is discussed, that doesn't require recursion and does no modifications to list. The function also visits every node of linked list only once.

**Trick :** Idea behind printing a list in reverse order without any recursive function or loop is to use [Carriage return \("r"\)](#). For this, we should have knowledge of length of list. Now, we should print n-1 blank space and then print 1st element then "r", further again n-2 blank space and 2nd node then "r" and so on..

**Carriage return ("r") :** It commands a printer (cursor or the display of a system console), to move the position of the cursor to the first position on the same line.

```
// C program to print reverse of list
```

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

/* Link list node */
struct Node {
    int data;
    struct Node* next;
};

/* Function to reverse the linked list */
void printReverse(struct Node** head_ref, int n)
{
    int j = 0;
    struct Node* current = *head_ref;
    while (current != NULL) {

        // For each node, print proper number
        // of spaces before printing it
        for (int i = 0; i < 2 * (n - j); i++)
            printf(" ");

        // use of carriage return to move back
        // and print.
        printf("%dr", current->data);

        current = current->next;
        j++;
    }
}

/* Function to push a node */
void push(struct Node** head_ref, int new_data)
{
    struct Node* new_node =
        (struct Node*)malloc(sizeof(struct Node));

    new_node->data = new_data;
    new_node->next = (*head_ref);
    (*head_ref) = new_node;
}

/* Function to print linked list and find its
   length */
int printList(struct Node* head)
{
    // i for finding length of list
    int i = 0;
    struct Node* temp = head;
```

```
while (temp != NULL) {
    printf("%d ", temp->data);
    temp = temp->next;
    i++;
}
return i;
}

/* Driver program to test above function*/
int main()
{
    /* Start with the empty list */
    struct Node* head = NULL;
    // list nodes are as 6 5 4 3 2 1
    push(&head, 1);
    push(&head, 2);
    push(&head, 3);
    push(&head, 4);
    push(&head, 5);
    push(&head, 6);

    printf("Given linked list:n");
    // printlist print the list and
    // return the size of list
    int n = printList(head);

    // print reverse list with help
    // of carriage return function
    printf("nReversed Linked list:n");
    printReverse(&head, n);

    return 0;
}
```

Output:

```
Given linked list:
6 5 4 3 2 1
Reversed Linked List:
1 2 3 4 5 6
```

**Input and Output Illustration :**

**Input : 6 5 4 3 2 1**  
**1st Iteration**                6  
**2nd Iteration**             5 6  
**3rd Iteration**          4 5 6

**4th Iteration**       **3 4 5 6**

**5th Iteration**    **2 3 4 5 6**

**Final Output** **1 2 3 4 5 6**

**NOTE:**Above program may not work on online compiler because they do not support anything like carriage return on their console.

**Reference :**

[stackoverflow/Carriage return](#)

**Source**

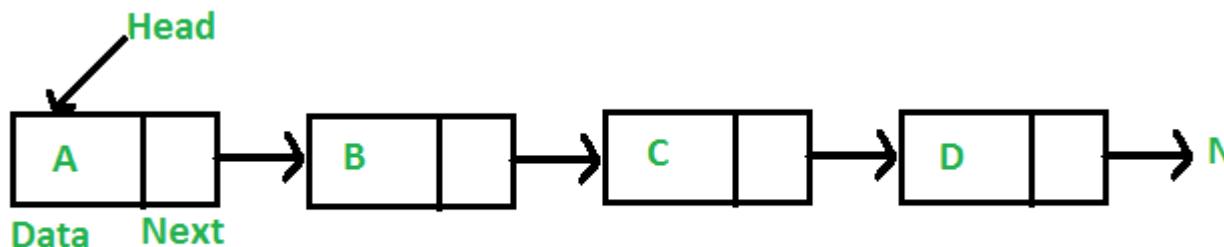
<https://www.geeksforgeeks.org/an-interesting-method-to-print-reverse-of-a-linked-list/>

# Chapter 10

## Applications of linked list data structure

Applications of linked list data structure - GeeksforGeeks

A linked list is a linear data structure, in which the elements are not stored at contiguous memory locations. The elements in a linked list are linked using pointers as shown in the below image:



### Applications of linked list in computer science –

1. Implementation of [stacks](#) and [queues](#)
2. Implementation of graphs : [Adjacency list representation of graphs](#) is most popular which is uses linked list to store adjacent vertices.
3. Dynamic memory allocation : We use linked list of free blocks.
4. Maintaining directory of names
5. Performing arithmetic operations on long integers
6. Manipulation of polynomials by storing constants in the node of linked list
7. representing sparse matrices

### Applications of linked list in real world-

1. *Image viewer* – Previous and next images are linked, hence can be accessed by next and previous button.

2. *Previous and next page in web browser* – We can access previous and next url searched in web browser by pressing back and next button since, they are linked as linked list.
3. *Music Player* – Songs in music player are linked to previous and next song, you can play songs either from starting or ending of the list.

### **Applications of Circular Linked Lists:**

1. Useful for implementation of queue. Unlike [this](#)implementation, we don't need to maintain two pointers for front and rear if we use circular linked list. We can maintain a pointer to the last inserted node and front can always be obtained as next of last.
2. Circular lists are useful in applications to repeatedly go around the list. For example, when multiple applications are running on a PC, it is common for the operating system to put the running applications on a list and then to cycle through them, giving each of them a slice of time to execute, and then making them wait while the CPU is given to another application. It is convenient for the operating system to use a circular list so that when it reaches the end of the list it can cycle around to the front of the list.
3. Circular Doubly Linked Lists are used for implementation of advanced data structures like [Fibonacci Heap](#).

### **An example problem:**

Design a data structure that supports following operations efficiently.

1. `getMin` : Gets minimum
2. `extractMin` : Removes minimum
3. `getMax` : Gets maximum
4. `extractMax` : Removes maximum
5. `insert` : Inserts an item. It may be assumed that the inserted item is always greater than maximum so far. For example, a valid insertion order is 10, 12, 13, 20, 50.

Doubly linked list is the best solution here. We maintain head and tail pointers, since inserted item is always greatest, we insert at tail. Deleting an item from head or tail can be done in  $O(1)$  time. So all operations take  $O(1)$  time.

### [\*\*Recent Articles on Linked List\*\*](#)

### **Source**

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

# Chapter 11

## Arrange consonants and vowels nodes in a linked list

Arrange consonants and vowels nodes in a linked list - GeeksforGeeks

Given a singly linked list, we need to arrange the consonants and vowel nodes of it in such a way that all the vowels nodes come before the consonants while maintaining the **order of their arrival**.

Examples:

```
Input : a -> b -> c -> e -> d ->
        o -> x -> i
Output : a -> e -> o -> i -> b ->
        c -> d -> x
```

### Solution :

The idea is to keep a marker of the latest vowel we found while traversing the list. If we find another vowel, we take it out of the chain and put it after the existing latest vowel.

Example: For linked list:

```
a -> b -> c -> e -> d -> o -> x -> i
```

say our latestVowel reference references the 'a' node, and that we currently reached the 'e' node. We do:

```
a -> e -> b -> c -> d -> o -> x -> i
```

So what was after the ‘a’ node is now after the ‘e’ node after deleting it, and linking ‘a’ directly to ‘e’.

To properly remove and add links, it’s best to use the node before the one you are checking. So if you have a *curr*, you will check *curr->next* node to see if it’s a vowel or not. If it is, we need to add it after the latestVowel node and then it’s easy to remove it from the chain by assigning its next to curr’s next. Also if a list only contains consonants, we simply return head.

```
/* C++ program to arrange consonants and
   vowels nodes in a linked list */
#include<bits/stdc++.h>
using namespace std;

/* A linked list node */
struct Node
{
    char data;
    struct Node *next;
};

/* Function to add new node to the List */
Node *newNode(char key)
{
    Node *temp = new Node;
    temp->data = key;
    temp->next = NULL;
    return temp;
}

// utility function to print linked list
void printlist(Node *head)
{
    if (! head)
    {
        cout << "Empty List\n";
        return;
    }
    while (head != NULL)
    {
        cout << head->data << " ";
        if (head->next)
            cout << "-> ";
        head = head->next;
    }
    cout << endl;
}

// utility function for checking vowel
```

```
bool isVowel(char x)
{
    return (x == 'a' || x == 'e' || x == 'i' ||
            x == 'o' || x == 'u');
}

/* function to arrange consonants and
   vowels nodes */
Node *arrange(Node *head)
{
    Node *newHead = head;

    // for keep track of vowel
    Node *latestVowel;

    Node *curr = head;

    // list is empty
    if (head == NULL)
        return NULL;

    // We need to discover the first vowel
    // in the list. It is going to be the
    // returned head, and also the initial
    // latestVowel.
    if (isVowel(head->data))

        // first element is a vowel. It will
        // also be the new head and the initial
        // latestVowel;
        latestVowel = head;

    else
    {

        // First element is not a vowel. Iterate
        // through the list until we find a vowel.
        // Note that curr points to the element
        // *before* the element with the vowel.
        while (curr->next != NULL &&
               !isVowel(curr->next->data))
            curr = curr->next;

        // This is an edge case where there are
        // only consonants in the list.
        if (curr->next == NULL)
            return head;
    }
}
```

```

// Set the initial latestVowel and the
// new head to the vowel item that we found.
// Relink the chain of consonants after
// that vowel item:
// old_head_consonant->consonant1->consonant2->
// vowel->rest_of_list becomes
// vowel->old_head_consonant->consonant1->
// consonant2->rest_of_list
latestVowel = newHead = curr->next;
curr->next = curr->next->next;
latestVowel->next = head;
}

// Now traverse the list. Curr is always the item
// *before* the one we are checking, so that we
// can use it to re-link.
while (curr != NULL && curr->next != NULL)
{
    if (isVowel(curr->next->data))
    {
        // The next discovered item is a vowel
        if (curr == latestVowel)
        {
            // If it comes directly after the
            // previous vowel, we don't need to
            // move items around, just mark the
            // new latestVowel and advance curr.
            latestVowel = curr = curr->next;
        }
        else
        {

            // But if it comes after an intervening
            // chain of consonants, we need to chain
            // the newly discovered vowel right after
            // the old vowel. Curr is not changed as
            // after the re-linking it will have a
            // new next, that has not been checked yet,
            // and we always keep curr at one before
            // the next to check.
            Node *temp = latestVowel->next;

            // Chain in new vowel
            latestVowel->next = curr->next;

            // Advance latestVowel
            latestVowel = latestVowel->next;
        }
    }
}

```

```
// Remove found vowel from previous place
curr->next = curr->next->next;

// Re-link chain of consonants after latestVowel
latestVowel->next = temp;
}

}

else
{

    // No vowel in the next element, advance curr.
    curr = curr->next;
}
}

return newHead;
}

// Driver code
int main()
{
    Node *head = newNode('a');
    head->next = newNode('b');
    head->next->next = newNode('c');
    head->next->next->next = newNode('e');
    head->next->next->next->next = newNode('d');
    head->next->next->next->next->next = newNode('o');
    head->next->next->next->next->next->next = newNode('x');
    head->next->next->next->next->next->next = newNode('i');

    printf("Linked list before :\n");
    printlist(head);

    head = arrange(head);

    printf("Linked list after :\n");
    printlist(head);

    return 0;
}
```

Output:

```
Linked list before :
a -> b -> c -> e -> d -> o -> x -> i
Linked list after :
a -> e -> o -> i -> b -> c -> d -> x
```

References:

[Stackoverflow](#)

## Source

<https://www.geeksforgeeks.org/arrange-consonants-vowels-nodes-linked-list/>

## Chapter 12

# Binary Search on Singly Linked List

Binary Search on Singly Linked List - GeeksforGeeks

Given a singly linked list and a key, find key using [binary search](#) approach.

To perform a Binary search based on Divide and Conquer Algorithm, determination of the middle element is important. Binary Search is usually fast and efficient for arrays because accessing the middle index between two given indices is easy and fast(Time Complexity  $O(1)$ ). But memory allocation for the singly linked list is dynamic and non-contiguous, which makes finding the middle element difficult. One approach could be of using [skip list](#), one could be traversing the linked list using one pointer.

**Prerequisite :** [Finding middle of a linked list](#).

**Note:** The approach and implementation provided below are to show how Binary Search can be implemented on a linked list. The implementation takes  $O(n)$  time.

**Approach :**

- Here, start node(set to Head of list), and the last node(set to NULL initially) are given.
- Middle is calculated using two pointers approach.
- If middle's data matches the required value of search, return it.
- Else if midele's data < value, move to upper half(setting last to midele's next).
- Else go to lower half(setting last to middle).
- The condition to come out is, either element found or entire list is traversed. When entire list is traversed, last points to start i.e. last  $\rightarrow$  next == start.

In main function, function **InsertAtHead** inserts value at the beginning of linked list. Inserting such values(for sake of simplicity) so that the list created is sorted.

Examples :

Input : Enter value to search : 7  
Output : Found

Input : Enter value to search : 12  
Output : Not Found

```
// CPP code to implement binary search
// on Singly Linked List
#include<stdio.h>
#include<stdlib.h>

struct Node
{
    int data;
    struct Node* next;
};

Node *newNode(int x)
{
    struct Node* temp = new Node;
    temp->data = x;
    temp->next = NULL;
    return temp;
}

// function to find out middle element
struct Node* middle(Node* start, Node* last)
{
    if (start == NULL)
        return NULL;

    struct Node* slow = start;
    struct Node* fast = start -> next;

    while (fast != last)
    {
        fast = fast -> next;
        if (fast != last)
        {
            slow = slow -> next;
            fast = fast -> next;
        }
    }

    return slow;
}
```

```
// Function for implementing the Binary
// Search on linked list
struct Node* binarySearch(Node *head, int value)
{
    struct Node* start = head;
    struct Node* last = NULL;

    do
    {
        // Find middle
        Node* mid = middle(start, last);

        // If middle is empty
        if (mid == NULL)
            return NULL;

        // If value is present at middle
        if (mid -> data == value)
            return mid;

        // If value is more than mid
        else if (mid -> data < value)
            start = mid -> next;

        // If the value is less than mid.
        else
            last = mid;

    } while (last == NULL ||
             last -> next != start);

    // value not present
    return NULL;
}

// Driver Code
int main()
{
    Node *head = newNode(1);
    head->next = newNode(4);
    head->next->next = newNode(7);
    head->next->next->next = newNode(8);
    head->next->next->next->next = newNode(9);
    head->next->next->next->next->next = newNode(10);
    int value = 7;
    if (binarySearch(head, value) == NULL)
        printf("Value not present\n");
    else
```

```
    printf("Present");
    return 0;
}
```

**Output:**

Present

**Time Complexity :**  $O(n)$

**Improved By :** [sunny94](#)

**Source**

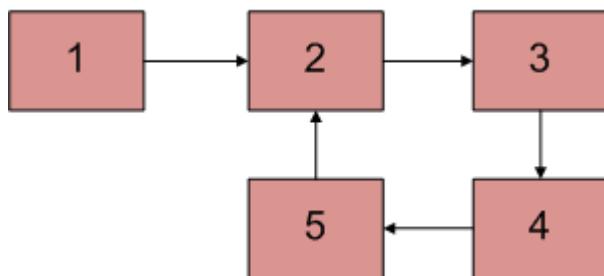
<https://www.geeksforgeeks.org/binary-search-on-singly-linked-list/>

# Chapter 13

## Brent's Cycle Detection Algorithm

Brent's Cycle Detection Algorithm - GeeksforGeeks

Given a linked list, check if the the linked list has loop or not. Below diagram shows a linked list with a loop.



We have discussed [Floyd's algorithm to detect cycle in linked list](#).

**Brent's cycle detection algorithm** is similar to [floyd's algorithm](#) as it also uses two pointer technique. But there is some difference in their approaches. Here we make one pointer stationary till every iteration and teleport it to other pointer **at every power of two**. The start of the cycle is determined by the **smallest power of two** at which they meet. This improves upon the constant factor of Floyd's algorithm by reducing the number of calls.

1. Move fast pointer (or second\_pointer) in powers of 2 until we find a loop. After every power, we reset slow pointer (or first\_pointer) to previous value of second pointer. Reset length to 0 after every every power.
2. The condition for loop testing is first\_pointer and second\_pointer become same. And loop is not present if second\_pointer becomes NULL.
3. When we come out of loop, we have length of loop.

4. We reset first\_pointer to head and second\_pointer to node at position head + length.
5. Now we move both pointers one by one to find beginning of loop.

**Comparison with Floyd's Algorithm:**

- 1) Finds the length of loop in first cycle detection loop itself. No extra work is required for this.
- 2) We only move second in every iteration and avoid moving first (which can be costly if moving to next node involves evaluating a function).

C++

```
// CPP program to implement Brent's cycle
// detection algorithm to detect cycle in
// a linked list.
#include <stdio.h>
#include <stdlib.h>

/* Link list node */
struct Node {
    int data;
    struct Node* next;
};

/* This function detects loop in the list
If loop was there in the list then it returns,
the first node of loop otherwise returns NULL */
struct Node* detectCycle(struct Node* head)
{
    // if head is null then no loop
    if (head == NULL)
        return NULL;

    struct Node* first_pointer = head;
    struct Node* second_pointer = head->next;
    int power = 1;
    int length = 1;

    // This loop runs till we find the loop.
    // If there is no loop then second_pointer
    // ends at NULL .
    while (second_pointer != NULL &&
           second_pointer != first_pointer) {

        // condition after which we will
        // update the power and length as
        // smallest power of two gives the
        // start of cycle.
        if (length == power) {
```

```

        // updating the power.
        power *= 2;

        // updating the length
        length = 0;

        first_pointer = second_pointer;
    }

    second_pointer = second_pointer->next;
    ++length;
}

// if it is null then no loop
if (second_pointer == NULL)
    return NULL;

// Otherwise length stores actual length
// of loop.
// If needed, we can also print length of
// loop.
// printf("Length of loop is %d\n", length);

// Now set first_pointer to the beginning
// and second_pointer to beginning plus
// cycle length which is length.
first_pointer = second_pointer = head;
while (length > 0) {
    second_pointer = second_pointer->next;
    --length;
}

// Now move both pointers at same speed so
// that they meet at the beginning of loop.
while (second_pointer != first_pointer) {
    second_pointer = second_pointer->next;
    first_pointer = first_pointer->next;
}

// If needed, we can also print length of
// loop.
// printf("Length of loop is %d", length);

return first_pointer;
}

struct Node* newNode(int key)

```

```
{
    struct Node* temp =
        (struct Node*)malloc(sizeof(struct Node));
    temp->data = key;
    temp->next = NULL;
    return temp;
}

// Driver program to test above function
int main()
{
    struct Node* head = newNode(50);
    head->next = newNode(20);
    head->next->next = newNode(15);
    head->next->next->next = newNode(4);
    head->next->next->next->next = newNode(10);

    // Create a loop for testing
    head->next->next->next->next->next =
        head->next->next;

    Node *res = detectCycle(head);
    if (res == NULL)
        printf("No loop");
    else
        printf("Loop is present at %d", res->data);
    return 0;
}
```

### Python3

```
# Python program to implement
# Brent's cycle detection
# algorithm to detect cycle
# in a linked list.

# Node class
class Node:

    # Constructor to initialize
    # the node object
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:

    # Function to initialize head
```

```

def __init__(self):
    self.head = None

# Function to insert a new Node
# at the beginning
def push(self, new_data):
    new_node = Node(new_data)
    new_node.next = self.head
    self.head = new_node

# Utility function to print
# the linked LinkedList
def printList(self):
    temp = self.head
    while(temp):
        print (temp.data,end=" ")
        temp = temp.next


def detectCycle(self):
    # if head is null
    # then no loop
    temp=self.head

    if not (temp):
        return False
    first_p=temp
    second_p=temp.next
    power = 1
    length = 1

    # This loop runs till we
    # find the loop. If there
    # is no loop then second
    # pointer ends at NULL
    while (second_p and second_p!= first_p):

        # condition after which
        # we will update the power
        # and length as smallest
        # power of two gives
        # the start of cycle.
        if (length == power):

            # updating the power.
            power *= 2

        # updating the length

```

```

length = 0

first_p = second_p

second_p = second_p.next
length=length+1
# if it is null then no loop
if not (second_p) :
    return

# Otherwise length stores
# actual length of loop.
# If needed, we can also
# print length of loop.
# print("Length of loop is ")
# print (length)

# Now set first_pointer
# to the beginning and
# second_pointer to
# beginning plus cycle
# length which is length.
first_p = second_p = self.head
while (length > 0):
    second_p = second_p.next
    length=length-1

    # Now move both pointers
    # at same speed so that
    # they meet at the
    # beginning of loop.
while (second_p!= first_p) :
    second_p = second_p.next
    first_p = first_p.next

return first_p

# Driver program for testing
llist = LinkedList()
llist.push(50)
llist.push(20)
llist.push(15)
llist.push(4)
llist.push(10)

# Create a loop for testing
llist.head.next.next.next.next = llist.head.next.next;
res=llist.detectCycle()

```

```
if( res.data):
    print ("loop found at ",end=' ')
    print (res.data)
else :
    print ("No Loop ")
# This code is contributed by Gitanjali
```

Output:

Loop is present at 15

**Time Complexity:**  $O(m + n)$  where **m** is the smallest index of the sequence which is the beginning of a cycle, and **n** is the cycle's length.

**Auxiliary Space :** – **O(1)** auxiliary

**References :**

[https://en.wikipedia.org/wiki/Cycle\\_detection#Brent's\\_algorithm](https://en.wikipedia.org/wiki/Cycle_detection#Brent's_algorithm)  
[github](https://github.com)

## Source

<https://www.geeksforgeeks.org/brents-cycle-detection-algorithm/>

## Chapter 14

# Bubble Sort On Doubly Linked List

Bubble Sort On Doubly Linked List - GeeksforGeeks

Sort the given doubly linked list using [bubble sort](#).

Examples:

Input : 5 4 3 2 1  
Output : 1 2 3 4 5

Input : 2 1 3 5 4  
Output : 1 2 3 4 5

### Explanation

As we do in the bubble sort, here also we check elements of two adjacent node whether they are in ascending order or not, if not then we swap the element. We do this until every element get its original position.

In 1st pass the largest element get its original position and in 2nd pass 2nd largest element get its original position and in 3rd pass 3rd largest element get its original position and so on.

And finally whole list get sorted.

Note: If the list is already sorted then it will do only one pass.

```
// CPP program to sort a doubly linked list using
// bubble sort
#include <iostream>
using namespace std;
```

```
// structure of a node
struct Node {
    int data;
    Node* prev;
    Node* next;
};

/* Function to insert a node at the beginning of a linked list */
void insertAtTheBegin(struct Node **start_ref, int data)
{
    struct Node *ptr1 = new Node;
    ptr1->data = data;
    ptr1->next = *start_ref;
    if (*start_ref != NULL)
        (*start_ref)->prev = ptr1;
    *start_ref = ptr1;
}

/* Function to print nodes in a given linked list */
void printList(struct Node *start)
{
    struct Node *temp = start;
    cout << endl;
    while (temp!=NULL)
    {
        cout << temp->data << " ";
        temp = temp->next;
    }
}

/* Bubble sort the given linked list */
void bubbleSort(struct Node *start)
{
    int swapped, i;
    struct Node *ptr1;
    struct Node *lptr = NULL;

    /* Checking for empty list */
    if (start == NULL)
        return;

    do
    {
        swapped = 0;
        ptr1 = start;

        while (ptr1->next != lptr)
        {
```

```
        if (ptr1->data > ptr1->next->data)
        {
            swap(ptr1->data, ptr1->next->data);
            swapped = 1;
        }
        ptr1 = ptr1->next;
    }
    lptr = ptr1;
}
while (swapped);
}

int main()
{
    int arr[] = {12, 56, 2, 11, 1, 90};
    int list_size, i;

    /* start with empty linked list */
    struct Node *start = NULL;

    /* Create linked list from the array arr[] .
     * Created linked list will be 1->11->2->56->12 */
    for (i = 0; i< 6; i++)
        insertAtTheBegin(&start, arr[i]);

    /* print list before sorting */
    printf("\n Linked list before sorting ");
    printList(start);

    /* sort the linked list */
    bubbleSort(start);

    /* print list after sorting */
    printf("\n Linked list after sorting ");
    printList(start);

    return 0;
}
```

**Output:**

```
Linked list before sorting
90 1 11 2 56 12
Linked list after sorting
1 2 11 12 56 90
```

**Source**

<https://www.geeksforgeeks.org/bubble-sort-on-doubly-linked-list/>

## Chapter 15

# C Program for Bubble Sort on Linked List

C Program for Bubble Sort on Linked List - GeeksforGeeks

Given a singly linked list, sort it using [bubble sort](#).

Input : 10->30->20->5  
Output : 5->10->20->30

Input : 20->4->3  
Output : 3->4->20

```
// C program to implement Bubble Sort on singly linked list
#include<stdio.h>
#include<stdlib.h>

/* structure for a node */
struct Node
{
    int data;
    struct Node *next;
};

/* Function to insert a node at the beginning of a linked list */
void insertAtTheBegin(struct Node **start_ref, int data);

/* Function to bubble sort the given linked list */
void bubbleSort(struct Node *start);

/* Function to swap data of two nodes a and b*/

```

```
void swap(struct Node *a, struct Node *b);

/* Function to print nodes in a given linked list */
void printList(struct Node *start);

int main()
{
    int arr[] = {12, 56, 2, 11, 1, 90};
    int list_size, i;

    /* start with empty linked list */
    struct Node *start = NULL;

    /* Create linked list from the array arr[].
     * Created linked list will be 1->11->2->56->12 */
    for (i = 0; i < 6; i++)
        insertAtTheBegin(&start, arr[i]);

    /* print list before sorting */
    printf("\n Linked list before sorting ");
    printList(start);

    /* sort the linked list */
    bubbleSort(start);

    /* print list after sorting */
    printf("\n Linked list after sorting ");
    printList(start);

    getchar();
    return 0;
}

/* Function to insert a node at the beginning of a linked list */
void insertAtTheBegin(struct Node **start_ref, int data)
{
    struct Node *ptr1 = (struct Node*)malloc(sizeof(struct Node));
    ptr1->data = data;
    ptr1->next = *start_ref;
    *start_ref = ptr1;
}

/* Function to print nodes in a given linked list */
void printList(struct Node *start)
{
    struct Node *temp = start;
    printf("\n");
```

```
while (temp!=NULL)
{
    printf("%d ", temp->data);
    temp = temp->next;
}
}

/* Bubble sort the given linked list */
void bubbleSort(struct Node *start)
{
    int swapped, i;
    struct Node *ptr1;
    struct Node *lptr = NULL;

    /* Checking for empty list */
    if (start == NULL)
        return;

    do
    {
        swapped = 0;
        ptr1 = start;

        while (ptr1->next != lptr)
        {
            if (ptr1->data > ptr1->next->data)
            {
                swap(ptr1, ptr1->next);
                swapped = 1;
            }
            ptr1 = ptr1->next;
        }
        lptr = ptr1;
    }
    while (swapped);
}

/* function to swap data of two nodes a and b*/
void swap(struct Node *a, struct Node *b)
{
    int temp = a->data;
    a->data = b->data;
    b->data = temp;
}
```

Output:

```
Linked list before sorting  
90 1 11 2 56 12  
Linked list after sorting  
1 2 11 12 56 90
```

**Improved By :** [YugandharTripathi](#)

### Source

<https://www.geeksforgeeks.org/c-program-bubble-sort-linked-list/>

## Chapter 16

# C Program to reverse each node value in Singly Linked List

C Program to reverse each node value in Singly Linked List - GeeksforGeeks

A [linked list](#) is a linear collection of data elements, in which each node points to the next node. Unlike an array, it doesn't have upper limit and hence extremely useful.

The task is to access value of each node of linked list and reverse them.

**Examples:**

Input : 56 87 12 49 35  
Output :65 78 21 94 53

Input : 128 87 12433 491 33  
Output :821 78 33421 194 33

**Algorithm:**

The task can be accomplished as:

1. Linearly traverse each node of the singly linked list.
2. Reverse the value of each node.
3. Store the reversed value in the current node.

```
// C program to reverse every node data in
// singly linked list.
#include <stdio.h>
#include <stdlib.h>

// A linked list node
struct Node {
```

```
int data;
struct Node* next;
};

// newNode function inserts the new node at
// the right side of linked list
struct Node* newNode(int key)
{
    struct Node* temp = new Node;
    temp->data = key;
    temp->next = NULL;
    return temp;
}

// reverse() will receive individual data item
// from reverseIndividualData() and will return
// the reversed number to calling function
int reverse(int number)
{
    int new_num = 0, rem;

    while (number != 0) {
        rem = number % 10;
        new_num = new_num * 10 + rem;
        number = number / 10;
    }

    return new_num;
}

void reverseIndividualData(struct Node* node)
{
    if (node == NULL)
        return;

    while (node != NULL) {

        /* function call to reverse(),
           reverseIndividualData(struct Node *node)
           will send the one data item at a time to
           reverse(node->data) function which will
           return updated value to node->data*/

        node->data = reverse(node->data);

        /* updating node pointer so as to get
           next value */
    }
}
```

```
        node = node->next;
    }
}

// Function to print nodes in linked list
void printList(struct Node* node)
{
    while (node != NULL) {
        printf("%d ", node->data);
        node = node->next;
    }
}

// Driver program to test above functions
int main()
{
    struct Node* head = NULL;
    head = newNode(56);
    head->next = newNode(87);
    head->next->next = newNode(12);
    head->next->next->next = newNode(49);
    head->next->next->next->next = newNode(35);

    printf("\nList before reversing individual data item \n");
    printList(head);

    reverseIndividualData(head);

    printf("\nList after reversing individual data item\n");
    printList(head);

    return 0;
}
```

**Output:**

```
List before reversing individual data item
56 87 12 49 35
List after reversing individual data item
65 78 21 94 53
```

**Source**

<https://www.geeksforgeeks.org/c-program-reverse-node-value-singly-linked-list/>

## Chapter 17

# Can we reverse a linked list in less than O(n)?

Can we reverse a linked list in less than O(n)? - GeeksforGeeks

It doesn't look possible to reverse a simple singly linked list. A simple singly linked list can only be reversed in O(n) time using recursive and iterative methods.

A memory efficient doubly linked list with head and tail pointers can also be reversed in O(1) time by swapping head and tail pointers.

A doubly linked list with head and tail pointers can also be reversed in O(1) time by swapping head and tail pointers. But we would have to traverse the list in forward direction using prev pointer and reverse direction using next pointer which may not be considered valid.

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

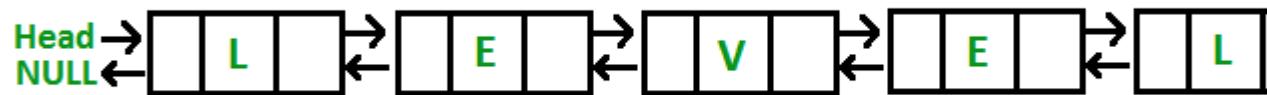
<https://www.geeksforgeeks.org/can-we-reverse-a-linked-list-in-less-than-on/>

# Chapter 18

## Check if a doubly linked list of characters is palindrome or not

Check if a doubly linked list of characters is palindrome or not - GeeksforGeeks

Given a doubly linked list of characters, write a function that returns true if the given doubly linked list is palindrome, else false.



1. Create a doubly linked list where each node contains only one character of a string.
2. Initialize two pointers **left** at starting of list and **right** at the end of the list.
3. Check the data at left node is equal to right node, if it is equal then increment left and decrement right till middle of the list, if at any stage it is not equal then return false.

```
// C++ program to check if doubly
// linked list is palindrome or not
#include<bits/stdc++.h>
using namespace std;

// Structure of node
struct Node
{
    char data;
    struct Node *next;
    struct Node *prev;
}
```

```
};

/* Given a reference (pointer to pointer) to
   the head of a list and an int, inserts a
   new node on the front of the list. */
void push(struct Node** head_ref, char new_data)
{
    struct Node* new_node = new Node;
    new_node->data = new_data;
    new_node->next = (*head_ref);
    new_node->prev = NULL;
    if ((*head_ref) != NULL)
        (*head_ref)->prev = new_node ;
    (*head_ref) = new_node;
}

// Function to check if list is palindrome or not
bool isPalindrome(struct Node *left)
{
    if (left == NULL)
        return true;

    // Find rightmost node
    struct Node *right = left;
    while (right->next != NULL)
        right = right->next;

    while (left != right)
    {
        if (left->data != right->data)
            return false;

        left = left->next;
        right = right->prev;
    }

    return true;
}

// Driver program
int main()
{
    struct Node* head = NULL;
    push(&head, 'l');
    push(&head, 'e');
    push(&head, 'v');
    push(&head, 'e');
    push(&head, 'l');
```

```
if (isPalindrome(head))
    printf("It is Palindrome");
else
    printf("Not Palindrome");

return 0;
}
```

Output:

It is Palindrome

Time complexity:  $O(n)$   
Auxiliary Space :  $O(1)$

**Related Post:**

- Function to check if a singly linked list is palindrome
- Check if a linked list of strings forms a palindrome

**Source**

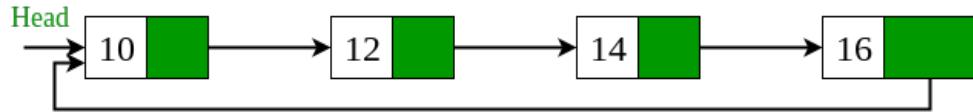
<https://www.geeksforgeeks.org/check-doubly-linked-list-characters-palindrome-not/>

# Chapter 19

## Check if a linked list is Circular Linked List

Check if a linked list is Circular Linked List - GeeksforGeeks

Given a singly linked list, find if the linked list is [circular](#) or not. A linked list is called circular if it is not NULL terminated and all nodes are connected in the form of a cycle. Below is an example of circular linked list.



An empty linked list is considered as circular.

Note that this problem is different from [cycle detection problem](#), here all nodes have to be part of cycle.

The idea is to store head of the linked list and traverse it. If we reach NULL, linked list is not circular. If we reach head again, linked list is circular.

```
// C++ program to check if linked list is circular
#include<bits/stdc++.h>
using namespace std;

/* Link list Node */
struct Node
{
    int data;
    struct Node* next;
};

/* This function returns true if given linked
```

```
list is circular, else false.*/
bool isCircular(struct Node *head)
{
    // An empty linked list is circular
    if (head == NULL)
        return true;

    // Next of head
    struct Node *node = head->next;

    // This loop would stop in both cases (1) If
    // Circular (2) Not circular
    while (node != NULL && node != head)
        node = node->next;

    // If loop stopped because of circular
    // condition
    return (node == head);
}

// Utility function to create a new node.
Node *newNode(int data)
{
    struct Node *temp = new Node;
    temp->data = data;
    temp->next = NULL;
    return temp;
}

/* Driver program to test above function*/
int main()
{
    /* Start with the empty list */
    struct Node* head = newNode(1);
    head->next = newNode(2);
    head->next->next = newNode(3);
    head->next->next->next = newNode(4);

    isCircular(head)? cout << "Yes\n" :
                    cout << "No\n" ;

    // Making linked list circular
    head->next->next->next->next = head;

    isCircular(head)? cout << "Yes\n" :
                    cout << "No\n" ;

    return 0;
}
```

}

Output :

No  
Yes

### Source

<https://www.geeksforgeeks.org/check-if-a-linked-list-is-circular-linked-list/>

## Chapter 20

# Check if a linked list of strings forms a palindrome

Check if a linked list of strings forms a palindrome - GeeksforGeeks

Given a linked list handling string data, check to see whether data is palindrome or not?  
For example,

```
Input  : a -> bc -> d -> dcba -> a -> NULL
Output : True
String "abcdcba" is palindrome.
```

```
Output : a -> bc -> d -> ba -> NULL
Output : False
String "abcdba" is not palindrome.
```

The idea is very simple. Construct a string out of given linked list and check if the constructed string is palindrome or not.

C/C++

```
// Program to check if a given linked list of strings
// form a palindrome
#include <bits/stdc++.h>
using namespace std;

/* Link list node */
struct Node
{
    string data;
    Node* next;
};


```

```
// A utility function to check if str is palindrome
// or not
bool isPalindromeUtil(string str)
{
    int length = str.length();

    // Match characters from beginning and
    // end.
    for (int i=0; i<length/2; i++)
        if (str[i] != str[length-i-1])
            return false;

    return true;
}

// Returns true if string formed by linked
// list is palindrome
bool isPalindrome(Node *node)
{
    // Append all nodes to form a string
    string str = "";
    while (node != NULL)
    {
        str.append(node->data);
        node = node->next;
    }

    // Check if the formed string is palindrome
    return isPalindromeUtil(str);
}

// A utility function to print a given linked list
void printList(Node *node)
{
    while (node != NULL)
    {
        cout << node->data << " -> ";
        node = node->next;
    }
    printf("NULL\n");
}

/* Function to create a new node with given data */
Node *newNode(const char *str)
{
    Node *new_node = new Node;
    new_node->data = str;
```

```
    new_node->next = NULL;
    return new_node;
}

/* Driver program to test above function*/
int main()
{
    Node *head = newNode("a");
    head->next = newNode("bc");
    head->next->next = newNode("d");
    head->next->next->next = newNode("dcb");
    head->next->next->next->next = newNode("a");

    isPalindrome(head)? printf("true\n"):
                      printf("false\n");

    return 0;
}
```

**Java**

```
// Java Program to check if a given linked list of strings
// form a palindrome

import java.util.Scanner;

// Linked List node
class Node
{
    String data;
    Node next;

    Node(String d)
    {
        data = d;
        next = null;
    }
}

class LinkedList_Palindrome
{
    Node head;

    // A utility function to check if str is palindrome
    // or not
    boolean isPalindromeUtil(String str)
    {
        int length = str.length();
```

```
// Match characters from beginning and
// end.
for (int i=0; i<length/2; i++)
    if (str.charAt(i) != str.charAt(length-i-1))
        return false;

return true;
}

// Returns true if string formed by linked
// list is palindrome
boolean isPalindrome()
{
    Node node = head;

    // Append all nodes to form a string
    String str = "";
    while (node != null)
    {
        str = str.concat(node.data);
        node = node.next;
    }

    // Check if the formed string is palindrome
    return isPalindromeUtil(str);
}

/* Driver program to test above function*/
public static void main(String[] args)
{
    LinkedList_Palindrome list = new LinkedList_Palindrome();
    list.head = new Node("a");
    list.head.next = new Node("bc");
    list.head.next.next = new Node("d");
    list.head.next.next.next = new Node("dcb");
    list.head.next.next.next.next = new Node("a");

    System.out.println(list.isPalindrome());
}

}
// This code has been contributed by Amit Khandelwal
```

### Python

```
# Python program to check if given linked list of
# strings form a palindrome
```

```
# Node class
class Node:

    # Constructor to initialize the node object
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:

    # Function to initialize head
    def __init__(self):
        self.head = None

    # A utility function to check if str is palindrome
    # or not
    def isPalindromeUtil(self, string):
        return (string == string[::-1])

    # Returns true if string formed by linked list is
    # palindrome
    def isPalindrome(self):
        node = self.head

        # Append all nodes to form a string
        temp = []
        while (node is not None):
            temp.append(node.data)
            node = node.next
        string = "".join(temp)
        return self.isPalindromeUtil(string)

    # Utility function to print the linked LinkedList
    def printList(self):
        temp = self.head
        while (temp):
            print temp.data,
            temp = temp.next

# Driver program to test above function
llist = LinkedList()
llist.head = Node('a')
llist.head.next = Node('bc')
llist.head.next.next = Node("d")
llist.head.next.next.next = Node("dcb")
llist.head.next.next.next.next = Node("a")
```

```
print "true" if llist.isPalindrome() else "false"  
# This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

Output:

```
true
```

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/check-linked-list-strings-form-palindrome/>

## Chapter 21

# Check if linked list is sorted (Iterative and Recursive)

Check if linked list is sorted (Iterative and Recursive) - GeeksforGeeks

Given a Linked List, task is to check whether the Linked List is sorted in Descending order or not?

Examples :

```
Input : 8 -> 7 -> 5 -> 2 -> 1
Output : Yes
Explanation :
In given linked list, starting from head,
8 > 7 > 5 > 2 > 1. So, it is sorted in reverse order

Input : 24 -> 12 -> 9 -> 11 -> 8 -> 2
Output : No
```

**Iterative Approach :** Traverse the linked list from head to end. For every newly encountered element, check **node -> data > node -> next -> data**. If True, do same for each node else return 0 and Print “No”.

```
// C++ program to check Linked List is sorted
// in descending order or not
#include <bits/stdc++.h>
using namespace std;

/* Linked list node */
struct Node
{
```

```
int data;
struct Node* next;
};

// function to Check Linked List is
// sorted in descending order or not
bool isSortedDesc(struct Node *head)
{
    if (head == NULL)
        return true;

    // Traverse the list till last node and return
    // false if a node is smaller than or equal
    // its next.
    for (Node *t=head; t->next != NULL; t=t->next)
        if (t->data <= t->next->data)
            return false;
    return true;
}

Node *newNode(int data)
{
    Node *temp = new Node;
    temp->next = NULL;
    temp->data = data;
}

// Driver program to test above
int main()
{
    struct Node *head = newNode(7);
    head->next = newNode(5);
    head->next->next = newNode(4);
    head->next->next->next = newNode(3);

    isSortedDesc(head) ? cout << "Yes" :
                        cout << "No";

    return 0;
}
```

**Output:**

Yes

Time Complexity : O(N), where N is the length of linked list.

**Recursive Approach :**

Check Recursively that **node -> data > node -> next -> data**, If not, return 0 that is our terminated condition to come out from recursion else Call Check\_List Function Recursively for next node.

```
// C++ program to recursively check Linked List
// is sorted in descending order or not
#include <bits/stdc++.h>
using namespace std;

/* Linked list node */
struct Node
{
    int data;
    struct Node* next;
};

// function to Check Linked List is
// sorted in descending order or not
bool isSortedDesc(struct Node *head)
{
    // Base cases
    if (head == NULL || head->next == NULL)
        return true;

    // Check first two nodes and recursively
    // check remaining.
    return (head->data > head->next->data &&
            isSortedDesc(head->next));
}

Node *newNode(int data)
{
    Node *temp = new Node;
    temp->next = NULL;
    temp->data = data;
}

// Driver program to test above
int main()
{
    struct Node *head = newNode(7);
    head->next = newNode(5);
    head->next->next = newNode(4);
    head->next->next->next = newNode(3);

    isSortedDesc(head) ? cout << "Yes" :
                        cout << "No";
}
```

```
    return 0;  
}
```

**Output:**

Yes

**Source**

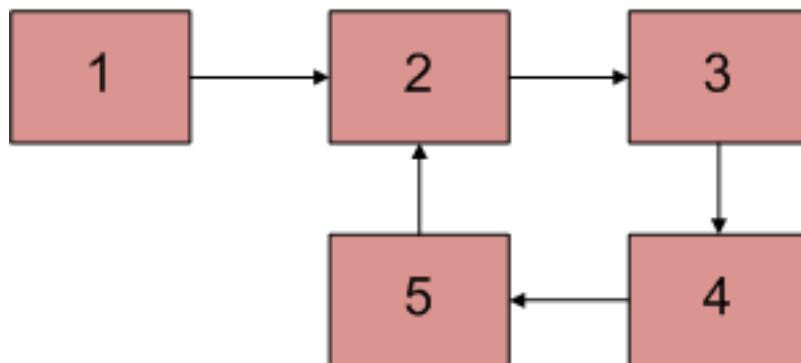
<https://www.geeksforgeeks.org/check-linked-list-sorting-order/>

## Chapter 22

# Check linked list with a loop is palindrome or not

Check linked list with a loop is palindrome or not - GeeksforGeeks

Given a linked list with a loop, the task is to find whether it is palindrome or not. You are not allowed to remove the loop.



Examples:

Input : 1 → 2 → 3 → 2  
/|\ \ |/  
----- 1

Output: Palindrome  
Linked list is 1 2 3 2 1 which is a palindrome.

Input : 1 → 2 → 3 → 4  
/|\ \ |/

```
----- 1
Output: Palindrome
Linked list is 1 2 3 4 1 which is a
not palindrome.
```

Algorithm:

1. Detect the loop using Floyd Cycle Detection Algorithm.
2. Then find the starting node of loop as discussed in [this](#)
3. Check linked list is palindrome or not as discussed in [this](#)

Below is C++ implementation.

```
// C++ program to check if a linked list with
// loop is palindrome or not.
#include<bits/stdc++.h>
using namespace std;

/* Link list node */
struct Node
{
    int data;
    struct Node * next;
};

/* Function to find loop starting node.
loop_node --> Pointer to one of the loop nodes
head --> Pointer to the start node of the linked list */
Node* getLoopstart(Node *loop_node, Node *head)
{
    Node *ptr1 = loop_node;
    Node *ptr2 = loop_node;

    // Count the number of nodes in loop
    unsigned int k = 1, i;
    while (ptr1->next != ptr2)
    {
        ptr1 = ptr1->next;
        k++;
    }

    // Fix one pointer to head
    ptr1 = head;

    // And the other pointer to k nodes after head
    ptr2 = head;
    for (i = 0; i < k; i++)

```

```
ptr2 = ptr2->next;

/* Move both pointers at the same pace,
they will meet at loop starting node */
while (ptr2 != ptr1)
{
    ptr1 = ptr1->next;
    ptr2 = ptr2->next;
}
return ptr1;
}

/* This function detects and find loop starting
node in the list*/
Node* detectAndgetLoopstarting(Node *head)
{
    Node *slow_p = head, *fast_p = head,*loop_start;

    //Start traversing list and detect loop
    while (slow_p && fast_p && fast_p->next)
    {
        slow_p = slow_p->next;
        fast_p = fast_p->next->next;

        /* If slow_p and fast_p meet then find
           the loop starting node*/
        if (slow_p == fast_p)
        {
            loop_start = getLoopstart(slow_p, head);
            break;
        }
    }

    // Return starting node of loop
    return loop_start;
}

// Utility function to check if a linked list with loop
// is palindrome with given starting point.
bool isPalindromeUtil(Node *head, Node* loop_start)
{
    Node *ptr = head;
    stack<int> s;

    // Traverse linked list until last node is equal
    // to loop_start and store the elements till start
    // in a stack
    int count = 0;
```

```
while (ptr != loop_start || count != 1)
{
    s.push(ptr->data);
    if (ptr == loop_start)
        count = 1;
    ptr = ptr->next;
}
ptr = head;
count = 0;

// Traverse linked list until last node is
// equal to loop_start second time
while (ptr != loop_start || count != 1)
{
    // Compare data of node with the top of stack
    // If equal then continue
    if (ptr->data == s.top())
        s.pop();

    // Else return false
    else
        return false;

    if (ptr == loop_start)
        count = 1;
    ptr = ptr->next;
}

// Return true if linked list is palindrome
return true;
}

// Function to find if linked list is palindrome or not
bool isPalindrome(Node* head)
{
    // Find the loop starting node
    Node* loop_start = detectAndgetLoopstarting(head);

    // Check if linked list is palindrome
    return isPalindromeUtil(head, loop_start);
}

Node *newNode(int key)
{
    Node *temp = new Node;
    temp->data = key;
    temp->next = NULL;
    return temp;
```

```
}

/* Driver program to test above function*/
int main()
{
    Node *head = newNode(50);
    head->next = newNode(20);
    head->next->next = newNode(15);
    head->next->next->next = newNode(20);
    head->next->next->next->next = newNode(50);

    /* Create a loop for testing */
    head->next->next->next->next->next = head->next->next;

    isPalindrome(head)? cout << "\nPalindrome"
                      : cout << "\nNot Palindrome";

    return 0;
}
```

Output:

Palindrome

## Source

<https://www.geeksforgeeks.org/check-linked-list-loop-palindrome-not/>

## Chapter 23

# Check whether the length of given linked list is Even or Odd

Check whether the length of given linked list is Even or Odd - GeeksforGeeks

Given a linklist, task is to make a function which check whether the length of linklist is even or odd.

Examples:

Input : 1->2->3->4->NULL  
Output : Even

Input : 1->2->3->4->5->NULL  
Output : Odd

### Method 1: Count the codes linearly

Traverse the entire Linked List and keep counting the number of nodes. As soon as the loop is finished, we can check if the count is even or odd. You may try it yourself.

### Method 2: Stepping 2 nodes at a time

Approach:

1. Take a pointer and move that pointer two nodes at a time
2. At the end, if the pointer is NULL then length is Even, else Odd.

```
// C program to check length
// of a given linklist
#include<stdio.h>
#include<stdlib.h>
```

```
// Defining structure
struct Node
{
    int data;
    struct Node* next;
};

// Function to check the length of linklist
int LinkedListLength(struct Node* head)
{
    while (head && head->next)
    {
        head = head->next->next;
    }
    if (!head)
        return 0;
    return 1;
}

// Push function
void push(struct Node** head, int info)
{
    // Allocating node
    struct Node* node = (struct Node*) malloc(sizeof(struct Node));

    // Info into node
    node->data = info;

    // Next of new node to head
    node->next = (*head);

    // head points to new node
    (*head) = node;
}

// Driver function
int main(void)
{
    struct Node* head = NULL;

    // Adding elements to Linked List
    push(&head, 4);
    push(&head, 5);
    push(&head, 7);
    push(&head, 2);
    push(&head, 9);
    push(&head, 6);
    push(&head, 1);
```

```
push(&head, 2);
push(&head, 0);
push(&head, 5);
push(&head, 5);
int check = LinkedListLength(head);

// Checking for length of
// linklist
if(check == 0)
{
    printf("Even\n");
}
else
{
    printf("Odd\n");
}
return 0;
}
```

Output:

Odd

**Time Complexity:** O(n)  
**Space Complexity:** O(1)

## Source

<https://www.geeksforgeeks.org/check-whether-the-length-of-given-linked-list-is-even-or-odd/>

## Chapter 24

# Circular Linked List | Set 1 (Introduction and Applications)

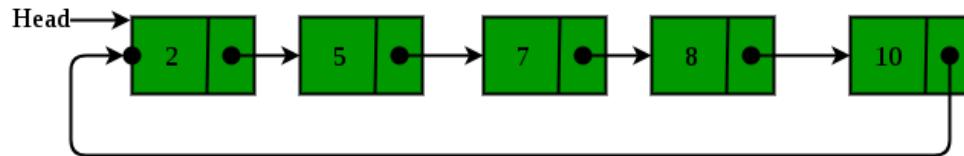
[Circular Linked List | Set 1 \(Introduction and Applications\) - GeeksforGeeks](#)

We have discussed singly and doubly linked lists in the following posts.

[Introduction to Linked List & Insertion](#)

[Doubly Linked List Introduction and Insertion](#)

*Circular linked list is a linked list where all nodes are connected to form a circle. There is no NULL at the end. A circular linked list can be a singly circular linked list or doubly circular linked list.*



### Advantages of Circular Linked Lists:

- 1) Any node can be a starting point. We can traverse the whole list by starting from any point. We just need to stop when the first visited node is visited again.
- 2) Useful for implementation of queue. Unlike [this](#) implementation, we don't need to maintain two pointers for front and rear if we use circular linked list. We can maintain a pointer to the last inserted node and front can always be obtained as next of last.
- 3) Circular lists are useful in applications to repeatedly go around the list. For example, when multiple applications are running on a PC, it is common for the operating system to put the running applications on a list and then to cycle through them, giving each of them a slice of time to execute, and then making them wait while the CPU is given to another application. It is convenient for the operating system to use a circular list so that when it reaches the end of the list it can cycle around to the front of the list.

4) Circular Doubly Linked Lists are used for implementation of advanced data structures like [Fibonacci Heap](#).

**Next Posts :**

[Circular Linked List | Set 2 \(Traversal\)](#)

[Circular Singly Linked List | Insertion](#)

**Source**

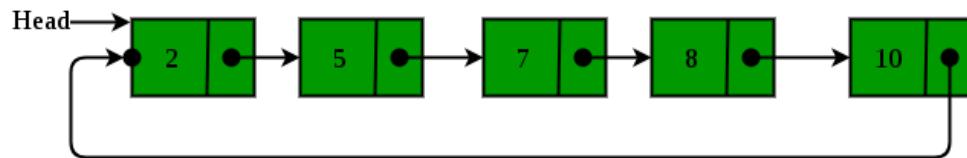
<https://www.geeksforgeeks.org/circular-linked-list/>

# Chapter 25

## Circular Linked List | Set 2 (Traversal)

Circular Linked List | Set 2 (Traversal) - GeeksforGeeks

We have discussed [Circular Linked List Introduction and Applications](#), in the previous post on Circular Linked List. In this post, traversal operation is discussed.



In a conventional linked list, we traverse the list from the head node and stop the traversal when we reach NULL. In a circular linked list, we stop traversal when we reach the first node again. Following is C code for linked list traversal.

```
/* Function to traverse a given Circular linked list and print nodes */
void printList(struct Node *first)
{
    struct Node *temp = first;

    // If linked list is not empty
    if (first != NULL)
    {
        // Keep printing nodes till we reach the first node again
        do
        {
            printf("%d ", temp->data);
            temp = temp->next;
        }
        while (temp != first);
    }
}
```

```
    }
}
```

**Complete C program to demonstrate traversal.** Following is complete C program to demonstrate traversal of circular linked list.

**C/C++**

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

/* structure for a node */
struct Node
{
    int data;
    struct Node *next;
};

/* Function to insert a node at the begining of a Circular
   linked list */
void push(struct Node **head_ref, int data)
{
    struct Node *ptr1 = (struct Node *)malloc(sizeof(struct Node));
    struct Node *temp = *head_ref;
    ptr1->data = data;
    ptr1->next = *head_ref;

    /* If linked list is not NULL then set the next of last node */
    if (*head_ref != NULL)
    {
        while (temp->next != *head_ref)
            temp = temp->next;
        temp->next = ptr1;
    }
    else
        ptr1->next = ptr1; /*For the first node */

    *head_ref = ptr1;
}

/* Function to print nodes in a given Circular linked list */
void printList(struct Node *head)
{
    struct Node *temp = head;
    if (head != NULL)
    {
        do
        {
```

```
        printf("%d ", temp->data);
        temp = temp->next;
    }
    while (temp != head);
}
}

/* Driver program to test above functions */
int main()
{
    /* Initialize lists as empty */
    struct Node *head = NULL;

    /* Created linked list will be 11->2->56->12 */
    push(&head, 12);
    push(&head, 56);
    push(&head, 2);
    push(&head, 11);

    printf("Contents of Circular Linked List\n ");
    printList(head);

    return 0;
}
```

### Python

```
# Python program to demonstrate circular linked list traversal

# Structure for a Node
class Node:

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

class CircularLinkedList:

    # Constructor to create an empty circular linked list
    def __init__(self):
        self.head = None

    # Function to insert a node at the beginning of a
    # circular linked list
    def push(self, data):
        ptr1 = Node(data)
        temp = self.head
```

```
ptr1.next = self.head

# If linked list is not None then set the next of
# last node
if self.head is not None:
    while(temp.next != self.head):
        temp = temp.next
    temp.next = ptr1

else:
    ptr1.next = ptr1 # For the first node

self.head = ptr1

# Function to print nodes in a given circular linked list
def printList(self):
    temp = self.head
    if self.head is not None:
        while(True):
            print "%d" %(temp.data),
            temp = temp.next
            if (temp == self.head):
                break

# Driver program to test above function

# Initialize list as empty
clist = CircularLinkedList()

# Created linked list will be 11->2->56->12
clist.push(12)
clist.push(56)
clist.push(2)
clist.push(11)

print "Contents of circular Linked List"
clist.printList()

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

Output:

```
Contents of Circular Linked List
11 2 56 12
```

You may like to see following posts on Circular Linked List

Split a Circular Linked List into two halves  
Sorted insert for circular linked list

We will soon be discussing implementation of insert delete operations for circular linked lists.

## Source

<https://www.geeksforgeeks.org/circular-linked-list-set-2-traversal/>

## Chapter 26

# Circular Queue | Set 2 (Circular Linked List Implementation)

[Circular Queue | Set 2 \(Circular Linked List Implementation\) - GeeksforGeeks](#)

Prerequisite – [Circular Singly Linked List](#)

We have discussed basics and how to implement circular queue using array in set 1.

[Circular Queue | Set 1 \(Introduction and Array Implementation\)](#)

In this post another method of circular queue implementation is discussed, using Circular Singly Linked List.

Operations on Circular Queue:

- **Front:** Get the front item from queue.
- **Rear:** Get the last item from queue.
- **enQueue(value)** This function is used to insert an element into the circular queue.  
In a circular queue, the new element is always inserted at Rear position.
  1. Create a new node dynamically and insert value into it.
  2. Check if front==NULL, if it is true then front = rear = (newly created node)
  3. If it is false then rare=(newly created node) and rear node always contains the address of the front node.
- **deQueue()** This function is used to delete an element from the circular queue. In a queue, the element is always deleted from front position.
  1. Check whether queue is empty or not means front == NULL.
  2. If it is empty then display Queue is empty. If queue is not empty then step 3
  3. Check if (front==rear) if it is true then set front = rear = NULL else move the front forward in queue, update address of front in rear node and return the element.

```
Elements in Circular Queue are: 14 22 6
Deleted value = 14
Deleted value = 22
Elements in Circular Queue are: 6
Elements in Circular Queue are: 6 9 20
```

**Time Complexity:** Time complexity of enQueue(), deQueue() operation is O(1) as there is no loop in any of the operation.

**Note :** In case of linked list implementation, a queue can be easily implemented without being circular. However in case of array implementation, we need a circular queue to save space.

## Source

<https://www.geeksforgeeks.org/circular-queue-set-2-circular-linked-list-implementation/>

## Chapter 27

# Circular Singly Linked List | Insertion

[Circular Singly Linked List | Insertion - GeeksforGeeks](#)

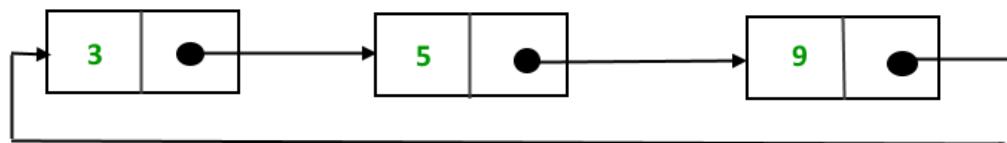
We have discussed Singly and Circular Linked List in the following post:

[Singly Linked List](#)

[Circular Linked List](#)

**Why Circular?** In a singly linked list, for accessing any node of linked list, we start traversing from the first node. If we are at any node in the middle of the list, then it is not possible to access nodes that precede the given node. This problem can be solved by slightly altering the structure of singly linked list. In a singly linked list, next part (pointer to next node) is NULL, if we utilize this link to point to the first node then we can reach preceding nodes. Refer [this](#) for more advantages of circular linked lists.

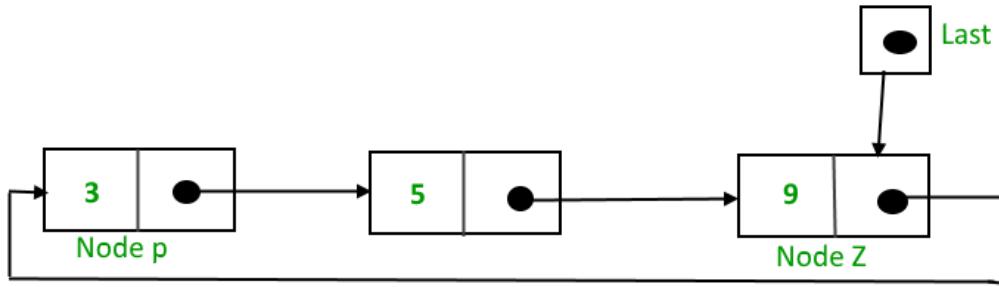
The structure thus formed is circular singly linked list look like this:



In this post, implementation and insertion of a node in a Circular Linked List using singly linked list are explained.

### Implementation

To implement a circular singly linked list, we take an external pointer that points to the last node of the list. If we have a pointer last pointing to the last node, then last->next will point to the first node.



The pointer *last* points to node Z and *last*  $\rightarrow$  next points to node P.

**Why have we taken a pointer that points to the last node instead of first node ?**  
 For insertion of node in the beginning we need traverse the whole list. Also, for insertion and the end, the whole list has to be traversed. If instead of *start* pointer we take a pointer to the last node then in both the cases there won't be any need to traverse the whole list. So insertion in the beginning or at the end takes constant time irrespective of the length of the list.

### Insertion

A node can be added in three ways:

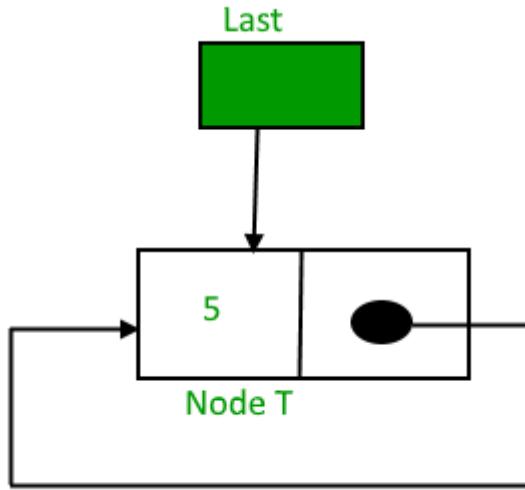
- Insertion in an empty list
- Insertion at the beginning of the list
- Insertion at the end of the list
- Insertion in between the nodes

### Insertion in an empty List

Initially when the list is empty, *last* pointer will be NULL.



After inserting a node T,



After insertion, T is the last node so pointer *last* points to node T. And Node T is first and last node, so T is pointing to itself.

Function to insert node in an empty List,

```
struct Node *addToEnd(struct Node *last, int data)
{
    // This function is only for empty list
    if (last != NULL)
        return last;

    // Creating a node dynamically.
    struct Node *last =
        (struct Node*)malloc(sizeof(struct Node));

    // Assigning the data.
    last -> data = data;

    // Note : list was empty. We link single node
    // to itself.
    last -> next = last;

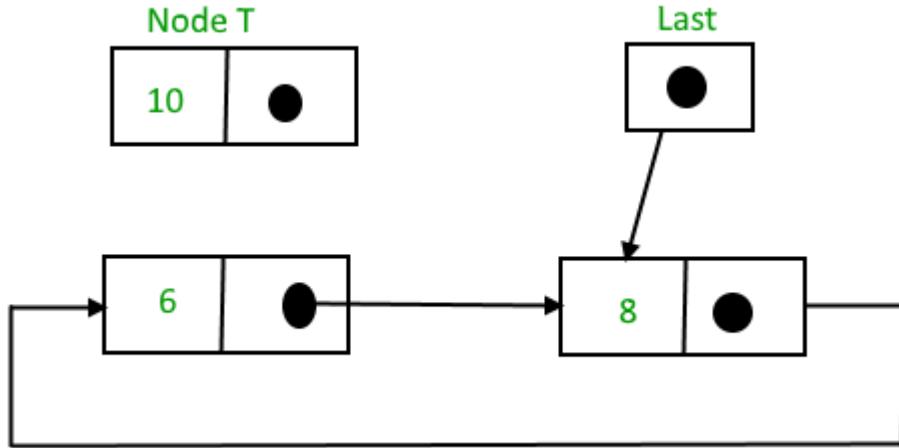
    return last;
}
```

### **Insertion at the beginning of the list**

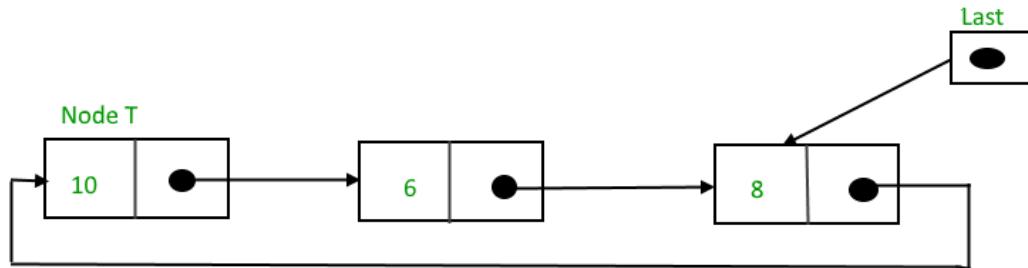
To Insert a node at the beginning of the list, follow these step:

1. Create a node, say T.

2. Make  $T \rightarrow \text{next} = \text{last} \rightarrow \text{next}$ .
3.  $\text{last} \rightarrow \text{next} = T$ .



After insertion,



Function to insert node in the beginning of the List,

```
struct Node *addBegin(struct Node *last, int data)
{
    if (last == NULL)
        return addToEmpty(last, data);

    // Creating a node dynamically.
    struct Node *temp
        = (struct Node *)malloc(sizeof(struct Node));

    // Assigning the data.
    temp->data = data;

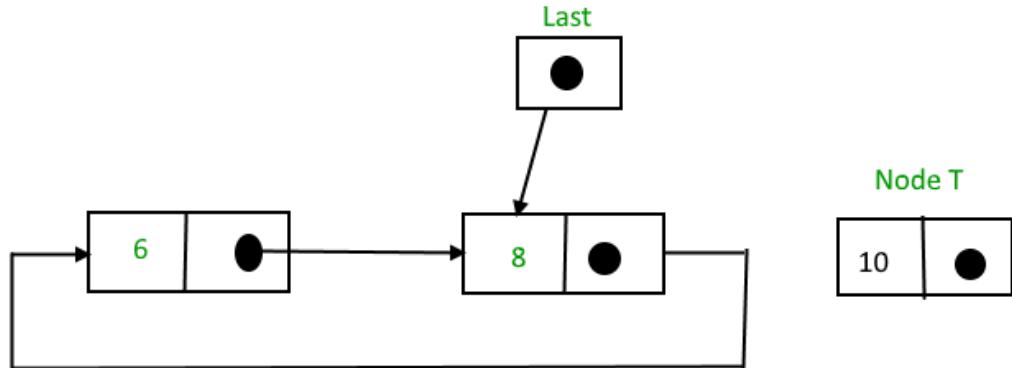
    // Adjusting the links.
    temp->next = last->next;
    last->next = temp;

    return last;
}
```

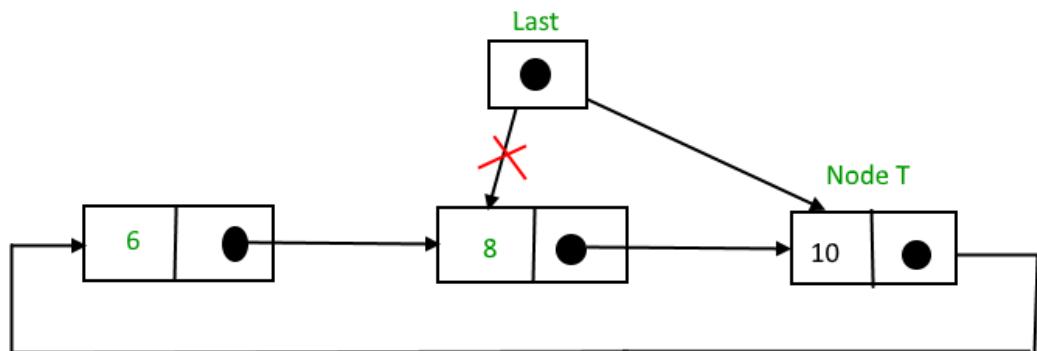
### Insertion at the end of the list

To Insert a node at the end of the list, follow these step:

1. Create a node, say T.
2. Make  $T \rightarrow \text{next} = \text{last} \rightarrow \text{next}$ ;
3.  $\text{last} \rightarrow \text{next} = T$ .
4.  $\text{last} = T$ .



After insertion,



Function to insert node in the end of the List,

```

struct Node *addEnd(struct Node *last, int data)
{
    if (last == NULL)
        return addToEmpty(last, data);

    // Creating a node dynamically.
    struct Node *temp =
        (struct Node *)malloc(sizeof(struct Node));

    // Assigning the data.
    temp -> data = data;

    // Adjusting the links.
    temp -> next = last;
    last -> next = temp;
}

```

```

temp -> next = last -> next;
last -> next = temp;
last = temp;

return last;
}

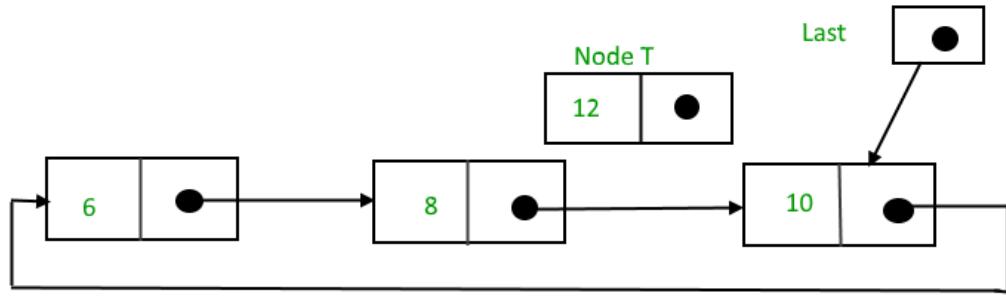
```

### Insertion in between the nodes

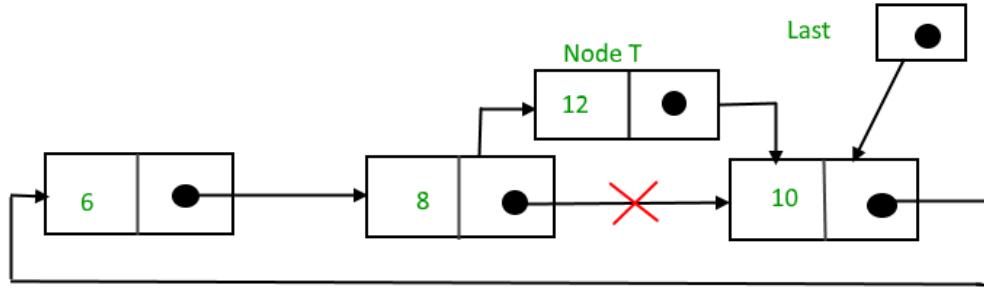
To Insert a node at the end of the list, follow these step:

1. Create a node, say T.
2. Search the node after which T need to be insert, say that node be P.
3. Make T -> next = P -> next;
4. P -> next = T.

Suppose 12 need to be insert after node having value 10,



After searching and insertion,



Function to insert node in the end of the List,

```

struct Node *addAfter(struct Node *last, int data, int item)
{
    if (last == NULL)
        return NULL;

    struct Node *temp, *p;

```

```
p = last -> next;

// Searching the item.
do
{
    if (p ->data == item)
    {
        // Creating a node dynamically.
        temp = (struct Node *)malloc(sizeof(struct Node));

        // Assigning the data.
        temp -> data = data;

        // Adjusting the links.
        temp -> next = p -> next;

        // Adding newly allocated node after p.
        p -> next = temp;

        // Checking for the last node.
        if (p == last)
            last = temp;

        return last;
    }
    p = p -> next;
} while (p != last -> next);

cout << item << " not present in the list." << endl;
return last;
}
```

Following is a complete program that uses all of the above methods to create a circular singly linked list.

```
#include<bits/stdc++.h>
using namespace std;

struct Node
{
    int data;
    struct Node *next;
};

struct Node *addToEnd(struct Node *last, int data)
{
    // This function is only for empty list
    if (last != NULL)
```

```
    return last;

// Creating a node dynamically.
struct Node *temp =
    (struct Node*)malloc(sizeof(struct Node));

// Assigning the data.
temp -> data = data;
last = temp;

// Creating the link.
last -> next = last;

return last;
}

struct Node *addBegin(struct Node *last, int data)
{
    if (last == NULL)
        return addToEmpty(last, data);

    struct Node *temp =
        (struct Node *)malloc(sizeof(struct Node));

    temp -> data = data;
    temp -> next = last -> next;
    last -> next = temp;

    return last;
}

struct Node *addEnd(struct Node *last, int data)
{
    if (last == NULL)
        return addToEmpty(last, data);

    struct Node *temp =
        (struct Node *)malloc(sizeof(struct Node));

    temp -> data = data;
    temp -> next = last -> next;
    last -> next = temp;
    last = temp;

    return last;
}

struct Node *addAfter(struct Node *last, int data, int item)
```

```

{
    if (last == NULL)
        return NULL;

    struct Node *temp, *p;
    p = last -> next;
    do
    {
        if (p -> data == item)
        {
            temp = (struct Node *)malloc(sizeof(struct Node));
            temp -> data = data;
            temp -> next = p -> next;
            p -> next = temp;

            if (p == last)
                last = temp;
            return last;
        }
        p = p -> next;
    } while(p != last -> next);

    cout << item << " not present in the list." << endl;
    return last;
}

void traverse(struct Node *last)
{
    struct Node *p;

    // If list is empty, return.
    if (last == NULL)
    {
        cout << "List is empty." << endl;
        return;
    }

    // Pointing to first Node of the list.
    p = last -> next;

    // Traversing the list.
    do
    {
        cout << p -> data << " ";
        p = p -> next;
    }
}

```

```
while(p != last->next);  
}  
  
// Driven Program  
int main()  
{  
    struct Node *last = NULL;  
  
    last = addToEmpty(last, 6);  
    last = addBegin(last, 4);  
    last = addBegin(last, 2);  
    last = addEnd(last, 8);  
    last = addEnd(last, 12);  
    last = addAfter(last, 10, 8);  
  
    traverse(last);  
  
    return 0;  
}
```

Output:

2 4 6 8 10 12

## Source

<https://www.geeksforgeeks.org/circular-singly-linked-list-insertion/>

## Chapter 28

# Clone a linked list with next and random pointer in O(1) space

Clone a linked list with next and random pointer in O(1) space - GeeksforGeeks

Given a linked list having two pointers in each node. The first one points to the next node of the list, however the other pointer is random and can point to any node of the list. Write a program that clones the given list in O(1) space, i.e., without any extra space.

Examples:

Input : Head of the below linked list

Output :  
A new linked list identical to the original list.

In the previous posts [Set-1](#) and [Set-2](#) various methods are discussed, and O(n) space complexity implementation is also available.

In this post we'll be implementing an algorithm that'd require no additional space as discussed in Set-1.

Below is the Algorithm:

- Create the copy of node 1 and insert it between node 1 & node 2 in original Linked List, create the copy of 2 and insert it between 2 & 3.. Continue in this fashion, add the copy of N after the Nth node
- Now copy the random link in this fashion

```
original->next->random= original->random->next; /*TRaverse  
TWO NODES*/
```

This works because original->next is nothing but copy of original and Original->random->next is nothing but copy of random.

- Now restore the original and copy linked lists in this fashion in a single loop.

```
original->next = original->next->next;
copy->next = copy->next->next;
```

- Ensure that original->next is NULL and return the cloned list

Below is the C++ implementation.

```
// C++ program to clone a linked list with next
// and arbit pointers in O(n) time
#include <bits/stdc++.h>
using namespace std;

// Structure of linked list Node
struct Node
{
    int data;
    Node *next,*random;
    Node(int x)
    {
        data = x;
        next = random = NULL;
    }
};

// Utility function to print the list.
void print(Node *start)
{
    Node *ptr = start;
    while (ptr)
    {
        cout << "Data = " << ptr->data << ", Random = "
            << ptr->random->data << endl;
        ptr = ptr->next;
    }
}

// This function clones a given linked list
// in O(1) space
Node* clone(Node *start)
{
    Node* curr = start, *temp;

    // insert additional node after
```

```
// every node of original list
while (curr)
{
    temp = curr->next;

    // Inserting node
    curr->next = new Node(curr->data);
    curr->next->next = temp;
    curr = temp;
}

curr = start;

// adjust the random pointers of the
// newly added nodes
while (curr)
{
    curr->next->random = curr->random->next;

    // move to the next newly added node by
    // skipping an original node
    curr = curr->next?curr->next->next:curr->next;
}

Node* original = start, *copy = start->next;

// save the start of copied linked list
temp = copy;

// now separate the original list and copied list
while (original && copy)
{
    original->next =
        original->next? original->next->next : original->next;

    copy->next = copy->next?copy->next->next:copy->next;
    original = original->next;
    copy = copy->next;
}

return temp;
}

// Driver code
int main()
{
    Node* start = new Node(1);
    start->next = new Node(2);
```

```
start->next->next = new Node(3);
start->next->next->next = new Node(4);
start->next->next->next->next = new Node(5);

// 1's random points to 3
start->random = start->next->next;

// 2's random points to 1
start->next->random = start;

// 3's and 4's random points to 5
start->next->next->random =
    start->next->next->next->next;
start->next->next->next->random =
    start->next->next->next->next;

// 5's random points to 2
start->next->next->next->next->random =
    start->next;

cout << "Original list : \n";
print(start);

cout << "\nCloned list : \n";
Node *cloned_list = clone(start);
print(cloned_list);

return 0;
}
```

Output:

```
Original list :
Data = 1, Random = 3
Data = 2, Random = 1
Data = 3, Random = 5
Data = 4, Random = 5
Data = 5, Random = 2

Cloned list :
Data = 1, Random = 3
Data = 2, Random = 1
Data = 3, Random = 5
Data = 4, Random = 5
Data = 5, Random = 2
```

## Source

<https://www.geeksforgeeks.org/clone-linked-list-next-random-pointer-o1-space/>

# Chapter 29

## Clone a linked list with next and random pointer | Set 1

Clone a linked list with next and random pointer | Set 1 - GeeksforGeeks

You are given a Double Link List with one pointer of each node pointing to the next node just like in a single link list. The second pointer however CAN point to any node in the list and not just the previous node. Now write a program in **O(n)** time to duplicate this list. That is, write a program which will create a copy of this list.

Let us call the second pointer as arbit pointer as it can point to any arbitrary node in the linked list.

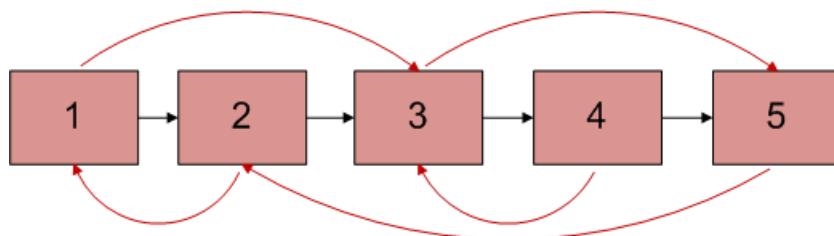


Figure 1

Arbitrary pointers are shown in red and next pointers in black

### Method 1 (Uses **O(n)** extra space)

This method stores the next and arbitrary mappings (of original list) in an array first, then modifies the original Linked List (to create copy), creates a copy. And finally restores the original list.

- 1) Create all nodes in copy linked list using next pointers.
- 2) Store the node and its next pointer mappings of original linked list.
- 3) Change next pointer of all nodes in original linked list to point to the corresponding node

in copy linked list.

Following diagram shows status of both Linked Lists after above 3 steps. The red arrow shows arbit pointer and black arrow shows next pointers.

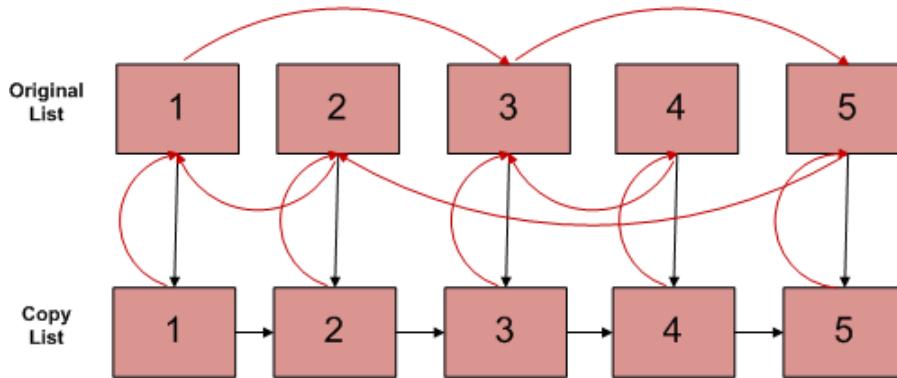


Figure 2

4) Change the arbit pointer of all nodes in copy linked list to point to corresponding node in original linked list.

5) Now construct the arbit pointer in copy linked list as below and restore the next pointer of nodes in the original linked list.

```
copy_list_node->arbit =
    copy_list_node->arbit->arbit->next;
copy_list_node = copy_list_node->next;
```

6) Restore the next pointers in original linked list from the stored mappings(in step 2).

Time Complexity:  $O(n)$

Auxiliary Space:  $O(n)$

### Method 2 (Uses Constant Extra Space)

Thanks to Saravanan Mani for providing this solution. This solution works using constant space.

1) Create the copy of node 1 and insert it between node 1 & node 2 in original Linked List, create the copy of 2 and insert it between 2 & 3.. Continue in this fashion, add the copy of N afte the Nth node

2) Now copy the arbitrary link in this fashion

```
original->next->arbitrary = original->arbitrary->next; /*TRAVERSE
TWO NODES*/
```

This works because original->next is nothing but copy of original and Original->arbitrary->next is nothing but copy of arbitrary.

3) Now restore the original and copy linked lists in this fashion in a single loop.

```
original->next = original->next->next;
copy->next = copy->next->next;
```

- 4) Make sure that last element of original->next is NULL.

Refer below post for implementation of this method.

**[Clone a linked list with next and random pointer in O\(1\) space](#)**

Time Complexity: O(n)

Auxiliary Space: O(1)

Refer Following Post for Hashing based Implementation.

**[Clone a linked list with next and random pointer | Set 2](#)**

Asked by Varun Bhatia. 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/a-linked-list-with-next-and-arbit-pointer/>

## Chapter 30

# Clone a linked list with next and random pointer | Set 2

Clone a linked list with next and random pointer | Set 2 - GeeksforGeeks

We have already discussed 2 different ways to clone a linked list. In [this](#)post, one more simple method to clone a linked list is discussed.

The idea is to use Hashing. Below is algorithm.

1. Traverse the original linked list and make a copy in terms of data.
2. Make a hash map of key value pair with original linked list node and copied linked list node.
3. Traverse the original linked list again and using the hash map adjust the next and random reference of cloned linked list nodes.

Below is the implementation of above approach.

C++

```
// C++ program to clone a linked list with
// random pointers
#include<bits/stdc++.h>
using namespace std;

// Linked List Node
class Node
{
public:
    int data;//Node data

    // Next and random reference
    Node *next, *random;
```

```
Node(int data)
{
    this->data = data;
    this->next = this->random = NULL;
}
};

// linked list class
class LinkedList
{
public:
    Node *head;// Linked list head reference

    LinkedList(Node *head)
    {
        this->head = head;
    }

    // push method to put data always at
    // the head in the linked list.
    void push(int data)
    {
        Node *node = new Node(data);
        node->next = head;
        head = node;
    }

    // Method to print the list.
    void print()
    {
        Node *temp = head;
        while (temp != NULL)
        {
            Node *random = temp->random;
            int randomData = (random != NULL)?
                random->data: -1;
            cout << "Data = " << temp->data
                << ", ";
            cout << "Random Data = " <<
                randomData << endl;
            temp = temp->next;
        }
        cout << endl;
    }

    // Actual clone method which returns
    // head reference of cloned linked
```

```
// list.
LinkedList* clone()
{
    // Initialize two references,
    // one with original list's head.
    Node *origCurr = head;
    Node *cloneCurr = NULL;

    // Hash map which contains node
    // to node mapping of original
    // and clone linked list.
    unordered_map<Node*, Node*> mymap;

    // Traverse the original list and
    // make a copy of that in the
    // clone linked list.
    while (origCurr != NULL)
    {
        cloneCurr = new Node(origCurr->data);
        mymap[origCurr] = cloneCurr;
        origCurr = origCurr->next;
    }

    // Adjusting the original list
    // reference again.
    origCurr = head;

    // Traversal of original list again
    // to adjust the next and random
    // references of clone list using
    // hash map.
    while (origCurr != NULL)
    {
        cloneCurr = mymap[origCurr];
        cloneCurr->next = mymap[origCurr->next];
        cloneCurr->random = mymap[origCurr->random];
        origCurr = origCurr->next;
    }

    // return the head reference of
    // the clone list.
    return new LinkedList(mymap[head]);
}

// driver code
int main()
{
```

```
// Pushing data in the linked list.  
LinkedList *mylist = new LinkedList(new Node(5));  
mylist->push(4);  
mylist->push(3);  
mylist->push(2);  
mylist->push(1);  
  
// Setting up random references.  
mylist->head->random = mylist->head->next->next;  
  
mylist->head->next->random =  
    mylist->head->next->next->next;  
  
mylist->head->next->next->random =  
    mylist->head->next->next->next->next;  
  
mylist->head->next->next->next->random =  
    mylist->head->next->next->next->next;  
  
mylist->head->next->next->next->random =  
    mylist->head->next;  
  
// Making a clone of the original  
// linked list.  
LinkedList *clone = mylist->clone();  
  
// Print the original and cloned  
// linked list.  
cout << "Original linked list\n";  
mylist->print();  
cout << "\nCloned linked list\n";  
clone->print();  
}  
// This code is contributed by Chhavi
```

### Java

```
// Java program to clone a linked list with random pointers  
import java.util.HashMap;  
import java.util.Map;  
  
// Linked List Node class  
class Node  
{  
    int data;//Node data  
    Node next, random;//Next and random reference  
  
    //Node constructor
```

```
public Node(int data)
{
    this.data = data;
    this.next = this.random = null;
}
}

// linked list class
class LinkedList
{
    Node head;//Linked list head reference

    // Linked list constructor
    public LinkedList(Node head)
    {
        this.head = head;
    }

    // push method to put data always at the head
    // in the linked list.
    public void push(int data)
    {
        Node node = new Node(data);
        node.next = this.head;
        this.head = node;
    }

    // Method to print the list.
    void print()
    {
        Node temp = head;
        while (temp != null)
        {
            Node random = temp.random;
            int randomData = (random != null)? random.data: -1;
            System.out.println("Data = " + temp.data +
                               ", Random data = "+ randomData);
            temp = temp.next;
        }
    }

    // Actual clone method which returns head
    // reference of cloned linked list.
    public LinkedList clone()
    {
        // Initialize two references, one with original
        // list's head.
        Node origCurr = this.head, cloneCurr = null;
```

```
// Hash map which contains node to node mapping of
// original and clone linked list.
Map<Node, Node> map = new HashMap<Node, Node>();

// Traverse the original list and make a copy of that
// in the clone linked list.
while (origCurr != null)
{
    cloneCurr = new Node(origCurr.data);
    map.put(origCurr, cloneCurr);
    origCurr = origCurr.next;
}

// Adjusting the original list reference again.
origCurr = this.head;

// Traversal of original list again to adjust the next
// and random references of clone list using hash map.
while (origCurr != null)
{
    cloneCurr = map.get(origCurr);
    cloneCurr.next = map.get(origCurr.next);
    cloneCurr.random = map.get(origCurr.random);
    origCurr = origCurr.next;
}

//return the head reference of the clone list.
return new LinkedList(map.get(this.head));
}

}

// Driver Class
class Main
{
    // Main method.
    public static void main(String[] args)
    {
        // Pushing data in the linked list.
        LinkedList list = new LinkedList(new Node(5));
        list.push(4);
        list.push(3);
        list.push(2);
        list.push(1);

        // Setting up random references.
        list.head.random = list.head.next.next;
        list.head.next.random =
```

```
list.head.next.next.next;
list.head.next.next.random =
    list.head.next.next.next.next;
list.head.next.next.next.random =
    list.head.next.next.next.next.next;
list.head.next.next.next.next.random =
    list.head.next;

// Making a clone of the original linked list.
LinkedList clone = list.clone();

// Print the original and cloned linked list.
System.out.println("Original linked list");
list.print();
System.out.println("\nCloned linked list");
clone.print();
}

}
```

Output:

```
Original linked list
Data = 1, Random data = 3
Data = 2, Random data = 4
Data = 3, Random data = 5
Data = 4, Random data = -1
Data = 5, Random data = 2

Cloned linked list
Data = 1, Random data = 3
Data = 2, Random data = 4
Data = 3, Random data = 5
Data = 4, Random data = -1
Data = 5, Random data = 2
```

Time complexity :  $O(n)$   
Auxiliary space :  $O(n)$

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/clone-linked-list-next-arbit-pointer-set-2/>

# Chapter 31

## Compare two strings represented as linked lists

Compare two strings represented as linked lists - GeeksforGeeks

Given two linked lists, represented as linked lists (every character is a node in linked list). Write a function compare() that works similar to strcmp(), i.e., it returns 0 if both strings are same, 1 if first linked list is lexicographically greater, and -1 if second string is lexicographically greater.

Examples:

```
Input: list1 = g->e->e->k->s->a  
       list2 = g->e->e->k->s->b  
Output: -1
```

```
Input: list1 = g->e->e->k->s->a  
       list2 = g->e->e->k->s  
Output: 1
```

```
Input: list1 = g->e->e->k->s  
       list2 = g->e->e->k->s  
Output: 0
```

C++

```
// C++ program to compare two strings represented as linked  
// lists  
#include<bits/stdc++.h>  
using namespace std;  
  
// Linked list Node structure
```

```
struct Node
{
    char c;
    struct Node *next;
};

// Function to create newNode in a linkedlist
Node* newNode(char c)
{
    Node *temp = new Node;
    temp->c = c;
    temp->next = NULL;
    return temp;
}

int compare(Node *list1, Node *list2)
{
    // Traverse both lists. Stop when either end of a linked
    // list is reached or current characters don't match
    while (list1 && list2 && list1->c == list2->c)
    {
        list1 = list1->next;
        list2 = list2->next;
    }

    // If both lists are not empty, compare mismatching
    // characters
    if (list1 && list2)
        return (list1->c > list2->c)? 1: -1;

    // If either of the two lists has reached end
    if (list1 && !list2) return 1;
    if (list2 && !list1) return -1;

    // If none of the above conditions is true, both
    // lists have reached end
    return 0;
}

// Driver program
int main()
{
    Node *list1 = newNode('g');
    list1->next = newNode('e');
    list1->next->next = newNode('e');
    list1->next->next->next = newNode('k');
    list1->next->next->next->next = newNode('s');
    list1->next->next->next->next = newNode('b');
```

```
Node *list2 = newNode('g');
list2->next = newNode('e');
list2->next->next = newNode('e');
list2->next->next->next = newNode('k');
list2->next->next->next->next = newNode('s');
list2->next->next->next->next = newNode('a');

cout << compare(list1, list2);

return 0;
}
```

**Java**

```
// Java program to compare two strings represented as a linked list

// Linked List Class
class LinkedList {

    Node head; // head of list
    static Node a, b;

    /* Node Class */
    static class Node {

        char data;
        Node next;

        // Constructor to create a new node
        Node(char d) {
            data = d;
            next = null;
        }
    }

    int compare(Node node1, Node node2) {

        if (node1 == null && node2 == null) {
            return 1;
        }
        while (node1 != null && node2 != null && node1.data == node2.data) {
            node1 = node1.next;
            node2 = node2.next;
        }

        // if the list are diffrent in size
        if (node1 != null && node2 != null) {
```

```
        return (node1.data > node2.data ? 1 : -1);
    }

    // if either of the list has reached end
    if (node1 != null && node2 == null) {
        return 1;
    }
    if (node1 == null && node2 != null) {
        return -1;
    }
    return 0;
}

public static void main(String[] args) {

    LinkedList list = new LinkedList();
    Node result = null;

    list.a = new Node('g');
    list.a.next = new Node('e');
    list.a.next.next = new Node('e');
    list.a.next.next.next = new Node('k');
    list.a.next.next.next.next = new Node('s');
    list.a.next.next.next.next.next = new Node('b');

    list.b = new Node('g');
    list.b.next = new Node('e');
    list.b.next.next = new Node('e');
    list.b.next.next.next = new Node('k');
    list.b.next.next.next.next = new Node('s');
    list.b.next.next.next.next.next = new Node('a');

    int value;
    value = list.compare(a, b);
    System.out.println(value);

}
}

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

### Python

```
# Python program to compare two strings represented as
# linked lists

# A linked list node structure
class Node:
```

```
# Constructor to create a new node
def __init__(self, key):
    self.c = key ;
    self.next = None

def compare(list1, list2):

    # Traverse both lists. Stop when either end of linked
    # list is reached or current characters don't match
    while(list1 and list2 and list1.c == list2.c):
        list1 = list1.next
        list2 = list2.next

    # If both lists are not empty, compare mismatching
    # characters
    if(list1 and list2):
        return 1 if list1.c > list2.c else -1

    # If either of the two lists has reached end
    if (list1 and not list2):
        return 1

    if (list2 and not list1):
        return -1
    return 0

# Driver program

list1 = Node('g')
list1.next = Node('e')
list1.next.next = Node('e')
list1.next.next.next = Node('k')
list1.next.next.next.next = Node('s')
list1.next.next.next.next.next = Node('b')

list2 = Node('g')
list2.next = Node('e')
list2.next.next = Node('e')
list2.next.next.next = Node('k')
list2.next.next.next.next = Node('s')
list2.next.next.next.next.next = Node('a')

print compare(list1, list2)

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

Output:

1

Thanks to Gaurav Ahirwar for suggesting above implementation.

### Source

<https://www.geeksforgeeks.org/compare-two-strings-represented-as-linked-lists/>

## Chapter 32

# Construct a Maximum Sum Linked List out of two Sorted Linked Lists having some Common nodes

Construct a Maximum Sum Linked List out of two Sorted Linked Lists having some Common nodes - GeeksforGeeks

Given two sorted linked lists, construct a linked list that contains maximum sum path from start to end. The result list may contain nodes from both input lists. When constructing the result list, we may switch to the other input list only at the point of intersection (which mean the two node with the same value in the lists). You are allowed to use O(1) extra space.

**Input:**

```
List1 = 1->3->30->90->120->240->511  
List2 = 0->3->12->32->90->125->240->249
```

```
Output: Following is maximum sum linked list out of two input lists  
list = 1->3->12->32->90->125->240->511  
we switch at 3 and 240 to get above maximum sum linked list
```

**We strongly recommend to minimize the browser and try this yourself first.**

The idea here in the below solution is to adjust next pointers after common nodes.

1. Start with head of both linked lists and find first common node. Use merging technique of sorted linked list for that.
2. Keep track of sum of the elements too while doing this and set head of result list based on greater sum till first common node.

3. After this till the current pointers of both lists don't become NULL we need to adjust the next of prev pointers based on greater sum.

This way it can be done in-place with constant extra space.

Time complexity of the below solution is O(n).

C++

```
// C++ program to construct the maximum sum linked
// list out of two given sorted lists
#include<iostream>
using namespace std;

//A linked list node
struct Node
{
    int data; //data belong to that node
    Node *next; //next pointer
};

// Push the data to the head of the linked list
void push(Node **head, int data)
{
    //Allocation memory to the new node
    Node *newnode = new Node;

    //Assigning data to the new node
    newnode->data = data;

    //Adjusting next pointer of the new node
    newnode->next = *head;

    //New node becomes the head of the list
    *head = newnode;
}

// Method that adjusts the pointers and prints the final list
void finalMaxSumList(Node *a, Node *b)
{
    Node *result = NULL;

    // Assigning pre and cur to the head of the
    // linked list.
    Node *pre1 = a, *curr1 = a;
    Node *pre2 = b, *curr2 = b;

    // Till either of the current pointers is not
    // NULL execute the loop
    while (curr1 != NULL || curr2 != NULL)
```

```
{  
    // Keeping 2 local variables at the start of every  
    // loop run to keep track of the sum between pre  
    // and cur pointer elements.  
    int sum1 = 0, sum2 = 0;  
  
    // Calculating sum by traversing the nodes of linked  
    // list as the merging of two linked list. The loop  
    // stops at a common node  
    while (curr1!=NULL && curr2!=NULL && curr1->data!=curr2->data)  
    {  
        if (curr1->data < curr2->data)  
        {  
            sum1 += curr1->data;  
            curr1 = curr1->next;  
        }  
        else // (curr2->data < curr1->data)  
        {  
            sum2 += curr2->data;  
            curr2 = curr2->next;  
        }  
    }  
  
    // If either of current pointers becomes NULL  
    // carry on the sum calculation for other one.  
    if (curr1 == NULL)  
    {  
        while (curr2 != NULL)  
        {  
            sum2 += curr2->data;  
            curr2 = curr2->next;  
        }  
    }  
    if (curr2 == NULL)  
    {  
        while (curr1 != NULL)  
        {  
            sum1 += curr1->data;  
            curr1 = curr1->next;  
        }  
    }  
  
    // First time adjustment of resultant head based on  
    // the maximum sum.  
    if (pre1 == a && pre2 == b)  
        result = (sum1 > sum2)? pre1 : pre2;  
  
    // If pre1 and pre2 don't contain the head pointers of
```

```
// lists adjust the next pointers of previous pointers.
else
{
    if (sum1 > sum2)
        pre2->next = pre1->next;
    else
        pre1->next = pre2->next;
}

// Adjusting previous pointers
pre1 = curr1, pre2 = curr2;

// If curr1 is not NULL move to the next.
if (curr1)
    curr1 = curr1->next;
// If curr2 is not NULL move to the next.
if (curr2)
    curr2 = curr2->next;
}

// Print the resultant list.
while (result != NULL)
{
    cout << result->data << " ";
    result = result->next;
}
}

//Main driver program
int main()
{
    //Linked List 1 : 1->3->30->90->110->120->NULL
    //Linked List 2 : 0->3->12->32->90->100->120->130->NULL
    Node *head1 = NULL, *head2 = NULL;
    push(&head1, 120);
    push(&head1, 110);
    push(&head1, 90);
    push(&head1, 30);
    push(&head1, 3);
    push(&head1, 1);

    push(&head2, 130);
    push(&head2, 120);
    push(&head2, 100);
    push(&head2, 90);
    push(&head2, 32);
    push(&head2, 12);
    push(&head2, 3);
```

```
    push(&head2, 0);

    finalMaxSumList(head1, head2);
    return 0;
}
```

**Java**

```
// Java program to construct a Maximum Sum Linked List out of
// two Sorted Linked Lists having some Common nodes
class LinkedList
{
    Node head; // head of list

    /* Linked list Node*/
    class Node
    {
        int data;
        Node next;
        Node(int d)
        {
            data = d;
            next = null;
        }
    }

    // Method to adjust pointers and print final list
    void finalMaxSumList(Node a, Node b)
    {
        Node result = null;

        /* assigning pre and cur to head
           of the linked list */
        Node pre1 = a, curr1 = a;
        Node pre2 = b, curr2 = b;

        /* Till either of current pointers is not null
           execute the loop */
        while (curr1 != null || curr2 != null)
        {
            // Keeping 2 local variables at the start of every
            // loop run to keep track of the sum between pre
            // and cur reference elements.
            int sum1 = 0, sum2 = 0;

            // Calculating sum by traversing the nodes of linked
            // list as the merging of two linked list. The loop
            // stops at a common node

```

```
while (curr1 != null && curr2 != null &&
       curr1.data != curr2.data)
{
    if (curr1.data<curr2.data)
    {
        sum1 += curr1.data;
        curr1 = curr1.next;
    }
    else
    {
        sum2 += curr2.data;
        curr2 = curr2.next;
    }
}

// If either of current pointers becomes null
// carry on the sum calculation for other one.
if (curr1 == null)
{
    while (curr2 != null)
    {
        sum2 += curr2.data;
        curr2 = curr2.next;
    }
}
if (curr2 == null)
{
    while(curr1 != null)
    {
        sum1 += curr1.data;
        curr1 = curr1.next;
    }
}

// First time adjustment of resultant head based on
// the maximum sum.
if (pre1 == a && pre2 == b)
    result = (sum1 > sum2) ? pre1 : pre2;

// If pre1 and pre2 don't contain the head references of
// lists adjust the next pointers of previous pointers.
else
{
    if (sum1 > sum2)
        pre2.next = pre1.next;
    else
        pre1.next = pre2.next;
}
```

```
}

// Adjusting previous pointers
pre1 = curr1;
pre2 = curr2;

// If curr1 is not NULL move to the next.
if (curr1 != null)
    curr1 = curr1.next;

// If curr2 is not NULL move to the next.
if (curr2 != null)
    curr2 = curr2.next;
}

while (result != null)
{
    System.out.print(result.data + " ");
    result = result.next;
}
System.out.println();
}

/* Inserts a node at start of linked list */
void push(int new_data)
{
    /* 1 & 2: Allocate the Node &
       Put in the data*/
    Node new_node = new Node(new_data);

    /* 3. Make next of new Node as head */
    new_node.next = head;

    /* 4. Move the head to point to new Node */
    head = new_node;
}

/*
 * Drier program to test above functions */
public static void main(String args[])
{
    LinkedList llist1 = new LinkedList();
    LinkedList llist2 = new LinkedList();

    //Linked List 1 : 1->3->30->90->110->120->NULL
    //Linked List 2 : 0->3->12->32->90->100->120->130->NULL

    llist1.push(120);
```

```
    llist1.push(110);
    llist1.push(90);
    llist1.push(30);
    llist1.push(3);
    llist1.push(1);

    llist2.push(130);
    llist2.push(120);
    llist2.push(100);
    llist2.push(90);
    llist2.push(32);
    llist2.push(12);
    llist2.push(3);
    llist2.push(0);

    llist1.finalMaxSumList(llist1.head, llist2.head);
}
} /* This code is contributed by Rajat Mishra */
```

### Python

```
# Python program to construct a Maximum Sum Linked List out of
# two Sorted Linked Lists having some Common nodes
class LinkedList(object):
    def __init__(self):
        # head of list
        self.head = None

    # Linked list Node
    class Node(object):
        def __init__(self, d):
            self.data = d
            self.next = None

    # Method to adjust pointers and print final list
    def finalMaxSumList(self, a, b):
        result = None
        # assigning pre and cur to head
        # of the linked list
        pre1 = a
        curr1 = a
        pre2 = b
        curr2 = b
        # Till either of current pointers is not null
        # execute the loop
        while curr1 != None or curr2 != None:
            # Keeping 2 local variables at the start of every
            # loop run to keep track of the sum between pre
```

```
# and cur reference elements.
sum1 = 0
sum2 = 0
# Calculating sum by traversing the nodes of linked
# list as the merging of two linked list. The loop
# stops at a common node
while curr1 != None and curr2 != None and curr1.data != curr2.data:
    if curr1.data < curr2.data:
        sum1 += curr1.data
        curr1 = curr1.next
    else:
        sum2 += curr2.data
        curr2 = curr2.next
# If either of current pointers becomes null
# carry on the sum calculation for other one.
if curr1 == None:
    while curr2 != None:
        sum2 += curr2.data
        curr2 = curr2.next
if curr2 == None:
    while curr1 != None:
        sum1 += curr1.data
        curr1 = curr1.next
# First time adjustment of resultant head based on
# the maximum sum.
if pre1 == a and pre2 == b:
    result = pre1 if (sum1 > sum2) else pre2
else:
    # If pre1 and pre2 don't contain the head references of
    # lists adjust the next pointers of previous pointers.
    if sum1 > sum2:
        pre2.next = pre1.next
    else:
        pre1.next = pre2.next
# Adjusting previous pointers
pre1 = curr1
pre2 = curr2
# If curr1 is not NULL move to the next.
if curr1 != None:
    curr1 = curr1.next
# If curr2 is not NULL move to the next.
if curr2 != None:
    curr2 = curr2.next

while result != None:
    print str(result.data),
    result = result.next
print ''
```

```
# Utility functions
# Inserts a new Node at front of the list.
def push(self, new_data):
    # 1 & 2: Allocate the Node &
    # Put in the data
    new_node = self.Node(new_data)
    # 3. Make next of new Node as head
    new_node.next = self.head
    # 4. Move the head to point to new Node
    self.head = new_node

# Driver program
llist1 = LinkedList()
llist2 = LinkedList()

# Linked List 1 : 1->3->30->90->110->120->NULL
# Linked List 2 : 0->3->12->32->90->100->120->130->NULL

llist1.push(120)
llist1.push(110)
llist1.push(90)
llist1.push(30)
llist1.push(3)
llist1.push(1)

llist2.push(130)
llist2.push(120)
llist2.push(100)
llist2.push(90)
llist2.push(32)
llist2.push(12)
llist2.push(3)
llist2.push(0)

llist1.finalMaxSumList(llist1.head, llist2.head)

# This code is contributed by BHAVYA JAIN
```

Output:

1 3 12 32 90 110 120 130

Time complexity =  $O(n)$  where  $n$  is the length of bigger linked list

Auxiliary space =  $O(1)$

However a problem in this solution is that the original lists are changed.

Exercise

1. Try this problem when auxiliary space is not a constraint.
2. Try this problem when we don't modify the actual list and create the resultant list.

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/maximum-sum-linked-list-two-sorted-linked-lists-common-nodes/>

## Chapter 33

# Construct a linked list from 2D matrix

Construct a linked list from 2D matrix - GeeksforGeeks

Given a matrix. Convert it into a linked list matrix such that each node is connected to its next right and down node.

Example:

Input : 2D matrix

```
1 2 3  
4 5 6  
7 8 9
```

Output :

```
1 -> 2 -> 3 -> NULL  
|     |     |  
v     v     v  
4 -> 5 -> 6 -> NULL  
|     |     |  
v     v     v  
7 -> 8 -> 9 -> NULL  
|     |     |  
v     v     v  
NULL NULL NULL
```

[Question Source : Factset Interview Experience | Set 9](#)

The idea is to construct a new node for every element of matrix and recursively create its down and right nodes.

C++

```
// CPP program to construct a linked list
// from given 2D matrix
#include <bits/stdc++.h>
using namespace std;

// struct node of linked list
struct Node {
    int data;
    Node* right, *down;
};

// returns head pointer of linked list
// constructed from 2D matrix
Node* construct(int arr[][][3], int i, int j,
                int m, int n)
{
    // return if i or j is out of bounds
    if (i > n - 1 || j > m - 1)
        return NULL;

    // create a new node for current i and j
    // and recursively allocate its down and
    // right pointers
    Node* temp = new Node();
    temp->data = arr[i][j];
    temp->right = construct(arr, i, j + 1, m, n);
    temp->down = construct(arr, i + 1, j, m, n);
    return temp;
}

// utility function for displaying
// linked list data
void display(Node* head)
{
    // pointer to move right
    Node* Rp;

    // pointer to move down
    Node* Dp = head;

    // loop till node->down is not NULL
    while (Dp) {
        Rp = Dp;

        // loop till node->right is not NULL
        while (Rp) {
            cout << Rp->data << " ";
            Rp = Rp->right;
        }
        Dp = Dp->down;
    }
}
```

```
        }
        cout << "\n";
        Dp = Dp->down;
    }
}

// driver program
int main()
{
    // 2D matrix
    int arr[][][3] = {
        { 1, 2, 3 },
        { 4, 5, 6 },
        { 7, 8, 9 }
    };

    int m = 3, n = 3;
    Node* head = construct(arr, 0, 0, m, n);
    display(head);
    return 0;
}
```

**Java**

```
// Java program to construct a linked list
// from given 2D matrix
public class Linked_list_2D_Matrix {

    // node of linked list
    static class Node {
        int data;
        Node right;
        Node down;
    };

    // returns head pointer of linked list
    // constructed from 2D matrix
    static Node construct(int arr[][][], int i, int j,
                          int m, int n) {

        // return if i or j is out of bounds
        if (i > n - 1 || j > m - 1)
            return null;

        // create a new node for current i and j
        // and recursively allocate its down and
        // right pointers
        Node temp = new Node();
```

```
temp.data = arr[i][j];
temp.right = construct(arr, i, j + 1, m, n);
temp.down = construct(arr, i + 1, j, m, n);
return temp;
}

// utility function for displaying
// linked list data
static void display(Node head) {

    // pointer to move right
    Node Rp;

    // pointer to move down
    Node Dp = head;

    // loop till node->down is not NULL
    while (Dp != null) {
        Rp = Dp;

        // loop till node->right is not NULL
        while (Rp != null) {
            System.out.print(Rp.data + " ");
            Rp = Rp.right;
        }
        System.out.println();
        Dp = Dp.down;
    }
}

// driver program
public static void main(String args[]) {
    // 2D matrix
    int arr[][] = { { 1, 2, 3 },
                    { 4, 5, 6 },
                    { 7, 8, 9 } };

    int m = 3, n = 3;
    Node head = construct(arr, 0, 0, m, n);
    display(head);
}

// This code is contributed by Sumit Ghosh
```

Output:

```
1 2 3  
4 5 6  
7 8 9
```

### Source

<https://www.geeksforgeeks.org/construct-linked-list-2d-matrix/>

## Chapter 34

# Construct a linked list from 2D matrix (Iterative Approach)

Construct a linked list from 2D matrix (Iterative Approach) - GeeksforGeeks

Given a matrix, the task is to construct a linked list matrix in which each node is connected to its right and down node.

**Example:**

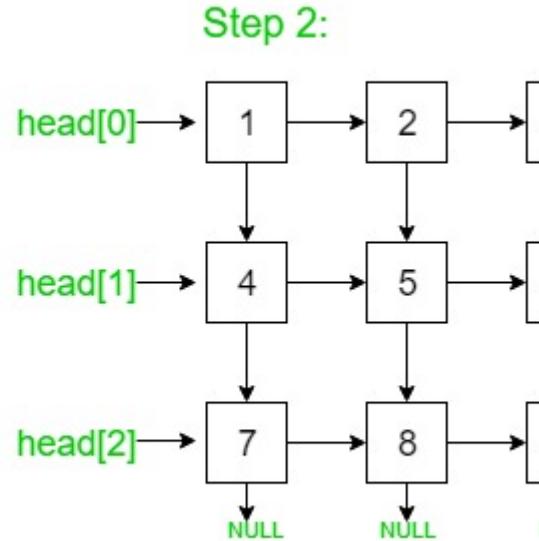
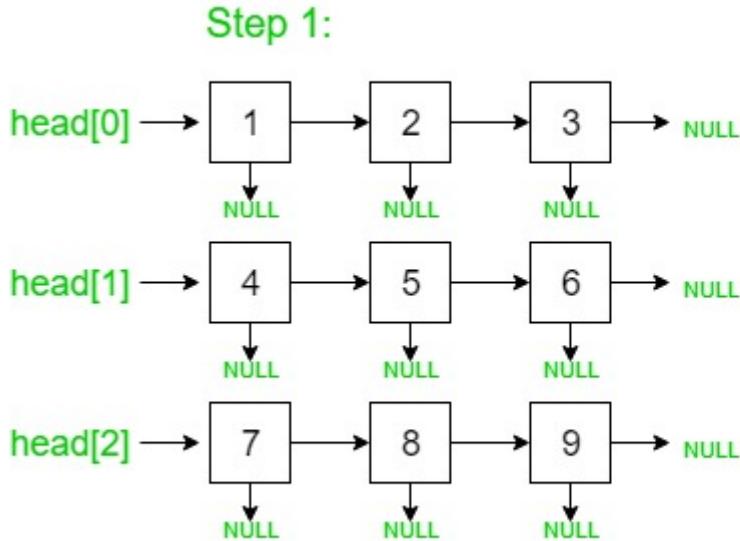
**Input:** [1 2 3  
        4 5 6  
        7 8 9]

**Output:**  
1 -> 2 -> 3 -> NULL  
| | |  
v v v  
4 -> 5 -> 6 -> NULL  
| | |  
v v v  
7 -> 8 -> 9 -> NULL  
| | |  
v v v  
NULL NULL NULL

A recursive solution for this problem has been already discussed in this [post](#). Below is an iterative approach for the problem:

- The idea is to create m linked lists (m = number of rows) whose each node stores its right node. The head pointers of each m linked lists are stored in an array of nodes.

- Then, traverse m lists, for every ith and (i+1)<sup>th</sup> list, set the down pointers of each node of i<sup>th</sup> list to its corresponding node of (i+1)<sup>th</sup> list.



Below is the implementation of the above approach:

```

// C++ program to construct a linked
// list from 2D matrix | Iterative Approach
#include <bits/stdc++.h>
using namespace std;

// struct node of linked list
struct node {
    int data;
    node *right, *down;
};

// utility function to create a new node with given data
node* newNode(int d)
{
    node* temp = new node;
    temp->data = d;
    temp->right = temp->down = NULL;
    return temp;
}
  
```

```
// utility function to print the linked list pointed to by head pointer
void display(node* head)
{
    node *rp, *dp = head;

    // loop until the down pointer is not NULL
    while (dp) {
        rp = dp;

        // loop until the right pointer is not NULL
        while (rp) {
            cout << rp->data << " ";
            rp = rp->right;
        }
        cout << endl;
        dp = dp->down;
    }
}

// function which constructs the linked list
// from the given matrix of size m * n
// and returns the head pointer of the linked list
node* constructLinkedMatrix(int mat[][][3], int m, int n)
{
    // stores the head of the linked list
    node* mainhead = NULL;

    // stores the head of linked lists of each row
    node* head[m];
    node *righttemp, *newptr;

    // Firstly, we create m linked lists
    // by setting all the right nodes of every row
    for (int i = 0; i < m; i++) {

        // initially set the head of ith row as NULL
        head[i] = NULL;
        for (int j = 0; j < n; j++) {
            newptr = newNode(mat[i][j]);

            // stores the mat[0][0] node as
            // the mainhead of the linked list
            if (!mainhead)
                mainhead = newptr;

            if (!head[i])
                head[i] = newptr;
            else
```

```
    righttemp->right = newptr;

    righttemp = newptr;
}
}

// Then, for every ith and (i+1)th list,
// we set the down pointers of
// every node of ith list
// with its corresponding
// node of (i+1)th list
for (int i = 0; i < m - 1; i++) {

    node *temp1 = head[i], *temp2 = head[i + 1];

    while (temp1 && temp2) {

        temp1->down = temp2;
        temp1 = temp1->right;
        temp2 = temp2->right;
    }
}

// return the mainhead pointer of the linked list
return mainhead;
}

// Driver program to test the above function
int main()
{
    int m, n; // m = rows and n = columns
    m = 3, n = 3;
    // 2D matrix
    int mat[][][3] = { { 1, 2, 3 },
                      { 4, 5, 6 },
                      { 7, 8, 9 } };

    node* head = constructLinkedMatrix(mat, m, n);
    display(head);

    return 0;
}
```

**Output:**

```
1 2 3
4 5 6
```

7 8 9

**Time complexity:**  $O(M * N)$

**Improved By :** [souravdutta123](#)

## Source

<https://www.geeksforgeeks.org/construct-a-linked-list-from-2d-matrix-iterative-approach/>

## Chapter 35

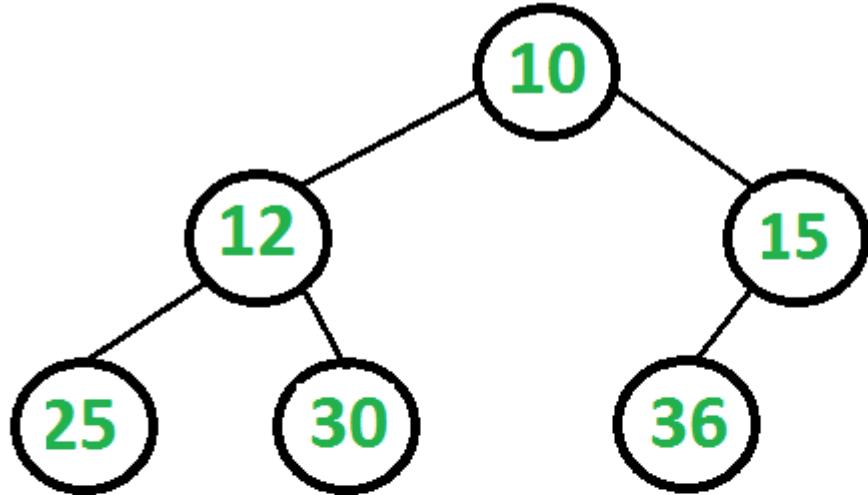
# Convert a Binary Tree to a Circular Doubly Link List

Convert a Binary Tree to a Circular Doubly Link List - GeeksforGeeks

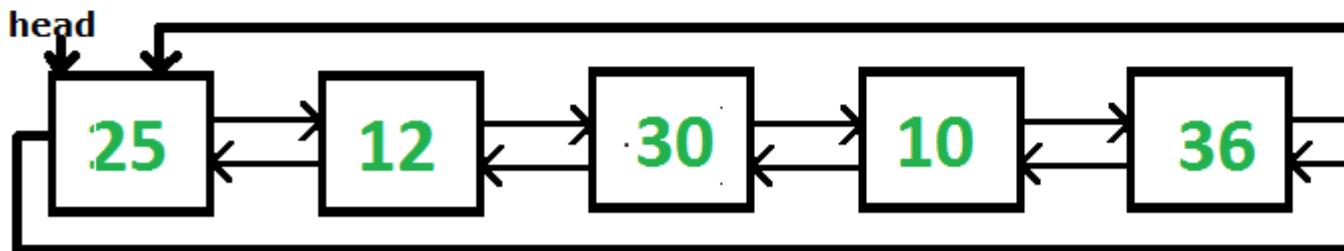
Given a Binary Tree, convert it to a Circular Doubly Linked List (In-Place).

- 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

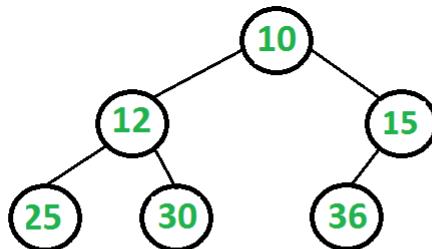
<https://www.geeksforgeeks.org/convert-a-binary-tree-to-a-circular-doubly-link-list/>

## Chapter 36

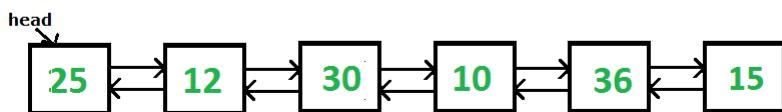
# Convert a given Binary Tree to Doubly Linked List | Set 1

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**

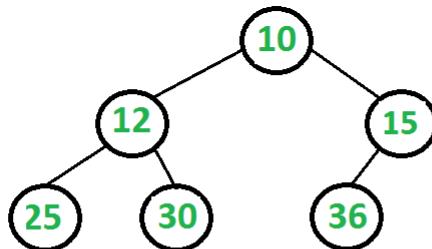
<https://www.geeksforgeeks.org/in-place-convert-a-given-binary-tree-to-doubly-linked-list/>

## Chapter 37

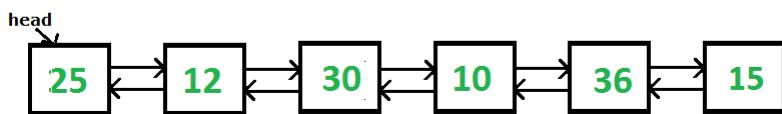
# Convert a given Binary Tree to Doubly Linked List | Set 2

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 Inorder Tree Traversal\n"
inorder(root)

# Static variable pre for function fixPrevPtr
fixPrevPtr.pre = None
head = BTToDLL(root)

print "\n\t DLL 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

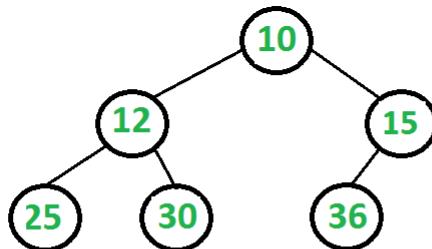
<https://www.geeksforgeeks.org/convert-a-given-binary-tree-to-doubly-linked-list-set-2/>

## Chapter 38

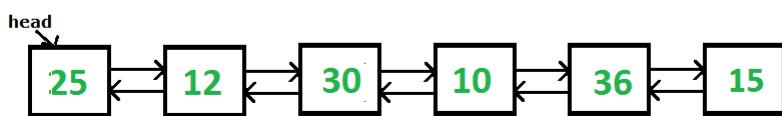
# Convert a given Binary Tree to Doubly Linked List | Set 3

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**

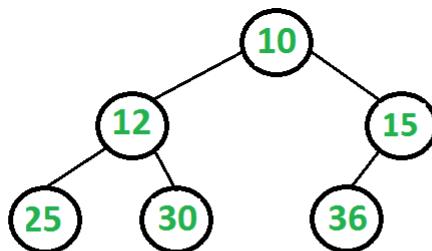
<https://www.geeksforgeeks.org/convert-given-binary-tree-doubly-linked-list-set-3/>

## Chapter 39

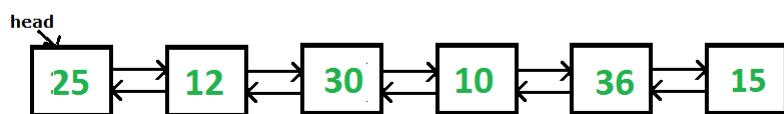
# Convert a given Binary Tree to Doubly Linked List | Set 4

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(int data)
{
    Node* node = 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 40

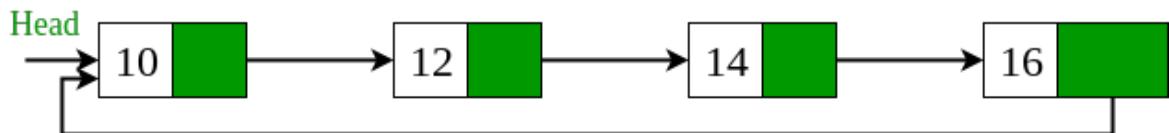
# Convert singly linked list into circular linked list

Convert singly linked list into circular linked list - GeeksforGeeks

Given a singly linked list, we have to convert it into circular linked list. For example, we have been given a singly linked list with four nodes and we want to convert this singly linked list into circular linked list.



The above singly linked list is converted into circular linked list.



**Approach:** The idea is to traverse the singly linked list and check if the node is the last node or not. If the node is the last node i.e pointing to NULL then make it point to the starting node i.e head node. Below is the implementation of this approach.

```
// Program for converting singly linked list
// into circular linked list.
#include <bits/stdc++.h>

/* Linked list node */
struct Node {
    int data;
    struct Node* next;
};
```

```
// Function that convert singly linked list
// into circular linked list.
struct Node* circular(struct Node* head)
{
    // declare a node variable start and
    // assign head node into start node.
    struct Node* start = head;

    // check that while head->next not equal
    // to NULL then head points to next node.
    while (head->next != NULL)
        head = head->next;

    // if head->next points to NULL then
    // start assign to the head->next node.
    head->next = start;
    return start;
}

void push(struct Node** head, int data)
{
    // Allocate dynamic memory for newNode.
    struct Node* newNode = (struct Node*)malloc
                           (sizeof(struct Node));

    // Assign the data into newNode.
    newNode->data = data;

    // newNode->next assign the address of
    // head node.
    newNode->next = (*head);

    // newNode become the headNode.
    (*head) = newNode;
}

// Function that display the elements of
// circular linked list.
void displayList(struct Node* node)
{
    struct Node* start = node;

    while (node->next != start) {
        printf("%d ", node->data);
        node = node->next;
    }
}
```

```
// Display the last node of circular
// linked list.
printf("%d ", node->data);
}

// Drier program to test the functions
int main()
{
    // Start with empty list
    struct Node* head = NULL;

    // Using push() function to construct
    // singly linked list
    // 17->22->13->14->15
    push(&head, 15);
    push(&head, 14);
    push(&head, 13);
    push(&head, 22);
    push(&head, 17);

    // Call the circular_list function that
    // convert singly linked list to circular
    // linked list.
    circular(head);

    printf("Display list: \n");
    displayList(head);

    return 0;
}
```

Output:

```
Display list:
17 22 13 14 15
```

## Source

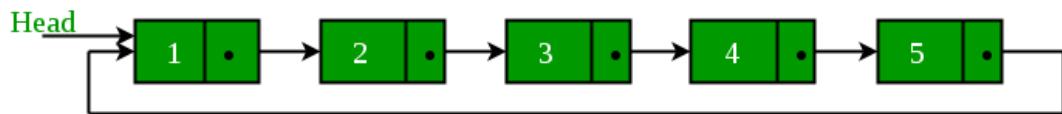
<https://www.geeksforgeeks.org/convert-singly-linked-list-circular-linked-list/>

# Chapter 41

## Count nodes in Circular linked list

Count nodes in Circular linked list - GeeksforGeeks

Given a circular linked list, count number of nodes in it. For example output is 5 for below list.



We use the concept used in [Circular Linked List | Set 2 \(Traversal\)](#). While traversing, we keep track of count of nodes.

```
// C program to count number of nodes in
// a circular linked list.
#include <stdio.h>
#include <stdlib.h>

/* structure for a node */
struct Node {
    int data;
    struct Node* next;
};

/* Function to insert a node at the begining
   of a Circular linked list */
void push(struct Node** head_ref, int data)
{
    struct Node* ptr1 = (struct Node*)malloc(sizeof(struct Node));
    struct Node* temp = *head_ref;
    ptr1->data = data;
```

```
ptr1->next = *head_ref;

/* If linked list is not NULL then set
   the next of last node */
if (*head_ref != NULL) {
    while (temp->next != *head_ref)
        temp = temp->next;
    temp->next = ptr1;
} else
    ptr1->next = ptr1; /*For the first node */

*head_ref = ptr1;
}

/* Function to print nodes in a given Circular
   linked list */
int countNodes(struct Node* head)
{
    struct Node* temp = head;
    int result = 0;
    if (head != NULL) {
        do {
            temp = temp->next;
            result++;
        } while (temp != head);
    }

    return result;
}

/* Driver program to test above functions */
int main()
{
    /* Initialize lists as empty */
    struct Node* head = NULL;
    push(&head, 12);
    push(&head, 56);
    push(&head, 2);
    push(&head, 11);

    printf("%d", countNodes(head));

    return 0;
}
```

## Source

<https://www.geeksforgeeks.org/count-nodes-circular-linked-list/>

## Chapter 42

# Count pairs from two linked lists whose sum is equal to a given value

Count pairs from two linked lists whose sum is equal to a given value - GeeksforGeeks

Given two linked lists(can be sorted or unsorted) of size **n1** and **n2** of distinct elements. Given a value **x**. The problem is to count all pairs from both lists whose sum is equal to the given value **x**.

**Note:** The pair has an element from each linked list.

Examples:

```
Input : list1 = 3->1->5->7
        list2 = 8->2->5->3
        x = 10
Output : 2
The pairs are:
(5, 5) and (7, 3)

Input : list1 = 4->3->5->7->11->2->1
        list2 = 2->3->4->5->6->8-12
        x = 9
Output : 5
```

**Method 1 (Naive Approach):** Using two loops pick elements from both the linked lists and check whether the sum of the pair is equal to **x** or not.

C/C++

```
// C++ implementation to count pairs from both linked
// lists whose sum is equal to a given value
#include <bits/stdc++.h>
using namespace std;

/* A Linked list node */
struct Node
{
    int data;
    struct Node* next;
};

// function to insert a node at the
// beginning of the linked list
void push(struct Node** head_ref, int new_data)
{
    /* allocate node */
    struct Node* new_node =
        (struct Node*) malloc(sizeof(struct Node));

    /* put in the data */
    new_node->data = new_data;

    /* link the old list to the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref) = new_node;
}

// function to count all pairs from both the linked lists
// whose sum is equal to a given value
int countPairs(struct Node* head1, struct Node* head2, int x)
{
    int count = 0;

    struct Node *p1, *p2;

    // traverse the 1st linked list
    for (p1 = head1; p1 != NULL; p1 = p1->next)

        // for each node of 1st list
        // traverse the 2nd list

        for (p2 = head2; p2 != NULL; p2 = p2->next)

            // if sum of pair is equal to 'x'
            // increment count
```

```
        if ((p1->data + p2->data) == x)
            count++;

        // required count of pairs
        return count;
    }

// Driver program to test above
int main()
{
    struct Node* head1 = NULL;
    struct Node* head2 = NULL;

    // create linked list1 3->1->5->7
    push(&head1, 7);
    push(&head1, 5);
    push(&head1, 1);
    push(&head1, 3);

    // create linked list2 8->2->5->3
    push(&head2, 3);
    push(&head2, 5);
    push(&head2, 2);
    push(&head2, 8);

    int x = 10;

    cout << "Count = "
        << countPairs(head1, head2, x);
    return 0;
}
```

### Java

```
// Java implementation to count pairs from both linked
// lists whose sum is equal to a given value

// Note : here we use java.util.LinkedList for
// linked list implementation

import java.util.Arrays;
import java.util.Iterator;
import java.util.LinkedList;

class GFG
{
    // method to count all pairs from both the linked lists
    // whose sum is equal to a given value
```

```
static int countPairs(LinkedList<Integer> head1, LinkedList<Integer> head2, int x)
{
    int count = 0;

    // traverse the 1st linked list
    Iterator<Integer> itr1 = head1.iterator();
    while(itr1.hasNext())
    {
        // for each node of 1st list
        // traverse the 2nd list
        Iterator<Integer> itr2 = head2.iterator();

        while(itr2.hasNext())
        {
            // if sum of pair is equal to 'x'
            // increment count
            if ((itr1.next() + itr2.next()) == x)
                count++;
        }
    }

    // required count of pairs
    return count;
}

// Driver method
public static void main(String[] args)
{
    Integer arr1[] = {3, 1, 5, 7};
    Integer arr2[] = {8, 2, 5, 3};

    // create linked list1 3->1->5->7
    LinkedList<Integer> head1 = new LinkedList<>(Arrays.asList(arr1));

    // create linked list2 8->2->5->3
    LinkedList<Integer> head2 = new LinkedList<>(Arrays.asList(arr2));

    int x = 10;

    System.out.println("Count = " + countPairs(head1, head2, x));
}
```

Output:

Count = 2

Time Complexity:  $O(n_1 \cdot n_2)$

Auxiliary Space:  $O(1)$

**Method 2 (Sorting):** Sort the 1st linked list in ascending order and the 2nd linked list in descending order using merge sort technique. Now traverse both the lists from left to right in the following way:

**Algorithm:**

```
countPairs(list1, list2, x)
    Initialize count = 0
    while list != NULL and list2 != NULL
        if (list1->data + list2->data) == x
            list1 = list1->next
            list2 = list2->next
            count++
        else if (list1->data + list2->data) > x
            list2 = list2->next
        else
            list1 = list1->next

    return count
```

For simplicity, the implementation given below assumes that list1 is sorted in ascending order and list2 is sorted in descending order.

**C/C++**

```
// C++ implementation to count pairs from both linked
// lists whose sum is equal to a given value
#include <bits/stdc++.h>
using namespace std;

/* A Linked list node */
struct Node
{
    int data;
    struct Node* next;
};

// function to insert a node at the
// beginning of the linked list
void push(struct Node** head_ref, int new_data)
{
    /* allocate node */
    struct Node* new_node =

```

```
(struct Node*) malloc(sizeof(struct Node));

/* put in the data */
new_node->data = new_data;

/* link the old list to the new node */
new_node->next = (*head_ref);

/* move the head to point to the new node */
(*head_ref) = new_node;
}

// function to count all pairs from both the linked
// lists whose sum is equal to a given value
int countPairs(struct Node* head1, struct Node* head2,
               int x)
{
    int count = 0;

    // sort head1 in ascending order and
    // head2 in descending order
    // sort (head1), sort (head2)
    // For simplicity both lists are considered to be
    // sorted in the respective orders

    // traverse both the lists from left to right
    while (head1 != NULL && head2 != NULL)
    {
        // if this sum is equal to 'x', then move both
        // the lists to next nodes and increment 'count'
        if ((head1->data + head2->data) == x)
        {
            head1 = head1->next;
            head2 = head2->next;
            count++;
        }

        // if this sum is greater than x, then
        // move head2 to next node
        else if ((head1->data + head2->data) > x)
            head2 = head2->next;

        // else move head1 to next node
        else
            head1 = head1->next;
    }

    // required count of pairs
}
```

```
    return count;
}

// Driver program to test above
int main()
{
    struct Node* head1 = NULL;
    struct Node* head2 = NULL;

    // create linked list1 1->3->5->7
    // assumed to be in ascending order
    push(&head1, 7);
    push(&head1, 5);
    push(&head1, 3);
    push(&head1, 1);

    // create linked list2 8->5->3->2
    // assumed to be in descending order
    push(&head2, 2);
    push(&head2, 3);
    push(&head2, 5);
    push(&head2, 8);

    int x = 10;

    cout << "Count = "
         << countPairs(head1, head2, x);
    return 0;
}
```

### Java

```
// Java implementation to count pairs from both linked
// lists whose sum is equal to a given value

// Note : here we use java.util.LinkedList for
// linked list implementation

import java.util.Arrays;
import java.util.Collections;
import java.util.Iterator;
import java.util.LinkedList;

class GFG
{
    // method to count all pairs from both the linked lists
    // whose sum is equal to a given value
    static int countPairs(LinkedList<Integer> head1, LinkedList<Integer> head2, int x)
```

```
{  
    int count = 0;  
  
    // sort head1 in ascending order and  
    // head2 in descending order  
    Collections.sort(head1);  
    Collections.sort(head2,Collections.reverseOrder());  
  
    // traverse both the lists from left to right  
    Iterator<Integer> itr1 = head1.iterator();  
    Iterator<Integer> itr2 = head2.iterator();  
  
    Integer num1 = itr1.hasNext() ? itr1.next() : null;  
    Integer num2 = itr2.hasNext() ? itr2.next() : null;  
  
    while(num1 != null && num2 != null)  
    {  
  
        // if this sum is equal to 'x', then move both  
        // the lists to next nodes and increment 'count'  
  
        if ((num1 + num2) == x)  
        {  
            num1 = itr1.hasNext() ? itr1.next() : null;  
            num2 = itr2.hasNext() ? itr2.next() : null;  
  
            count++;  
        }  
  
        // if this sum is greater than x, then  
        // move itr2 to next node  
        else if ((num1 + num2) > x)  
            num2 = itr2.hasNext() ? itr2.next() : null;  
  
        // else move itr1 to next node  
        else  
            num1 = itr1.hasNext() ? itr1.next() : null;  
  
    }  
  
    // required count of pairs  
    return count;  
}  
  
// Driver method  
public static void main(String[] args)  
{  
    Integer arr1[] = {3, 1, 5, 7};
```

```
Integer arr2[] = {8, 2, 5, 3};

// create linked list1 3->1->5->7
LinkedList<Integer> head1 = new LinkedList<>(Arrays.asList(arr1));

// create linked list2 8->2->5->3
LinkedList<Integer> head2 = new LinkedList<>(Arrays.asList(arr2));

int x = 10;

System.out.println("Count = " + countPairs(head1, head2, x));
}
}
```

Output:

```
Count = 2
```

Time Complexity:  $O(n_1 \log n_1) + O(n_2 \log n_2)$

Auxiliary Space:  $O(1)$

Sorting will change the order of nodes. If order is important, then copy of the linked lists can be created and used.

**Method 3 (Hashing):** Hash table is implemented using [unordered\\_set in C++](#). We store all first linked list elements in hash table. For elements of second linked list, we subtract every element from  $x$  and check the result in hash table. If result is present, we increment the **count**.

C++

```
// C++ implementation to count pairs from both linked
// lists whose sum is equal to a given value
#include <bits/stdc++.h>
using namespace std;

/* A Linked list node */
struct Node
{
    int data;
    struct Node* next;
};

// function to insert a node at the
// beginning of the linked list
void push(struct Node** head_ref, int new_data)
{
    /* allocate node */

```

```
struct Node* new_node =
    (struct Node*) malloc(sizeof(struct Node));

/* put in the data */
new_node->data = new_data;

/* link the old list to the new node */
new_node->next = (*head_ref);

/* move the head to point to the new node */
(*head_ref) = new_node;
}

// function to count all pairs from both the linked
// lists whose sum is equal to a given value
int countPairs(struct Node* head1, struct Node* head2,
               int x)
{
    int count = 0;

    unordered_set<int> us;

    // insert all the elements of 1st list
    // in the hash table(unordered_set 'us')
    while (head1 != NULL)
    {
        us.insert(head1->data);

        // move to next node
        head1 = head1->next;
    }

    // for each element of 2nd list
    while (head2 != NULL)
    {
        // find (x - head2->data) in 'us'
        if (us.find(x - head2->data) != us.end())
            count++;

        // move to next node
        head2 = head2->next;
    }
    // required count of pairs
    return count;
}

// Driver program to test above
int main()
```

```
{  
    struct Node* head1 = NULL;  
    struct Node* head2 = NULL;  
  
    // create linked list1 3->1->5->7  
    push(&head1, 7);  
    push(&head1, 5);  
    push(&head1, 1);  
    push(&head1, 3);  
  
    // create linked list2 8->2->5->3  
    push(&head2, 3);  
    push(&head2, 5);  
    push(&head2, 2);  
    push(&head2, 8);  
  
    int x = 10;  
  
    cout << "Count = "  
        << countPairs(head1, head2, x);  
    return 0;  
}
```

**Java**

```
// Java implementation to count pairs from both linked  
// lists whose sum is equal to a given value  
  
// Note : here we use java.util.LinkedList for  
// linked list implementation  
  
import java.util.Arrays;  
import java.util.HashSet;  
import java.util.Iterator;  
import java.util.LinkedList;  
  
class GFG  
{  
    // method to count all pairs from both the linked lists  
    // whose sum is equal to a given value  
    static int countPairs(LinkedList<Integer> head1, LinkedList<Integer> head2, int x)  
    {  
        int count = 0;  
  
        HashSet<Integer> us = new HashSet<Integer>();  
  
        // insert all the elements of 1st list  
        // in the hash table(unordered_set 'us')  
}
```

```
Iterator<Integer> itr1 = head1.iterator();
while (itr1.hasNext())
{
    us.add(itr1.next());
}

Iterator<Integer> itr2 = head2.iterator();
// for each element of 2nd list
while (itr2.hasNext())
{
    // find (x - head2->data) in 'us'
    if (us.add(x - itr2.next()))
        count++;

}

// required count of pairs
return count;
}

// Driver method
public static void main(String[] args)
{
    Integer arr1[] = {3, 1, 5, 7};
    Integer arr2[] = {8, 2, 5, 3};

    // create linked list1 3->1->5->7
    LinkedList<Integer> head1 = new LinkedList<>(Arrays.asList(arr1));

    // create linked list2 8->2->5->3
    LinkedList<Integer> head2 = new LinkedList<>(Arrays.asList(arr2));

    int x = 10;

    System.out.println("Count = " + countPairs(head1, head2, x));
}
```

Output:

Count = 2

Time Complexity: O(n1 + n2)

Auxiliary Space: O(n1), hash table should be created of the array having smaller size so as to reduce the space complexity.

**Source**

<https://www.geeksforgeeks.org/count-pairs-two-linked-lists-whose-sum-equal-given-value/>

## Chapter 43

# 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

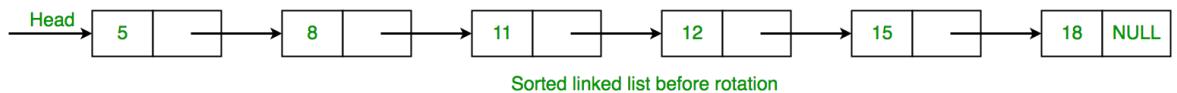
<https://www.geeksforgeeks.org/count-pairs-in-a-binary-tree-whose-sum-is-equal-to-a-given-value-x/>

## Chapter 44

# Count rotations in sorted and rotated linked list

Count rotations in sorted and rotated linked list - GeeksforGeeks

Given a linked list of n nodes which is first sorted, then rotated by k elements. Find the value of k.



The idea is to traverse singly linked list to check condition whether current node value is greater than value of next node. If the given condition is true, then break the loop. Otherwise increase the counter variable and increase the node by node->next. Below is the implementation of this approach.

```
// Program for count number of rotations in
// sorted linked list.
#include <bits/stdc++.h>
using namespace std;
/* Linked list node */
struct Node {
    int data;
    struct Node* next;
};
```

```
// Function that count number of
// rotation in singly linked list.
int countRotation(struct Node* head)
{
    // declare count variable and assign it 1.
    int count = 0;

    // declare a min variable and assign to
    // data of head node.
    int min = head->data;

    // check that while head not equal to NULL.
    while (head != NULL) {

        // if min value is greater then head->data
        // then it breaks the while loop and
        // return the value of count.
        if (min > head->data)
            break;

        count++;

        // head assign the next value of head.
        head = head->next;
    }
    return count;
}

// Function to push element in linked list.
void push(struct Node** head, int data)
{
    // Allocate dynamic memory for newNode.
    struct Node* newNode = new Node;

    // Assign the data into newNode.
    newNode->data = data;

    // newNode->next assign the address of
    // head node.
    newNode->next = (*head);

    // newNode become the headNode.
    (*head) = newNode;
}

// Display linked list.
void printList(struct Node* node)
```

```
{  
    while (node != NULL) {  
        printf("%d ", node->data);  
        node = node->next;  
    }  
}  
  
// Drier functions  
int main()  
{  
    // Create a node and initialize with NULL  
    struct Node* head = NULL;  
  
    // push() insert node in linked list.  
    // 15 -> 18 -> 5 -> 8 -> 11 -> 12  
    push(&head, 12);  
    push(&head, 11);  
    push(&head, 8);  
    push(&head, 5);  
    push(&head, 18);  
    push(&head, 15);  
  
    printList(head);  
    cout << endl;  
    cout << "Linked list rotated elements: "  
  
    // Function call countRotation()  
    cout << countRotation(head) << endl;  
  
    return 0;  
}
```

Output:

```
Linked List:  
15 18 5 8 11 12  
Linked list rotated elements: 2
```

## Source

<https://www.geeksforgeeks.org/count-rotations-sorted-rotated-linked-list/>

## Chapter 45

# Count triplets in a sorted doubly linked list whose sum is equal to a given value x

Count triplets in a sorted doubly linked list whose sum is equal to a given value x - Geeks-forGeeks

Given a sorted doubly linked list of distinct nodes(no two nodes have the same data) and a value **x**. Count triplets in the list that sum up to a given value **x**.

Examples:

```
Input : DLL: 1 <-> 2 <-> 4 <-> 5 <-> 6 <-> 8 <-> 9  
        x = 17  
Output : 2  
The triplets are:  
(2, 6, 9) and (4, 5, 8)
```

```
Input : DLL: 1 <-> 2 <-> 4 <-> 5 <-> 6 <-> 8 <-> 9  
        x = 15  
Output : 5
```

### Method 1 (Naive Approach):

Using three nested loops generate all triplets and check whether elements in the triplet sum up to **x** or not.

```
// C++ implementation to count triplets in a sorted doubly linked list  
// whose sum is equal to a given value 'x'  
#include <bits/stdc++.h>  
  
using namespace std;
```

```
// structure of node of doubly linked list
struct Node {
    int data;
    struct Node* next, *prev;
};

// function to count triplets in a sorted doubly linked list
// whose sum is equal to a given value 'x'
int countTriplets(struct Node* head, int x)
{
    struct Node* ptr1, *ptr2, *ptr3;
    int count = 0;

    // generate all possible triplets
    for (ptr1 = head; ptr1 != NULL; ptr1 = ptr1->next)
        for (ptr2 = ptr1->next; ptr2 != NULL; ptr2 = ptr2->next)
            for (ptr3 = ptr2->next; ptr3 != NULL; ptr3 = ptr3->next)

                // if elements in the current triplet sum up to 'x'
                if ((ptr1->data + ptr2->data + ptr3->data) == x)

                    // increment count
                    count++;

    // required count of triplets
    return count;
}

// A utility function to insert a new node at the
// beginning of doubly linked list
void insert(struct Node** head, int data)
{
    // allocate node
    struct Node* temp = new Node();

    // put in the data
    temp->data = data;
    temp->next = temp->prev = NULL;

    if ((*head) == NULL)
        (*head) = temp;
    else {
        temp->next = *head;
        (*head)->prev = temp;
        (*head) = temp;
    }
}
```

```
// Driver program to test above
int main()
{
    // start with an empty doubly linked list
    struct Node* head = NULL;

    // insert values in sorted order
    insert(&head, 9);
    insert(&head, 8);
    insert(&head, 6);
    insert(&head, 5);
    insert(&head, 4);
    insert(&head, 2);
    insert(&head, 1);

    int x = 17;

    cout << "Count = "
        << countTriplets(head, x);
    return 0;
}
```

Output:

```
Count = 2
```

Time Complexity:  $O(n^3)$

Auxiliary Space:  $O(1)$

#### Method 2 (Hashing):

Create a hash table with **(key, value)** tuples represented as **(node data, node pointer tuples)**. Traverse the doubly linked list and store each node's data and its pointer pair(tuple) in the hash table. Now, generate each possible pair of nodes. For each pair of nodes, calculate the **p\_sum**(sum of data in the two nodes) and check whether **(x-p\_sum)** exists in the hash table or not. If it exists, then also verify that the two nodes in the pair are not same to the node associated with **(x-p\_sum)** in the hash table and finally increment **count**. Return **(count / 3)** as each triplet is counted 3 times in the above process.

```
// C++ implementation to count triplets in a sorted doubly linked list
// whose sum is equal to a given value 'x'
#include <bits/stdc++.h>

using namespace std;

// structure of node of doubly linked list
struct Node {
```

```
int data;
struct Node* next, *prev;
};

// function to count triplets in a sorted doubly linked list
// whose sum is equal to a given value 'x'
int countTriplets(struct Node* head, int x)
{
    struct Node* ptr, *ptr1, *ptr2;
    int count = 0;

    // unordered_map 'um' implemented as hash table
    unordered_map<int, Node*> um;

    // insert the <node data, node pointer> tuple in 'um'
    for (ptr = head; ptr != NULL; ptr = ptr->next)
        um[ptr->data] = ptr;

    // generate all possible pairs
    for (ptr1 = head; ptr1 != NULL; ptr1 = ptr1->next)
        for (ptr2 = ptr1->next; ptr2 != NULL; ptr2 = ptr2->next) {

            // p_sum - sum of elements in the current pair
            int p_sum = ptr1->data + ptr2->data;

            // if 'x-p_sum' is present in 'um' and either of the two nodes
            // are not equal to the 'um[x-p_sum]' node
            if (um.find(x - p_sum) != um.end() && um[x - p_sum] != ptr1
                && um[x - p_sum] != ptr2)

                // increment count
                count++;
        }

        // required count of triplets
        // division by 3 as each triplet is counted 3 times
        return (count / 3);
}

// A utility function to insert a new node at the
// beginning of doubly linked list
void insert(struct Node** head, int data)
{
    // allocate node
    struct Node* temp = new Node();

    // put in the data
    temp->data = data;
```

```
temp->next = temp->prev = NULL;

if ((*head) == NULL)
    (*head) = temp;
else {
    temp->next = *head;
    (*head)->prev = temp;
    (*head) = temp;
}
}

// Driver program to test above
int main()
{
    // start with an empty doubly linked list
    struct Node* head = NULL;

    // insert values in sorted order
    insert(&head, 9);
    insert(&head, 8);
    insert(&head, 6);
    insert(&head, 5);
    insert(&head, 4);
    insert(&head, 2);
    insert(&head, 1);

    int x = 17;

    cout << "Count = "
        << countTriplets(head, x);
    return 0;
}
```

Output:

```
Count = 2
```

Time Complexity:  $O(n^2)$

Auxiliary Space:  $O(n)$

### Method 3 Efficient Approach(Use of two pointers):

Traverse the doubly linked list from left to right. For each **current** node during the traversal, initialize two pointers **first** = pointer to the node next to the **current** node and **last** = pointer to the last node of the list. Now, count pairs in the list from **first** to **last** pointer that sum up to value ( $x - \text{current node's data}$ ) (algorithm described in [this](#) post). Add this count to the **total\_count** of triplets. Pointer to the **last** node can be found only once in the beginning.

```
// C++ implementation to count triplets in a sorted doubly linked list
// whose sum is equal to a given value 'x'
#include <bits/stdc++.h>

using namespace std;

// structure of node of doubly linked list
struct Node {
    int data;
    struct Node* next, *prev;
};

// function to count pairs whose sum equal to given 'value'
int countPairs(struct Node* first, struct Node* second, int value)
{
    int count = 0;

    // The loop terminates when either of two pointers
    // become NULL, or they cross each other (second->next
    // == first), or they become same (first == second)
    while (first != NULL && second != NULL &&
           first != second && second->next != first) {

        // pair found
        if ((first->data + second->data) == value) {

            // increment count
            count++;

            // move first in forward direction
            first = first->next;

            // move second in backward direction
            second = second->prev;
        }

        // if sum is greater than 'value'
        // move second in backward direction
        else if ((first->data + second->data) > value)
            second = second->prev;

        // else move first in forward direction
        else
            first = first->next;
    }

    // required count of pairs
    return count;
}
```

```
}  
  
// function to count triplets in a sorted doubly linked list  
// whose sum is equal to a given value 'x'  
int countTriplets(struct Node* head, int x)  
{  
    // if list is empty  
    if (head == NULL)  
        return 0;  
  
    struct Node* current, *first, *last;  
    int count = 0;  
  
    // get pointer to the last node of  
    // the doubly linked list  
    last = head;  
    while (last->next != NULL)  
        last = last->next;  
  
    // traversing the doubly linked list  
    for (current = head; current != NULL; current = current->next) {  
  
        // for each current node  
        first = current->next;  
  
        // count pairs with sum(x - current->data) in the range  
        // first to last and add it to the 'count' of triplets  
        count += countPairs(first, last, x - current->data);  
    }  
  
    // required count of triplets  
    return count;  
}  
  
// A utility function to insert a new node at the  
// beginning of doubly linked list  
void insert(struct Node** head, int data)  
{  
    // allocate node  
    struct Node* temp = new Node();  
  
    // put in the data  
    temp->data = data;  
    temp->next = temp->prev = NULL;  
  
    if ((*head) == NULL)  
        (*head) = temp;  
    else {
```

```
    temp->next = *head;
    (*head)->prev = temp;
    (*head) = temp;
}
}

// Driver program to test above
int main()
{
    // start with an empty doubly linked list
    struct Node* head = NULL;

    // insert values in sorted order
    insert(&head, 9);
    insert(&head, 8);
    insert(&head, 6);
    insert(&head, 5);
    insert(&head, 4);
    insert(&head, 2);
    insert(&head, 1);

    int x = 17;

    cout << "Count = "
        << countTriplets(head, x);
    return 0;
}
```

Output:

Count = 2

Time Complexity:  $O(n^2)$   
Auxiliary Space:  $O(1)$

## Source

<https://www.geeksforgeeks.org/count-triplets-sorted-doubly-linked-list-whose-sum-equal-given-value-x/>

## Chapter 46

# Create a Doubly Linked List from a Ternary Tree

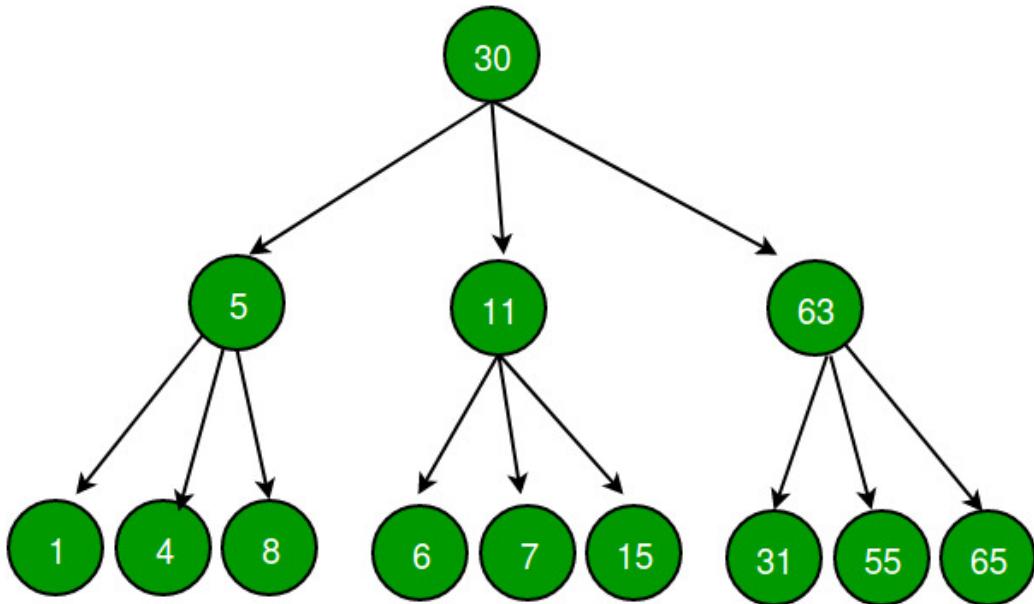
Create a Doubly Linked List from a Ternary Tree - GeeksforGeeks

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);

//recurve 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 47

# Data Structure design to perform required operations

Data Structure design to perform required operations - GeeksforGeeks

Design a data structure which can do following operations

1. add() in  $O(n)$
2. getMinimum() in  $O(1)$
3. deleteMinimum() in  $O(1)$

Source :[MakeMyTrip Interview.](#)

1. Maintain a linkedlist with elements in increasing order.
2. Move head to next position in case of delete Min operation.
3. Return First element in case of get Min Operation.

```
// Java code for linked list to
// perform required operations
import java.util.*;

// Node class
class Node
{
    int data;
    Node next;

    Node(int d)
    {
        data = d;
        next = null;
    }
}
```

```
}

// main class
class MinDS
{
    Node start;

    public MinDS()
    {
        start = null;
    }

    // Function to add element
    void addElement(int d)
    {
        Node tmp = new Node(d);

        // If linked list is empty
        if (start == null)
        {
            start = tmp;
            return;
        }

        // If head itself is greater
        if (d < start.data)
        {
            tmp.next = start;
            start = tmp;
            return;
        }

        // If need to insert somewhere in middle
        Node prev = start;
        Node ptr = start.next;
        while (ptr != null)
        {
            if (d < ptr.data)
            {
                tmp.next = ptr;
                prev.next = tmp;
                return;
            }
            else
            {
                prev = ptr;
                ptr = ptr.next;
            }
        }
    }
}
```

```
        }
        prev.next = tmp;
    }

    // Function to get minimum
    int getMin()
    {
        return start.data;
    }

    // Function to delete minimum
    int delMin()
    {
        int min = start.data;
        start = start.next;
        return min;
    }

    // Function to print elements
    void print()
    {
        Node ptr = start;
        System.out.print("Elements: ");
        while (ptr != null)
        {
            System.out.print(ptr.data + ", ");
            ptr = ptr.next;
        }
        System.out.println("\n");
    }

    // Driver code
    public static void main(String[] args)
    {
        MinDS x = new MinDS();
        x.addElement(10);
        x.addElement(20);
        x.addElement(5);
        x.addElement(15);
        x.print();

        System.out.println("Get Min: " + x.getMin());
        System.out.println("Del Min: " + x.delMin());
        x.print();

        System.out.println("Min: " + x.getMin());
    }
}
```

**Output:**

Elements: 5, 10, 15, 20,

Get Min: 5

Del Min: 5

Elements: 10, 15, 20,

Min: 10

**Source**

<https://www.geeksforgeeks.org/data-structure-design-perform-required-operations/>

## Chapter 48

# Decimal Equivalent of Binary Linked List

Decimal Equivalent of Binary Linked List - GeeksforGeeks

Given a singly linked list of 0s and 1s find its decimal equivalent.

```
Input  : 0->0->0->1->1->0->0->1->0
Output : 50
```

```
Input  : 1->0->0
Output : 4
```

Decimal Value of an empty linked list is considered as 0.

Initialize result as 0. Traverse the linked list and for each node, multiply the result by 2 and add node's data to it.

```
// C++ Program to find decimal value of
// binary linked list
#include<iostream>
using namespace std;

/* Link list Node */
struct Node
{
    bool data;
    struct Node* next;
};

/* Returns decimal value of binary linked list */
int decimalValue(struct Node *head)
```

```
{  
    // Initialized result  
    int res = 0;  
  
    // Traverse linked list  
    while (head != NULL)  
    {  
        // Multiply result by 2 and add  
        // head's data  
        res = (res << 1) + head->data;  
  
        // Move next  
        head = head->next;  
    }  
    return res;  
}  
  
// Utility function to create a new node.  
Node *newNode(bool data)  
{  
    struct Node *temp = new Node;  
    temp->data = data;  
    temp->next = NULL;  
    return temp;  
}  
  
/* Drier program to test above function*/  
int main()  
{  
    /* Start with the empty list */  
    struct Node* head = newNode(1);  
    head->next = newNode(0);  
    head->next->next = newNode(1);  
    head->next->next->next = newNode(1);  
  
    cout << "Decimal value is "  
        << decimalValue(head);  
  
    return 0;  
}
```

Output :

```
Decimal value is 11
```

## Source

<https://www.geeksforgeeks.org/decimal-equivalent-of-binary-linked-list/>

## Chapter 49

# Delete N nodes after M nodes of a linked list

Delete N nodes after M nodes of a linked list - GeeksforGeeks

Given a linked list and two integers M and N. Traverse the linked list such that you retain M nodes then delete next N nodes, continue the same till end of the linked list.

Difficulty Level: Rookie

Examples:

Input:

M = 2, N = 2

Linked List: 1->2->3->4->5->6->7->8

Output:

Linked List: 1->2->5->6

Input:

M = 3, N = 2

Linked List: 1->2->3->4->5->6->7->8->9->10

Output:

Linked List: 1->2->3->6->7->8

Input:

M = 1, N = 1

Linked List: 1->2->3->4->5->6->7->8->9->10

Output:

Linked List: 1->3->5->7->9

The main part of the problem is to maintain proper links between nodes, make sure that all corner cases are handled. Following is C implementation of function skipMdeleteN() that skips M nodes and delete N nodes till end of list. It is assumed that M cannot be 0.

C

```
// C program to delete N nodes after M nodes of a linked list
#include <stdio.h>
#include <stdlib.h>

// A linked list node
struct Node
{
    int data;
    struct Node *next;
};

/* Function to insert a node at the beginning */
void push(struct Node ** head_ref, int new_data)
{
    /* allocate node */
    struct Node* new_node = (struct Node*) malloc(sizeof(struct Node));

    /* put in the data */
    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;
}

/* Function to print linked list */
void printList(struct Node *head)
{
    struct Node *temp = head;
    while (temp != NULL)
    {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}

// Function to skip M nodes and then delete N nodes of the linked list.
void skipMdeleteN(struct Node *head, int M, int N)
{
    struct Node *curr = head, *t;
    int count;

    // The main loop that traverses through the whole list
```

```
while (curr)
{
    // Skip M nodes
    for (count = 1; count<M && curr!= NULL; count++)
        curr = curr->next;

    // If we reached end of list, then return
    if (curr == NULL)
        return;

    // Start from next node and delete N nodes
    t = curr->next;
    for (count = 1; count<=N && t!= NULL; count++)
    {
        struct node *temp = t;
        t = t->next;
        free(temp);
    }
    curr->next = t; // Link the previous list with remaining nodes

    // Set current pointer for next iteration
    curr = t;
}

// Driver program to test above functions
int main()
{
    /* Create following linked list
     * 1->2->3->4->5->6->7->8->9->10 */
    struct Node* head = NULL;
    int M=2, N=3;
    push(&head, 10);
    push(&head, 9);
    push(&head, 8);
    push(&head, 7);
    push(&head, 6);
    push(&head, 5);
    push(&head, 4);
    push(&head, 3);
    push(&head, 2);
    push(&head, 1);

    printf("M = %d, N = %d \nGiven Linked list is :\n", M, N);
    printList(head);

    skipMdeleteN(head, M, N);
}
```

```
printf("\nLinked list after deletion is :\n");
printList(head);

return 0;
}
```

### Python

```
# Python program to delete M nodes after N nodes

# Node class
class Node:

    # Constructor to initialize the node object
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:

    # Function to initialize head
    def __init__(self):
        self.head = None

    # Function to insert a new node at the beginning
    def push(self, new_data):
        new_node = Node(new_data)
        new_node.next = self.head
        self.head = new_node

    # Utility function to print the linked LinkedList
    def printList(self):
        temp = self.head
        while(temp):
            print temp.data,
            temp = temp.next

    def skipMdeleteN(self, M, N):
        curr = self.head

        # The main loop that traverses through the
        # whole list
        while(curr):
            # Skip M nodes
            for count in range(1, M):
                if curr is None:
                    return
                curr = curr.next
```

```
if curr is None :
    return

# Start from next node and delete N nodes
t = curr.next
for count in range(1, N+1):
    if t is None:
        break
    t = t.next

# Link the previous list with remaining nodes
curr.next = t
# Set Current pointer for next iteration
curr = t

# Driver program to test above function

# Create following linked list
# 1->2->3->4->5->6->7->8->9->10
llist = LinkedList()
M = 2
N = 3
llist.push(10)
llist.push(9)
llist.push(8)
llist.push(7)
llist.push(6)
llist.push(5)
llist.push(4)
llist.push(3)
llist.push(2)
llist.push(1)

print "M = %d, N = %d\nGiven Linked List is:" %(M, N)
llist.printList()
print

llist.skipMdeleteN(M, N)

print "\nLinked list after deletion is"
llist.printList()

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

Output:

```
M = 2, N = 3
```

Given Linked list is :

1 2 3 4 5 6 7 8 9 10

Linked list after deletion is :

1 2 6 7

Time Complexity: O(n) where n is number of nodes in linked list.

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/delete-n-nodes-after-m-nodes-of-a-linked-list/>

## Chapter 50

# Delete a Doubly Linked List node at a given position

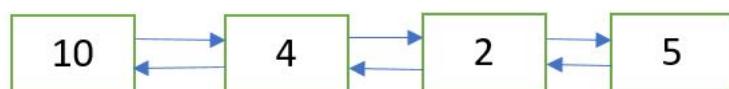
Delete a Doubly Linked List node at a given position - GeeksforGeeks

Given a doubly linked list and a position **n**. The task is to delete the node at the given position **n** from the beginning.

Initial doubly linked list



Doubly Linked List after deletion of node at position **n = 2**



**Approach:** Following are the steps:

1. Get the pointer to the node at position **n** by traversing the doubly linked list up to the **nth** node from the beginning.
2. Delete the node using the pointer obtained in Step 1. Refer [this](#) post.

```
/* C++ implementation to delete a doubly Linked List node
   at the given position */
#include <bits/stdc++.h>

using namespace std;
```

```
/* a node of the doubly linked list */
struct Node {
    int data;
    struct Node* next;
    struct Node* prev;
};

/* Function to delete a node in a Doubly Linked List.
   head_ref --> pointer to head node pointer.
   del --> pointer to node to be deleted. */
void deleteNode(struct Node** head_ref, struct Node* del)
{
    /* base case */
    if (*head_ref == NULL || del == NULL)
        return;

    /* If node to be deleted is head node */
    if (*head_ref == del)
        *head_ref = del->next;

    /* Change next only if node to be deleted is NOT
       the last node */
    if (del->next != NULL)
        del->next->prev = del->prev;

    /* Change prev only if node to be deleted is NOT
       the first node */
    if (del->prev != NULL)
        del->prev->next = del->next;

    /* Finally, free the memory occupied by del*/
    free(del);
}

/* Function to delete the node at the given position
   in the doubly linked list */
void deleteNodeAtGivenPos(struct Node** head_ref, int n)
{
    /* if list in NULL or invalid position is given */
    if (*head_ref == NULL || n <= 0)
        return;

    struct Node* current = *head_ref;
    int i;

    /* traverse up to the node at position 'n' from
       the beginning */
    for (int i = 1; current != NULL && i < n; i++)

```

```
    current = current->next;

    /* if 'n' is greater than the number of nodes
       in the doubly linked list */
    if (current == NULL)
        return;

    /* delete the node pointed to by 'current' */
    deleteNode(head_ref, current);
}

/* Function to insert a node at the beginning
   of the Doubly Linked List */
void push(struct Node** head_ref, int new_data)
{
    /* allocate node */
    struct Node* new_node =
        (struct Node*)malloc(sizeof(struct Node));

    /* put in the data */
    new_node->data = new_data;

    /* since we are adding at the beginning,
       prev is always NULL */
    new_node->prev = NULL;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* change prev of head node to new node */
    if ((*head_ref) != NULL)
        (*head_ref)->prev = new_node;

    /* move the head to point to the new node */
    (*head_ref) = new_node;
}

/* Function to print nodes in a given doubly
   linked list */
void printList(struct Node* head)
{
    while (head != NULL) {
        cout << head->data << " ";
        head = head->next;
    }
}

/* Driver program to test above functions*/
```

```
int main()
{
    /* Start with the empty list */
    struct Node* head = NULL;

    /* Create the doubly linked list 10<->8<->4<->2<->5 */
    push(&head, 5);
    push(&head, 2);
    push(&head, 4);
    push(&head, 8);
    push(&head, 10);

    cout << "Doubly linked list before deletion:n";
    printList(head);

    int n = 2;

    /* delete node at the given position 'n' */
    deleteNodeAtGivenPos(&head, n);

    cout << "\nDoubly linked list after deletion:n";
    printList(head);

    return 0;
}
```

Output:

```
Doubly linked list before deletion:
10 8 4 2 5
Doubly linked list after deletion:
10 4 2 5
```

Time Complexity: O(n), in worst case where n is the number of nodes in the doubly linked list.

## Source

<https://www.geeksforgeeks.org/delete-doubly-linked-list-node-given-position/>

## Chapter 51

# Delete a Linked List node at a given position

Delete a Linked List node at a given position - GeeksforGeeks

Given a singly linked list and a position, delete a linked list node at the given position.

**Example:**

Input: position = 1, Linked List = 8->2->3->1->7  
Output: Linked List = 8->3->1->7

Input: position = 0, Linked List = 8->2->3->1->7  
Output: Linked List = 2->3->1->7

If node to be deleted is root, simply delete it. To delete a middle node, we must have pointer to the node previous to the node to be deleted. So if positions is not zero, we run a loop position-1 times and get pointer to the previous node.

Below is the implementation of above idea.

**C/C++**

```
// A complete working C program to delete a node in a linked list
// at a given position
#include <stdio.h>
#include <stdlib.h>

// A linked list node
struct Node
{
    int data;
    struct Node *next;
}
```

```
};

/* Given a reference (pointer to pointer) to the head of a list
   and an int, inserts a new node on the front of the list. */
void push(struct Node** head_ref, int new_data)
{
    struct Node* new_node = (struct Node*) malloc(sizeof(struct Node));
    new_node->data = new_data;
    new_node->next = (*head_ref);
    (*head_ref) = new_node;
}

/* Given a reference (pointer to pointer) to the head of a list
   and a position, deletes the node at the given position */
void deleteNode(struct Node **head_ref, int position)
{
    // If linked list is empty
    if (*head_ref == NULL)
        return;

    // Store head node
    struct Node* temp = *head_ref;

    // If head needs to be removed
    if (position == 0)
    {
        *head_ref = temp->next; // Change head
        free(temp);           // free old head
        return;
    }

    // Find previous node of the node to be deleted
    for (int i=0; temp!=NULL && i<position-1; i++)
        temp = temp->next;

    // If position is more than number of nodes
    if (temp == NULL || temp->next == NULL)
        return;

    // Node temp->next is the node to be deleted
    // Store pointer to the next of node to be deleted
    struct Node *next = temp->next->next;

    // Unlink the node from linked list
    free(temp->next); // Free memory

    temp->next = next; // Unlink the deleted node from list
}
```

```
// This function prints contents of linked list starting from
// the given node
void printList(struct Node *node)
{
    while (node != NULL)
    {
        printf(" %d ", node->data);
        node = node->next;
    }
}

/* Drier program to test above functions*/
int main()
{
    /* Start with the empty list */
    struct Node* head = NULL;

    push(&head, 7);
    push(&head, 1);
    push(&head, 3);
    push(&head, 2);
    push(&head, 8);

    puts("Created Linked List: ");
    printList(head);
    deleteNode(&head, 4);
    puts("\nLinked List after Deletion at position 4: ");
    printList(head);
    return 0;
}
```

**Java**

```
// A complete working Java program to delete a node in a linked list
// at a given position
class LinkedList
{
    Node head; // head of list

    /* Linked list Node*/
    class Node
    {
        int data;
        Node next;
        Node(int d)
        {
            data = d;
```

```
        next = null;
    }
}

/* Inserts a new Node at front of the list. */
public void push(int new_data)
{
    /* 1 & 2: Allocate the Node &
       Put in the data*/
    Node new_node = new Node(new_data);

    /* 3. Make next of new Node as head */
    new_node.next = head;

    /* 4. Move the head to point to new Node */
    head = new_node;
}

/* Given a reference (pointer to pointer) to the head of a list
   and a position, deletes the node at the given position */
void deleteNode(int position)
{
    // If linked list is empty
    if (head == null)
        return;

    // Store head node
    Node temp = head;

    // If head needs to be removed
    if (position == 0)
    {
        head = temp.next;    // Change head
        return;
    }

    // Find previous node of the node to be deleted
    for (int i=0; temp!=null && i<position-1; i++)
        temp = temp.next;

    // If position is more than number of nodes
    if (temp == null || temp.next == null)
        return;

    // Node temp->next is the node to be deleted
    // Store pointer to the next of node to be deleted
    Node next = temp.next.next;
```

```
        temp.next = next; // Unlink the deleted node from list
    }

/* This function prints contents of linked list starting from
   the given node */
public void printList()
{
    Node tnode = head;
    while (tnode != null)
    {
        System.out.print(tnode.data+" ");
        tnode = tnode.next;
    }
}

/* Drier program to test above functions. Ideally this function
   should be in a separate user class. It is kept here to keep
   code compact */
public static void main(String[] args)
{
    /* Start with the empty list */
    LinkedList llist = new LinkedList();

    llist.push(7);
    llist.push(1);
    llist.push(3);
    llist.push(2);
    llist.push(8);

    System.out.println("\nCreated Linked list is: ");
    llist.printList();

    llist.deleteNode(4); // Delete node at position 4

    System.out.println("\nLinked List after Deletion at position 4: ");
    llist.printList();
}
}
```

### Python

```
# Python program to delete a node in a linked list
# at a given position

# Node class
class Node:

    # Constructor to initialize the node object
```

```
def __init__(self, data):
    self.data = data
    self.next = None

class LinkedList:

    # Constructor to initialize head
    def __init__(self):
        self.head = None

    # Function to insert a new node at the beginning
    def push(self, new_data):
        new_node = Node(new_data)
        new_node.next = self.head
        self.head = new_node

    # Given a reference to the head of a list
    # and a position, delete the node at a given position
    def deleteNode(self, position):

        # If linked list is empty
        if self.head == None:
            return

        # Store head node
        temp = self.head

        # If head needs to be removed
        if position == 0:
            self.head = temp.next
            temp = None
            return

        # Find previous node of the node to be deleted
        for i in range(position - 1):
            temp = temp.next
            if temp is None:
                break

        # If position is more than number of nodes
        if temp is None:
            return
        if temp.next is None:
            return

        # Node temp.next is the node to be deleted
        # store pointer to the next of node to be deleted
        next = temp.next.next
```

```
# Unlink the node from linked list
temp.next = None

temp.next = next

# Utility function to print the linked LinkedList
def printList(self):
    temp = self.head
    while(temp):
        print " %d " %(temp.data),
        temp = temp.next

# Driver program to test above function
llist = LinkedList()
llist.push(7)
llist.push(1)
llist.push(3)
llist.push(2)
llist.push(8)

print "Created Linked List: "
llist.printList()
llist.deleteNode(4)
print "\nLinked List after Deletion at position 4: "
llist.printList()

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

Output:

```
Created Linked List:
8 2 3 1 7
Linked List after Deletion at position 4:
8 2 3 1
```

Thanks to **Hemanth Kumar** for suggesting 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/delete-a-linked-list-node-at-a-given-position/>

## Chapter 52

# Delete a given node in Linked List under given constraints

Delete a given node in Linked List under given constraints - GeeksforGeeks

Given a Singly Linked List, write a function to delete a given node. Your function must follow following constraints:

- 1) It must accept pointer to the start node as first parameter and node to be deleted as second parameter i.e., pointer to head node is not global.
- 2) It should not return pointer to the head node.
- 3) It should not accept pointer to pointer to head node.

You may assume that the Linked List never becomes empty.

Let the function name be `deleteNode()`. In a straightforward implementation, the function needs to modify head pointer when the node to be deleted is first node. As discussed in [previous post](#), when a function modifies the head pointer, the function must use one of the [given approaches](#), we can't use any of those approaches here.

### Solution

We explicitly handle the case when node to be deleted is first node, we copy the data of next node to head and delete the next node. The cases when deleted node is not the head node can be handled normally by finding the previous node and changing next of previous node. Following is C implementation.

C

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

/* structure of a linked list node */
struct Node
{
    int data;
    struct Node *next;
```

```
};

void deleteNode(struct Node *head, struct Node *n)
{
    // When node to be deleted is head node
    if(head == n)
    {
        if(head->next == NULL)
        {
            printf("There is only one node. The list can't be made empty ");
            return;
        }

        /* Copy the data of next node to head */
        head->data = head->next->data;

        // store address of next node
        n = head->next;

        // Remove the link of next node
        head->next = head->next->next;

        // free memory
        free(n);

        return;
    }

    // When not first node, follow the normal deletion process

    // find the previous node
    struct Node *prev = head;
    while(prev->next != NULL && prev->next != n)
        prev = prev->next;

    // Check if node really exists in Linked List
    if(prev->next == NULL)
    {
        printf("\n Given node is not present in Linked List");
        return;
    }

    // Remove node from Linked List
    prev->next = prev->next->next;

    // Free memory
    free(n);
```

```
        return;
    }

/* Utility function to insert a node at the begining */
void push(struct Node **head_ref, int new_data)
{
    struct Node *new_node =
        (struct Node *)malloc(sizeof(struct Node));
    new_node->data = new_data;
    new_node->next = *head_ref;
    *head_ref = new_node;
}

/* Utility function to print a linked list */
void printList(struct Node *head)
{
    while(head!=NULL)
    {
        printf("%d ",head->data);
        head=head->next;
    }
    printf("\n");
}

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

    /* Create following linked list
       12->15->10->11->5->6->2->3 */
    push(&head,3);
    push(&head,2);
    push(&head,6);
    push(&head,5);
    push(&head,11);
    push(&head,10);
    push(&head,15);
    push(&head,12);

    printf("Given Linked List: ");
    printList(head);

    /* Let us delete the node with value 10 */
    printf("\nDeleting node %d: ", head->next->next->data);
    deleteNode(head, head->next->next);
}
```

```
printf("\nModified Linked List: ");
printList(head);

/* Let us delete the the first node */
printf("\nDeleting first node ");
deleteNode(head, head);

printf("\nModified Linked List: ");
printList(head);

getchar();
return 0;
}
```

**Java**

```
// Java program to delete a given node in linked list under given constraints

class LinkedList {

    static Node head;

    static class Node {

        int data;
        Node next;

        Node(int d) {
            data = d;
            next = null;
        }
    }

    void deleteNode(Node node, Node n) {

        // When node to be deleted is head node
        if (node == n) {
            if (node.next == null) {
                System.out.println("There is only one node. The list "
                    + "can't be made empty ");
                return;
            }

            /* Copy the data of next node to head */
            node.data = node.next.data;

            // store address of next node
            n = node.next;
        }
    }
}
```

```
// Remove the link of next node
node.next = node.next.next;

// free memory
System.gc();

return;
}

// When not first node, follow the normal deletion process
// find the previous node
Node prev = node;
while (prev.next != null && prev.next != n) {
    prev = prev.next;
}

// Check if node really exists in Linked List
if (prev.next == null) {
    System.out.println("Given node is not present in Linked List");
    return;
}

// Remove node from Linked List
prev.next = prev.next.next;

// Free memory
System.gc();

return;
}

/* Utility function to print a linked list */
void printList(Node head) {
    while (head != null) {
        System.out.print(head.data + " ");
        head = head.next;
    }
    System.out.println("");
}

public static void main(String[] args) {
    LinkedList list = new LinkedList();
    list.head = new Node(12);
    list.head.next = new Node(15);
    list.head.next.next = new Node(10);
    list.head.next.next.next = new Node(11);
    list.head.next.next.next.next = new Node(5);
```

```
list.head.next.next.next.next = new Node(6);
list.head.next.next.next.next = new Node(2);
list.head.next.next.next.next.next = new Node(3);

System.out.println("Given Linked List :");
list.printList(head);
System.out.println("");

// Let us delete the node with value 10
System.out.println("Deleting node :" + head.next.next.data);
list.deleteNode(head, head.next.next);

System.out.println("Modified Linked list :");
list.printList(head);
System.out.println("");

// Lets delete the first node
System.out.println("Deleting first Node");
list.deleteNode(head, head);
System.out.println("Modified Linked List");
list.printList(head);

}

}

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

Output:

```
Given Linked List: 12 15 10 11 5 6 2 3

Deleting node 10:
Modified Linked List: 12 15 11 5 6 2 3

Deleting first node
Modified Linked List: 15 11 5 6 2 3
```

## Source

<https://www.geeksforgeeks.org/delete-a-given-node-in-linked-list-under-given-constraints/>

## Chapter 53

# Delete a linked list using recursion

Delete a linked list using recursion - GeeksforGeeks

Delete the given linked list using recursion

### Method

- 1) If head equal to NULL then linked list is empty, we simply return.
- 2) Recursively delete linked list after head node.
- 3) Delete head node.

```
// C++ program to recursively delete a linked list
#include <bits/stdc++.h>

/* Link list node */
struct Node {
    int data;
    struct Node* next;
};

/* Recursive Function to delete the entire linked list */
void deleteList(struct Node* head)
{
    if (head == NULL)
        return;
    deleteList(head->next);
    free(head);
}

/* Given a reference (pointer to pointer) to
   the head of a list and an int, push a new
   node on the front of the list. */
```

```
void push(struct Node** head_ref, int new_data)
{
    struct Node* new_node = new Node;
    new_node->data = new_data;
    new_node->next = (*head_ref);
    (*head_ref) = new_node;
}

/* Driver program to test count function*/
int main()
{
    /* Start with the empty list */
    struct Node* head = NULL;

    /* Use push() to construct below list
     * 1->12->1->4->1 */
    push(&head, 1);
    push(&head, 4);
    push(&head, 12);
    push(&head, 1);
    printf("\n Deleting linked list");
    deleteList(head);
    printf("\nLinked list deleted");
    return 0;
}
```

Output:

```
Deleting linked list
Linked list deleted
```

## Source

<https://www.geeksforgeeks.org/delete-linked-list-using-recursion/>

## Chapter 54

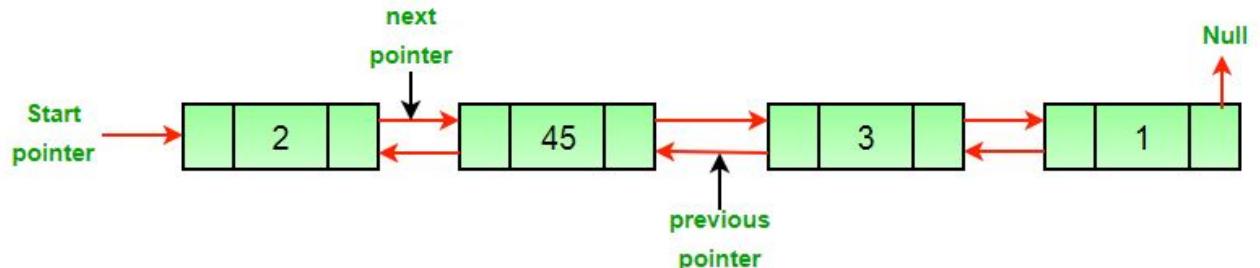
# Delete a node in a Doubly Linked List

Delete a node in a Doubly Linked List - GeeksforGeeks

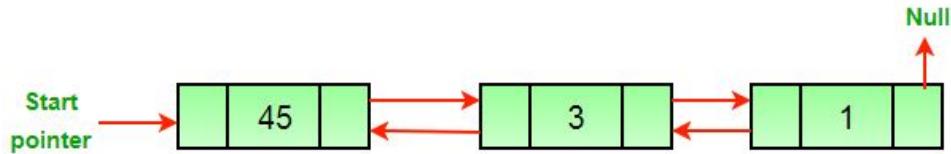
[Doubly Link List Set 1| Introduction and Insertion](#)

Write a function to delete a given node in a doubly linked list.

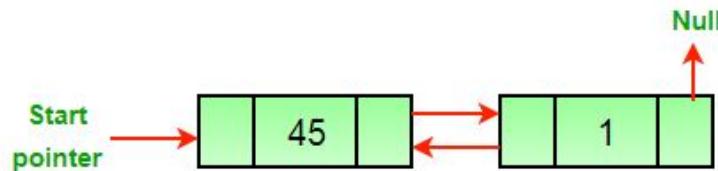
(a) Original Doubly Linked List



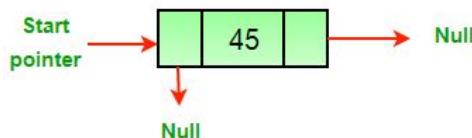
(a) After deletion of head node



(a) After deletion of middle node



(a) After deletion of last node



### Algorithm

Let the node to be deleted is *del*.

- 1) If node to be deleted is head node, then change the head pointer to next current head.
- 2) Set *next* of previous to *del*, if previous to *del* exists.
- 3) Set *prev* of next to *del*, if next to *del* exists.

C

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

/* a node of the doubly linked list */
struct Node
```

```
{  
    int data;  
    struct Node *next;  
    struct Node *prev;  
};  
  
/* Function to delete a node in a Doubly Linked List.  
   head_ref --> pointer to head node pointer.  
   del --> pointer to node to be deleted. */  
void deleteNode(struct Node **head_ref, struct Node *del)  
{  
    /* base case */  
    if(*head_ref == NULL || del == NULL)  
        return;  
  
    /* If node to be deleted is head node */  
    if(*head_ref == del)  
        *head_ref = del->next;  
  
    /* Change next only if node to be deleted is NOT the last node */  
    if(del->next != NULL)  
        del->next->prev = del->prev;  
  
    /* Change prev only if node to be deleted is NOT the first node */  
    if(del->prev != NULL)  
        del->prev->next = del->next;  
  
    /* Finally, free the memory occupied by del*/  
    free(del);  
    return;  
}  
  
/* UTILITY FUNCTIONS */  
/* Function to insert a node at the beginning of the Doubly Linked List */  
void push(struct Node** head_ref, int new_data)  
{  
    /* allocate node */  
    struct Node* new_node =  
        (struct Node*) malloc(sizeof(struct Node));  
  
    /* put in the data */  
    new_node->data = new_data;  
  
    /* since we are adding at the begining,  
       prev is always NULL */  
    new_node->prev = NULL;  
  
    /* link the old list off the new node */
```

```
new_node->next = (*head_ref);

/* change prev of head node to new node */
if((*head_ref) != NULL)
    (*head_ref)->prev = new_node ;

/* move the head to point to the new node */
(*head_ref)      = new_node;
}

/* Function to print nodes in a given doubly linked list
   This function is same as printList() of singly linked lsit */
void printList(struct Node *node)
{
    while(node!=NULL)
    {
        printf("%d ", node->data);
        node = node->next;
    }
}

/* Drier program to test above functions*/
int main()
{
    /* Start with the empty list */
    struct Node* head = NULL;

    /* Let us create the doubly linked list 10<->8<->4<->2 */
    push(&head, 2);
    push(&head, 4);
    push(&head, 8);
    push(&head, 10);

    printf("\n Original Linked list ");
    printList(head);

    /* delete nodes from the doubly linked list */
    deleteNode(&head, head); /*delete first node*/
    deleteNode(&head, head->next); /*delete middle node*/
    deleteNode(&head, head->next); /*delete last node*/

    /* Modified linked list will be NULL<-8->NULL */
    printf("\n Modified Linked list ");
    printList(head);

    getchar();
}
```

Java

```
// Java program to delete a node from doubly linked list

class LinkedList {

    static Node head = null;

    class Node {

        int data;
        Node next, prev;

        Node(int d) {
            data = d;
            next = prev = null;
        }
    }

    /*Function to delete a node in a Doubly Linked List.
    head_ref --> pointer to head node pointer.
    del --> pointer to node to be deleted. */
    void deleteNode(Node head_ref, Node del) {

        /* base case */
        if (head == null || del == null) {
            return;
        }

        /* If node to be deleted is head node */
        if (head == del) {
            head = del.next;
        }

        /* Change next only if node to be deleted is NOT the last node */
        if (del.next != null) {
            del.next.prev = del.prev;
        }

        /* Change prev only if node to be deleted is NOT the first node */
        if (del.prev != null) {
            del.prev.next = del.next;
        }

        /* Finally, free the memory occupied by del*/
        return;
    }
}
```

```
/* UTILITY FUNCTIONS */
/* Function to insert a node at the beginning of the Doubly Linked List */
void push(Node head_ref, int new_data) {

    /* allocate node */
    Node new_node = new Node(new_data);

    /* since we are adding at the begining,
       prev is always NULL */
    new_node.prev = null;

    /* link the old list off the new node */
    new_node.next = (head);

    /* change prev of head node to new node */
    if ((head) != null) {
        (head).prev = new_node;
    }

    /* move the head to point to the new node */
    (head) = new_node;
}

/*Function to print nodes in a given doubly linked list
This function is same as printList() of singly linked lsit */
void printList(Node node) {
    while (node != null) {
        System.out.print(node.data + " ");
        node = node.next;
    }
}

public static void main(String[] args) {
    LinkedList list = new LinkedList();

    /* Let us create the doubly linked list 10<->8<->4<->2 */
    list.push(head, 2);
    list.push(head, 4);
    list.push(head, 8);
    list.push(head, 10);

    System.out.println("Original Linked list ");
    list.printList(head);

    /* delete nodes from the doubly linked list */
    list.deleteNode(head, head); /*delete first node*/

    list.deleteNode(head, head.next); /*delete middle node*/
```

```
list.deleteNode(head, head.next); /*delete last node*/
System.out.println("");

/* Modified linked list will be NULL<-8->NULL */
System.out.println("Modified Linked List");
list.printList(head);
}

}

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

### Python

```
# Program to delete a node in doubly linked list

# for Garbage collection
import gc

# A node of the doublly linked list
class Node:

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

class DoublyLinkedList:
    # Constructor for empty Doubly Linked List
    def __init__(self):
        self.head = None

    # Function to delete a node in a Doubly Linked List.
    # head_ref --> pointer to head node pointer.
    # dele --> pointer to node to be deleted

    def deleteNode(self, dele):

        # Base Case
        if self.head is None or dele is None:
            return

        # If node to be deleted is head node
        if self.head == dele:
            self.head = dele.next

        # Change next only if node to be deleted is NOT
```

```
# the last node
if dele.next is not None:
    dele.next.prev = dele.prev

# Change prev only if node to be deleted is NOT
# the first node
if dele.prev is not None:
    dele.prev.next = dele.next
# Finally, free the memory occupied by dele
# Call python garbage collector
gc.collect()

# Given a reference to the head of a list and an
# integer, inserts a new node on the front of list
def push(self, new_data):

    # 1. Allocates node
    # 2. Put the data in it
    new_node = Node(new_data)

    # 3. Make next of new node as head and
    # previous as None (already None)
    new_node.next = self.head

    # 4. change prev of head node to new_node
    if self.head is not None:
        self.head.prev = new_node

    # 5. move the head to point to the new node
    self.head = new_node

def printList(self, node):
    while(node is not None):
        print node.data,
        node = node.next

# Driver program to test the above functions

# Start with empty list
dll = DoublyLinkedList()

# Let us create the doubly linked list 10<->8<->4<->2
dll.push(2);
dll.push(4);
dll.push(8);
```

```
dll.push(10);

print "\n Original Linked List",
dll.printList(dll.head)

# delete nodes from doubly linked list
dll.deleteNode(dll.head)
dll.deleteNode(dll.head.next)
dll.deleteNode(dll.head.next)
# Modified linked list will be NULL<-8->NULL
print "\n Modified Linked List",
dll.printList(dll.head)

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

Time Complexity: O(1)

Space Complexity: O(1)

**Improved By :** [RiskCatalyst](#)

## Source

<https://www.geeksforgeeks.org/delete-a-node-in-a-doubly-linked-list/>

## Chapter 55

# Delete all occurrences of a given key in a doubly linked list

Delete all occurrences of a given key in a doubly linked list - GeeksforGeeks

Given a doubly linked list and a key  $x$ . The problem is to delete all occurrences of the given key  $x$  from the doubly linked list.

Examples:

```
Input : DLL: 2 <-> 2 <-> 10 <-> 8 <-> 4 <-> 2 <-> 5 <-> 2  
      x = 2  
Output : 10 <-> 8 <-> 4 <-> 5
```

Algorithm:

```
delAllOccurOfGivenKey(head_ref, x)  
    if head_ref == NULL  
        return  
    Initialize current = head_ref  
    Declare next  
    while current != NULL  
        if current->data == x  
            next = current->next  
            deleteNode(head_ref, current)  
            current = next  
        else  
            current = current->next
```

The algorithm for **deleteNode(head\_ref, current)** (which deletes the node using the pointer to the node) is discussed in [this](#) post.

```
/* C++ implementation to delete all occurrences
   of a given key in a doubly linked list */
#include <bits/stdc++.h>

using namespace std;

/* a node of the doubly linked list */
struct Node {
    int data;
    struct Node* next;
    struct Node* prev;
};

/* Function to delete a node in a Doubly Linked List.
   head_ref --> pointer to head node pointer.
   del --> pointer to node to be deleted. */
void deleteNode(struct Node** head_ref, struct Node* del)
{
    /* base case */
    if (*head_ref == NULL || del == NULL)
        return;

    /* If node to be deleted is head node */
    if (*head_ref == del)
        *head_ref = del->next;

    /* Change next only if node to be deleted
       is NOT the last node */
    if (del->next != NULL)
        del->next->prev = del->prev;

    /* Change prev only if node to be deleted
       is NOT the first node */
    if (del->prev != NULL)
        del->prev->next = del->next;

    /* Finally, free the memory occupied by del*/
    free(del);
}

/* function to delete all occurrences of the given
   key 'x' */
void deleteAllOccurOfX(struct Node** head_ref, int x)
{
    /* if list is empty */
    if ((*head_ref) == NULL)
        return;
```

```
struct Node* current = *head_ref;
struct Node* next;

/* traverse the list up to the end */
while (current != NULL) {

    /* if node found with the value 'x' */
    if (current->data == x) {

        /* save current's next node in the
           pointer 'next' */
        next = current->next;

        /* delete the node pointed to by
           'current' */
        deleteNode(head_ref, current);

        /* update current */
        current = next;
    }

    /* else simply move to the next node */
    else
        current = current->next;
}

/* Function to insert a node at the beginning
   of the Doubly Linked List */
void push(struct Node** head_ref, int new_data)
{
    /* allocate node */
    struct Node* new_node =
        (struct Node*)malloc(sizeof(struct Node));

    /* put in the data */
    new_node->data = new_data;

    /* since we are adding at the begining,
       prev is always NULL */
    new_node->prev = NULL;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* change prev of head node to new node */
    if ((*head_ref) != NULL)
        (*head_ref)->prev = new_node;
```

```
/* move the head to point to the new node */
(*head_ref) = new_node;
}

/* Function to print nodes in a given doubly
   linked list */
void printList(struct Node* head)
{
    /* if list is empty */
    if (head == NULL)
        cout << "Doubly Linked list empty";

    while (head != NULL) {
        cout << head->data << " ";
        head = head->next;
    }
}

/* Driver program to test above functions*/
int main()
{
    /* Start with the empty list */
    struct Node* head = NULL;

    /* Create the doubly linked list:
       2<->2<->10<->8<->4<->2<->5<->2 */
    push(&head, 2);
    push(&head, 5);
    push(&head, 2);
    push(&head, 4);
    push(&head, 8);
    push(&head, 10);
    push(&head, 2);
    push(&head, 2);

    cout << "Original Doubly linked list:n";
    printList(head);

    int x = 2;

    /* delete all occurrences of 'x' */
    deleteAllOccur0fX(&head, x);

    cout << "\nDoubly linked list after deletion of "
        << x << ":n";
    printList(head);
```

```
    return 0;  
}
```

Output:

```
Original Doubly linked list:  
2 2 10 8 4 2 5 2  
Doubly linked list after deletion of 2:  
10 8 4 5
```

Time Complexity: O(n)

## Source

<https://www.geeksforgeeks.org/delete-occurrences-given-key-doubly-linked-list/>

## Chapter 56

# Delete all occurrences of a given key in a linked list

Delete all occurrences of a given key in a linked list - GeeksforGeeks

Given a singly linked list, delete all occurrences of a given key in it. For example, consider the following list.

```
Input: 2 -> 2 -> 1 -> 8 -> 2 -> 3 -> 2 -> 7  
      Key to delete = 2  
Output: 1 -> 8 -> 3 -> 7
```

This is mainly an extension of [this post which deletes first occurrence of a given key](#).

We need to first check for all occurrences at head node and change the head node appropriately. Then we need to check for all occurrences inside a loop and delete them one by one. Following is C implementation for the same.

```
// C Program to delete all occurrences of a given key in linked list  
#include <stdio.h>  
#include <stdlib.h>  
  
// A linked list node  
struct Node  
{  
    int data;  
    struct Node *next;  
};  
  
/* Given a reference (pointer to pointer) to the head of a list  
   and an int, inserts a new node on the front of the list. */
```

```
void push(struct Node** head_ref, int new_data)
{
    struct Node* new_node = (struct Node*) malloc(sizeof(struct Node));
    new_node->data = new_data;
    new_node->next = (*head_ref);
    (*head_ref) = new_node;
}

/* Given a reference (pointer to pointer) to the head of a list and
   a key, deletes all occurrence of the given key in linked list */
void deleteKey(struct Node **head_ref, int key)
{
    // Store head node
    struct Node* temp = *head_ref, *prev;

    // If head node itself holds the key or multiple occurrences of key
    while (temp != NULL && temp->data == key)
    {
        *head_ref = temp->next; // Changed head
        free(temp); // free old head
        temp = *head_ref; // Change Temp
    }

    // Delete occurrences other than head
    while (temp != NULL)
    {
        // Search for the key to be deleted, keep track of the
        // previous node as we need to change 'prev->next'
        while (temp != NULL && temp->data != key)
        {
            prev = temp;
            temp = temp->next;
        }

        // If key was not present in linked list
        if (temp == NULL) return;

        // Unlink the node from linked list
        prev->next = temp->next;

        free(temp); // Free memory

        //Update Temp for next iteration of outer loop
        temp = prev->next;
    }
}

// This function prints contents of linked list starting from
```

```
// the given node
void printList(struct Node *node)
{
    while (node != NULL)
    {
        printf(" %d ", node->data);
        node = node->next;
    }
}

/* Drier program to test above functions*/
int main()
{
    /* Start with the empty list */
    struct Node* head = NULL;

    push(&head, 7);
    push(&head, 2);
    push(&head, 3);
    push(&head, 2);
    push(&head, 8);
    push(&head, 1);
    push(&head, 2);
    push(&head, 2);

    int key = 2; // key to delete

    puts("Created Linked List: ");
    printList(head);

    deleteKey(&head, key);
    puts("\nLinked List after Deletion of 1: ");

    printList(head);
    return 0;
}
```

Output:

```
Created Linked List:
2 2 1 8 2 3 2 7
Linked List after Deletion of 1:
1 8 3 7
```

This article is contributed by **Saransh**. 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/delete-occurrences-given-key-linked-list/>

## Chapter 57

# Delete all the nodes from the doubly linked list that are greater than a given value

Delete all the nodes from the doubly linked list that are greater than a given value - Geeks-forGeeks

Given a doubly linked list containing N nodes and a number X, the task is to delete all the nodes from the list that are greater than the given value X.

**Examples:**

**Input:** 10 <=> 8 <=> 4 <=> 11 <=> 9, X = 9

**Output:** 8 <=> 4 <=> 9

**Input:** 4 <=> 4 <=> 2 <=> 1 <=> 3 <=> 5, X = 2

**Output:** 2 <=> 1

**Approach:** Traverse the nodes of the doubly linked list one by one and get the pointer of the nodes having data value greater than x. Delete these nodes by following the approach used in [this](#) post.

```
// C++ implementation to delete all
// the nodes from the doubly
// linked list that are greater than
// the specified value x
#include <bits/stdc++.h>

using namespace std;

// Node of the doubly linked list
struct Node {
```

```
int data;
Node *prev, *next;
};

// function to insert a node at the beginning
// of the Doubly Linked List
void push(Node** head_ref, int new_data)
{
    // allocate node
    Node* new_node = (Node*)malloc(sizeof(struct Node));

    // put in the data
    new_node->data = new_data;

    // since we are adding at the begining,
    // prev is always NULL
    new_node->prev = NULL;

    // link the old list off the new node
    new_node->next = (*head_ref);

    // change prev of head node to new node
    if ((*head_ref) != NULL)
        (*head_ref)->prev = new_node;

    // move the head to point to the new node
    (*head_ref) = new_node;
}

// function to delete a node in a Doubly Linked List.
// head_ref --> pointer to head node pointer.
// del --> pointer to node to be deleted
void deleteNode(Node** head_ref, Node* del)
{
    // base case
    if (*head_ref == NULL || del == NULL)
        return;

    // If node to be deleted is head node
    if (*head_ref == del)
        *head_ref = del->next;

    // Change next only if node to be
    // deleted is NOT the last node
    if (del->next != NULL)
        del->next->prev = del->prev;

    // Change prev only if node to be
```

```
// deleted is NOT the first node
if (del->prev != NULL)
    del->prev->next = del->next;

// Finally, free the memory occupied by del
free(del);

return;
}

// function to delete all the nodes
// from the doubly linked
// list that are greater than the
// specified value x
void deleteGreaterNodes(Node** head_ref, int x)
{
    Node* ptr = *head_ref;
    Node* next;

    while (ptr != NULL) {
        next = ptr->next;
        // if true, delete node 'ptr'
        if (ptr->data > x)
            deleteNode(head_ref, ptr);
        ptr = next;
    }
}

// function to print nodes in a
// given doubly linked list
void printList(Node* head)
{
    while (head != NULL) {
        cout << head->data << " ";
        head = head->next;
    }
}

// Driver program to test above
int main()
{
    // start with the empty list
    Node* head = NULL;

    // create the doubly linked list
    // 10<->8<->4<->11<->9
    push(&head, 9);
    push(&head, 11);
```

```
push(&head, 4);
push(&head, 8);
push(&head, 10);

int x = 9;

cout << "Original List: ";
printList(head);

deleteGreaterNodes(&head, x);

cout << "\nModified List: ";
printList(head);
}
```

**Output:**

```
Original List: 10 8 4 11 9
Modified List: 8 4 9
```

**Time Complexity:** O(N)

**Source**

<https://www.geeksforgeeks.org/delete-all-the-nodes-from-the-doubly-linked-list-that-are-greater-than-a-given-value>

## Chapter 58

# Delete all the nodes from the list that are greater than x

Delete all the nodes from the list that are greater than x - GeeksforGeeks

Given a linked list, the problem is to delete all the nodes from the list that are greater than the specified value x.

Examples:

```
Input : list: 7->3->4->8->5->1
        x = 6
Output : 3->4->5->1
```

```
Input : list: 1->8->7->3->7->10
        x = 7
Output : 1->7->3->7
```

**Source:** [Microsoft Interview Experience | Set 169.](#)

**Approach:** This is mainly a variation of the post which [deletes first occurrence of a given key](#).

We need to first check for all occurrences at head node which are greater than 'x', delete them and change the head node appropriately. Then we need to check for all occurrences inside a loop and delete them one by one.

```
// C++ implementation to delete all the nodes from the list
// that are greater than the specified value x
#include <bits/stdc++.h>

using namespace std;
```

```
// structure of a node
struct Node {
    int data;
    Node* next;
};

// function to get a new node
Node* getNode(int data)
{
    Node* newNode = new Node;
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}

// function to delete all the nodes from the list
// that are greater than the specified value x
void deleteGreaterNodes(Node** head_ref, int x)
{
    Node *temp = *head_ref, *prev;

    // If head node itself holds the value greater than 'x'
    while (temp != NULL && temp->data > x) {
        *head_ref = temp->next; // Changed head
        free(temp); // free old head
        temp = *head_ref; // Change temp
    }

    // Delete occurrences other than head
    while (temp != NULL) {

        // Search for the node to be deleted,
        // keep track of the previous node as we
        // need to change 'prev->next'
        while (temp != NULL && temp->data <= x) {
            prev = temp;
            temp = temp->next;
        }

        // If required value node was not present
        // in linked list
        if (temp == NULL)
            return;

        // Unlink the node from linked list
        prev->next = temp->next;

        delete temp; // Free memory
    }
}
```

```
// Update Temp for next iteration of
// outer loop
temp = prev->next;
}
}

// function to print a linked list
void printList(Node* head)
{
    while (head) {
        cout << head->data << " ";
        head = head->next;
    }
}

// Driver program to test above
int main()
{
    // Create list: 7->3->4->8->5->1
    Node* head = getNode(7);
    head->next = getNode(3);
    head->next->next = getNode(4);
    head->next->next->next = getNode(8);
    head->next->next->next->next = getNode(5);
    head->next->next->next->next->next = getNode(1);

    int x = 6;

    cout << "Original List: ";
    printList(head);

    deleteGreaterNodes(&head, x);

    cout << "\nModified List: ";
    printList(head);

    return 0;
}
```

Output:

```
Original List: 7 3 4 8 5 1
Modified List: 3 4 5 1
```

Time Complexity: O(n).

## Source

<https://www.geeksforgeeks.org/delete-nodes-list-greater-x/>

## Chapter 59

# Delete alternate nodes of a Linked List

Delete alternate nodes of a Linked List - GeeksforGeeks

Given a Singly Linked List, starting from the second node delete all alternate nodes of it. For example, if the given linked list is 1->2->3->4->5 then your function should convert it to 1->3->5, and if the given linked list is 1->2->3->4 then convert it to 1->3.

### Method 1 (Iterative)

Keep track of previous of the node to be deleted. First change the next link of previous node and then free the memory allocated for the node.

C/C++

```
// C program to remove alternate nodes of a linked list
#include<stdio.h>
#include<stdlib.h>

/* A linked list node */
struct Node
{
    int data;
    struct Node *next;
};

/* deletes alternate nodes of a list starting with head */
void deleteAlt(struct Node *head)
{
    if (head == NULL)
        return;

    /* Initialize prev and node to be deleted */
    struct Node *prev = head;
```

```

struct Node *node = head->next;

while (prev != NULL && node != NULL)
{
    /* Change next link of previous node */
    prev->next = node->next;

    /* Free memory */
    free(node);

    /* Update prev and node */
    prev = prev->next;
    if (prev != NULL)
        node = prev->next;
}
}

/* UTILITY FUNCTIONS TO TEST fun1() and fun2() */
/* Given a reference (pointer to pointer) to the head
   of a list and an int, push a new node on the front
   of the list. */
void push(struct Node** head_ref, int new_data)
{
    /* allocate node */
    struct Node* new_node =
        (struct Node*) malloc(sizeof(struct Node));

    /* put in the data */
    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;
}

/* Function to print nodes in a given linked list */
void printList(struct Node *node)
{
    while (node != NULL)
    {
        printf("%d ", node->data);
        node = node->next;
    }
}

/* Drier program to test above functions */

```

```
int main()
{
    /* Start with the empty list */
    struct Node* head = NULL;

    /* Using push() to construct below list
       1->2->3->4->5 */
    push(&head, 5);
    push(&head, 4);
    push(&head, 3);
    push(&head, 2);
    push(&head, 1);

    printf("\nList before calling deleteAlt() \n");
    printList(head);

    deleteAlt(head);

    printf("\nList after calling deleteAlt() \n");
    printList(head);

    return 0;
}
```

**Java**

```
// Java program to delete alternate nodes of a linked list
class LinkedList
{
    Node head; // head of list

    /* Linked list Node*/
    class Node
    {
        int data;
        Node next;
        Node(int d) {data = d; next = null; }
    }

    void deleteAlt()
    {
        if (head == null)
            return;

        Node prev = head;
        Node now = head.next;

        while (prev != null && now != null)
```

```
{  
    /* Change next link of previous node */  
    prev.next = now.next;  
  
    /* Free node */  
    now = null;  
  
    /*Update prev and now */  
    prev = prev.next;  
    if (prev != null)  
        now = prev.next;  
}  
}  
  
/* Utility functions */  
  
/* Inserts a new Node at front of the list. */  
public void push(int new_data)  
{  
    /* 1 & 2: Allocate the Node &  
       Put in the data*/  
    Node new_node = new Node(new_data);  
  
    /* 3. Make next of new Node as head */  
    new_node.next = head;  
  
    /* 4. Move the head to point to new Node */  
    head = new_node;  
}  
  
/* Function to print linked list */  
void printList()  
{  
    Node temp = head;  
    while(temp != null)  
    {  
        System.out.print(temp.data+" ");  
        temp = temp.next;  
    }  
    System.out.println();  
}  
  
/* Driver program to test above functions */  
public static void main(String args[])  
{  
    LinkedList llist = new LinkedList();  
}
```

```
/* Constructed Linked List is 1->2->3->4->5->null */
llist.push(5);
llist.push(4);
llist.push(3);
llist.push(2);
llist.push(1);

System.out.println("Linked List before calling deleteAlt() ");
llist.printList();

llist.deleteAlt();

System.out.println("Linked List after calling deleteAlt() ");
llist.printList();
}

/*
 * This code is contributed by Rajat Mishra */

```

Output:

```
List before calling deleteAlt()
1 2 3 4 5
List after calling deleteAlt()
1 3 5
```

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

### Method 2 (Recursive)

Recursive code uses the same approach as method 1. The recursive code is simple and short, but causes O(n) recursive function calls for a linked list of size n.

```
/* deletes alternate nodes of a list starting with head */
void deleteAlt(struct Node *head)
{
    if (head == NULL)
        return;

    struct Node *node = head->next;

    if (node == NULL)
        return;

    /* Change the next link of head */
    head->next = node->next;

    /* free memory allocated for node */
    free(node);
```

```
/* Recursively call for the new next of head */
deleteAlt(head->next);
}
```

Time Complexity: O(n)

### Source

<https://www.geeksforgeeks.org/delete-alternate-nodes-of-a-linked-list/>

## Chapter 60

# Delete last occurrence of an item from linked list

Delete last occurrence of an item from linked list - GeeksforGeeks

Given a linked list and a key to be deleted. Delete last occurrence of key from linked. The list may have duplicates.

Examples:

Input: 1->2->3->5->2->10, key = 2  
Output: 1->2->3->5->10

The idea is to traverse the linked list from beginning to end. While traversing, keep track of last occurrence key. After traversing the complete list, delete last occurrence by copying data of next node and deleting the next node.

```
// A C++ program to demonstrate deletion of last
// Node in singly linked list
#include <bits/stdc++.h>

// A linked list Node
struct Node {
    int key;
    struct Node* next;
};

void deleteLast(Node* head, int key)
{
    // Initialize previous of Node to be deleted
    Node* x = NULL;

    // Start from head and find the Node to be
```

```
// deleted
Node* temp = head;
while (temp) {
    // If we found the key, update xv
    if (temp->key == key)
        x = temp;

    temp = temp->next;
}

// key occurs at-least once
if (x != NULL) {

    // Copy key of next Node to x
    x->key = x->next->key;

    // Store and unlink next
    temp = x->next;
    x->next = x->next->next;

    // Free memory for next
    delete temp;
}
}

/* Utility function to create a new node with
   given key */
Node* newNode(int key)
{
    Node* temp = new Node;
    temp->key = key;
    temp->next = NULL;
    return temp;
}

// This function prints contents of linked list
// starting from the given Node
void printList(struct Node* node)
{
    while (node != NULL) {
        printf(" %d ", node->key);
        node = node->next;
    }
}

/* Drier program to test above functions*/
int main()
{
```

```
/* Start with the empty list */
struct Node* head = newNode(1);
head->next = newNode(2);
head->next->next = newNode(3);
head->next->next->next = newNode(5);
head->next->next->next->next = newNode(2);
head->next->next->next->next = newNode(10);

puts("Created Linked List: ");
printList(head);
deleteLast(head, 2);
puts("\nLinked List after Deletion of 1: ");
printList(head);
return 0;
}
```

Output:

```
Created Linked List:
1 2 3 5 2 10
Linked List after Deletion of 1:
1 2 3 5 10
```

**The above solution doesn't work when the node to be deleted is the last node.**

Following solution handles all cases.

```
// A C program to demonstrate deletion of last
// Node in singly linked list
#include <stdio.h>
#include <stdlib.h>

// A linked list Node
struct Node {
    int data;
    struct Node* next;
};

// Function to delete the last occurrence
void deleteLast(struct Node* head, int x)
{
    struct Node *temp = head, *ptr = NULL;
    while (temp) {

        // If found key, update
        if (temp->data == x)
            ptr = temp;
        temp = temp->next;
    }
}
```

```
}

// If the last occurrence is the last node
if (ptr != NULL && ptr->next == NULL) {
    temp = head;
    while (temp->next != ptr)
        temp = temp->next;
    temp->next = NULL;
}

// If it is not the last node
if (ptr != NULL && ptr->next != NULL) {
    ptr->data = ptr->next->data;
    temp = ptr->next;
    ptr->next = ptr->next->next;
    free(temp);
}
}

/* Utility function to create a new node with
given key */
struct Node* newNode(int x)
{
    struct Node* node = malloc(sizeof(struct Node));
    node->data = x;
    node->next = NULL;
    return node;
}

// This function prints contents of linked list
// starting from the given Node
void display(struct Node* head)
{
    struct Node* temp = head;
    if (head == NULL) {
        printf("NULL\n");
        return;
    }
    while (temp != NULL) {
        printf("%d --> ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}

/* Drier program to test above functions*/
int main()
{
```

```
struct Node* head = newNode(1);
head->next = newNode(2);
head->next->next = newNode(3);
head->next->next->next = newNode(4);
head->next->next->next->next = newNode(5);
head->next->next->next->next->next = newNode(4);
head->next->next->next->next->next->next = newNode(4);
printf("Created Linked list: ");
display(head);
deleteLast(head, 4);
printf("List after deletion of 4: ");
display(head);
return 0;
}
```

Output:

```
Created Linked List:
1 2 3 4 5 4 4
Linked List after Deletion of 1:
1 2 3 4 5 4
```

## Source

<https://www.geeksforgeeks.org/delete-last-occurrence-of-an-item-from-linked-list/>

# Chapter 61

## Delete middle of linked list

Delete middle of linked list - GeeksforGeeks

Given a singly linked list, delete middle of the linked list. For example, if given linked list is 1->2->3->4->5 then linked list should be modified to 1->2->4->5

If there are even nodes, then there would be two middle nodes, we need to delete the second middle element. For example, if given linked list is 1->2->3->4->5->6 then it should be modified to 1->2->3->5->6.

If the input linked list is NULL, then it should remain NULL.

If the input linked list has 1 node, then this node should be deleted and new head should be returned.

A Simple Solution is to first count number of nodes in linked list, then delete  $n/2$ 'th node using the simple deletion process.

The above solution requires two traversals of linked list. We can delete middle node using one traversal. The idea is to use two pointers, slow\_ptr and fast\_ptr. Both pointers start from head of list. When fast\_ptr reaches end, slow\_ptr reaches middle. This idea is same as the one used in method 2 of [this](#) post. The additional thing in this post is to keep track of previous of middle so that we can delete middle.

Below is C++ implementation.

```
// C++ program to delete middle of a linked list
#include<bits/stdc++.h>
using namespace std;

/* Link list Node */
struct Node
{
    int data;
    struct Node* next;
};


```

```
// Deletes middle node and returns head of the
// modified list
struct Node* deleteMid(struct Node *head)
{
    // Base cases
    if (head == NULL)
        return NULL;
    if (head->next == NULL)
    {
        delete head;
        return NULL;
    }

    // Initialize slow and fast pointers to reach
    // middle of linked list
    struct Node *slow_ptr = head;
    struct Node *fast_ptr = head;

    // Find the middle and previous of middle.
    struct Node *prev; // To store previous of slow_ptr
    while (fast_ptr != NULL && fast_ptr->next != NULL)
    {
        fast_ptr = fast_ptr->next->next;
        prev = slow_ptr;
        slow_ptr = slow_ptr->next;
    }

    //Delete the middle node
    prev->next = slow_ptr->next;
    delete slow_ptr;

    return head;
}

// A utility function to print a given linked list
void printList(struct Node *ptr)
{
    while (ptr != NULL)
    {
        cout << ptr->data << "->";
        ptr = ptr->next;
    }
    cout << "NULL\n";
}

// Utility function to create a new node.
Node *newNode(int data)
```

```
{  
    struct Node *temp = new Node;  
    temp->data = data;  
    temp->next = NULL;  
    return temp;  
}  
  
/* Drier program to test above function*/  
int main()  
{  
    /* Start with the empty list */  
    struct Node* head = newNode(1);  
    head->next = newNode(2);  
    head->next->next = newNode(3);  
    head->next->next->next = newNode(4);  
  
    cout << "Given Linked List\n";  
    printList(head);  
  
    head = deleteMid(head);  
  
    cout << "Linked List after deletion of middle\n";  
    printList(head);  
  
    return 0;  
}
```

Output :

```
Given Linked List  
1->2->3->4->NULL  
Linked List after deletion of middle  
1->2->4->NULL
```

## Source

<https://www.geeksforgeeks.org/delete-middle-of-linked-list/>

## Chapter 62

# Delete multiple occurrences of key in Linked list using double pointer

Delete multiple occurrences of key in Linked list using double pointer - GeeksforGeeks

Given a singly linked list, delete all occurrences of a given key in it. For example, consider the following list.

```
Input: 2 -> 2 -> 4 -> 3 -> 2  
      Key to delete = 2  
Output: 4 -> 3
```

This is mainly an alternative of this post which deletes multiple occurrences of a given key using separate condition loops for head and remaining nodes. Here we use a double pointer approach to use a single loop irrespective of the position of the element (head, tail or between). The original method to delete a node from a linked list without an extra check for the head was explained by Linus Torvalds in his “25th Anniversary of Linux” TED talk. This article uses that logic to delete multiple occurrences of the key without an extra check for the head.

Explanation:

1. Store address of head in a double pointer till we find a non “key” node. This takes care of the 1st while loop to handle the special case of the head.
2. If a node is not “key” node then store the address of node->next in pp.
3. if we find a “key” node later on then change pp (ultimately node->next) to point to current node->next.

Following is C++ implementation for the same.

```
// CPP program to delete multiple
```

```
// occurrences of a key using single
// loop.
#include <iostream>
using namespace std;

// A linked list node
struct Node {
    int data;
    struct Node* next;
};

struct Node* head = NULL;

void printList(struct Node* node)
{
    while (node != NULL) {
        printf(" %d ", node->data);
        node = node->next;
    }
}

void push(int new_data)
{
    struct Node* new_node = new Node;
    new_node->data = new_data;
    new_node->next = (head);
    (head) = new_node;
}

void deleteEntry(int key)
{
    // Start from head
    struct Node** pp = &head;
    while (*pp) {

        struct Node* entry = *pp;

        // If key found, then put
        // next at the address of pp
        // delete entry.
        if (entry->data == key) {
            *pp = entry->next;
            delete (entry);
        }

        // Else move to next
        else
            pp = &(entry->next);
    }
}
```

```
        }
    }

int main()
{
    push(2);
    push(2);
    push(4);
    push(3);
    push(2);

    int key = 2; // key to delete

    puts("Created Linked List: ");
    printList(head);
    printf("\n");
    deleteEntry(key);
    printf("\nLinked List after Deletion of 2: \n");
    printList(head);
    return 0;
}
```

**Output:**

Created Linked List:

2 3 4 2 2

Linked List after Deletion of 2:

3 4

**Source**

<https://www.geeksforgeeks.org/delete-multiple-occurrences-of-key-in-linked-list-using-double-pointer/>

## Chapter 63

# Delete nodes which have a greater value on right side

Delete nodes which have a greater value on right side - GeeksforGeeks

Given a singly linked list, remove all the nodes which have a greater value on right side.

Examples:

a) The list 12->15->10->11->5->6->2->3->NULL should be changed to 15->11->6->3->NULL. Note that 12, 10, 5 and 2 have been deleted because there is a greater value on the right side.

When we examine 12, we see that after 12 there is one node with value greater than 12 (i.e. 15), so we delete 12.

When we examine 15, we find no node after 15 that has value greater than 15 so we keep this node.

When we go like this, we get 15->6->3

b) The list 10->20->30->40->50->60->NULL should be changed to 60->NULL. Note that 10, 20, 30, 40 and 50 have been deleted because they all have a greater value on the right side.

c) The list 60->50->40->30->20->10->NULL should not be changed.

### Method 1 (Simple)

Use two loops. In the outer loop, pick nodes of the linked list one by one. In the inner loop, check if there exist a node whose value is greater than the picked node. If there exists a node whose value is greater, then delete the picked node.

Time Complexity:  $O(n^2)$

### Method 2 (Use Reverse)

Thanks to Paras for providing the below algorithm.

1. Reverse the list.
2. Traverse the reversed list. Keep max till now. If next node is less than max, then delete the next node, otherwise max = next node.
3. Reverse the list again to retain the original order.

Time Complexity: O(n)

Thanks to R.Srinivasan for providing below code.

C

```
// C program to delete nodes which have a greater value on
// right side
#include <stdio.h>
#include <stdlib.h>

/* structure of a linked list node */
struct Node {
    int data;
    struct Node* next;
};

/* prototype for utility functions */
void reverseList(struct Node** headref);
void _delLesserNodes(struct Node* head);

/* Deletes nodes which have a node with greater value node
   on left side */
void delLesserNodes(struct Node** head_ref)
{
    /* 1) Reverse the linked list */
    reverseList(head_ref);

    /* 2) In the reversed list, delete nodes which have a node
       with greater value node on left side. Note that head
       node is never deleted because it is the leftmost node.*/
    _delLesserNodes(*head_ref);

    /* 3) Reverse the linked list again to retain the
       original order */
    reverseList(head_ref);
}

/* Deletes nodes which have greater value node(s) on left side */
void _delLesserNodes(struct Node* head)
{
    struct Node* current = head;

    /* Initialize max */
    struct Node* maxnode = head;
    struct Node* temp;

    while (current != NULL && current->next != NULL) {
```

```
/* If current is smaller than max, then delete current */
if (current->next->data < maxnode->data) {
    temp = current->next;
    current->next = temp->next;
    free(temp);
}

/* If current is greater than max, then update max and
   move current */
else {
    current = current->next;
    maxnode = current;
}
}

/* Utility function to insert a node at the begining */
void push(struct Node** head_ref, int new_data)
{
    struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));
    new_node->data = new_data;
    new_node->next = *head_ref;
    *head_ref = new_node;
}

/* Utility function to reverse a linked list */
void reverseList(struct Node** headref)
{
    struct Node* current = *headref;
    struct Node* prev = NULL;
    struct Node* next;
    while (current != NULL) {
        next = current->next;
        current->next = prev;
        prev = current;
        current = next;
    }
    *headref = prev;
}

/* Utility function to print a linked list */
void printList(struct Node* head)
{
    while (head != NULL) {
        printf("%d ", head->data);
        head = head->next;
    }
    printf("\n");
```

```
}

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

    /* Create following linked list
       12->15->10->11->5->6->2->3 */
    push(&head, 3);
    push(&head, 2);
    push(&head, 6);
    push(&head, 5);
    push(&head, 11);
    push(&head, 10);
    push(&head, 15);
    push(&head, 12);

    printf("Given Linked List \n");
    printList(head);

    delLesserNodes(&head);

    printf("Modified Linked List \n");
    printList(head);

    return 0;
}
```

### Java

```
// Java program to delete nodes which have a greater value on
// right side
class LinkedList {
    Node head; // head of list

    /* Linked list Node*/
    class Node {
        int data;
        Node next;
        Node(int d)
        {
            data = d;
            next = null;
        }
    }

    /* Deletes nodes which have a node with greater
```

```
    value node on left side */
void delLesserNodes()
{
    /* 1.Reverse the linked list */
reverseList();

    /* 2) In the reversed list, delete nodes which
       have a node with greater value node on left
       side. Note that head node is never deleted
       because it is the leftmost node.*/
    _delLesserNodes();

    /* 3) Reverse the linked list again to retain
       the original order */
reverseList();
}

/* Deletes nodes which have greater value node(s)
   on left side */
void _delLesserNodes()
{
    Node current = head;

    /* Initialise max */
    Node maxnode = head;
    Node temp;

    while (current != null && current.next != null) {
        /* If current is smaller than max, then delete
           current */
        if (current.next.data < maxnode.data) {
            temp = current.next;
            current.next = temp.next;
            temp = null;
        }

        /* If current is greater than max, then update
           max and move current */
        else {
            current = current.next;
            maxnode = current;
        }
    }
}

/* Utility functions */

/* Inserts a new Node at front of the list. */
```

```
void push(int new_data)
{
    /* 1 & 2: Allocate the Node &
       Put in the data*/
    Node new_node = new Node(new_data);

    /* 3. Make next of new Node as head */
    new_node.next = head;

    /* 4. Move the head to point to new Node */
    head = new_node;
}

/* Function to reverse the linked list */
void reverseList()
{
    Node current = head;
    Node prev = null;
    Node next;
    while (current != null) {
        next = current.next;
        current.next = prev;
        prev = current;
        current = next;
    }
    head = prev;
}

/* Function to print linked list */
void printList()
{
    Node temp = head;
    while (temp != null) {
        System.out.print(temp.data + " ");
        temp = temp.next;
    }
    System.out.println();
}

/* Drier program to test above functions */
public static void main(String args[])
{
    LinkedList llist = new LinkedList();

    /* Constructed Linked List is 12->15->10->11->
       5->6->2->3 */
    llist.push(3);
    llist.push(2);
```

```
llist.push(6);
llist.push(5);
llist.push(11);
llist.push(10);
llist.push(15);
llist.push(12);

System.out.println("Given Linked List");
llist.printList();

llist.delLesserNodes();

System.out.println("Modified Linked List");
llist.printList();
}

} /* This code is contributed by Rajat Mishra */
```

Output:

```
Given Linked List
12 15 10 11 5 6 2 3
Modified Linked List
15 11 6 3
```

Source:

<https://www.geeksforgeeks.org/forum/topic/amazon-interview-question-for-software-engineer-developer-about-linked-lists/>

## Source

<https://www.geeksforgeeks.org/delete-nodes-which-have-a-greater-value-on-right-side/>

## Chapter 64

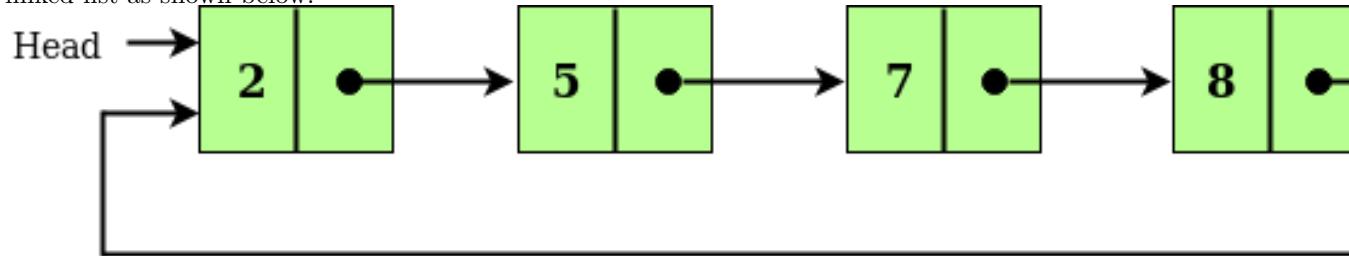
# Deletion from a Circular Linked List

Deletion from a Circular Linked List - GeeksforGeeks

We have already discussed about circular linked list and traversal in a circular linked list in the below articles:

[Introduction to circular linked list](#)  
[Traversal in a circular linked list](#)

In this article we will learn about deleting a node from a circular linked list. Consider the linked list as shown below:



We will be given a node and our task is to delete that node from the circular linked list.

Examples:

```
Input : 2->5->7->8->10->(head node)
        data = 5
Output : 2->7->8->10->(head node)
```

```
Input : 2->5->7->8->10->(head node)
        7
Output : 2->5->8->10->2(head node)
```

**Algorithm**

**Case 1:** List is empty.

- If the list is empty we will simply return.

**Case 2:** List is not empty

- If the list is not empty then we define two pointers **curr** and **prev** and initialize the pointer **curr** with the **head** node.
- Traverse the list using **curr** to find the node to be deleted and before moving curr to next node, everytime set **prev = curr**.
- If the node is found, check if it is the only node in the list. If yes, set **head = NULL** and **free(curr)**.
- If the list has more than one node, check if it is the first node of the list. Condition to check this( **curr == head**). If yes, then move **prev** until it reaches the last node. After **prev** reaches the last node, set **head = head -> next** and **prev -> next = head**. Delete curr.
- If curr is not first node, we check if it is the last node in the list. Condition to check this is (**curr -> next == head**).
- If curr is the last node. Set **prev -> next = head** and delete the node curr by **free(curr)**.
- If the node to be deleted is neither the first node nor the last node, then set **prev -> next = temp -> next** and delete curr.

**Complete C program to demonstrate deletion in Circular Linked List:**

```
// C program to delete a given key from
// linked list.
#include<stdio.h>
#include<stdlib.h>

/* structure for a node */
struct Node
{
    int data;
    struct Node *next;
};

/* Function to insert a node at the beginning of
   a Circular linked list */
void push(struct Node **head_ref, int data)
{
    // Create a new node and make head as next
    // of it.
    struct Node *ptr1 =
        (struct Node *)malloc(sizeof(struct Node));
    ptr1->data = data;
    ptr1->next = *head_ref;
```

```

/* If linked list is not NULL then set the
   next of last node */
if (*head_ref != NULL)
{
    // Find the node before head and update
    // next of it.
    struct Node *temp = *head_ref;
    while (temp->next != *head_ref)
        temp = temp->next;
    temp->next = ptr1;
}
else
    ptr1->next = ptr1; /*For the first node */

*head_ref = ptr1;
}

/* Function to print nodes in a given
   circular linked list */
void printList(struct Node *head)
{
    struct Node *temp = head;
    if (head != NULL)
    {
        do
        {
            printf("%d ", temp->data);
            temp = temp->next;
        }
        while (temp != head);
    }

    printf("\n");
}

/* Function to delete a given node from the list */
void deleteNode(struct Node *head, int key)
{
    if (head == NULL)
        return;

    // Find the required node
    struct Node *curr = head, *prev;
    while (curr->data != key)
    {
        if (curr->next == head)
        {
            printf("\nGiven node is not found"

```

```
        " in the list!!!");
    break;
}

prev = curr;
curr = curr -> next;
}

// Check if node is only node
if (curr->next == head)
{
    head = NULL;
    free(curr);
    return;
}

// If more than one node, check if
// it is first node
if (curr == head)
{
    prev = head;
    while (prev -> next != head)
        prev = prev -> next;
    head = curr->next;
    prev->next = head;
    free(curr);
}

// check if node is last node
else if (curr -> next == head)
{
    prev->next = head;
    free(curr);
}
else
{
    prev->next = curr->next;
    free(curr);
}
}

/* Driver program to test above functions */
int main()
{
    /* Initialize lists as empty */
    struct Node *head = NULL;

    /* Created linked list will be 2->5->7->8->10 */
}
```

```
push(&head, 2);
push(&head, 5);
push(&head, 7);
push(&head, 8);
push(&head, 10);

printf("List Before Deletion: ");
printList(head);

deleteNode(head, 7);

printf("List After Deletion: ");
printList(head);

return 0;
}
```

**Output:**

```
List Before Deletion: 10 8 7 5 2
List After Deletion: 10 8 5 2
```

**Source**

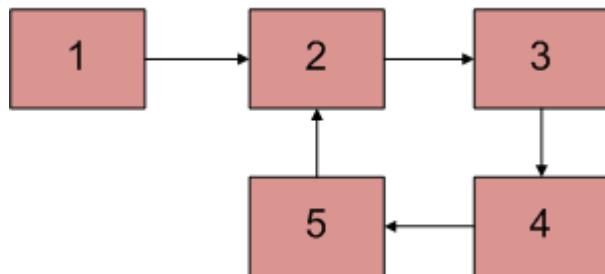
<https://www.geeksforgeeks.org/deletion-circular-linked-list/>

## Chapter 65

# Detect and Remove Loop in a Linked List

[Detect and Remove Loop in a Linked List - GeeksforGeeks](#)

Write a function *detectAndRemoveLoop()* that checks whether a given Linked List contains loop and if loop is present then removes the loop and returns true. And if the list doesn't contain loop then returns false. Below diagram shows a linked list with a loop. *detectAndRemoveLoop()* must change the below list to 1->2->3->4->5->NULL.



We also recommend to read following post as a prerequisite of the solution discussed here.

[Write a C function to detect loop in a linked list](#)

Before trying to remove the loop, we must detect it. Techniques discussed in the above post can be used to detect loop. To remove loop, all we need to do is to get pointer to the last node of the loop. For example, node with value 5 in the above diagram. Once we have pointer to the last node, we can make the next of this node as NULL and loop is gone. We can easily use Hashing or Visited node techniques (discussed in the above mentioned post) to get the pointer to the last node. Idea is simple: the very first node whose next is already visited (or hashed) is the last node.

We can also use Floyd Cycle Detection algorithm to detect and remove the loop. In the Floyd's algo, the slow and fast pointers meet at a loop node. We can use this loop node

to remove cycle. There are following two different ways of removing loop when Floyd's algorithm is used for Loop detection.

**Method 1 (Check one by one)**

We know that Floyd's Cycle detection algorithm terminates when fast and slow pointers meet at a common point. We also know that this common point is one of the loop nodes (2 or 3 or 4 or 5 in the above diagram). We store the address of this in a pointer variable say ptr2. Then we start from the head of the Linked List and check for nodes one by one if they are reachable from ptr2. When we find a node that is reachable, we know that this node is the starting node of the loop in Linked List and we can get pointer to the previous of this node.

C

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

/* Link list node */
struct Node
{
    int data;
    struct Node* next;
};

/* Function to remove loop. Used by detectAndRemoveLoop() */
void removeLoop(struct Node *, struct Node *);

/* This function detects and removes loop in the list
   If loop was there in the list then it returns 1,
   otherwise returns 0 */
int detectAndRemoveLoop(struct Node *list)
{
    struct Node *slow_p = list, *fast_p = list;

    while (slow_p && fast_p && fast_p->next)
    {
        slow_p = slow_p->next;
        fast_p = fast_p->next->next;

        /* If slow_p and fast_p meet at some point then there
           is a loop */
        if (slow_p == fast_p)
        {
            removeLoop(slow_p, list);

            /* Return 1 to indicate that loop is found */
            return 1;
        }
    }
}
```

```

/* Return 0 to indicate that there is no loop*/
return 0;
}

/* Function to remove loop.
loop_node --> Pointer to one of the loop nodes
head --> Pointer to the start node of the linked list */
void removeLoop(struct Node *loop_node, struct Node *head)
{
    struct Node *ptr1;
    struct Node *ptr2;

    /* Set a pointer to the beginning of the Linked List and
       move it one by one to find the first node which is
       part of the Linked List */
    ptr1 = head;
    while (1)
    {
        /* Now start a pointer from loop_node and check if it ever
           reaches ptr2 */
        ptr2 = loop_node;
        while (ptr2->next != loop_node && ptr2->next != ptr1)
            ptr2 = ptr2->next;

        /* If ptr2 reached ptr1 then there is a loop. So break the
           loop */
        if (ptr2->next == ptr1)
            break;

        /* If ptr2 didn't reach ptr1 then try the next node after ptr1 */
        ptr1 = ptr1->next;
    }

    /* After the end of loop ptr2 is the last node of the loop. So
       make next of ptr2 as NULL */
    ptr2->next = NULL;
}

/* Function to print linked list */
void printList(struct Node *node)
{
    while (node != NULL)
    {
        printf("%d ", node->data);
        node = node->next;
    }
}

```

```
struct Node *newNode(int key)
{
    struct Node *temp = (struct Node*)malloc(sizeof(struct Node));
    temp->data = key;
    temp->next = NULL;
    return temp;
}

/* Drier program to test above function*/
int main()
{
    struct Node *head = newNode(50);
    head->next = newNode(20);
    head->next->next = newNode(15);
    head->next->next->next = newNode(4);
    head->next->next->next->next = newNode(10);

    /* Create a loop for testing */
    head->next->next->next->next->next = head->next->next;

    detectAndRemoveLoop(head);

    printf("Linked List after removing loop \n");
    printList(head);
    return 0;
}
```

### Java

```
// Java program to detect and remove loop in linked list

class LinkedList {

    static Node head;

    static class Node {

        int data;
        Node next;

        Node(int d) {
            data = d;
            next = null;
        }
    }

    // Function that detects loop in the list
```

```

int detectAndRemoveLoop(Node node) {
    Node slow = node, fast = node;
    while (slow != null && fast != null && fast.next != null) {
        slow = slow.next;
        fast = fast.next.next;

        // If slow and fast meet at same point then loop is present
        if (slow == fast) {
            removeLoop(slow, node);
            return 1;
        }
    }
    return 0;
}

// Function to remove loop
void removeLoop(Node loop, Node curr) {
    Node ptr1 = null, ptr2 = null;

    /* Set a pointer to the begining of the Linked List and
       move it one by one to find the first node which is
       part of the Linked List */
    ptr1 = curr;
    while (1 == 1) {

        /* Now start a pointer from loop_node and check if it ever
           reaches ptr2 */
        ptr2 = loop;
        while (ptr2.next != loop && ptr2.next != ptr1) {
            ptr2 = ptr2.next;
        }

        /* If ptr2 reached ptr1 then there is a loop. So break the
           loop */
        if (ptr2.next == ptr1) {
            break;
        }

        /* If ptr2 didn't reach ptr1 then try the next node after ptr1 */
        ptr1 = ptr1.next;
    }

    /* After the end of loop ptr2 is the last node of the loop. So
       make next of ptr2 as NULL */
    ptr2.next = null;
}

// Function to print the linked list

```

```
void printList(Node node) {
    while (node != null) {
        System.out.print(node.data + " ");
        node = node.next;
    }
}

// Driver program to test above functions
public static void main(String[] args) {
    LinkedList list = new LinkedList();
    list.head = new Node(50);
    list.head.next = new Node(20);
    list.head.next.next = new Node(15);
    list.head.next.next.next = new Node(4);
    list.head.next.next.next.next = new Node(10);

    // Creating a loop for testing
    head.next.next.next.next = head.next.next;
    list.detectAndRemoveLoop(head);
    System.out.println("Linked List after removing loop : ");
    list.printList(head);
}
}

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

### Python

```
# Python program to detect and remove loop in linked list

# Node class
class Node:

    # Constructor to initialize the node object
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:

    # Function to initialize head
    def __init__(self):
        self.head = None

    def detectAndRemoveLoop(self):
        slow_p = fast_p = self.head
        while(slow_p and fast_p and fast_p.next):
            slow_p = slow_p.next
            fast_p = fast_p.next.next
```

```
fast_p = fast_p.next.next

# If slow_p and fast_p meet at some point
# then there is a loop
if slow_p == fast_p:
    self.removeLoop(slow_p)

# Return 1 to indicate that loop is found
return 1

# Return 0 to indicate that there is no loop
return 0

# Function to remove loop
# loop_node -> Pointer to one of the loop nodes
# head --> Pointer to the start node of the
# linked list
def removeLoop(self, loop_node):

    # Set a pointer to the beginning of the linked
    # list and move it one by one to find the first
    # node which is part of the linked list
    ptr1 = self.head
    while(1):
        # Now start a pointer from loop_node and check
        # if it ever reaches ptr2
        ptr2 = loop_node
        while(ptr2.next!= loop_node and ptr2.next !=ptr1):
            ptr2 = ptr2.next

        # If ptr2 reached ptr1 then there is a loop.
        # So break the loop
        if ptr2.next == ptr1 :
            break

        ptr1 = ptr1.next

    # After the end of loop ptr2 is the last node of
    # the loop. So make next of ptr2 as NULL
    ptr2.next = None
# Function to insert a new node at the beginning
def push(self, new_data):
    new_node = Node(new_data)
    new_node.next = self.head
    self.head = new_node

# Utility function to print the linked LinkedList
def printList(self):
```

```
temp = self.head
while(temp):
    print temp.data,
    temp = temp.next

# Driver program
llist = LinkedList()
llist.push(10)
llist.push(4)
llist.push(15)
llist.push(20)
llist.push(50)

# Create a loop for testing
llist.head.next.next.next.next = llist.head.next.next

llist.detectAndRemoveLoop()

print "Linked List after removing loop"
llist.printList()

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

Output:

```
Linked List after removing loop
50 20 15 4 10
```

### Method 2 (Better Solution)

This method is also dependent on Floyd's Cycle detection algorithm.

- 1) Detect Loop using Floyd's Cycle detection algo and get the pointer to a loop node.
- 2) Count the number of nodes in loop. Let the count be k.
- 3) Fix one pointer to the head and another to kth node from head.
- 4) Move both pointers at the same pace, they will meet at loop starting node.
- 5) Get pointer to the last node of loop and make next of it as NULL.

Thanks to WgpShashank for suggesting this method.

C

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

/* Link list node */
struct Node
{
    int data;
```

```

    struct Node* next;
};

/* Function to remove loop. */
void removeLoop(struct Node *, struct Node *);

/* This function detects and removes loop in the list
   If loop was there in the list then it returns 1,
   otherwise returns 0 */
int detectAndRemoveLoop(struct Node *list)
{
    struct Node  *slow_p = list, *fast_p = list;

    while (slow_p && fast_p && fast_p->next)
    {
        slow_p = slow_p->next;
        fast_p  = fast_p->next->next;

        /* If slow_p and fast_p meet at some point then there
           is a loop */
        if (slow_p == fast_p)
        {
            removeLoop(slow_p, list);

            /* Return 1 to indicate that loop is found */
            return 1;
        }
    }

    /* Return 0 to indicate that there is no loop*/
    return 0;
}

/* Function to remove loop.
   loop_node --> Pointer to one of the loop nodes
   head --> Pointer to the start node of the linked list */
void removeLoop(struct Node *loop_node, struct Node *head)
{
    struct Node *ptr1 = loop_node;
    struct Node *ptr2 = loop_node;

    // Count the number of nodes in loop
    unsigned int k = 1, i;
    while (ptr1->next != ptr2)
    {
        ptr1 = ptr1->next;
        k++;
    }
}

```

```
// Fix one pointer to head
ptr1 = head;

// And the other pointer to k nodes after head
ptr2 = head;
for (i = 0; i < k; i++)
    ptr2 = ptr2->next;

/* Move both pointers at the same pace,
   they will meet at loop starting node */
while (ptr2 != ptr1)
{
    ptr1 = ptr1->next;
    ptr2 = ptr2->next;
}

// Get pointer to the last node
ptr2 = ptr2->next;
while (ptr2->next != ptr1)
    ptr2 = ptr2->next;

/* Set the next node of the loop ending node
   to fix the loop */
ptr2->next = NULL;
}

/* Function to print linked list */
void printList(struct Node *node)
{
    while (node != NULL)
    {
        printf("%d ", node->data);
        node = node->next;
    }
}

struct Node *newNode(int key)
{
    struct Node *temp = (struct Node*)malloc(sizeof(struct Node));
    temp->data = key;
    temp->next = NULL;
    return temp;
}

/* Driver program to test above function*/
int main()
{
```

```
struct Node *head = newNode(50);
head->next = newNode(20);
head->next->next = newNode(15);
head->next->next->next = newNode(4);
head->next->next->next->next = newNode(10);

/* Create a loop for testing */
head->next->next->next->next->next = head->next->next;

detectAndRemoveLoop(head);

printf("Linked List after removing loop \n");
printList(head);
return 0;
}
```

**Java**

```
// Java program to detect and remove loop in linked list

class LinkedList {

    static Node head;

    static class Node {

        int data;
        Node next;

        Node(int d) {
            data = d;
            next = null;
        }
    }

    // Function that detects loop in the list
    int detectAndRemoveLoop(Node node) {
        Node slow = node, fast = node;
        while (slow != null && fast != null && fast.next != null) {
            slow = slow.next;
            fast = fast.next.next;

            // If slow and fast meet at same point then loop is present
            if (slow == fast) {
                removeLoop(slow, node);
                return 1;
            }
        }
    }
}
```

```
        return 0;
    }

    // Function to remove loop
    void removeLoop(Node loop, Node head) {
        Node ptr1 = loop;
        Node ptr2 = loop;

        // Count the number of nodes in loop
        int k = 1, i;
        while (ptr1.next != ptr2) {
            ptr1 = ptr1.next;
            k++;
        }

        // Fix one pointer to head
        ptr1 = head;

        // And the other pointer to k nodes after head
        ptr2 = head;
        for (i = 0; i < k; i++) {
            ptr2 = ptr2.next;
        }

        /* Move both pointers at the same pace,
           they will meet at loop starting node */
        while (ptr2 != ptr1) {
            ptr1 = ptr1.next;
            ptr2 = ptr2.next;
        }

        // Get pointer to the last node
        ptr2 = ptr2.next;
        while (ptr2.next != ptr1) {
            ptr2 = ptr2.next;
        }

        /* Set the next node of the loop ending node
           to fix the loop */
        ptr2.next = null;
    }

    // Function to print the linked list
    void printList(Node node) {
        while (node != null) {
            System.out.print(node.data + " ");
            node = node.next;
        }
    }
}
```

```
}

// Driver program to test above functions
public static void main(String[] args) {
    LinkedList list = new LinkedList();
    list.head = new Node(50);
    list.head.next = new Node(20);
    list.head.next.next = new Node(15);
    list.head.next.next.next = new Node(4);
    list.head.next.next.next.next = new Node(10);

    // Creating a loop for testing
    head.next.next.next.next = head.next.next;
    list.detectAndRemoveLoop(head);
    System.out.println("Linked List after removing loop : ");
    list.printList(head);
}
}

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

### Python

```
# Python program to detect and remove loop in linked list

# Node class
class Node:

    # Constructor to initialize the node object
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:

    # Function to initialize head
    def __init__(self):
        self.head = None

    def detectAndRemoveLoop(self):
        slow_p = fast_p = self.head

        while(slow_p and fast_p and fast_p.next):
            slow_p = slow_p.next
            fast_p = fast_p.next.next

            # If slow_p and fast_p meet at some point then
            # there is a loop
```

```
if slow_p == fast_p:  
    self.removeLoop(slow_p)  
  
    # Return 1 to indicate that loop is found  
    return 1  
  
# Return 0 to indicate that there is no loop  
return 0  
  
# Function to remove loop  
# loop_node --> pointer to one of the loop nodes  
# head --> Pointer to the start node of the linked list  
def removeLoop(self, loop_node):  
    ptr1 = loop_node  
    ptr2 = loop_node  
  
    # Count the number of nodes in loop  
    k = 1  
    while(ptr1.next != ptr2):  
        ptr1 = ptr1.next  
        k += 1  
  
    # Fix one pointer to head  
    ptr1 = self.head  
  
    # And the other pointer to k nodes after head  
    ptr2 = self.head  
    for i in range(k):  
        ptr2 = ptr2.next  
  
    # Move both pointers at the same place  
    # they will meet at loop starting node  
    while(ptr2 != ptr1):  
        ptr1 = ptr1.next  
        ptr2 = ptr2.next  
  
    # Get pointer to the last node  
    ptr2 = ptr2.next  
    while(ptr2.next != ptr1):  
        ptr2 = ptr2.next  
  
    # Set the next node of the loop ending node  
    # to fix the loop  
    ptr2.next = None  
  
# Function to insert a new node at the beginning  
def push(self, new_data):  
    new_node = Node(new_data)
```

```
new_node.next = self.head
self.head = new_node

# Utility function to print the linked LinkedList
def printList(self):
    temp = self.head
    while(temp):
        print temp.data,
        temp = temp.next

# Driver program
llist = LinkedList()
llist.push(10)
llist.push(4)
llist.push(15)
llist.push(20)
llist.push(50)

# Create a loop for testing
llist.head.next.next.next.next = llist.head.next.next

llist.detectAndRemoveLoop()

print "Linked List after removing loop"
llist.printList()

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

Output:

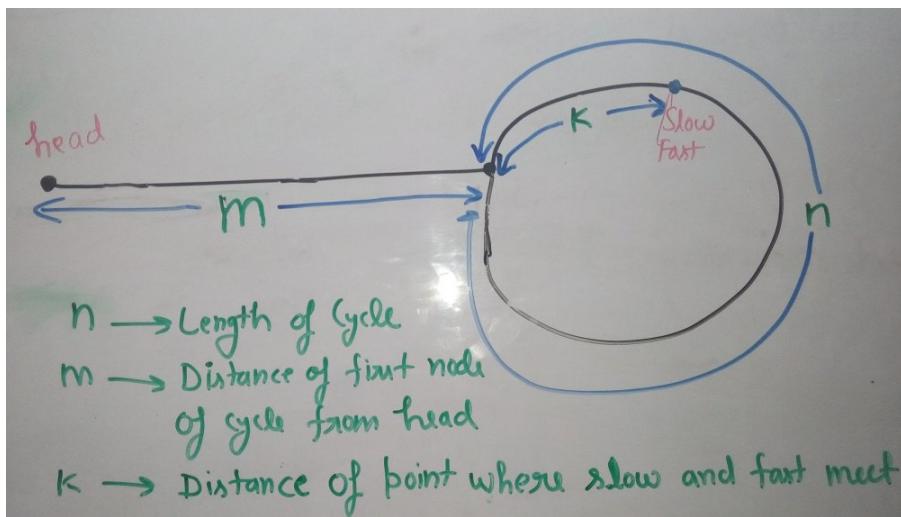
```
Linked List after removing loop
50 20 15 4 10
```

### Method 3 (Optimized Method 2: Without Counting Nodes in Loop)

We do not need to count number of nodes in Loop. After detecting the loop, if we start slow pointer from head and move both slow and fast pointers at same speed until fast don't meet, they would meet at the beginning of the loop.

#### How does this work?

Let slow and fast meet at some point after Floyd's Cycle finding algorithm. Below diagram shows the situation when cycle is found.



We can conclude below from above diagram

Distance traveled by fast pointer =  $2 * (\text{Distance traveled by slow pointer})$

$$(m + n*x + k) = 2*(m + n*y + k)$$

Note that before meeting the point shown above, fast was moving at twice speed.

$x \rightarrow$  Number of complete cyclic rounds made by fast pointer before they meet first time

$y \rightarrow$  Number of complete cyclic rounds made by slow pointer before they meet first time

From above equation, we can conclude below

$$m + k = (x - 2y)*n$$

Which means  $m+k$  is a multiple of  $n$ .

So if we start moving both pointers again at **same speed** such that one pointer (say slow) begins from head node of linked list and other pointer (say fast) begins from meeting point. When slow pointer reaches beginning of loop (has made  $m$  steps), fast pointer would have made also moved  $m$  steps as they are now moving same pace. Since  $m+k$  is a multiple of  $n$

and fast starts from k, they would meet at the beginning. Can they meet before also? No because slow pointer enters the cycle first time after m steps.

C++

```
// C++ program to detect and remove loop
#include<bits/stdc++.h>
using namespace std;

struct Node
{
    int key;
    struct Node *next;
};

Node *newNode(int key)
{
    Node *temp = new Node;
    temp->key = key;
    temp->next = NULL;
    return temp;
}

// A utility function to print a linked list
void printList(Node *head)
{
    while (head != NULL)
    {
        cout << head->key << " ";
        head = head->next;
    }
    cout << endl;
}

// Function to detect and remove loop
// in a linked list that may contain loop
void detectAndRemoveLoop(Node *head)
{
    // If list is empty or has only one node
    // without loop
    if (head == NULL || head->next == NULL)
        return;

    Node *slow = head, *fast = head;

    // Move slow and fast 1 and 2 steps
    // ahead respectively.
    slow = slow->next;
    fast = fast->next->next;
```

```
// Search for loop using slow and
// fast pointers
while (fast && fast->next)
{
    if (slow == fast)
        break;
    slow = slow->next;
    fast = fast->next->next;
}

/* If loop exists */
if (slow == fast)
{
    slow = head;
    while (slow->next != fast->next)
    {
        slow = slow->next;
        fast = fast->next;
    }

    /* since fast->next is the looping point */
    fast->next = NULL; /* remove loop */
}
}

/* Driver program to test above function*/
int main()
{
    Node *head = newNode(50);
    head->next = head;
    head->next = newNode(20);
    head->next->next = newNode(15);
    head->next->next->next = newNode(4);
    head->next->next->next->next = newNode(10);

    /* Create a loop for testing */
    head->next->next->next->next->next = head->next->next;

    detectAndRemoveLoop(head);

    printf("Linked List after removing loop \n");
    printList(head);

    return 0;
}
```

Java

```
// Java program to detect and remove loop in linked list

class LinkedList {

    static Node head;

    static class Node {

        int data;
        Node next;

        Node(int d) {
            data = d;
            next = null;
        }
    }

    // Function that detects loop in the list
    void detectAndRemoveLoop(Node node) {

        // If list is empty or has only one node
        // without loop
        if (node == null || node.next == null)
            return;

        Node slow = node, fast = node;

        // Move slow and fast 1 and 2 steps
        // ahead respectively.
        slow = slow.next;
        fast = fast.next.next;

        // Search for loop using slow and fast pointers
        while (fast != null && fast.next != null) {
            if (slow == fast)
                break;

            slow = slow.next;
            fast = fast.next.next;
        }

        /* If loop exists */
        if (slow == fast) {
            slow = node;
            while (slow.next != fast.next) {
                slow = slow.next;
                fast = fast.next;
            }
        }
    }
}
```

```
        /* since fast->next is the looping point */
        fast.next = null; /* remove loop */
    }
}

// Function to print the linked list
void printList(Node node) {
    while (node != null) {
        System.out.print(node.data + " ");
        node = node.next;
    }
}

// Driver program to test above functions
public static void main(String[] args) {
    LinkedList list = new LinkedList();
    list.head = new Node(50);
    list.head.next = new Node(20);
    list.head.next.next = new Node(15);
    list.head.next.next.next = new Node(4);
    list.head.next.next.next.next = new Node(10);

    // Creating a loop for testing
    head.next.next.next.next = head.next.next;
    list.detectAndRemoveLoop(head);
    System.out.println("Linked List after removing loop : ");
    list.printList(head);
}
}

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

### Python

```
# Python program to detect and remove loop

# Node class
class Node:

    # Constructor to initialize the node object
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:

    # Function to initialize head
```

```
def __init__(self):
    self.head = None

# Function to insert a new node at the beginning
def push(self, new_data):
    new_node = Node(new_data)
    new_node.next = self.head
    self.head = new_node


def detectAndRemoveLoop(self):

    if self.head is None :
        return
    if self.head.next is None :
        return

    slow = self.head
    fast = self.head

    # Move slow and fast 1 and 2 steps respectively
    slow = slow.next
    fast = fast.next.next

    # Search for loop using slow and fast pointers
    while (fast is not None):
        if fast.next is None:
            break
        if slow == fast :
            break
        slow = slow.next
        fast = fast.next.next

    # if loop exists
    if slow == fast :
        slow = self.head
        while (slow.next != fast.next):
            slow = slow.next
            fast = fast.next

    # Since fast.next is the looping point
    fast.next = None # Remove loop

# Utility function to print the linked LinkedList
def printList(self):
    temp = self.head
    while(temp):
```

```
        print temp.data,
        temp = temp.next

# Driver program
llist = LinkedList()
llist.head = Node(50)
llist.head.next = Node(20)
llist.head.next.next = Node(15)
llist.head.next.next.next = Node(4)
llist.head.next.next.next.next = Node(10)

#Create a loop for testing
llist.head.next.next.next.next = llist.head.next.next

llist.detectAndRemoveLoop()

print "Linked List after removing loop"
llist.printList()

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

Output:

```
Linked List after removing loop
50 20 15 4 10
```

#### Method 4 Hashing: Hash the address of the linked list nodes

We can hash the addresses of the linked list nodes in an unordered map and just check if the element already exists in the map. If it exists, we have reached a node which already exists by a cycle, hence we need to make the last node's next pointer NULL.

C++

```
// C++ program to detect and remove loop
#include<bits/stdc++.h>
using namespace std;

struct Node
{
    int key;
    struct Node *next;
};

Node *newNode(int key)
{
    Node *temp = new Node;
    temp->key = key;
```

```

        temp->next = NULL;
        return temp;
    }

// A utility function to print a linked list
void printList(Node *head)
{
    while (head != NULL)
    {
        cout << head->key << " ";
        head = head->next;
    }
    cout << endl;
}

// Function to detect and remove loop
// in a linked list that may contain loop
void hashAndRemove(Node *head)
{
    //hash map to hash addresses of the linked list nodes
    unordered_map<Node*, int> node_map;
    //pointer to last node
    Node* last = NULL;
    while(head!=NULL){
        //if node not present in the map, insert it in the map
        if(node_map.find(head) == node_map.end()){
            node_map[head]++;
            last = head;
            head = head->next;
        }
        //if present, it is a cycle, make the last node's next pointer NULL
        else{
            last->next = NULL;
            head = head->next;
        }
    }
}
/* Driver program to test above function*/
int main()
{
    Node *head = newNode(50);
    head->next = head;
    head->next = newNode(20);
    head->next->next = newNode(15);
    head->next->next->next = newNode(4);
    head->next->next->next->next = newNode(10);

    /* Create a loop for testing */
}

```

```
head->next->next->next->next->next = head->next->next;  
  
//printList(head);  
hashAndRemove(head);  
  
printf("Linked List after removing loop \n");  
printList(head);  
  
return 0;  
}
```

We Thank Shubham Agrawal for suggesting this solution.

Thanks to Gaurav Ahirwar for suggesting above solution.

**Improved By :** [Shubham Agrawal 8](#)

## Source

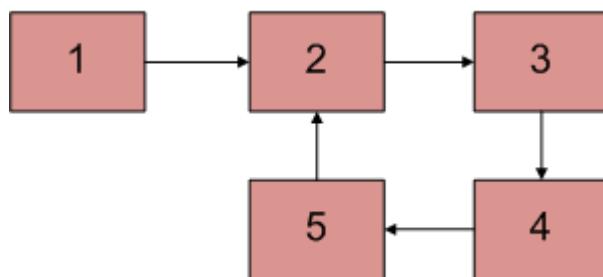
<https://www.geeksforgeeks.org/detect-and-remove-loop-in-a-linked-list/>

# Chapter 66

## Detect loop in a linked list

Detect loop in a linked list - GeeksforGeeks

Given a linked list, check if the the linked list has loop or not. Below diagram shows a linked list with a loop.



Following are different ways of doing this

### Use Hashing:

Traverse the list one by one and keep putting the node addresses in a Hash Table. At any point, if NULL is reached then return false and if next of current node points to any of the previously stored nodes in Hash then return true.

C++

```
// C++ program to detect loop in a linked list
#include<bits/stdc++.h>
using namespace std;

/* Link list node */
struct Node
{
    int data;
    struct Node* next;
}
```

```

};

void push(struct Node** head_ref, int new_data)
{
    /* allocate node */
    struct Node* new_node = new Node;

    /* put in the data */
    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;
}

// Returns true if there is a loop in linked list
// else returns false.
bool detectLoop(struct Node *h)
{
    unordered_set<Node *> s;
    while (h != NULL)
    {
        // If this node is already present
        // in hashmap it means there is a cycle
        // (Because you we encountering the
        // node for the second time).
        if (s.find(h) != s.end())
            return true;

        // If we are seeing the node for
        // the first time, insert it in hash
        s.insert(h);

        h = h->next;
    }

    return false;
}

/* Drier program to test above function*/
int main()
{
    /* Start with the empty list */
    struct Node* head = NULL;

    push(&head, 20);
}

```

```
push(&head, 4);
push(&head, 15);
push(&head, 10);

/* Create a loop for testing */
head->next->next->next->next = head;

if (detectLoop(head))
    cout << "Loop found";
else
    cout << "No Loop";

return 0;
}

// This code is contributed by Geetanjali
```

**Java**

```
// Java program to detect loop in a linked list
import java.util.*;

public class LinkedList {

    static Node head; // head of list

    /* Linked list Node*/
    static class Node
    {
        int data;
        Node next;
        Node(int d) {data = d; next = null; }
    }

    /* Inserts a new Node at front of the list. */
    static public void push(int new_data)
    {
        /* 1 & 2: Allocate the Node &
           Put in the data*/
        Node new_node = new Node(new_data);

        /* 3. Make next of new Node as head */
        new_node.next = head;

        /* 4. Move the head to point to new Node */
        head = new_node;
    }

    // Returns true if there is a loop in linked
```

```
// list else returns false.
static boolean detectLoop(Node h)
{
    HashSet<Node> s = new HashSet<Node>();
    while (h != null)
    {
        // If we have already has this node
        // in hashmap it means their is a cycle
        // (Because you we encountering the
        // node second time).
        if (s.contains(h))
            return true;

        // If we are seeing the node for
        // the first time, insert it in hash
        s.add(h);

        h = h.next;
    }

    return false;
}

/* Driver program to test above function */
public static void main(String[] args)
{
    LinkedList llist = new LinkedList();

    llist.push(20);
    llist.push(4);
    llist.push(15);
    llist.push(10);

    /*Create loop for testing */
    llist.head.next.next.next = llist.head;

    if (detectLoop(llist.head))
        System.out.println("Loop found");
    else
        System.out.println("No Loop");

}
}

// This code is contributed by Arnav Kr. Mandal.
```

**Python3**

```
# Python program to detect loop
# in the linked list

# Node class
class Node:

    # Constructor to initialize
    # the node object
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:

    # Function to initialize head
    def __init__(self):
        self.head = None

    # Function to insert a new
    # node at the beginning
    def push(self, new_data):
        new_node = Node(new_data)
        new_node.next = self.head
        self.head = new_node

    # Utility function to print
    # the linked LinkedList
    def printList(self):
        temp = self.head
        while(temp):
            print (temp.data,end=" ")
            temp = temp.next


def detectLoop(self):
    s = set()
    temp=self.head
    while (temp):

        # If we have already has
        # this node in hashmap it
        # means their is a cycle
        # (Because you we encountering
        # the node second time).
        if (temp in s):
            return True

        # If we are seeing the node for
```

```
# the first time, insert it in hash
s.add(temp)

temp = temp.next

return False

# Driver program for testing
llist = LinkedList()
llist.push(20)
llist.push(4)
llist.push(15)
llist.push(10)

# Create a loop for testing
llist.head.next.next.next = llist.head;

if( llist.detectLoop()):
    print ("Loop found")
else :
    print ("No Loop ")

# This code is contributed by Gitanjali.
```

Output :

Loop Found

#### **Mark Visited Nodes:**

This solution requires modifications to basic linked list data structure. Have a visited flag with each node. Traverse the linked list and keep marking visited nodes. If you see a visited node again then there is a loop. This solution works in O(n) but requires additional information with each node.

A variation of this solution that doesn't require modification to basic data structure can be implemented using hash. Just store the addresses of visited nodes in a hash and if you see an address that already exists in hash then there is a loop.

#### **Floyd's Cycle-Finding Algorithm:**

This is the fastest method. Traverse linked list using two pointers. Move one pointer by one and other pointer by two. If these pointers meet at same node then there is a loop. If pointers do not meet then linked list doesn't have loop.

#### **Implementation of Floyd's Cycle-Finding Algorithm:**

C/C++

```

// C program to detect loop in a linked list
#include<stdio.h>
#include<stdlib.h>

/* Link list node */
struct Node
{
    int data;
    struct Node* next;
};

void push(struct Node** head_ref, int new_data)
{
    /* allocate node */
    struct Node* new_node =
        (struct Node*) malloc(sizeof(struct Node));

    /* put in the data */
    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;
}

int detectloop(struct Node *list)
{
    struct Node *slow_p = list, *fast_p = list;

    while (slow_p && fast_p && fast_p->next )
    {
        slow_p = slow_p->next;
        fast_p = fast_p->next->next;
        if (slow_p == fast_p)
        {
            printf("Found Loop");
            return 1;
        }
    }
    return 0;
}

/* Drier program to test above function*/
int main()
{
    /* Start with the empty list */

```

```
struct Node* head = NULL;

push(&head, 20);
push(&head, 4);
push(&head, 15);
push(&head, 10);

/* Create a loop for testing */
head->next->next->next->next = head;
detectloop(head);

return 0;
}
```

**Java**

```
// Java program to detect loop in a linked list
class LinkedList
{
    Node head; // head of list

    /* Linked list Node*/
    class Node
    {
        int data;
        Node next;
        Node(int d) {data = d; next = null; }
    }

    /* Inserts a new Node at front of the list. */
    public void push(int new_data)
    {
        /* 1 & 2: Allocate the Node &
           Put in the data*/
        Node new_node = new Node(new_data);

        /* 3. Make next of new Node as head */
        new_node.next = head;

        /* 4. Move the head to point to new Node */
        head = new_node;
    }

    int detectLoop()
    {
        Node slow_p = head, fast_p = head;
        while (slow_p != null && fast_p != null && fast_p.next != null) {
            slow_p = slow_p.next;
            fast_p = fast_p.next.next;
        }
        if (slow_p == fast_p)
            return 1;
        else
            return 0;
    }
}
```

```
        fast_p = fast_p.next.next;
        if (slow_p == fast_p) {
            System.out.println("Found loop");
            return 1;
        }
    }
    return 0;
}

/* Drier program to test above functions */
public static void main(String args[])
{
    LinkedList llist = new LinkedList();

    llist.push(20);
    llist.push(4);
    llist.push(15);
    llist.push(10);

    /*Create loop for testing */
    llist.head.next.next.next = llist.head;

    llist.detectLoop();
}
}
/* This code is contributed by Rajat Mishra. */
```

### Python

```
# Python program to detect loop in the linked list

# Node class
class Node:

    # Constructor to initialize the node object
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:

    # Function to initialize head
    def __init__(self):
        self.head = None

    # Function to insert a new node at the beginning
    def push(self, new_data):
        new_node = Node(new_data)
```

```
new_node.next = self.head
self.head = new_node

# Utility function to print the linked LinkedList
def printList(self):
    temp = self.head
    while(temp):
        print temp.data,
        temp = temp.next


def detectLoop(self):
    slow_p = self.head
    fast_p = self.head
    while(slow_p and fast_p and fast_p.next):
        slow_p = slow_p.next
        fast_p = fast_p.next.next
        if slow_p == fast_p:
            print "Found Loop"
            return

# Driver program for testing
llist = LinkedList()
llist.push(20)
llist.push(4)
llist.push(15)
llist.push(10)

# Create a loop for testing
llist.head.next.next.next = llist.head
llist.detectLoop()

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

Output:

Found loop

**Time Complexity:** O(n)

**Auxiliary Space:** O(1)

**How does above algorithm work?**

Please See : [How does Floyd's slow and fast pointers approach work?](#)

**References:**

[http://en.wikipedia.org/wiki/Cycle\\_detection](http://en.wikipedia.org/wiki/Cycle_detection)

[http://ostermiller.org/find\\_loop\\_singly\\_linked\\_list.html](http://ostermiller.org/find_loop_singly_linked_list.html)

**Improved By :** [PrasadPawar, ashwinvivek](#)

## Source

<https://www.geeksforgeeks.org/detect-loop-in-a-linked-list/>

## Chapter 67

# Double elements and append zeros in linked list

Double elements and append zeros in linked list - GeeksforGeeks

Given a linked list with some two adjacent repeating nodes before a zero, task is to double the first and make next 0. After this, append all the zeros to tail.

**Prerequisite:** [Basics of implementation of Singly Linked List](#)

Examples :

```
Input : 4 -> 4 -> 0 -> 2 -> 3 -> 4 ->
        3 -> 3 -> 0 -> 4 ->
Output : 8-> 2-> 3-> 4-> 6-> 4-> 0->
        0-> 0-> 0->
```

Explanation :

First, after doubling the first element and making second element 0 before all zeros.

8 -> 0 -> 0 -> 2 -> 3 -> 4 -> 6 -> 0  
-> 0 -> 4 ->

Next :

8 -> 6 -> 5 -> 6 -> 0 -> 0 -> 0 ->  
0 -> 0 -> 0 -> 0 ->

```
Input : 0 -> 4 -> 4 -> 0 -> 3 -> 3 -> 0
        -> 5 -> 0 -> 0 -> 6 ->
Output : 8 -> 6 -> 5 -> 6 -> 0 -> 0 -> 0 ->
        -> 0 -> 0 -> 0 -> 0 ->
```

Traverse through the linked list, and wherever there are two adjacent same data of nodes

before a 0 (e.g. 4 -> 4 -> 0), then, double first element and make another as 0 (e.g. 8 -> 0 -> 0 ->). Finally, traverse the linked list and linearly point all the zeros to tail.

```
// Java code to modify linked list
import java.util.*;

// Linked List Node
class Node
{
    int data;
    Node next;

    // Constructor
    public Node(int data)
    {
        this.data = data;
        next = null;
    }
}

// Class to perform operations
// on linked list
class GfG
{
    // Recursive function to double the one of two
    // elements and make next one as 0,
    // which are equal before 0
    public static void changeTwoBefore0(Node head)
    {
        // there should be atleast three elements
        // to perform required operation
        if (head == null || head.next == null ||
            head.next.next == null)
            return;

        // when two continuous elements
        // are same
        if ((head.data == head.next.data) &&
            (head.next.next.data == 0))
        {

            int temp = head.data;
            head.data = 2*temp;
            head.next.data = 0;

            if (head.next.next.next != null)
                head = head.next.next.next;
        }
    }
}
```

```
        else
            return;
    }
    else
        head = head.next;

    // recursive call to changeTwoBefore0
    // for next element
    changeTwoBefore0(head);
}

// function to append zeros at tail
public static Node appendZero(Node head)
{
    if (head == null || head.next == null)
        return head;

    // Find tail node
    Node tail = head;
    while (tail.next != null)
        tail = tail.next;
    Node origTail = tail;

    // Case when starting nodes have 0 values
    // we need to change head in this case.
    Node curr = head;
    while (curr.next != null && curr.data == 0)
    {
        tail.next = curr;
        tail = curr;
        curr = curr.next;
    }
    head = curr;

    // Now moving other 0s to end
    Node prev = curr;
    curr = curr.next;

    // We check until original tail
    while (curr != origTail)
    {
        // If current data is 0, append
        // after tail and update tail.
        if (curr.data == 0)
        {
            tail.next = curr;
            tail = curr;
            prev.next = curr.next;
        }
    }
}
```

```
        }
        else
            prev = curr;

        // We always move current
        curr = curr.next;
    }

    // Finally making sure that linked
    // list is null terminated.
    tail.next = null;

    return head;
}

public static Node doubleAndAppend0(Node head)
{
    // Change two same nodes before 0
    changeTwoBefore0(head);

    // Move all 0s to end
    return appendZero(head);
}

// function to display the nodes
public static void display(Node head)
{
    while (head != null)
    {
        System.out.print(head.data + " -> ");
        head = head.next;
    }
}

// Driver code
public static void main(String[] args)
{
    Node head = new Node(4);
    head.next = new Node(4);
    head.next.next = new Node(0);
    head.next.next.next = new Node(2);
    head.next.next.next.next = new Node(3);
    head.next.next.next.next.next = new Node(4);
    head.next.next.next.next.next.next = new Node(3);
    head.next.next.next.next.next.next.next = new Node(3);
    head.next.next.next.next.next.next.next.next = new Node(0);
    head.next.next.next.next.next.next.next.next.next = new Node(4);
```

```
System.out.println("Original linked list :");
display(head);

head = doubleAndAppend0(head);

System.out.println("\nModified linked list :");
display(head);
}
}
```

Output :

```
Original linked list :
4 -> 4 -> 0 -> 2 -> 3 -> 4 -> 3 -> 3 -> 0 -> 4 ->
Modified linked list :
8 -> 2 -> 3 -> 4 -> 6 -> 4 -> 0 -> 0 -> 0 -> 0 ->
```

**Time complexity :**  $O(n)$ , where  $n$  is the number of nodes of linked list.

## Source

<https://www.geeksforgeeks.org/double-elements-and-append-zeros-in-linked-list/>

## Chapter 68

# Doubly Circular Linked List | Set 1 (Introduction and Insertion)

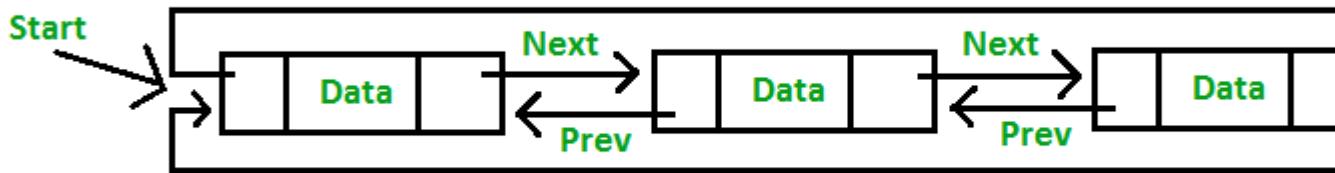
Doubly Circular Linked List | Set 1 (Introduction and Insertion) - GeeksforGeeks

Prerequisite: [Doubly Linked list](#), [Circular Linked List](#)

Circular Doubly Linked List has properties of both doubly linked list and circular linked list in which two consecutive elements are linked or connected by previous and next pointer and the last node points to first node by next pointer and also the first node points to last node by previous pointer.

Following is representation of a Circular doubly linked list node in C/C++:

```
// Structure of the node
struct node
{
    int data;
    struct node *next; // Pointer to next node
    struct node *prev; // Pointer to previous node
};
```

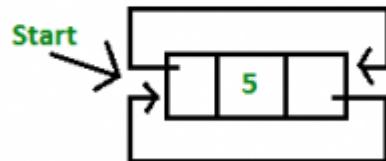


### Insertion in Circular Doubly Linked List

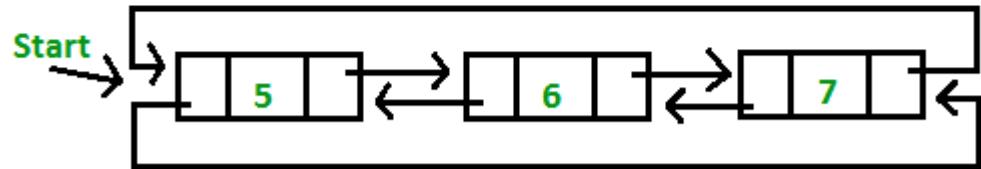
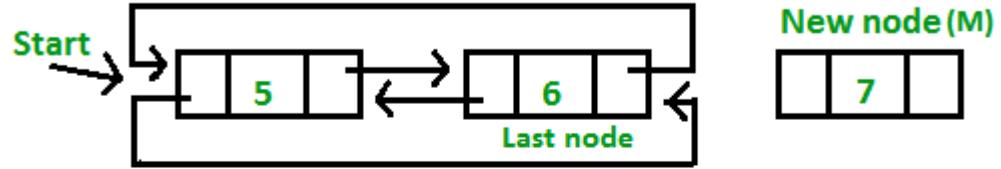
#### 1. Insertion at the end of list or in an empty list

- **Empty List (start = NULL):** A node(Say N) is inserted with data = 5, so previous pointer of N points to N and next pointer of N also points to N. But now start pointer points to the first node the list.

**Start** → **NULL**



- **List initially contain some nodes, start points to first node of the List:** A node(Say M) is inserted with data = 7, so previous pointer of M points to last node, next pointer of M points to first node and last node's next pointer points to this M node and first node's previous pointer points to this M node.



```

// Function to insert at the end
void insertEnd(struct Node** start, int value)
{
    // If the list is empty, create a single node
    // circular and doubly list
    if (*start == NULL)
    {
        struct Node* new_node = new Node;
        new_node->data = value;
        new_node->next = new_node->prev = new_node;
        *start = new_node;
        return;
    }

    // If list is not empty

    /* Find last node */
    Node *last = (*start)->prev;

    // Create Node dynamically
    struct Node* new_node = new Node;
    new_node->data = value;

    // Start is going to be next of new_node
    new_node->next = *start;

    // Make new node previous of start
    (*start)->prev = new_node;
}

```

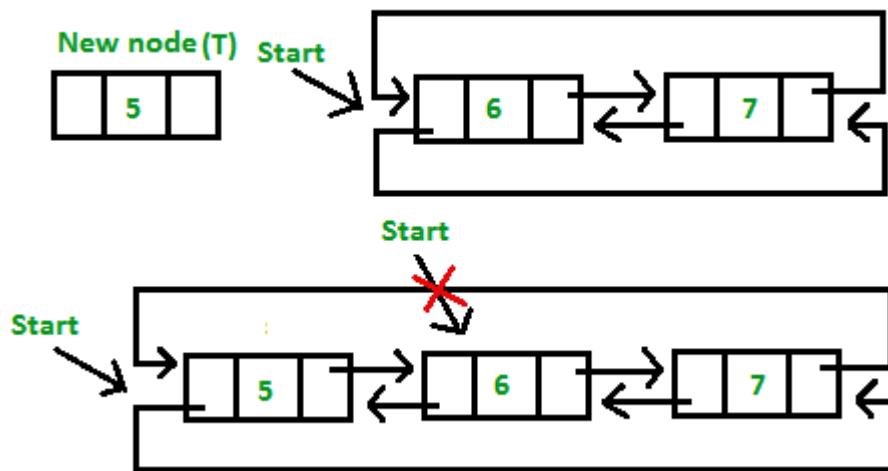
```

    // Make last previous of new node
    new_node->prev = last;

    // Make new node next of old last
    last->next = new_node;
}

```

2. **Insertion at the beginning of the list:** To insert a node at the beginning of the list, create a node(Say T) with data = 5, T next pointer points to first node of the list, T previous pointer points to last node the list, last node's next pointer points to this T node, first node's previous pointer also points this T node and at last don't forget to shift 'Start' pointer to this T node.



```

// Function to insert Node at the beginning
// of the List,
void insertBegin(struct Node** start, int value)
{
    // Pointer points to last Node
    struct Node *last = (*start)->prev;

    struct Node* new_node = new Node;
    new_node->data = value;    // Inserting the data

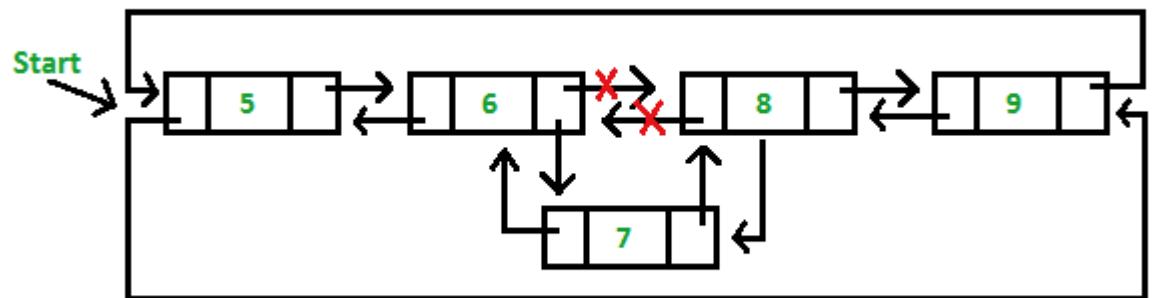
    // setting up previous and next of new node
    new_node->next = *start;
    new_node->prev = last;

    // Update next and previous pointers of start
    // and last.
    last->next = (*start)->prev = new_node;
}

```

```
// Update start pointer
*start = new_node;
}
```

3. **Insertion in between the nodes of the list:** To insert a node in between the list, two data values are required one after which new node will be inserted and another is the data of the new node.



```
// Function to insert node with value as value1.
// The new node is inserted after the node with
// with value2
void insertAfter(struct Node** start, int value1,
                  int value2)
{
    struct Node* new_node = new Node;
    new_node->data = value1; // Inserting the data

    // Find node having value2 and next node of it
    struct Node *temp = *start;
    while (temp->data != value2)
        temp = temp->next;
    struct Node *next = temp->next;

    // insert new_node between temp and next.
    temp->next = new_node;
    new_node->prev = temp;
    new_node->next = next;
    next->prev = new_node;
}
```

Following is a complete program that uses all of the above methods to create a circular doubly linked list.

```
// C++ program to illustrate inserting a Node in
// a Circular Doubly Linked list in beginning, end
// and middle
#include <bits/stdc++.h>
using namespace std;

// Structure of a Node
struct Node
{
    int data;
    struct Node *next;
    struct Node *prev;
};

// Function to insert at the end
void insertEnd(struct Node** start, int value)
{
    // If the list is empty, create a single node
    // circular and doubly list
    if (*start == NULL)
    {
        struct Node* new_node = new Node;
        new_node->data = value;
        new_node->next = new_node->prev = new_node;
        *start = new_node;
        return;
    }

    // If list is not empty

    /* Find last node */
    Node *last = (*start)->prev;

    // Create Node dynamically
    struct Node* new_node = new Node;
    new_node->data = value;

    // Start is going to be next of new_node
    new_node->next = *start;

    // Make new node previous of start
    (*start)->prev = new_node;

    // Make last previous of new node
    new_node->prev = last;

    // Make new node next of old last
    last->next = new_node;
```

```
}

// Function to insert Node at the beginning
// of the List,
void insertBegin(struct Node** start, int value)
{
    // Pointer points to last Node
    struct Node *last = (*start)->prev;

    struct Node* new_node = new Node;
    new_node->data = value;    // Inserting the data

    // setting up previous and next of new node
    new_node->next = *start;
    new_node->prev = last;

    // Update next and previous pointers of start
    // and last.
    last->next = (*start)->prev = new_node;

    // Update start pointer
    *start = new_node;
}

// Function to insert node with value as value1.
// The new node is inserted after the node with
// with value2
void insertAfter(struct Node** start, int value1,
                int value2)
{
    struct Node* new_node = new Node;
    new_node->data = value1; // Inserting the data

    // Find node having value2 and next node of it
    struct Node *temp = *start;
    while (temp->data != value2)
        temp = temp->next;
    struct Node *next = temp->next;

    // insert new_node between temp and next.
    temp->next = new_node;
    new_node->prev = temp;
    new_node->next = next;
    next->prev = new_node;
}

void display(struct Node* start)
```

```
{  
    struct Node *temp = start;  
  
    printf("\nTraversal in forward direction \n");  
    while (temp->next != start)  
    {  
        printf("%d ", temp->data);  
        temp = temp->next;  
    }  
    printf("%d ", temp->data);  
  
    printf("\nTraversal in reverse direction \n");  
    Node *last = start->prev;  
    temp = last;  
    while (temp->prev != last)  
    {  
        printf("%d ", temp->data);  
        temp = temp->prev;  
    }  
    printf("%d ", temp->data);  
}  
  
/* Driver program to test above functions*/  
int main()  
{  
    /* Start with the empty list */  
    struct Node* start = NULL;  
  
    // Insert 5. So linked list becomes 5->NULL  
    insertEnd(&start, 5);  
  
    // Insert 4 at the beginning. So linked  
    // list becomes 4->5  
    insertBegin(&start, 4);  
  
    // Insert 7 at the end. So linked list  
    // becomes 4->5->7  
    insertEnd(&start, 7);  
  
    // Insert 8 at the end. So linked list  
    // becomes 4->5->7->8  
    insertEnd(&start, 8);  
  
    // Insert 6, after 5. So linked list  
    // becomes 4->5->6->7->8  
    insertAfter(&start, 6, 5);  
  
    printf("Created circular doubly linked list is: ");
```

```
    display(start);  
  
    return 0;  
}
```

Output:

```
Created circular doubly linked list is:  
Traversal in forward direction  
4 5 6 7 8  
Traversal in reverse direction  
8 7 6 5 4
```

Following are advantages and disadvantages of circular doubly linked list:

**Advantages:**

- List can be traversed from both the directions i.e. from head to tail or from tail to head.
- Jumping from head to tail or from tail to head is done in constant time O(1).
- Circular Doubly Linked Lists are used for implementation of advanced data structures like [Fibonacci Heap](#).

**Disadvantages**

- It takes slightly extra memory in each node to accommodate previous pointer.
- Lots of pointers involved while implementing or doing operations on a list. So, pointers should be handled carefully otherwise data of the list may get lost.

**Applications of Circular doubly linked list**

- Managing songs playlist in media player applications.
- Managing shopping cart in online shopping.

**Source**

<https://www.geeksforgeeks.org/doubly-circular-linked-list-set-1-introduction-and-insertion/>

## Chapter 69

# Doubly Circular Linked List | Set 2 (Deletion)

Doubly Circular Linked List | Set 2 (Deletion) - GeeksforGeeks

We have discussed [doubly circular linked list introduction and its insertion](#).

Let us formulate the problem statement to understand the deletion process. Given a '*key*', delete the first occurrence of this key in circular doubly linked list.

### Algorithm

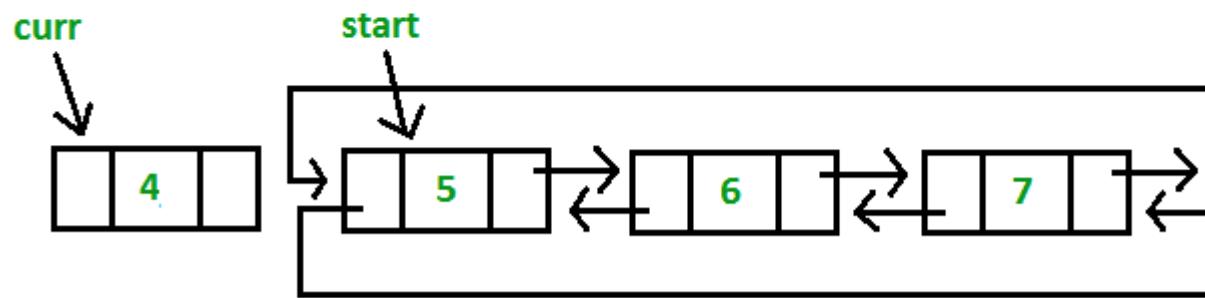
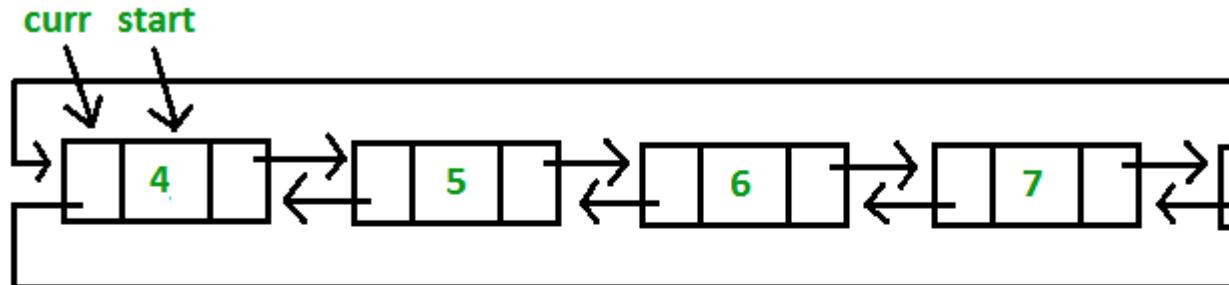
**Case 1:** Empty List(*start* = NULL)

- If the list is empty, simply return.

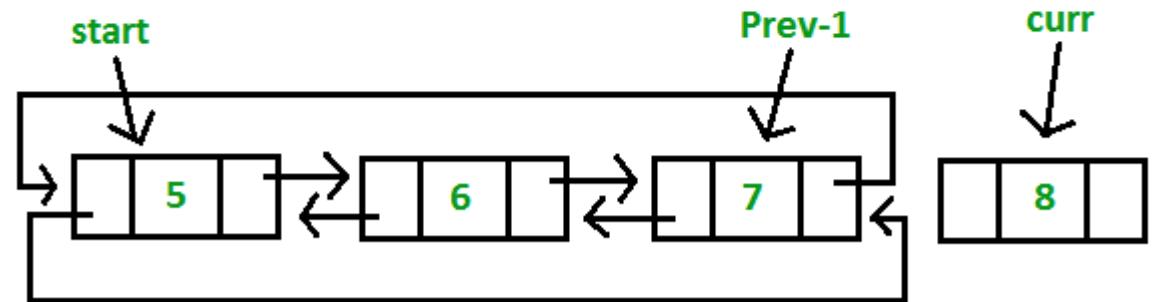
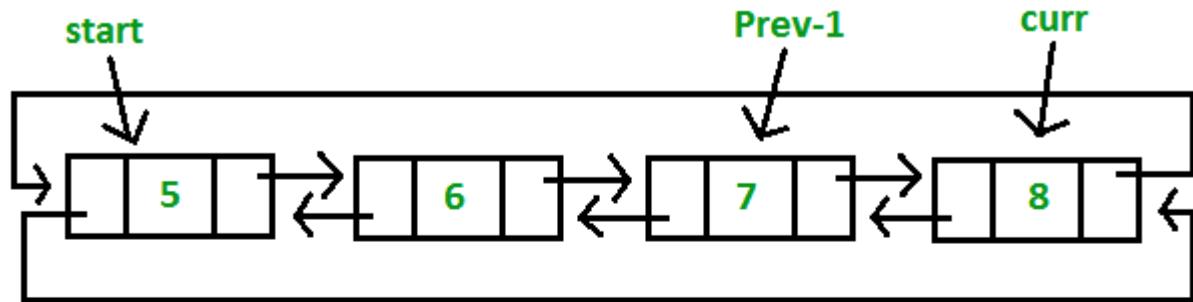
**Case 2:** List initially contain some nodes, start points to first node of the List

1. If the list is not empty then we define two pointers **curr** and **prev\_1** and initialize the pointer **curr** points to first node of the list and **prev\_1** = NULL.
2. Traverse the list using *curr* pointer to find the node to be deleted and before moving *curr* to next node, every time set *prev\_1* = *curr*.
3. If the node is found, check if it is the only node in the list. If yes, set *start* = NULL and free the node pointing by *curr*.
4. If the list has more than one node, check if it is the first node of the list. Condition to check this is (*curr* == *start*). If yes, then move *prev\_1* to the last node(*prev\_1* = *start* -> *prev*). After *prev\_1* reaches the last node, set *start* = *start* -> next and *prev\_1* -> *next* = *start* and *start* -> *prev* = *prev\_1*. Free the node pointing by *curr*.
5. If *curr* is not first node, we check if it is the last node in the list. Condition to check this is (*curr* -> *next* == *start*). If yes, set *prev\_1* -> *next* = *start* and *start* -> *prev* = *prev\_1*. Free the node pointing by *curr*.
6. If the node to be deleted is neither the first node nor the last node, declare one more pointer **temp** and initialize the pointer **temp** points to next of *curr* pointer (*temp* = *curr*->*next*). Now set, *prev\_1* -> *next* = *temp* and *temp* ->*prev* = *prev\_1*. Free the node pointing by *curr*.

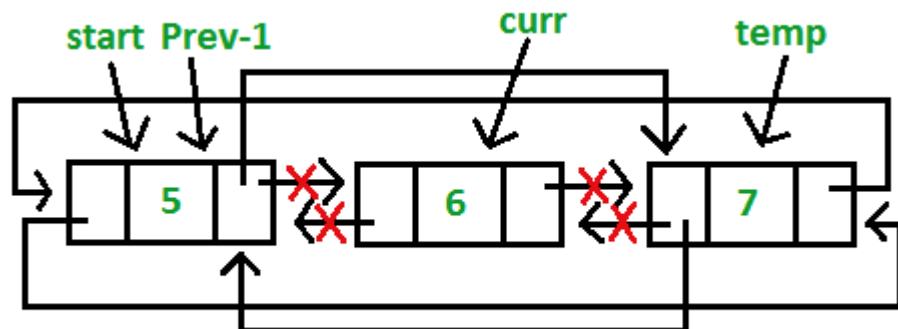
- If the given *key*(Say 4) matches with first node of the list(Step 4):



- If the given *key*(Say 8) matches with last node of the list(Step 5):



- If the given *key*(Say 6) matches with middle node of the list(Step 6):



```
// C++ program to delete a given key from
// circular doubly linked list.
#include<bits/stdc++.h>
using namespace std;

// Structure of a Node
```

```
struct Node
{
    int data;
    struct Node *next;
    struct Node *prev;
};

// Function to insert node in the list
void insert(struct Node** start, int value)
{
    // If the list is empty, create a single node
    // circular and doubly list
    if (*start == NULL)
    {
        struct Node* new_node = new Node;
        new_node->data = value;
        new_node->next = new_node->prev = new_node;
        *start = new_node;
        return;
    }

    // If list is not empty

    /* Find last node */
    Node *last = (*start)->prev;

    // Create Node dynamically
    struct Node* new_node = new Node;
    new_node->data = value;

    // Start is going to be next of new_node
    new_node->next = *start;

    // Make new node previous of start
    (*start)->prev = new_node;

    // Make last previous of new node
    new_node->prev = last;

    // Make new node next of old last
    last->next = new_node;
}

// Function to delete a given node from the list
void deleteNode(struct Node **start, int key)
{
    // If list is empty
    if (*start == NULL)
```

```
return;

// Find the required node
// Declare two pointers and initialize them
struct Node *curr = *start, *prev_1 = NULL;
while (curr -> data != key)
{
    // If node is not present in the list
    if (curr->next == *start)
    {
        printf("\nList doesn't have node with value = %d", key);
        return;
    }

    prev_1 = curr;
    curr = curr -> next;
}

// Check if node is the only node in list
if (curr -> next == *start && prev_1 == NULL)
{
    (*start) = NULL;
    free(curr);
    return;
}

// If list has more than one node,
// check if it is the first node
if (curr == *start)
{
    // Move prev_1 to last node
    prev_1 = (*start) -> prev;

    // Move start ahead
    *start = (*start) -> next;

    // Adjust the pointers of prev_1 and start node
    prev_1 -> next = *start;
    (*start) -> prev = prev_1;
    free(curr);
}

// check if it is the last node
else if (curr->next == *start)
{
    // Adjust the pointers of prev_1 and start node
    prev_1 -> next = *start;
    (*start) -> prev = prev_1;
```

```
        free(curr);
    }
else
{
    // create new pointer, points to next of curr node
    struct Node *temp = curr -> next;

    // Adjust the pointers of prev_1 and temp node
    prev_1 -> next = temp;
    temp -> prev = prev_1;
    free(curr);
}
}

// Function to display list elements
void display(struct Node* start)
{
    struct Node *temp = start;

    while (temp->next != start)
    {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("%d ", temp->data);
}

// Driver program to test above functions
int main()
{
    // Start with the empty list
    struct Node *start = NULL;

    // Created linked list will be 4->5->6->7->8
    insert(&start, 4);
    insert(&start, 5);
    insert(&start, 6);
    insert(&start, 7);
    insert(&start, 8);

    printf("List Before Deletion: ");
    display(start);

    // Delete the node which is not present in list
    deleteNode(&start, 9);
    printf("\nList After Deletion: ");
    display(start);
```

```
// Delete the first node
deleteNode(&start, 4);
printf("\nList After Deleting %d: ", 4);
display(start);

// Delete the last node
deleteNode(&start, 8);
printf("\nList After Deleting %d: ", 8);
display(start);

// Delete the middle node
deleteNode(&start, 6);
printf("\nList After Deleting %d: ", 6);
display(start);

return 0;
}
```

Output:

```
List Before Deletion: 4 5 6 7 8
List doesn't have node with value = 9
List After Deletion: 4 5 6 7 8
List After Deleting 4: 5 6 7 8
List After Deleting 8: 5 6 7
List After Deleting 6: 5 7
```

## Source

<https://www.geeksforgeeks.org/doubly-circular-linked-list-set-2-deletion/>

# Chapter 70

## Doubly Linked List | Set 1 (Introduction and Insertion)

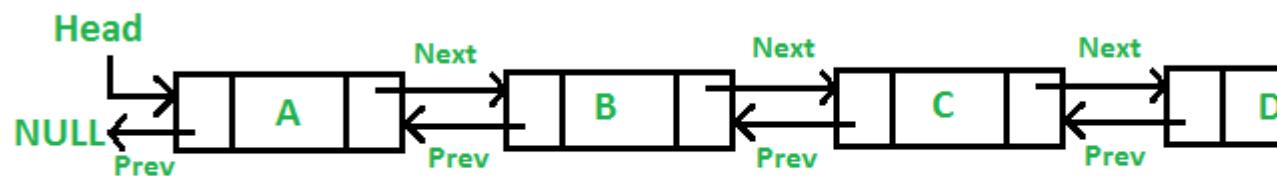
Doubly Linked List | Set 1 (Introduction and Insertion) - GeeksforGeeks

We strongly recommend to refer following post as a prerequisite of this post.

[Linked List Introduction](#)

[Inserting a node in Singly Linked List](#)

A Doubly Linked List (DLL) contains an extra pointer, typically called *previous pointer*, together with next pointer and data which are there in singly linked list.



Following is representation of a DLL node in C language.

C

```
/* Node of a doubly linked list */
struct Node {
    int data;
    struct Node* next; // Pointer to next node in DLL
    struct Node* prev; // Pointer to previous node in DLL
};
```

### Java

```
// Class for Doubly Linked List
public class DLL {
    Node head; // head of list

    /* Doubly Linked list Node*/
    class Node {
        int data;
        Node prev;
        Node next;

        // Constructor to create a new node
        // next and prev is by default initialized as null
        Node(int d) { data = d; }
    }
}
```

### Python3

```
# Node of a doubly linked list
class Node:
    def __init__(self, next=None, prev=None, data=None):
        self.next = next # reference to next node in DLL
        self.prev = prev # reference to previous node in DLL
        self.data = data
```

Following are advantages/disadvantages of doubly linked list over singly linked list.

#### Advantages over singly linked list

- 1) A DLL can be traversed in both forward and backward direction.
- 2) The delete operation in DLL is more efficient if pointer to the node to be deleted is given.
- 3) We can quickly insert a new node before a given node.

In singly linked list, to delete a node, pointer to the previous node is needed. To get this previous node, sometimes the list is traversed. In DLL, we can get the previous node using previous pointer.

#### Disadvantages over singly linked list

- 1) Every node of DLL Require extra space for an previous pointer. It is possible to implement DLL with single pointer though (See [this](#) and [this](#)).
- 2) All operations require an extra pointer previous to be maintained. For example, in insertion, we need to modify previous pointers together with next pointers. For example in following functions for insertions at different positions, we need 1 or 2 extra steps to set previous pointer.

#### Insertion

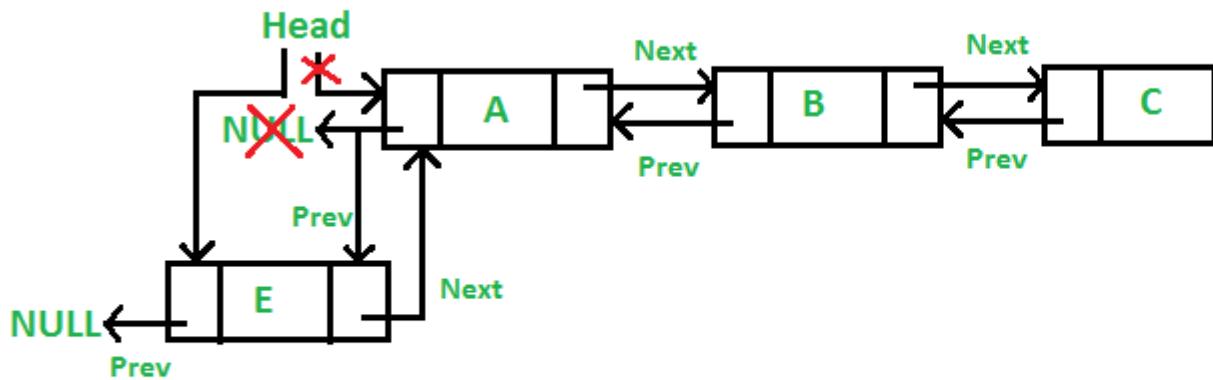
A node can be added in four ways

- 1) At the front of the DLL
- 2) After a given node.

- 3) At the end of the DLL
- 4) Before a given node.

**1) Add a node at the front: (A 5 steps process)**

The new node is always added before the head of the given Linked List. And newly added node becomes the new head of DLL. For example if the given Linked List is 10152025 and we add an item 5 at the front, then the Linked List becomes 510152025. Let us call the function that adds at the front of the list is push(). The push() must receive a pointer to the head pointer, because push must change the head pointer to point to the new node (See this)



Following are the 5 steps to add node at the front.

C

```
/* Given a reference (pointer to pointer) to the head of a list
   and an int, inserts a new node on the front of the list. */
void push(struct Node** head_ref, int new_data)
{
    /* 1. allocate node */
    struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));

    /* 2. put in the data */
    new_node->data = new_data;

    /* 3. Make next of new node as head and previous as NULL */
    new_node->next = (*head_ref);
```

```
new_node->prev = NULL;

/* 4. change prev of head node to new node */
if ((*head_ref) != NULL)
    (*head_ref)->prev = new_node;

/* 5. move the head to point to the new node */
(*head_ref) = new_node;
}
```

### Java

```
// Adding a node at the front of the list
public void push(int new_data)
{
    /* 1. allocate node
     * 2. put in the data */
    Node new_Node = new Node(new_data);

    /* 3. Make next of new node as head and previous as NULL */
    new_Node.next = head;
    new_Node.prev = null;

    /* 4. change prev of head node to new node */
    if (head != null)
        head.prev = new_Node;

    /* 5. move the head to point to the new node */
    head = new_Node;
}
```

### Python3

```
# Adding a node at the front of the list
def push(self, new_data):

    # 1 & 2: Allocate the Node & Put in the data
    new_node = Node(data = new_data)

    # 3. Make next of new node as head and previous as NULL
    new_node.next = self.head
    new_node.prev = None

    # 4. change prev of head node to new node
    if self.head is not None:
        self.head.prev = new_node
```

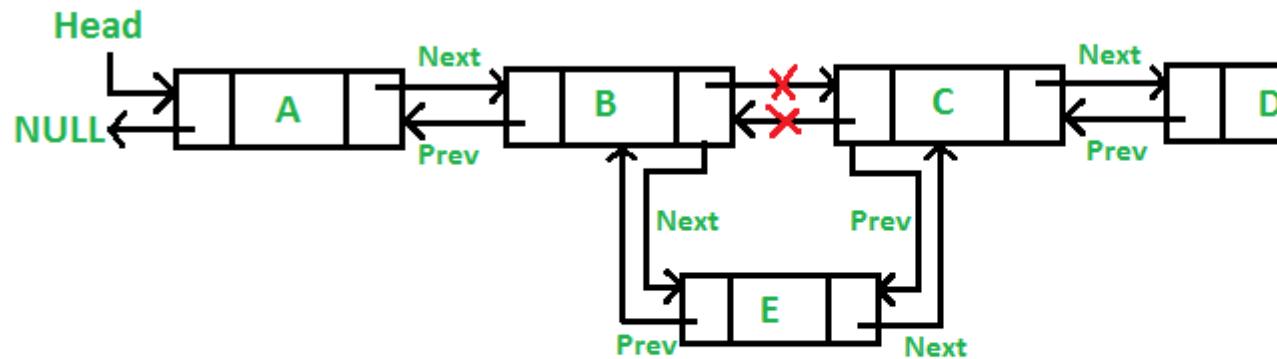
```
# 5. move the head to point to the new node
self.head = new_node

# This code is contributed by jatinreaper
```

Four steps of the above five steps are same as [the 4 steps used for inserting at the front in singly linked list](#). The only extra step is to change previous of head.

## 2) Add a node after a given node.: (A 7 steps process)

We are given pointer to a node as prev\_node, and the new node is inserted after the given node.



C

```
/* Given a node as prev_node, insert a new node after the given node */
void insertAfter(struct Node* prev_node, int new_data)
{
    /*1. check if the given prev_node is NULL */
    if (prev_node == NULL) {
        printf("the given previous node cannot be NULL");
        return;
    }

    /* 2. allocate new node */
    struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));

    /* 3. put in the data */
    new_node->data = new_data;
```

```
/* 4. Make next of new node as next of prev_node */
new_node->next = prev_node->next;

/* 5. Make the next of prev_node as new_node */
prev_node->next = new_node;

/* 6. Make prev_node as previous of new_node */
new_node->prev = prev_node;

/* 7. Change previous of new_node's next node */
if (new_node->next != NULL)
    new_node->next->prev = new_node;
}
```

### Java

```
/* Given a node as prev_node, insert a new node after the given node */
public void InsertAfter(Node prev_Node, int new_data)
{

    /*1. check if the given prev_node is NULL */
    if (prev_Node == null) {
        System.out.println("The given previous node cannot be NULL ");
        return;
    }

    /* 2. allocate node
     * 3. put in the data */
    Node new_node = new Node(new_data);

    /* 4. Make next of new node as next of prev_node */
    new_node.next = prev_Node.next;

    /* 5. Make the next of prev_node as new_node */
    prev_Node.next = new_node;

    /* 6. Make prev_node as previous of new_node */
    new_node.prev = prev_Node;

    /* 7. Change previous of new_node's next node */
    if (new_node.next != null)
        new_node.next.prev = new_node;
}
```

### Python3

```
# Given a node as prev_node, insert
```

```
# a new node after the given node

def insertAfter(self, prev_node, new_data):

    # 1. check if the given prev_node is NULL
    if prev_node is None:
        print("This node doesn't exist in DLL")
        return

    #2. allocate node & 3. put in the data
    new_node = Node(data = new_data)

    # 4. Make next of new node as next of prev_node
    new_node.next = prev_node.next

    # 5. Make the next of prev_node as new_node
    prev_node.next = new_node

    # 6. Make prev_node as previous of new_node
    new_node.prev = prev_node

    # 7. Change previous of new_node's next node */
    if new_node.next is not None:
        new_node.next.prev = new_node

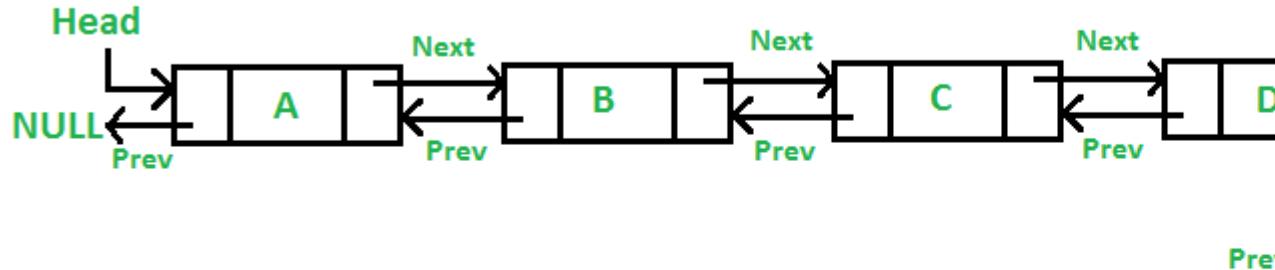
# This code is contributed by jatinreaper
```

Five of the above steps step process are same as [the 5 steps used for inserting after a given node in singly linked list](#). The two extra steps are needed to change previous pointer of new node and previous pointer of new node's next node.

### 3) Add a node at the end: (7 steps process)

The new node is always added after the last node of the given Linked List. For example if the given DLL is 510152025 and we add an item 30 at the end, then the DLL becomes 51015202530.

Since a Linked List is typically represented by the head of it, we have to traverse the list till end and then change the next of last node to new node.



Following are the 7 steps to add node at the end.

C

```

/* Given a reference (pointer to pointer) to the head
   of a DLL and an int, appends a new node at the end */
void append(struct Node** head_ref, int new_data)
{
    /* 1. allocate node */
    struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));

    struct Node* last = *head_ref; /* used in step 5*/

    /* 2. put in the data */
    new_node->data = new_data;

    /* 3. This new node is going to be the last node, so
       make next of it as NULL*/
    new_node->next = NULL;

    /* 4. If the Linked List is empty, then make the new
       node as head */
    if (*head_ref == NULL) {
        new_node->prev = NULL;
        *head_ref = new_node;
        return;
    }

    /* 5. Else traverse till the last node */
    while (last->next != NULL)
        last = last->next;
}

```

```
/* 6. Change the next of last node */
last->next = new_node;

/* 7. Make last node as previous of new node */
new_node->prev = last;

return;
}
```

**Java**

```
// Add a node at the end of the list
void append(int new_data)
{
    /* 1. allocate node
     * 2. put in the data */
    Node new_node = new Node(new_data);

    Node last = head; /* used in step 5*/

    /* 3. This new node is going to be the last node, so
     * make next of it as NULL*/
    new_node.next = null;

    /* 4. If the Linked List is empty, then make the new
     * node as head */
    if (head == null) {
        new_node.prev = null;
        head = new_node;
        return;
    }

    /* 5. Else traverse till the last node */
    while (last.next != null)
        last = last.next;

    /* 6. Change the next of last node */
    last.next = new_node;

    /* 7. Make last node as previous of new node */
    new_node.prev = last;
}
```

**Python3**

```
# Add a node at the end of the DLL
```

```

def append(self, new_data):

    # 1. allocate node 2. put in the data
    new_node = Node(data = new_data)
    last = self.head

    # 3. This new node is going to be the
    # last node, so make next of it as NULL
    new_node.next = None

    # 4. If the Linked List is empty, then
    # make the new node as head
    if self.head is None:
        new_node.prev = None
        self.head = new_node
        return

    # 5. Else traverse till the last node
    while (last.next is not None):
        last = last.next

    # 6. Change the next of last node
    last.next = new_node
    # 7. Make last node as previous of new node */
    new_node.prev = last

# This code is contributed by jatinreaper

```

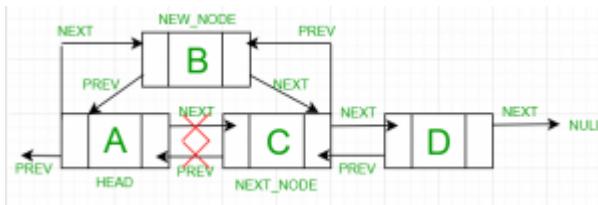
Six of the above 7 steps are same as [the 6 steps used for inserting after a given node in singly linked list](#). The one extra step is needed to change previous pointer of new node.

#### 4) Add a node before a given node:

##### Steps

Let the pointer to this given node be `next_node` and the data of the new node to be added as `new_data`.

1. Check if the `next_node` is `NULL` or not. If it's `NULL`, return from the function because any new node can not be added before a `NULL`
2. Allocate memory for the new node, let it be called `new_node`
3. Set `new_node->data = new_data`
4. Set the previous pointer of this `new_node` as the previous node of the `next_node`, `new_node->prev = next_node->prev`
5. Set the previous pointer of the `next_node` as the `new_node`, `next_node->prev = new_node`
6. Set the next pointer of this `new_node` as the `next_node`, `new_node->next = next_node;`
7. If the previous node of the `new_node` is not `NULL`, then set the next pointer of this previous node as `new_node`, `new_node->prev->next = new_node`



Below is the implementation of the above approach:

C++

```

// A complete working C program to demonstrate all
// insertion before a given node
#include <stdio.h>
#include <stdlib.h>

// A linked list node
struct Node {
    int data;
    struct Node* next;
    struct Node* prev;
};

/* Given a reference (pointer to pointer) to the head of a list
and an int, inserts a new node on the front of the list. */
void push(struct Node** head_ref, int new_data)
{
    struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));

    new_node->data = new_data;

    new_node->next = (*head_ref);
    new_node->prev = NULL;

    if ((*head_ref) != NULL)
        (*head_ref)->prev = new_node;

    (*head_ref) = new_node;
}

/* Given a node as next_node, insert a new node before the given node */
void insertBefore(struct Node* next_node, int new_data)
{
    /*1. check if the given new_node is NULL */
    if (next_node == NULL) {
        printf("the given next node cannot be NULL");
        return;
    }
}

```

```
/* 2. allocate new node */
struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));

/* 3. put in the data */
new_node->data = new_data;

/* 4. Make prev of new node as prev of next_node */
new_node->prev = next_node->prev;

/* 5. Make the prev of next_node as new_node */
next_node->prev = new_node;

/* 6. Make next_node as next of new_node */
new_node->next = next_node;

/* 7. Change next of new_node's previous node */
if (new_node->prev != NULL)
    new_node->prev->next = new_node;
}

// This function prints contents of linked list starting from the given node
void printList(struct Node* node)
{
    struct Node* last;
    printf("\nTraversal in forward direction \n");
    while (node != NULL) {
        printf(" %d ", node->data);
        last = node;
        node = node->next;
    }

    printf("\nTraversal in reverse direction \n");
    while (last != NULL) {
        printf(" %d ", last->data);
        last = last->prev;
    }
}

/* Driver program to test above functions*/
int main()
{
    /* Start with the empty list */
    struct Node* head = NULL;
    push(&head, 7);

    push(&head, 1);
```

```
push(&head, 4);

// Insert 8, before 1. So linked list becomes 4->8->1->7->NULL
insertBefore(head->next, 8);

printf("Created DLL is: ");
printList(head);

getchar();
return 0;
}
```

Output:

```
Created DLL is:
Traversal in forward direction
4 8 1 7
Traversal in reverse direction
7 1 8 4
```

**A complete working program to test above functions.**

Following is complete C program to test above functions.

**C**

```
// A complete working C program to demonstrate all insertion methods
#include <stdio.h>
#include <stdlib.h>

// A linked list node
struct Node {
    int data;
    struct Node* next;
    struct Node* prev;
};

/* Given a reference (pointer to pointer) to the head of a list
   and an int, inserts a new node on the front of the list. */
void push(struct Node** head_ref, int new_data)
{
    /* 1. allocate node */
    struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));

    /* 2. put in the data */
    new_node->data = new_data;
```

```

/* 3. Make next of new node as head and previous as NULL */
new_node->next = (*head_ref);
new_node->prev = NULL;

/* 4. change prev of head node to new node */
if ((*head_ref) != NULL)
    (*head_ref)->prev = new_node;

/* 5. move the head to point to the new node */
(*head_ref) = new_node;
}

/* Given a node as prev_node, insert a new node after the given node */
void insertAfter(struct Node* prev_node, int new_data)
{
    /*1. check if the given prev_node is NULL */
    if (prev_node == NULL) {
        printf("the given previous node cannot be NULL");
        return;
    }

    /* 2. allocate new node */
    struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));

    /* 3. put in the data */
    new_node->data = new_data;

    /* 4. Make next of new node as next of prev_node */
    new_node->next = prev_node->next;

    /* 5. Make the next of prev_node as new_node */
    prev_node->next = new_node;

    /* 6. Make prev_node as previous of new_node */
    new_node->prev = prev_node;

    /* 7. Change previous of new_node's next node */
    if (new_node->next != NULL)
        new_node->next->prev = new_node;
}

/* Given a reference (pointer to pointer) to the head
   of a DLL and an int, appends a new node at the end */
void append(struct Node** head_ref, int new_data)
{
    /* 1. allocate node */
    struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));

```

```

struct Node* last = *head_ref; /* used in step 5*/

/* 2. put in the data */
new_node->data = new_data;

/* 3. This new node is going to be the last node, so
   make next of it as NULL*/
new_node->next = NULL;

/* 4. If the Linked List is empty, then make the new
   node as head */
if (*head_ref == NULL) {
    new_node->prev = NULL;
    *head_ref = new_node;
    return;
}

/* 5. Else traverse till the last node */
while (last->next != NULL)
    last = last->next;

/* 6. Change the next of last node */
last->next = new_node;

/* 7. Make last node as previous of new node */
new_node->prev = last;

return;
}

// This function prints contents of linked list starting from the given node
void printList(struct Node* node)
{
    struct Node* last;
    printf("\nTraversal in forward direction \n");
    while (node != NULL) {
        printf(" %d ", node->data);
        last = node;
        node = node->next;
    }

    printf("\nTraversal in reverse direction \n");
    while (last != NULL) {
        printf(" %d ", last->data);
        last = last->prev;
    }
}

```

```
/* Drier program to test above functions*/
int main()
{
    /* Start with the empty list */
    struct Node* head = NULL;

    // Insert 6. So linked list becomes 6->NULL
    append(&head, 6);

    // Insert 7 at the beginning. So linked list becomes 7->6->NULL
    push(&head, 7);

    // Insert 1 at the beginning. So linked list becomes 1->7->6->NULL
    push(&head, 1);

    // Insert 4 at the end. So linked list becomes 1->7->6->4->NULL
    append(&head, 4);

    // Insert 8, after 7. So linked list becomes 1->7->8->6->4->NULL
    insertAfter(head->next, 8);

    printf("Created DLL is: ");
    printList(head);

    getchar();
    return 0;
}
```

### Java

```
// A complete working Java program to demonstrate all

// Class for Doubly Linked List
public class DLL {
    Node head; // head of list

    /* Doubly Linked list Node*/
    class Node {
        int data;
        Node prev;
        Node next;

        // Constructor to create a new node
        // next and prev is by default initialized as null
        Node(int d) { data = d; }
    }

    // Adding a node at the front of the list
```

```

public void push(int new_data)
{
    /* 1. allocate node
     * 2. put in the data */
    Node new_Node = new Node(new_data);

    /* 3. Make next of new node as head and previous as NULL */
    new_Node.next = head;
    new_Node.prev = null;

    /* 4. change prev of head node to new node */
    if (head != null)
        head.prev = new_Node;

    /* 5. move the head to point to the new node */
    head = new_Node;
}

/* Given a node as prev_node, insert a new node after the given node */
public void InsertAfter(Node prev_Node, int new_data)
{
    /*1. check if the given prev_node is NULL */
    if (prev_Node == null) {
        System.out.println("The given previous node cannot be NULL ");
        return;
    }

    /* 2. allocate node
     * 3. put in the data */
    Node new_node = new Node(new_data);

    /* 4. Make next of new node as next of prev_node */
    new_node.next = prev_Node.next;

    /* 5. Make the next of prev_node as new_node */
    prev_Node.next = new_node;

    /* 6. Make prev_node as previous of new_node */
    new_node.prev = prev_Node;

    /* 7. Change previous of new_node's next node */
    if (new_node.next != null)
        new_node.next.prev = new_node;
}

// Add a node at the end of the list
void append(int new_data)

```

```
{
    /* 1. allocate node
     * 2. put in the data */
    Node new_node = new Node(new_data);

    Node last = head; /* used in step 5*/

    /* 3. This new node is going to be the last node, so
     * make next of it as NULL*/
    new_node.next = null;

    /* 4. If the Linked List is empty, then make the new
     * node as head */
    if (head == null) {
        new_node.prev = null;
        head = new_node;
        return;
    }

    /* 5. Else traverse till the last node */
    while (last.next != null)
        last = last.next;

    /* 6. Change the next of last node */
    last.next = new_node;

    /* 7. Make last node as previous of new node */
    new_node.prev = last;
}

// This function prints contents of linked list starting from the given node
public void printlist(Node node)
{
    Node last = null;
    System.out.println("Traversal in forward Direction");
    while (node != null) {
        System.out.print(node.data + " ");
        last = node;
        node = node.next;
    }
    System.out.println();
    System.out.println("Traversal in reverse direction");
    while (last != null) {
        System.out.print(last.data + " ");
        last = last.prev;
    }
}
```

```
/* Drier program to test above functions*/
public static void main(String[] args)
{
    /* Start with the empty list */
    DLL dll = new DLL();

    // Insert 6. So linked list becomes 6->NULL
    dll.append(6);

    // Insert 7 at the beginning. So linked list becomes 7->6->NULL
    dll.push(7);

    // Insert 1 at the beginning. So linked list becomes 1->7->6->NULL
    dll.push(1);

    // Insert 4 at the end. So linked list becomes 1->7->6->4->NULL
    dll.append(4);

    // Insert 8, after 7. So linked list becomes 1->7->8->6->4->NULL
    dll.InsertAfter(dll.head.next, 8);

    System.out.println("Created DLL is: ");
    dll.printlist(dll.head);
}
}

// This code is contributed by Sumit Ghosh
```

### Python

```
# A complete working Python program to demonstrate all
# insertion methods

# A linked list node
class Node:

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

# Class to create a Doubly Linked List
class DoublyLinkedList:

    # Constructor for empty Doubly Linked List
    def __init__(self):
        self.head = None
```

```
# Given a reference to the head of a list and an
# integer, inserts a new node on the front of list
def push(self, new_data):

    # 1. Allocates node
    # 2. Put the data in it
    new_node = Node(new_data)

    # 3. Make next of new node as head and
    # previous as None (already None)
    new_node.next = self.head

    # 4. change prev of head node to new_node
    if self.head is not None:
        self.head.prev = new_node

    # 5. move the head to point to the new node
    self.head = new_node

# Given a node as prev_node, insert a new node after
# the given node
def insertAfter(self, prev_node, new_data):

    # 1. Check if the given prev_node is None
    if prev_node is None:
        print "the given previous node cannot be NULL"
        return

    # 2. allocate new node
    # 3. put in the data
    new_node = Node(new_data)

    # 4. Make net of new node as next of prev node
    new_node.next = prev_node.next

    # 5. Make prev_node as previous of new_node
    prev_node.next = new_node

    # 6. Make prev_node ass previous of new_node
    new_node.prev = prev_node

    # 7. Change previous of new_nodes's next node
    if new_node.next is not None:
        new_node.next.prev = new_node

# Given a reference to the head of DLL and integer,
# appends a new node at the end
```

```
def append(self, new_data):

    # 1. Allocates node
    # 2. Put in the data
    new_node = Node(new_data)

    # 3. This new node is going to be the last node,
    # so make next of it as None
    new_node.next = None

    # 4. If the Linked List is empty, then make the
    # new node as head
    if self.head is None:
        new_node.prev = None
        self.head = new_node
        return

    # 5. Else traverse till the last node
    last = self.head
    while(last.next is not None):
        last = last.next

    # 6. Change the next of last node
    last.next = new_node

    # 7. Make last node as previous of new node
    new_node.prev = last

    return

# This function prints contents of linked list
# starting from the given node
def printList(self, node):

    print "\nTraversal in forward direction"
    while(node is not None):
        print " % d" %(node.data),
        last = node
        node = node.next

    print "\nTraversal in reverse direction"
    while(last is not None):
        print " % d" %(last.data),
        last = last.prev

# Driver program to test above functions

# Start with empty list
```

```
llist = DoublyLinkedList()

# Insert 6. So the list becomes 6->None
llist.append(6)

# Insert 7 at the beginning.
# So linked list becomes 7->6->None
llist.push(7)

# Insert 1 at the beginning.
# So linked list becomes 1->7->6->None
llist.push(1)

# Insert 4 at the end.
# So linked list becomes 1->7->6->4->None
llist.append(4)

# Insert 8, after 7.
# So linked list becomes 1->7->8->6->4->None
llist.insertAfter(llist.head.next, 8)

print "Created DLL is: ",
llist.printList(llist.head)

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

Output:

```
Created DLL is:
Traversal in forward direction
1 7 8 6 4
Traversal in reverse direction
4 6 8 7 1
```

Also see – [Delete a node in double Link List](#)

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

**Improved By :** [jatinreaper](#)

## Source

<https://www.geeksforgeeks.org/doubly-linked-list/>

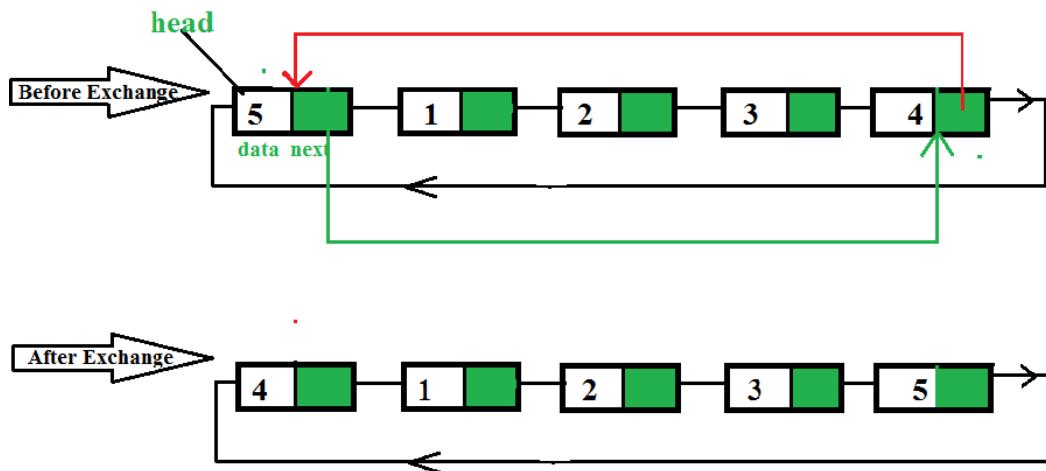
## Chapter 71

# Exchange first and last nodes in Circular Linked List

Exchange first and last nodes in Circular Linked List - GeeksforGeeks

Given a [Circular linked list](#) exchange the first and the last node. The task should be done with only one extra node, you can not declare more than one extra node and also you are not allowed to declare any other temporary variable.

**Note:** Extra node means need of a node to traverse a list.



Examples:

Input : 5 4 3 2 1  
Output : 1 4 3 2 5

Input : 6 1 2 3 4 5 6 7 8 9  
Output : 9 1 2 3 4 5 6 7 8 6

We first find pointer to previous of last node. Let this node be p. Now we change next links so that the last and first nodes are swapped.

```
// CPP program to exchange first and
// last node in circular linked list
#include<bits/stdc++.h>
using namespace std;

struct Node
{
    int data;
    struct Node *next;
};

struct Node *addToEmpty(struct Node *head, int data)
{
    // This function is only for empty list
    if (head != NULL)
        return head;

    // Creating a node dynamically.
    struct Node *temp =
        (struct Node*)malloc(sizeof(struct Node));

    // Assigning the data.
    temp -> data = data;
    head = temp;

    // Creating the link.
    head -> next = head;

    return head;
}

struct Node *addBegin(struct Node *head, int data)
{
    if (head == NULL)
        return addToEmpty(head, data);

    struct Node *temp =
        (struct Node *)malloc(sizeof(struct Node));

    temp -> data = data;
    temp -> next = head -> next;
    head -> next = temp;

    return head;
}
```

```
/* function for traversing the list */
void traverse(struct Node *head)
{
    struct Node *p;

    // If list is empty, return.
    if (head == NULL)
    {
        cout << "List is empty." << endl;
        return;
    }

    // Pointing to first Node of the list.
    p = head;

    // Traversing the list.
    do
    {
        cout << p -> data << " ";
        p = p -> next;

    } while(p != head);
}

/* Function to exchange first and last node*/
struct Node *exchangeNodes(struct Node *head)
{
    // Find pointer to previous of last node
    struct Node *p = head;
    while (p->next->next != head)
        p = p->next;

    /* Exchange first and last nodes using
       head and p */
    p->next->next = head->next;
    head->next = p->next;
    p->next = head;
    head = head->next;

    return head;
}

// Driven Program
int main()
{
    int i;
    struct Node *head = NULL;
```

```
head = addToEmpty(head, 6);

for (i = 5; i > 0; i--)
head = addBegin(head, i);
cout << "List Before: ";
traverse(head);
cout << endl;

cout << "List After: ";
head = exchangeNodes(head);
traverse(head);

return 0;
}
```

Output:

```
List Before: 6 1 2 3 4 5
List After: 5 1 2 3 4 6
```

## Source

<https://www.geeksforgeeks.org/exchange-first-last-node-circular-linked-list/>

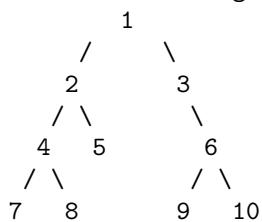
## Chapter 72

# 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.

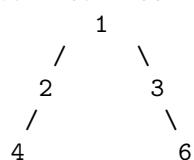
Let the following be input binary tree



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

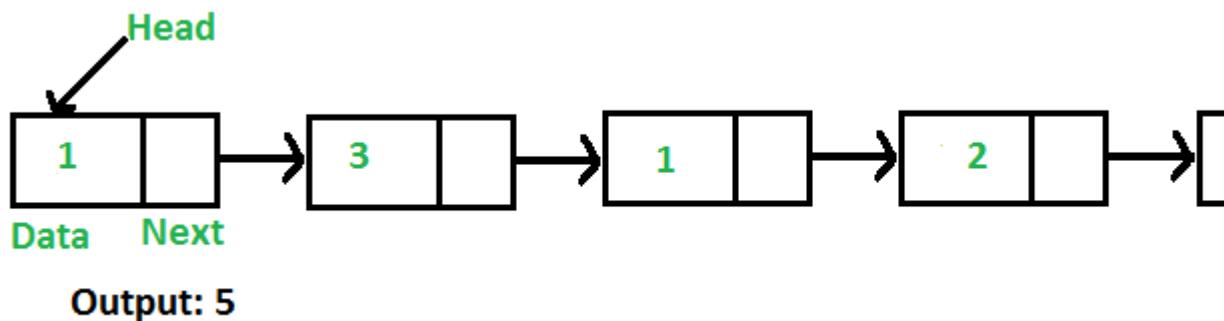
<https://www.geeksforgeeks.org/connect-leaves-doubly-linked-list/>

## Chapter 73

# Find Length of a Linked List (Iterative and Recursive)

Find Length of a Linked List (Iterative and Recursive) - GeeksforGeeks

Write a C function to count number of nodes in a given singly linked list.



For example, the function should return 5 for linked list 1->3->1->2->1.

### Iterative Solution

- 1) Initialize count as 0
- 2) Initialize a node pointer, current = head.
- 3) Do following while current is not NULL
  - a) current = current -> next
  - b) count++;
- 4) Return count

Following are C/C++, Java and Python implementations of above algorithm to find count of nodes.

C/C++

```
// Iterative C program to find length or count of nodes in a linked list
#include<stdio.h>
#include<stdlib.h>

/* Link list node */
struct Node
{
    int data;
    struct Node* next;
};

/* Given a reference (pointer to pointer) to the head
   of a list and an int, push a new node on the front
   of the list. */
void push(struct Node** head_ref, int new_data)
{
    /* allocate node */
    struct Node* new_node =
        (struct Node*) malloc(sizeof(struct Node));

    /* put in the data */
    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;
}

/* Counts no. of nodes in linked list */
int getCount(struct Node* head)
{
    int count = 0; // Initialize count
    struct Node* current = head; // Initialize current
    while (current != NULL)
    {
        count++;
        current = current->next;
    }
    return count;
}

/* Drier program to test count function*/
int main()
{
```

```
/* Start with the empty list */
struct Node* head = NULL;

/* Use push() to construct below list
   1->2->1->3->1 */
push(&head, 1);
push(&head, 3);
push(&head, 1);
push(&head, 2);
push(&head, 1);

/* Check the count function */
printf("count of nodes is %d", getCount(head));
return 0;
}
```

**Java**

```
// Java program to count number of nodes in a linked list

/* Linked list Node*/
class Node
{
    int data;
    Node next;
    Node(int d) { data = d; next = null; }
}

// Linked List class
class LinkedList
{
    Node head; // head of list

    /* Inserts a new Node at front of the list. */
    public void push(int new_data)
    {
        /* 1 & 2: Allocate the Node &
           Put in the data*/
        Node new_node = new Node(new_data);

        /* 3. Make next of new Node as head */
        new_node.next = head;

        /* 4. Move the head to point to new Node */
        head = new_node;
    }

    /* Returns count of nodes in linked list */
}
```

```
public int getCount()
{
    Node temp = head;
    int count = 0;
    while (temp != null)
    {
        count++;
        temp = temp.next;
    }
    return count;
}

/* Drier program to test above functions. Ideally
   this function should be in a separate user class.
   It is kept here to keep code compact */
public static void main(String[] args)
{
    /* Start with the empty list */
    LinkedList llist = new LinkedList();
    llist.push(1);
    llist.push(3);
    llist.push(1);
    llist.push(2);
    llist.push(1);

    System.out.println("Count of nodes is " +
                       llist.getCount());
}
}
```

### Python

```
# A complete working Python program to find length of a
# Linked List iteratively

# Node class
class Node:
    # Function to initialise the node object
    def __init__(self, data):
        self.data = data # Assign data
        self.next = None # Initialize next as null

# Linked List class contains a Node object
class LinkedList:

    # Function to initialize head
    def __init__(self):
```

```
        self.head = None

# This function is in LinkedList class. It inserts
# a new node at the beginning of Linked List.
def push(self, new_data):

    # 1 & 2: Allocate the Node &
    #         Put in the data
    new_node = Node(new_data)

    # 3. Make next of new Node as head
    new_node.next = self.head

    # 4. Move the head to point to new Node
    self.head = new_node

# This function counts number of nodes in Linked List
# iteratively, given 'node' as starting node.
def getCount(self):
    temp = self.head # Initialise temp
    count = 0 # Initialise count

    # Loop while end of linked list is not reached
    while (temp):
        count += 1
        temp = temp.next
    return count

# Code execution starts here
if __name__=='__main__':
    llist = LinkedList()
    llist.push(1)
    llist.push(3)
    llist.push(1)
    llist.push(2)
    llist.push(1)
    print ("Count of nodes is :",llist.getCount())
```

Output:

```
count of nodes is 5
```

### Recursive Solution

```
int getCount(head)
1) If head is NULL, return 0.
2) Else return 1 + getCount(head->next)
```

Following are C/C++, Java and Python implementations of above algorithm to find count of nodes.

### C/C++

```
// Recursive C program to find length or count of nodes in a linked list
#include<stdio.h>
#include<stdlib.h>

/* Link list node */
struct Node
{
    int data;
    struct Node* next;
};

/* Given a reference (pointer to pointer) to the head
   of a list and an int, push a new node on the front
   of the list. */
void push(struct Node** head_ref, int new_data)
{
    /* allocate node */
    struct Node* new_node =
        (struct Node*) malloc(sizeof(struct Node));

    /* put in the data */
    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;
}

/* Counts the no. of occurrences of a node
   (search_for) in a linked list (head)*/
int getCount(struct Node* head)
{
    // Base case
    if (head == NULL)
        return 0;
```

```
// count is 1 + count of remaining list
return 1 + getCount(head->next);
}

/* Drier program to test count function*/
int main()
{
    /* Start with the empty list */
    struct Node* head = NULL;

    /* Use push() to construct below list
     * 1->2->1->3->1 */
    push(&head, 1);
    push(&head, 3);
    push(&head, 1);
    push(&head, 2);
    push(&head, 1);

    /* Check the count function */
    printf("count of nodes is %d", getCount(head));
    return 0;
}
```

**Java**

```
// Recursive Java program to count number of nodes in
// a linked list

/* Linked list Node*/
class Node
{
    int data;
    Node next;
    Node(int d) { data = d;  next = null; }
}

// Linked List class
class LinkedList
{
    Node head; // head of list

    /* Inserts a new Node at front of the list. */
    public void push(int new_data)
    {
        /* 1 & 2: Allocate the Node &
           Put in the data*/
        Node new_node = new Node(new_data);
```

```
/* 3. Make next of new Node as head */
new_node.next = head;

/* 4. Move the head to point to new Node */
head = new_node;
}

/* Returns count of nodes in linked list */
public int getCountRec(Node node)
{
    // Base case
    if (node == null)
        return 0;

    // Count is this node plus rest of the list
    return 1 + getCountRec(node.next);
}

/* Wrapper over getCountRec() */
public int getCount()
{
    return getCountRec(head);
}

/* Drier program to test above functions. Ideally
   this function should be in a separate user class.
   It is kept here to keep code compact */
public static void main(String[] args)
{
    /* Start with the empty list */
    LinkedList llist = new LinkedList();
    llist.push(1);
    llist.push(3);
    llist.push(1);
    llist.push(2);
    llist.push(1);

    System.out.println("Count of nodes is " +
                       llist.getCount());
}
```

### Python

```
# A complete working Python program to find length of a
# Linked List recursively

# Node class
```

```
class Node:  
    # Function to initialise the node object  
    def __init__(self, data):  
        self.data = data # Assign data  
        self.next = None # Initialize next as null  
  
# Linked List class contains a Node object  
class LinkedList:  
  
    # Function to initialize head  
    def __init__(self):  
        self.head = None  
  
    # This function is in LinkedList class. It inserts  
    # a new node at the beginning of Linked List.  
    def push(self, new_data):  
  
        # 1 & 2: Allocate the Node &  
        #         Put in the data  
        new_node = Node(new_data)  
  
        # 3. Make next of new Node as head  
        new_node.next = self.head  
  
        # 4. Move the head to point to new Node  
        self.head = new_node  
  
    # This function counts number of nodes in Linked List  
    # recursively, given 'node' as starting node.  
    def getCountRec(self, node):  
        if (not node): # Base case  
            return 0  
        else:  
            return 1 + self.getCountRec(node.next)  
  
    # A wrapper over getCountRec()  
    def getCount(self):  
        return self.getCountRec(self.head)  
  
# Code execution starts here  
if __name__=='__main__':  
    llist = LinkedList()  
    llist.push(1)  
    llist.push(3)  
    llist.push(1)  
    llist.push(2)
```

```
llist.push(1)
print 'Count of nodes is :',llist.getCount()
```

Output:

```
count of nodes is 5
```

This article is contributed by **Ravi**. 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-length-of-a-linked-list-iterative-and-recursive/>

## Chapter 74

# Find a triplet from three linked lists with sum equal to a given number

Find a triplet from three linked lists with sum equal to a given number - GeeksforGeeks

Given three linked lists, say a, b and c, find one node from each list such that the sum of the values of the nodes is equal to a given number.

For example, if the three linked lists are 12->6->29, 23->5->8 and 90->20->59, and the given number is 101, the output should be tripel “6 5 90”.

In the following solutions, size of all three linked lists is assumed same for simplicity of analysis. The following solutions work for linked lists of different sizes also.

A simple method to solve this problem is to run three nested loops. The outermost loop picks an element from list a, the middle loop picks an element from b and the innermost loop picks from c. The innermost loop also checks whether the sum of values of current nodes of a, b and c is equal to given number. The time complexity of this method will be  $O(n^3)$ .

Sorting can be used to reduce the time complexity to  $O(n*n)$ . Following are the detailed steps.

- 1) Sort list b in ascending order, and list c in descending order.
- 2) After the b and c are sorted, one by one pick an element from list a and find the pair by traversing both b and c. See isSumSorted() in the following code. The idea is similar to Quadratic algorithm of [3 sum problem](#).

Following code implements step 2 only. The solution can be easily modified for unsorted lists by adding the merge sort code discussed [here](#).

C/C++

```
// C/C++ program to find a triplet from three linked lists with
```

```
// sum equal to a given number
#include<stdio.h>
#include<stdlib.h>
#include<stdbool.h>

/* Link list node */
struct Node
{
    int data;
    struct Node* next;
};

/* A utility function to insert a node at the beginning of a
   linked list*/
void push (struct Node** head_ref, int new_data)
{
    /* allocate node */
    struct Node* new_node =
        (struct Node*) malloc(sizeof(struct Node));

    /* put in the data */
    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;
}

/* A function to check if there are three elements in a, b
   and c whose sum is equal to givenNumber. The function
   assumes that the list b is sorted in ascending order
   and c is sorted in descending order. */
bool isSumSorted(struct Node *headA, struct Node *headB,
                 struct Node *headC, int givenNumber)
{
    struct Node *a = headA;

    // Traverse through all nodes of a
    while (a != NULL)
    {
        struct Node *b = headB;
        struct Node *c = headC;

        // For every node of list a, pick two nodes
        // from lists b and c
        while (b != NULL && c != NULL)
```

```
{  
    // If this a triplet with given sum, print  
    // it and return true  
    int sum = a->data + b->data + c->data;  
    if (sum == givenNumber)  
    {  
        printf ("Triplet Found: %d %d %d ", a->data,  
                b->data, c->data);  
        return true;  
    }  
  
    // If sum of this triplet is smaller, look for  
    // greater values in b  
    else if (sum < givenNumber)  
        b = b->next;  
    else // If sum is greater, look for smaller values in c  
        c = c->next;  
    }  
    a = a->next; // Move ahead in list a  
}  
  
printf ("No such triplet");  
return false;  
}  
  
/* Drier program to test above function*/  
int main()  
{  
    /* Start with the empty list */  
    struct Node* headA = NULL;  
    struct Node* headB = NULL;  
    struct Node* headC = NULL;  
  
    /*create a linked list 'a' 10->15->5->20 */  
    push (&headA, 20);  
    push (&headA, 4);  
    push (&headA, 15);  
    push (&headA, 10);  
  
    /*create a sorted linked list 'b' 2->4->9->10 */  
    push (&headB, 10);  
    push (&headB, 9);  
    push (&headB, 4);  
    push (&headB, 2);  
  
    /*create another sorted linked list 'c' 8->4->2->1 */  
    push (&headC, 1);
```

```
push (&headC, 2);
push (&headC, 4);
push (&headC, 8);

int givenNumber = 25;

isSumSorted (headA, headB, headC, givenNumber);

return 0;
}
```

**Java**

```
// Java program to find a triplet from three linked lists with
// sum equal to a given number
class LinkedList
{
    Node head; // head of list

    /* Linked list Node*/
    class Node
    {
        int data;
        Node next;
        Node(int d) {data = d; next = null; }
    }

    /* A function to check if there are three elements in a, b
       and c whose sum is equal to givenNumber. The function
       assumes that the list b is sorted in ascending order and
       c is sorted in descending order. */
    boolean isSumSorted(LinkedList la, LinkedList lb, LinkedList lc,
                        int givenNumber)
    {
        Node a = la.head;

        // Traverse all nodes of la
        while (a != null)
        {
            Node b = lb.head;
            Node c = lc.head;

            // for every node in la pick 2 nodes from lb and lc
            while (b != null && c!=null)
            {
                int sum = a.data + b.data + c.data;
                if (sum == givenNumber)
                {
```

```
        System.out.println("Triplet found " + a.data +
                            " " + b.data + " " + c.data);
        return true;
    }

    // If sum is smaller then look for greater value of b
    else if (sum < givenNumber)
        b = b.next;

    else
        c = c.next;
    }
    a = a.next;
}
System.out.println("No Triplet found");
return false;
}

/* Given a reference (pointer to pointer) to the head
   of a list and an int, push a new node on the front
   of the list. */
void push(int new_data)
{
    /* 1 & 2: Allocate the Node &
       Put in the data*/
    Node new_node = new Node(new_data);

    /* 3. Make next of new Node as head */
    new_node.next = head;

    /* 4. Move the head to point to new Node */
    head = new_node;
}

/* Drier program to test above functions */
public static void main(String args[])
{
    LinkedList llist1 = new LinkedList();
    LinkedList llist2 = new LinkedList();
    LinkedList llist3 = new LinkedList();

    /* Create Linked List llist1 100->15->5->20 */
    llist1.push(20);
    llist1.push(5);
    llist1.push(15);
    llist1.push(100);
```

```
/*create a sorted linked list 'b' 2->4->9->10 */
llist2.push(10);
llist2.push(9);
llist2.push(4);
llist2.push(2);

/*create another sorted linked list 'c' 8->4->2->1 */
llist3.push(1);
llist3.push(2);
llist3.push(4);
llist3.push(8);

int givenNumber = 25;
llist1.isSumSorted(llist1,llist2,llist3,givenNumber);
}
} /* This code is contributed by Rajat Mishra */
```

Output:

Triplet Found: 15 2 8

Time complexity: The linked lists b and c can be sorted in  $O(n\log n)$  time using Merge Sort (See [this](#)). The step 2 takes  $O(n^2)$  time. So the overall time complexity is  $O(n\log n) + O(n\log n) + O(n^2) = O(n^2)$ .

In this approach, the linked lists b and c are sorted first, so their original order will be lost. If we want to retain the original order of b and c, we can create copy of b and c.

This article is compiled by **Abhinav Priyadarshi** 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/find-a-triplet-from-three-linked-lists-with-sum-equal-to-a-given-number/>

## Chapter 75

# Find common elements in three linked lists

Find common elements in three linked lists - GeeksforGeeks

Given three linked lists, find all common element among the three linked lists.

Examples:

Input :

```
10 15 20 25 12  
10 12 13 15  
10 12 15 24 25 26
```

Output : 10 12 15

Input :

```
1 2 3 4 5  
1 2 3 4 6 9 8  
1 2 4 5 10
```

Output : 1 2 4

### Method 1 : (Simple)

Use three-pointers to iterate the given three linked lists and if any element common print that element.

Time complexity of the above solution will be  $O(N^3)$

### Method 2 : (Use Merge Sort)

In this method, we first sort the three lists and then we traverse the sorted lists to get the intersection.

Following are the steps to be followed to get intersection of three lists:

- 1) Sort the first Linked List using merge sort. This step takes  $O(m \log m)$  time. Refer [this post](#) for details of this step.

- 2) Sort the second Linked List using merge sort. This step takes  $O(n \log n)$  time. Refer [this post](#) for details of this step.
- 3) Sort the third Linked List using merge sort. This step takes  $O(p \log p)$  time. Refer [this post](#) for details of this step.
- 3) Linearly scan three sorted lists to get the intersection. This step takes  $O(m + n + p)$  time. This step can be implemented using the same algorithm as sorted arrays algorithm discussed [here](#).

Time complexity of this method is  $O(m \log m + n \log n + p \log p)$  which is better than method 1's time complexity.

### Method 3 : (Hashing)

Following are the steps to be followed to get intersection of three lists using hashing:

- 1) Create an empty hash table. Iterate through the first linked list and mark all the element frequency as 1 in the hash table. This step takes  $O(m)$  time.
- 2) Iterate through the second linked list and if current element frequency is 1 in hash table mark it as 2. This step takes  $O(n)$  time.
- 3) Iterate the third linked list and if the current element frequency is 2 in hash table mark it as 3. This step takes  $O(p)$  time.
- 4) Now iterate first linked list again to check the frequency of elements. if an element with frequency three exist in hash table, it will be present in the intersection of three linked lists. This step takes  $O(m)$  time.

Time complexity of this method is  $O(m + n + p)$  which is better than time complexity of method 1 and 2.

Below is the C++ implementation of the above idea.

```
// C++ program to find common element
// in three unsorted linked list
#include <bits/stdc++.h>
#define max 1000000
using namespace std;

/* Link list node */
struct Node {
    int data;
    struct Node* next;
};

/* A utility function to insert a node at the
beginning of a linked list */
void push(struct Node** head_ref, int new_data)
{
    struct Node* new_node =
        (struct Node *)malloc(sizeof(struct Node));
    new_node->data = new_data;
    new_node->next = (*head_ref);
    (*head_ref) = new_node;
```

```
}

/* print the common element in between
given three linked list*/
void Common(struct Node* head1,
            struct Node* head2, struct Node* head3)
{

    // Creating empty hash table;
    unordered_map<int, int> hash;

    struct Node* p = head1;
    while (p != NULL) {

        // set frequency by 1
        hash[p->data] = 1;
        p = p->next;
    }

    struct Node* q = head2;
    while (q != NULL) {

        // if the element is already exist in the
        // linked list set its frequency 2
        if (hash.find(q->data) != hash.end())
            hash[q->data] = 2;
        q = q->next;
    }

    struct Node* r = head3;
    while (r != NULL) {
        if (hash.find(r->data) != hash.end() &&
            hash[r->data] == 2)

            // if the element frquency is 2 it means
            // its present in both the first and second
            // linked list set its frquency 3
            hash[r->data] = 3;
        r = r->next;
    }

    for (auto x : hash) {

        // if current frequency is 3 its means
        // element is common in all the given
        // linked list
        if (x.second == 3)
```

```
        cout << x.first << " ";
    }
}

// Driver code
int main()
{
    // first list
    struct Node* head1 = NULL;
    push(&head1, 20);
    push(&head1, 5);
    push(&head1, 15);
    push(&head1, 10);

    // second list
    struct Node* head2 = NULL;
    push(&head2, 10);
    push(&head2, 20);
    push(&head2, 15);
    push(&head2, 8);

    // third list
    struct Node* head3 = NULL;
    push(&head3, 10);
    push(&head3, 2);
    push(&head3, 15);
    push(&head3, 20);

    Common(head1, head2, head3);

    return 0;
}
```

**Output:**

10 15 20

Time Complexity :  $O(m + n + p)$

**Source**

<https://www.geeksforgeeks.org/find-common-elements-in-three-linked-lists/>

## Chapter 76

# Find first node of loop in a linked list

Find first node of loop in a linked list - GeeksforGeeks

Write a function *findFirstLoopNode()* that checks whether a given Linked List contains loop. If loop is present then it returns point to first node of loop. Else it returns NULL.

Example :

Input : Head of bellow linked list

Output : Pointer to node 2

We have discussed [Floyd's loop detection algorithm](#). Below are steps to find first node of loop.

1. If a loop is found, initialize slow pointer to head, let fast pointer be at its position.
2. Move both slow and fast pointers one node at a time.
3. The point at which they meet is the start of the loop.

```
// C++ program to return first node of loop.  
#include <bits/stdc++.h>  
using namespace std;  
  
struct Node {  
    int key;  
    struct Node* next;  
};  
  
Node* newNode(int key)  
{
```

```
Node* temp = new Node;
temp->key = key;
temp->next = NULL;
return temp;
}

// A utility function to print a linked list
void printList(Node* head)
{
    while (head != NULL) {
        cout << head->key << " ";
        head = head->next;
    }
    cout << endl;
}

// Function to detect and remove loop
// in a linked list that may contain loop
Node* detectAndRemoveLoop(Node* head)
{
    // If list is empty or has only one node
    // without loop
    if (head == NULL || head->next == NULL)
        return NULL;

    Node *slow = head, *fast = head;

    // Move slow and fast 1 and 2 steps
    // ahead respectively.
    slow = slow->next;
    fast = fast->next->next;

    // Search for loop using slow and
    // fast pointers
    while (fast && fast->next) {
        if (slow == fast)
            break;
        slow = slow->next;
        fast = fast->next->next;
    }

    // If loop does not exist
    if (slow != fast)
        return NULL;

    // If loop exists. Start slow from
    // head and fast from meeting point.
    slow = head;
```

```
while (slow != fast) {
    slow = slow->next;
    fast = fast->next;
}

return slow;
}

/* Driver program to test above function*/
int main()
{
    Node* head = newNode(50);
    head->next = newNode(20);
    head->next->next = newNode(15);
    head->next->next->next = newNode(4);
    head->next->next->next->next = newNode(10);

    /* Create a loop for testing */
    head->next->next->next->next->next = head->next->next;

    Node* res = detectAndRemoveLoop(head);
    if (res == NULL)
        cout << "Loop does not exist";
    else
        cout << "Loop starting node is " << res->key;

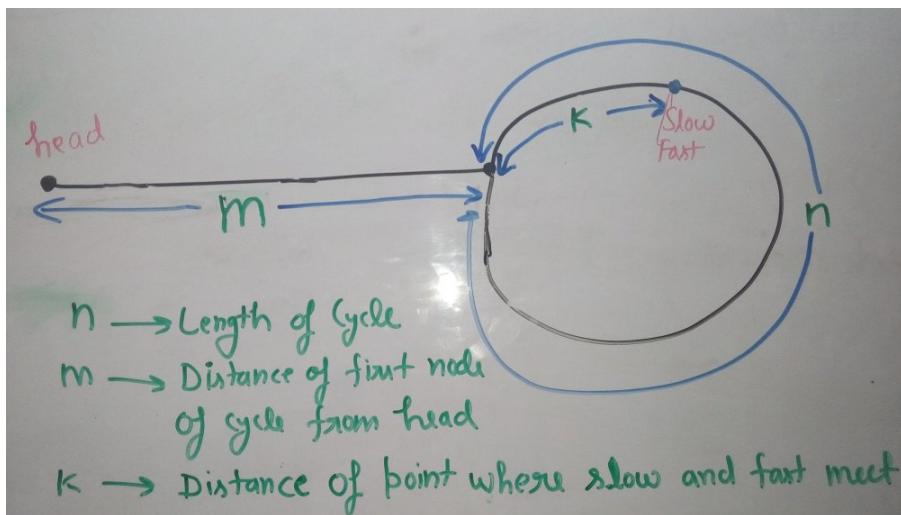
    return 0;
}
```

**Output:**

```
Loop starting node is 15
```

**How does this approach work?**

Let slow and fast meet at some point after Floyd's Cycle finding algorithm. Below diagram shows the situation when cycle is found.



We can conclude below from above diagram

Distance traveled by fast pointer =  $2 * (\text{Distance traveled by slow pointer})$

$$(m + n*x + k) = 2*(m + n*y + k)$$

Note that before meeting the point shown above, fast was moving at twice speed.

$x \rightarrow$  Number of complete cyclic rounds made by fast pointer before they meet first time

$y \rightarrow$  Number of complete cyclic rounds made by slow pointer before they meet first time

From above equation, we can conclude below

$$m + k = (x - 2y)*n$$

Which means  $m+k$  is a multiple of  $n$ .

So if we start moving both pointers again at **same speed** such that one pointer (say slow) begins from head node of linked list and other pointer (say fast) begins from meeting point. When slow pointer reaches beginning of loop (has made  $m$  steps), fast pointer would have made also moved  $m$  steps as they are now moving same pace. Since  $m+k$  is a multiple of  $n$  and fast starts from  $k$ , they would meet at the beginning. Can they meet before also? No because slow pointer enters the cycle first time after  $m$  steps.

## Source

<https://www.geeksforgeeks.org/find-first-node-of-loop-in-a-linked-list/>

## Chapter 77

# Find kth node from Middle towards Head of a Linked List

Find kth node from Middle towards Head of a Linked List - GeeksforGeeks

Given a Linked List and a number K. The task is to print the value of the K-th node from the middle towards the beginning of the List. If no such element exists, then print “-1”.

**Note:** Position of middle node is:  $(n/2)+1$ , where n is the total number of nodes in the list.

**Examples:**

```
Input : List is 1->2->3->4->5->6->7
        K= 2
Output : 2
```

```
Input : list is 7->8->9->10->11->12
        K = 3
Output : 7
```

Traverse the List from beginning to end and count the total number of nodes. Now, suppose  $n$  is the total number of nodes in the List. Therefore, the middle node will be at the position  $(n/2)+1$ . Now, the task remains to print the node at  $(n/2 + 1 - k)^{th}$  position from the head of the List.

Below is the implementation of the above idea:

```
// CPP program to find kth node from middle
// towards Head of the Linked List

#include <bits/stdc++.h>

using namespace std;
```

```
// Linked list node
struct Node {
    int data;
    struct Node* next;
};

/* Given a reference (pointer to
   pointer) to the head of a list
   and an int, push a new node on
   the front of the list. */
void push(struct Node** head_ref,
          int new_data)
{
    struct Node* new_node = new Node;
    new_node->data = new_data;
    new_node->next = (*head_ref);
    (*head_ref) = new_node;
}

// Function to count number of nodes
int getCount(struct Node* head)
{
    int count = 0; // Initialize count
    struct Node* current = head; // Initialize current
    while (current != NULL) {
        count++;
        current = current->next;
    }
    return count;
}

// Function to get the kth node from the mid
// towards begin of the linked list
int printKthfrommid(struct Node* head_ref, int k)
{
    // Get the count of total number of
    // nodes in the linked list
    int n = getCount(head_ref);

    int reqNode = ((n / 2 + 1) - k);

    // If no such node exists, return -1
    if (reqNode <= 0) {
        return -1;
    }

    // Find node at position reqNode
```

```
else {
    struct Node* current = head_ref;

    // the index of the
    // node we're currently
    // looking at
    int count = 1;
    while (current != NULL) {
        if (count == reqNode)
            return (current->data);
        count++;
        current = current->next;
    }
}

// Driver code
int main()
{
    // start with empty list
    struct Node* head = NULL;
    int k = 2;

    // create linked list
    // 1->2->3->4->5->6->7
    push(&head, 7);
    push(&head, 6);
    push(&head, 5);
    push(&head, 4);
    push(&head, 3);
    push(&head, 2);
    push(&head, 1);

    cout << printKthfrommid(head, 2);

    return 0;
}
```

**Output:**

2

**Time Complexity:**  $O(n)$ , where  $n$  is the length of the list.  
**Auxiliary Space:**  $O(1)$

## Source

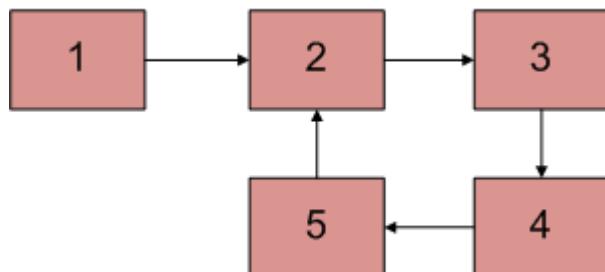
<https://www.geeksforgeeks.org/find-kth-node-from-middle-towards-head-of-a-linked-list/>

## Chapter 78

# Find length of loop in linked list

Find length of loop in linked list - GeeksforGeeks

Write a function *detectAndCountLoop()* that checks whether a given Linked List contains loop and if loop is present then returns count of nodes in loop. For example, loop is present in below linked list and length of loop is 4. If loop is not present, then function should return 0.



We know that [Floyd's Cycle detection algorithm](#) terminates when fast and slow pointers meet at a common point. We also know that this common point is one of the loop nodes (2 or 3 or 4 or 5 in the above diagram). We store the address of this in a pointer variable say *ptr2*. Then we start from the head of the Linked List and check for nodes one by one if they are reachable from *ptr2*. When we find a node that is reachable, we know that this node is the starting node of the loop in Linked List and we can get pointer to the previous of this node.

```
// C program to count number of nodes
// in loop in a linked list if loop is
// present
#include<stdio.h>
#include<stdlib.h>

/* Link list node */
struct Node
```

```
{  
    int data;  
    struct Node* next;  
};  
  
// Returns count of nodes present in loop.  
int countNodes(struct Node *n)  
{  
    int res = 1;  
    struct Node *temp = n;  
    while (temp->next != n)  
    {  
        res++;  
        temp = temp->next;  
    }  
    return res;  
}  
  
/* This function detects and counts loop  
nodes in the list. If loop is not there  
in then returns 0 */  
int countNodesinLoop(struct Node *list)  
{  
    struct Node *slow_p = list, *fast_p = list;  
  
    while (slow_p && fast_p && fast_p->next)  
    {  
        slow_p = slow_p->next;  
        fast_p = fast_p->next->next;  
  
        /* If slow_p and fast_p meet at some point  
        then there is a loop */  
        if (slow_p == fast_p)  
            return countNodes(slow_p);  
    }  
  
    /* Return 0 to indicate that there is no loop*/  
    return 0;  
}  
  
struct Node *newNode(int key)  
{  
    struct Node *temp =  
        (struct Node*)malloc(sizeof(struct Node));  
    temp->data = key;  
    temp->next = NULL;  
    return temp;  
}
```

```
/* Driver program to test above function*/
int main()
{
    struct Node *head = newNode(1);
    head->next = newNode(2);
    head->next->next = newNode(3);
    head->next->next->next = newNode(4);
    head->next->next->next->next = newNode(5);

    /* Create a loop for testing */
    head->next->next->next->next->next = head->next;

    printf("%d \n", countNodesinLoop(head));

    return 0;
}
```

Output :

4

**Related Articles :**

[Detect loop in a linked list](#)

[Detect and Remove Loop in a Linked List](#)

**Improved By :** [subtleseeker](#)

**Source**

<https://www.geeksforgeeks.org/find-length-of-loop-in-linked-list/>

## Chapter 79

# Find middle of singly linked list Recursively

Find middle of singly linked list Recursively - GeeksforGeeks

Given a singly linked list and the task is to find middle of linked list.

Examples:

Input : 1->2->3->4->5  
Output : 3

Input : 1->2->3->4->5->6  
Output : 4

We have already discussed [Iterative Solution](#). In this post iterative solution is discussed. Count total number of nodes in the list in recursive manner and do half of this, suppose this value is n. Then rolling back through recursion decrement n by one for each call. Return the node where n is zero.

```
// C++ program for Recursive approach to find
// middle of singly linked list
#include <iostream>
using namespace std;

// Tree Node Structure
struct Node
{
    int data;
    struct Node* next;
};

// Function to get the middle of
// the linked list
Node* getMiddle(Node* head)
{
    if (head == NULL)
        return head;

    Node* slow = head;
    Node* fast = head;

    while (fast->next != NULL && fast->next->next != NULL) {
        slow = slow->next;
        fast = fast->next->next;
    }

    return slow;
}
```

```
// Create new Node
Node* newLNode(int data)
{
    Node* temp = new Node;
    temp->data = data;
    temp->next = NULL;
    return temp;
}

// Function for finding midpoint recursively
void midpoint_util(Node* head, int* n, Node** mid)
{
    // If we reached end of linked list
    if (head == NULL)
    {
        *n = (*n) / 2;
        return;
    }

    *n = *n + 1;

    midpoint_util(head->next, n, mid);

    // Rolling back, decrement n by one
    *n = *n - 1;
    if (*n == 0)
    {
        // Final answer
        *mid = head;
    }
}

Node* midpoint(Node* head)
{
    Node* mid = NULL;
    int n = 1;
    midpoint_util(head, &n, &mid);
    return mid;
}

int main()
{
    Node* head = newLNode(1);
    head->next = newLNode(2);
    head->next->next = newLNode(3);
    head->next->next->next = newLNode(4);
```

```
head->next->next->next->next = newLNode(5);
Node* result = midpoint(head);
cout << result->data << endl;
return 0;
}
```

Output:

3

## Source

<https://www.geeksforgeeks.org/find-middle-singly-linked-list-recursively/>

## Chapter 80

# Find modular node in a linked list

Find modular node in a linked list - GeeksforGeeks

Given a singly linked list and a number k, find the last node whose  $n \% k == 0$ , where n is the number of nodes in the list.

Examples:

```
Input : list = 1->2->3->4->5->6->7
        k = 3
Output : 6
```

```
Input : list = 3->7->1->9->8
        k = 2
Output : 9
```

1. Take a pointer modularNode and initialize it with NULL. Traverse the linked list.
2. For every  $i \% k = 0$ , update modularNode.

C++

```
// C++ program to find modular node in a linked list
#include <bits/stdc++.h>

/* Linked list node */
struct Node {
    int data;
    Node* next;
};
```

```
/* Function to create a new node with given data */
Node* newNode(int data)
{
    Node* new_node = new Node;
    new_node->data = data;
    new_node->next = NULL;
    return new_node;
}

/* Function to find modular node in the linked list */
Node* modularNode(Node* head, int k)
{
    // Corner cases
    if (k <= 0 || head == NULL)
        return NULL;

    // Traverse the given list
    int i = 1;
    Node* modularNode = NULL;
    for (Node* temp = head; temp != NULL; temp = temp->next) {
        if (i % k == 0)
            modularNode = temp;

        i++;
    }
    return modularNode;
}

/* Driver program to test above function */
int main(void)
{
    Node* head = newNode(1);
    head->next = newNode(2);
    head->next->next = newNode(3);
    head->next->next->next = newNode(4);
    head->next->next->next->next = newNode(5);
    int k = 2;
    Node* answer = modularNode(head, k);
    printf("\nModular node is ");
    if (answer != NULL)
        printf("%d\n", answer->data);
    else
        printf("null\n");
    return 0;
}
```

Java

```
// A Java program to find modular node in a linked list
public class GFG
{
    // A Linkedlist node
    static class Node{
        int data;
        Node next;
        Node(int data){
            this.data = data;
        }
    }

    // Function to find modular node in the linked list
    static Node modularNode(Node head, int k)
    {
        // Corner cases
        if (k <= 0 || head == null)
            return null;

        // Traverse the given list
        int i = 1;
        Node modularNode = null;
        for (Node temp = head; temp != null; temp = temp.next) {
            if (i % k == 0)
                modularNode = temp;

            i++;
        }
        return modularNode;
    }

    // Driver code to test above function
    public static void main(String[] args)
    {
        Node head = new Node(1);
        head.next = new Node(2);
        head.next.next = new Node(3);
        head.next.next.next = new Node(4);
        head.next.next.next.next = new Node(5);
        int k = 2;
        Node answer = modularNode(head, k);
        System.out.print("Modular node is ");
        if (answer != null)
            System.out.println(answer.data);
        else
            System.out.println("null");
    }
}
```

```
// This code is contributed by Sumit Ghosh
```

Output:

```
Modular node is 4
```

### Source

<https://www.geeksforgeeks.org/find-modular-node-linked-list/>

## Chapter 81

# Find pair for given sum in a sorted singly linked without extra space

Find pair for given sum in a sorted singly linked without extra space - GeeksforGeeks

Given a sorted singly linked list and a value x, the task is to find pair whose sum is equal to x. We are not allowed to use any extra space and expected time complexity is O(n).

Examples:

```
Input : head = 3-->6-->7-->8-->9-->10-->11 , x=17
Output: (6, 11), (7, 10), (8, 9)
```

Hint : We are allowed to modify original linked list

A **simple solution** for this problem is to take each element one by one and traverse the remaining list in forward direction to find second element whose sum is equal to given value x. Time complexity for this approach will be  $O(n^2)$ .

An **efficient solution** for this problem is based on ideas discussed in below articles.

[Find pair in doubly linked list](#) : We use the same algorithm that traverses linked list from both ends.

[XOR Linked list](#) : In singly linked list, we can traverse list only in forward direction. We use XOR concept to convert a singly linked list to doubly linked list.

Below are steps :

- First we need to convert our singly linked list into doubly linked list. Here we are given singly linked list structure node which have only **next** pointer not **prev** pointer, so to convert our singly linked list into doubly linked list we use [memory efficient doubly linked list \( XOR linked list \)](#).

- In XOR linked list each **next** pointer of singly linked list contains XOR of **next** and **prev** pointer.
- After converting singly linked list into doubly linked list we initialize two pointers variables to find the candidate elements in the sorted doubly linked list. Initialize **first** with start of doubly linked list i.e; **first = head** and initialize **second** with last node of doubly linked list i.e; **second = last\_node**.
- Here we don't have random access, so to initialize pointer, we traverse the list till last node and assign last node to second.
- If current sum of **first** and **second** is less than x, then we move **first** in forward direction. If current sum of **first** and **second** element is greater than x, then we move **second** in backward direction.
- Loop termination conditions are also different from arrays. The loop terminates when either of two pointers become NULL, or they cross each other (**first==next\_node**), or they become same (**first == second**).

```
// C++ program to find pair with given sum in a singly
// linked list in O(n) time and no extra space.
#include<bits/stdc++.h>
using namespace std;

/* Link list node */
struct Node
{
    int data;

    /* also constains XOR of next and
       previous node after conversion*/
    struct Node* next;
};

/* Given a reference (pointer to pointer) to the head
   of a list and an int, push a new node on the front
   of the list. */
void insert(struct Node** head_ref, int new_data)
{
    /* allocate node */
    struct Node* new_node =
        (struct Node*) malloc(sizeof(struct Node));

    /* put in the data */
    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;
}
```

```
/* returns XORed value of the node addresses */
struct Node* XOR (struct Node *a, struct Node *b)
{
    return (struct Node*) ((uintptr_t) (a) ^ (uintptr_t) (b));
}

// Utility function to convert singly linked list
// into XOR doubly linked list
void convert(struct Node *head)
{
    // first we store address of next node in it
    // then take XOR of next node and previous node
    // and store it in next pointer
    struct Node *next_node;

    // prev node stores the address of previously
    // visited node
    struct Node *prev = NULL;

    // traverse list and store xor of address of
    // next_node and prev node in next pointer of node
    while (head != NULL)
    {
        // address of next node
        next_node = head->next;

        // xor of next_node and prev node
        head->next = XOR(next_node, prev);

        // update previous node
        prev = head;

        // move head forward
        head = next_node;
    }
}

// function to Find pair whose sum is equal to
// given value x
void pairSum(struct Node *head, int x)
{
    // initialize first
    struct Node *first = head;

    // next_node and prev node to calculate xor again
    // and find next and prev node while moving forward
    // and backward direction from both the corners
```

```

struct Node *next_node = NULL, *prev = NULL;

// traverse list to initialize second pointer
// here we need to move in forward direction so to
// calculate next address we have to take xor
// with prev pointer because  $(a \oplus b) \oplus b = a$ 
struct Node *second = head;
while (second->next != prev)
{
    struct Node *temp = second;
    second = XOR(second->next, prev);
    prev = temp;
}

// now traverse from both the corners
next_node = NULL;
prev = NULL;

// here if we want to move forward then we must
// know the prev address to calculate next node
// and if we want to move backward then we must
// know the next_node address to calculate prev node
bool flag = false;
while (first != NULL && second != NULL &&
       first != second && first != next_node)
{
    if ((first->data + second->data)==x)
    {
        cout << "(" << first->data << ","
           << second->data << ")" << endl;

        flag = true;

        // move first in forward
        struct Node *temp = first;
        first = XOR(first->next, prev);
        prev = temp;

        // move second in backward
        temp = second;
        second = XOR(second->next, next_node);
        next_node = temp;
    }
    else
    {
        if ((first->data + second->data) < x)
        {
            // move first in forward

```

```
        struct Node *temp = first;
        first = XOR(first->next, prev);
        prev = temp;
    }
    else
    {
        // move second in backward
        struct Node *temp = second;
        second = XOR(second->next, next_node);
        next_node = temp;
    }
}
if (flag == false)
    cout << "No pair found" << endl;
}

// Driver program to run the case
int main()
{
    /* Start with the empty list */
    struct Node* head = NULL;
    int x = 17;

    /* Use insert() to construct below list
    3-->6-->7-->8-->9-->10-->11 */
    insert(&head, 11);
    insert(&head, 10);
    insert(&head, 9);
    insert(&head, 8);
    insert(&head, 7);
    insert(&head, 6);
    insert(&head, 3);

    // convert singly linked list into XOR doubly
    // linked list
    convert(head);
    pairSum(head, x);
    return 0;
}
```

Output:

(6,11) , (7,10) , (8,9)

Time complexity : O(n)

If linked list is not sorted, then we can sort the list as a first step. But in that case overall time complexity would become  $O(n \log n)$ . We can use Hashing in such cases if extra space is not a constraint. The hashing based solution is same as method 2 [here](#).

## Source

<https://www.geeksforgeeks.org/find-pair-given-sum-sorted-singly-linked-without-extra-space/>

## Chapter 82

# Find pairs with given sum in doubly linked list

Find pairs with given sum in doubly linked list - GeeksforGeeks

Given a sorted doubly linked list of positive distinct elements, the task is to find pairs in doubly linked list whose sum is equal to given value x, without using any extra space ?

```
Input : head : 1 <-> 2 <-> 4 <-> 5 <-> 6 <-> 8 <-> 9  
       x = 7  
Output: (6, 1), (5,2)
```

Expected time complexity is O(n) and auxiliary space is O(1).

A **simple approach** for this problem is to one by one pick each node and find second element whose sum is equal to x in the remaining list by traversing in forward direction. Time complexity for this problem will be  $O(n^2)$  , n is total number of nodes in doubly linked list.

An **efficient solution** for this problem is same as [this](#) article. Here is the algorithm :

- Initialize two pointer variables to find the candidate elements in the sorted doubly linked list. Initialize **first** with start of doubly linked list i.e; **first=head** and initialize **second** with last node of doubly linked list i.e; **second=last\_node**.
- We initialize **first** and **second** pointers as first and last nodes. Here we don't have random access, so to find second pointer, we traverse the list to initialize second.
- If current sum of **first** and **second** is less than x, then we move **first** in forward direction. If current sum of **first** and **second** element is greater than x, then we move **second** in backward direction.
- Loop termination conditions are also different from arrays. The loop terminates when either of two pointers become NULL, or they cross each other (**second->next = first**), or they become same (**first == second**)

```
// C++ program to find a pair with given sum x.
#include<bits/stdc++.h>
using namespace std;

// structure of node of doubly linked list
struct Node
{
    int data;
    struct Node *next, *prev;
};

// Function to find pair whose sum equal to given value x.
void pairSum(struct Node *head, int x)
{
    // Set two pointers, first to the beginning of DLL
    // and second to the end of DLL.
    struct Node *first = head;
    struct Node *second = head;
    while (second->next != NULL)
        second = second->next;

    // To track if we find a pair or not
    bool found = false;

    // The loop terminates when either of two pointers
    // become NULL, or they cross each other (second->next
    // == first), or they become same (first == second)
    while (first != NULL && second != NULL &&
           first != second && second->next != first)
    {
        // pair found
        if ((first->data + second->data) == x)
        {
            found = true;
            cout << "(" << first->data << ", "
                  << second->data << ")" << endl;

            // move first in forward direction
            first = first->next;

            // move second in backward direction
            second = second->prev;
        }
        else
        {
            if ((first->data + second->data) < x)
                first = first->next;
            else

```

```
        second = second->prev;
    }
}

// if pair is not present
if (found == false)
    cout << "No pair found";
}

// A utility function to insert a new node at the
// beginning of doubly linked list
void insert(struct Node **head, int data)
{
    struct Node *temp = new Node;
    temp->data = data;
    temp->next = temp->prev = NULL;
    if (!(*head))
        (*head) = temp;
    else
    {
        temp->next = *head;
        (*head)->prev = temp;
        (*head) = temp;
    }
}

// Driver program
int main()
{
    struct Node *head = NULL;
    insert(&head, 9);
    insert(&head, 8);
    insert(&head, 6);
    insert(&head, 5);
    insert(&head, 4);
    insert(&head, 2);
    insert(&head, 1);
    int x = 7;

    pairSum(head, x);

    return 0;
}
```

Output:

(6,1)

(5, 2)

Time complexity : O(n)

Auxiliary space : O(1)

If linked list is not sorted, then we can sort the list as a first step. But in that case overall time complexity would become  $O(n \log n)$ . We can use Hashing in such cases if extra space is not a constraint. The hashing based solution is same as method 2 [here](#).

### Source

<https://www.geeksforgeeks.org/find-pairs-given-sum-doubly-linked-list/>

## Chapter 83

# Find smallest and largest elements in singly linked list

Find smallest and largest elements in singly linked list - GeeksforGeeks

Given a singly linked list of n nodes and find the smallest and largest elements in linked list.

Examples:

```
Input : 15 14 13 22 17
Output : Linked list are:
         17 -> 22 -> 13 -> 14 -> 15 -> NULL
         Maximum element in linked list: 22
         Minimum element in linked list: 13

Input : 20 25 23 68 54 13 45
Output : Linked list are:
         45 -> 13 -> 54 -> 68 -> 23 -> 25 -> 20 -> NULL
         Maximum element in linked list: 68
         Minimum element in linked list: 13
```

The idea is to traverse the linked list while head not equal to NULL and initialise the **max** and **min** variable to **INT\_MIN** and **INT\_MAX** respectively. After that check a condition that if max value is less then head value then head value is assign to max or min value is greater then head value then head value is assign to min otherwise head point to next node. Continue this process until head not equal to NULL.

```
// Program to find smallest and largest
// elements in singly linked list.
#include <bits/stdc++.h>
```

```
using namespace std;
/* Linked list node */
struct Node {
    int data;
    struct Node* next;
};

// Function that returns the largest element
// from the linked list.
int largestElement(struct Node* head)
{
    // Declare a max variable and initialize
    // it with INT_MIN value.
    // INT_MIN is integer type and its value
    // is -32767 or less.
    int max = INT_MIN;

    // Check loop while head not equal to NULL
    while (head != NULL) {

        // If max is less then head->data then
        // assign value of head->data to max
        // otherwise node point to next node.
        if (max < head->data)
            max = head->data;
        head = head->next;
    }
    return max;
}

// Function that returns smallest element
// from the linked list.
int smallestElement(struct Node* head)
{
    // Declare a min variable and initialize
    // it with INT_MAX value.
    // INT_MAX is integer type and its value
    // is 32767 or greater.
    int min = INT_MAX;

    // Check loop while head not equal to NULL
    while (head != NULL) {

        // If min is greater then head->data then
        // assign value of head->data to min
        // otherwise node point to next node.
        if (min > head->data)
            min = head->data;
    }
}
```

```
        head = head->next;
    }
    return min;
}

// Function that push the element in linked list.
void push(struct Node** head, int data)
{
    // Allocate dynamic memory for newNode.
    struct Node* newNode =
        (struct Node*)malloc(sizeof(struct Node));

    // Assign the data into newNode.
    newNode->data = data;

    // newNode->next assign the address of
    // head node.
    newNode->next = (*head);

    // newNode become the headNode.
    (*head) = newNode;
}

// Display linked list.
void printList(struct Node* head)
{
    while (head != NULL) {
        printf("%d -> ", head->data);
        head = head->next;
    }
    cout << "NULL" << endl;
}

// Drier program to test the functions
int main()
{
    // Start with empty list
    struct Node* head = NULL;

    // Using push() function to construct
    // singly linked list
    // 17->22->13->14->15
    push(&head, 15);
    push(&head, 14);
    push(&head, 13);
    push(&head, 22);
    push(&head, 17);
```

```
cout << "Linked list is : " << endl;

// Call printList() function to display
// the linked list.
printList(head);
cout << "Maximum element in linked list:";

// Call largestElement() function to get largest
// element in linked list.
cout << largestElement(head) << endl;
cout << "Minimum element in linked list:";

// Call smallestElement() function to get smallest
// element in linked list.
cout << smallestElement(head) << endl;

return 0;
}
```

Output:

```
Linked list is :
17 -> 22 -> 13 -> 14 -> 15 -> NULL
Maximum element in linked list: 22
Minimum element in linked list: 13
```

## Source

<https://www.geeksforgeeks.org/find-smallest-largest-elements-singly-linked-list/>

## Chapter 84

# Find the first non-repeating character from a stream of characters

Find the first non-repeating character from a stream of characters - GeeksforGeeks

Given a stream of characters, find the first non-repeating character from stream. You need to tell the first non-repeating character in O(1) time at any moment.

If we follow the first approach discussed [here](#), then we need to store the stream so that we can traverse it one more time to find the first non-repeating character at any moment. If we use extended approach discussed in the [same post](#), we need to go through the count array every time first non-repeating element is queried. We can find the first non-repeating character from stream at any moment without traversing any array.

The idea is to use a DLL (**Doubly Linked List**) to efficiently get the first non-repeating character from a stream. The DLL contains all non-repeating characters in order, i.e., the head of DLL contains first non-repeating character, the second node contains the second non-repeating and so on.

We also maintain two arrays: one array is to maintain characters that are already visited two or more times, we call it `repeated[]`, the other array is array of pointers to linked list nodes, we call it `inDLL[]`. The size of both arrays is equal to alphabet size which is typically 256.

1. Create an empty DLL. Also create two arrays `inDLL[]` and `repeated[]` of size 256. `inDLL` is an array of pointers to DLL nodes. `repeated[]` is a boolean array, `repeated[x]` is true if `x` is repeated two or more times, otherwise false. `inDLL[x]` contains pointer to a DLL node if character `x` is present in DLL, otherwise `NULL`.
2. Initialize all entries of `inDLL[]` as `NULL` and `repeated[]` as false.
3. To get the first non-repeating character, return character at head of DLL.
4. Following are steps to process a new character ‘`x`’ in stream.

- If repeated[x] is true, ignore this character (x is already repeated two or more times in the stream)
- If repeated[x] is false and inDLL[x] is NULL (x is seen first time). Append x to DLL and store address of new DLL node in inDLL[x].
- If repeated[x] is false and inDLL[x] is not NULL (x is seen second time). Get DLL node of x using inDLL[x] and remove the node. Also, mark inDLL[x] as NULL and repeated[x] as true.

Note that appending a new node to DLL is O(1) operation if we maintain tail pointer. Removing a node from DLL is also O(1). So both operations, addition of new character and finding first non-repeating character take O(1) time.

### C/C++

```
// A C++ program to find first non-repeating character
// from a stream of characters
#include <iostream>
#define MAX_CHAR 256
using namespace std;

// A linked list node
struct node
{
    char a;
    struct node *next, *prev;
};

// A utility function to append a character x at the end
// of DLL. Note that the function may change head and tail
// pointers, that is why pointers to these pointers are passed.
void appendNode(struct node **head_ref, struct node **tail_ref,
                char x)
{
    struct node *temp = new node;
    temp->a = x;
    temp->prev = temp->next = NULL;

    if (*head_ref == NULL)
    {
        *head_ref = *tail_ref = temp;
        return;
    }
    (*tail_ref)->next = temp;
    temp->prev = *tail_ref;
    *tail_ref = temp;
}

// A utility function to remove a node 'temp' fromt DLL.
```

```

// Note that the function may change head and tail pointers,
// that is why pointers to these pointers are passed.
void removeNode(struct node **head_ref, struct node **tail_ref,
                 struct node *temp)
{
    if (*head_ref == NULL)
        return;

    if (*head_ref == temp)
        *head_ref = (*head_ref)->next;
    if (*tail_ref == temp)
        *tail_ref = (*tail_ref)->prev;
    if (temp->next != NULL)
        temp->next->prev = temp->prev;
    if (temp->prev != NULL)
        temp->prev->next = temp->next;

    delete(temp);
}

void findFirstNonRepeating()
{
    // inDLL[x] contains pointer to a DLL node if x is present
    // in DLL. If x is not present, then inDLL[x] is NULL
    struct node *inDLL[MAX_CHAR];

    // repeated[x] is true if x is repeated two or more times.
    // If x is not seen so far or x is seen only once. then
    // repeated[x] is false
    bool repeated[MAX_CHAR];

    // Initialize the above two arrays
    struct node *head = NULL, *tail = NULL;
    for (int i = 0; i < MAX_CHAR; i++)
    {
        inDLL[i] = NULL;
        repeated[i] = false;
    }

    // Let us consider following stream and see the process
    char stream[] = "geeksforgeeksandgeeksquizfor";
    for (int i = 0; stream[i]; i++)
    {
        char x = stream[i];
        cout << "Reading " << x << " from stream n";

        // We process this character only if it has not occurred
        // or occurred only once. repeated[x] is true if x is

```

```
// repeated twice or more.s
if (!repeated[x])
{
    // If the character is not in DLL, then add this at
    // the end of DLL.
    if (inDLL[x] == NULL)
    {
        appendNode(&head, &tail, stream[i]);
        inDLL[x] = tail;
    }
    else // Otherwise remove this character from DLL
    {
        removeNode(&head, &tail, inDLL[x]);
        inDLL[x] = NULL;
        repeated[x] = true; // Also mark it as repeated
    }
}

// Print the current first non-repeating character from
// stream
if (head != NULL)
    cout << "First non-repeating character so far is "
        << head->a << endl;
}
}

/* Driver program to test above function */
int main()
{
    findFirstNonRepeating();
    return 0;
}
```

### Java

```
//A Java program to find first non-repeating character
//from a stream of characters

import java.util.ArrayList;
import java.util.List;

public class NonReapeatingC
{
    final static int MAX_CHAR = 256;

    static void findFirstNonRepeating()
    {
        // inDLL[x] contains pointer to a DLL node if x is present
```

```
// in DLL. If x is not present, then inDLL[x] is NULL
List<Character> inDLL =new ArrayList<Character>();

// repeated[x] is true if x is repeated two or more times.
// If x is not seen so far or x is seen only once. then
// repeated[x] is false
boolean[] repeated =new boolean[MAX_CHAR];

// Let us consider following stream and see the process
String stream = "geeksforgeeksandgeeksquizfor";
for (int i=0;i < stream.length();i++)
{
    char x = stream.charAt(i);
    System.out.println("Reading "+ x +" from stream n");

    // We process this character only if it has not occurred
    // or occurred only once. repeated[x] is true if x is
    // repeated twice or more.s
    if(!repeated[x])
    {
        // If the character is not in DLL, then add this at
        // the end of DLL.
        if(!(inDLL.contains(x)))
        {
            inDLL.add(x);
        }
        else      // Otherwise remove this character from DLL
        {
            inDLL.remove((Character)x);
            repeated[x] = true; // Also mark it as repeated
        }
    }

    // Print the current first non-repeating character from
    // stream
    if(inDLL.size() != 0)
    {
        System.out.print("First non-repeating character so far is ");
        System.out.println(inDLL.get(0));
    }
}

/* Driver program to test above function */
public static void main(String[] args)
{
    findFirstNonRepeating();
}
```

```
}
```

//This code is contributed by Sumit Ghosh

### Python

```
# A Python program to find first non-repeating character from
# a stream of characters
MAX_CHAR = 256

def findFirstNonRepeating():

    # inDLL[x] contains pointer to a DLL node if x is present
    # in DLL. If x is not present, then inDLL[x] is NULL
    inDLL = [] * MAX_CHAR

    # repeated[x] is true if x is repeated two or more times.
    # If x is not seen so far or x is seen only once. then
    # repeated[x] is false
    repeated = [False] * MAX_CHAR

    # Let us consider following stream and see the process
    stream = "geeksforgeeksandgeeksquizfor"
    for i in xrange(len(stream)):
        x = stream[i]
        print "Reading " + x + " from stream"

        # We process this character only if it has not occurred
        # or occurred only once. repeated[x] is true if x is
        # repeated twice or more.s
        if not repeated[ord(x)]:

            # If the character is not in DLL, then add this
            # at the end of DLL
            if not x in inDLL:
                inDLL.append(x)
            else:
                inDLL.remove(x)

        if len(inDLL) != 0:
            print "First non-repeating character so far is ",
            print str(inDLL[0])

    # Driver program
    findFirstNonRepeating()

# This code is contributed by BHAVYA JAIN
```

Output:

```
Reading g from stream
First non-repeating character so far is g
Reading e from stream
First non-repeating character so far is g
Reading e from stream
First non-repeating character so far is g
Reading k from stream
First non-repeating character so far is g
Reading s from stream
First non-repeating character so far is g
Reading f from stream
First non-repeating character so far is g
Reading o from stream
First non-repeating character so far is g
Reading r from stream
First non-repeating character so far is g
Reading g from stream
First non-repeating character so far is k
Reading e from stream
First non-repeating character so far is k
Reading e from stream
First non-repeating character so far is k
Reading k from stream
First non-repeating character so far is s
Reading s from stream
First non-repeating character so far is f
Reading a from stream
First non-repeating character so far is f
Reading n from stream
First non-repeating character so far is f
Reading d from stream
First non-repeating character so far is f
Reading g from stream
First non-repeating character so far is f
Reading e from stream
First non-repeating character so far is f
Reading e from stream
First non-repeating character so far is f
Reading k from stream
First non-repeating character so far is f
Reading s from stream
First non-repeating character so far is f
Reading q from stream
First non-repeating character so far is f
Reading u from stream
First non-repeating character so far is f
Reading i from stream
First non-repeating character so far is f
```

```
Reading z from stream
First non-repeating character so far is f
Reading f from stream
First non-repeating character so far is o
Reading o from stream
First non-repeating character so far is r
Reading r from stream
First non-repeating character so far is a
```

This article is contributed by [Amit Jain](#). 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-first-non-repeating-character-stream-characters/>

## Chapter 85

# Find the fractional (or $n/k - th$ ) node in linked list

Find the fractional (or  $n/k - th$ ) node in linked list - GeeksforGeeks

Given a singly linked list and a number k, write a function to find the  $(n/k)$ -th element, where n is the number of elements in the list. We need to consider ceil value in case of decimals.

Examples:

```
Input : list = 1->2->3->4->5->6
        k = 2
Output : 3
Since n = 6 and k = 2, we print (6/2)-th node
which is 3.
```

```
Input : list = 2->7->9->3->5
        k = 3
Output : 7
Since n is 5 and k is 3, we print ceil(5/3)-th
node which is 2nd node, i.e., 7.
```

1. Take two pointers temp and fractionalNode and initialize them with null and head respectively.
2. For every k jumps of the temp pointer, make one jump of the fractionalNode pointer.

C++

```
// C++ program to find fractional node in a linked list
#include <bits/stdc++.h>
```

```
/* Linked list node */
struct Node {
    int data;
    Node* next;
};

/* Function to create a new node with given data */
Node* newNode(int data)
{
    Node* new_node = new Node;
    new_node->data = data;
    new_node->next = NULL;
    return new_node;
}

/* Function to find fractional node in the linked list */
Node* fractionalNodes(Node* head, int k)
{
    // Corner cases
    if (k <= 0 || head == NULL)
        return NULL;

    Node* fractionalNode = NULL;

    // Traverse the given list
    int i = 0;
    for (Node* temp = head; temp != NULL; temp = temp->next) {

        // For every k nodes, we move fractionalNode one
        // step ahead.
        if (i % k == 0) {

            // First time we see a multiple of k
            if (fractionalNode == NULL)
                fractionalNode = head;

            else
                fractionalNode = fractionalNode->next;
        }
        i++;
    }
    return fractionalNode;
}

// A utility function to print a linked list
void printList(Node* node)
{
```

```
while (node != NULL) {
    printf("%d ", node->data);
    node = node->next;
}
printf("\n");

/* Driver program to test above function */
int main(void)
{
    Node* head = newNode(1);
    head->next = newNode(2);
    head->next->next = newNode(3);
    head->next->next->next = newNode(4);
    head->next->next->next->next = newNode(5);
    int k = 2;

    printf("List is ");
    printList(head);

    Node* answer = fractionalNodes(head, k);
    printf("\nFractional node is ");
    printf("%d\n", answer->data);

    return 0;
}
```

### Java

```
// Java program to find fractional node in
// a linked list
public class FractionalNodell
{
    /* Linked list node */
    static class Node{
        int data;
        Node next;

        //Constructor
        Node (int data){
            this.data = data;
        }
    }

    /* Function to find fractional node in the
       linked list */
    static Node fractionalNodes(Node head, int k)
    {
```

```
// Corner cases
if (k <= 0 || head == null)
    return null;

Node fractionalNode = null;

// Traverse the given list
int i = 0;
for (Node temp = head; temp != null;
        temp = temp.next){

    // For every k nodes, we move
    // fractionalNode one step ahead.
    if (i % k == 0){

        // First time we see a multiple of k
        if (fractionalNode == null)
            fractionalNode = head;
        else
            fractionalNode = fractionalNode.next;
    }
    i++;
}
return fractionalNode;
}

// A utility function to print a linked list
static void printList(Node node)
{
    while (node != null)
    {
        System.out.print(node.data+" ");
        node = node.next;
    }
    System.out.println();
}

/* Driver program to test above function */
public static void main(String[] args) {
    Node head = new Node(1);
    head.next = new Node(2);
    head.next.next = new Node(3);
    head.next.next.next = new Node(4);
    head.next.next.next.next = new Node(5);
    int k =2;

    System.out.print("List is ");
    printList(head);
}
```

```
Node answer = fractionalNodes(head, k);
System.out.println("Fractional node is "+
                    answer.data);
}
// This code is contributed by Sumit Ghosh
```

Output:

```
List is 1 2 3 4 5
Fractional node is 3
```

## Source

<https://www.geeksforgeeks.org/find-fractional-nk-th-node-linked-list/>

## Chapter 86

# Find the largest node in Doubly linked list

Find the largest node in Doubly linked list - GeeksforGeeks

Given a doubly linked list, find the largest node in the doubly linked list.

Examples:

```
Input : 10->8->4->23->67->88
        Largest node is: 88
Output : 88
```

```
Input : 34->2->78->18->120->39->7
        Largest node  is: 120
Output :120
```

### Approach Used :

1. Initialize the temp and max pointer to head nodes.
2. Traverse the whole list.
3. if temp's data is greater than max's data, then put max = temp.
4. move on next node .

```
/* Program to find the largest
   nodes in doubly linked list */
#include <stdio.h>
#include <stdlib.h>

// Create node of the doubly linked list
struct Node {
    int data;
```

```
struct Node* next;
struct Node* prev;
};

/* UTILITY FUNCTIONS */
/* Function to insert a node at the
beginning of the Doubly Linked List */
void push(struct Node** head_ref, int new_data)
{
    /* allocate node */
    struct Node* new_node =
        (struct Node*)malloc(sizeof(struct Node));

    /* put in the data */
    new_node->data = new_data;

    /* since we are adding at the
    begining, prev is always NULL */
    new_node->prev = NULL;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* change prev of head node to new node */
    if ((*head_ref) != NULL)
        (*head_ref)->prev = new_node;

    /* move the head to point to the new node */
    (*head_ref) = new_node;
}

/* Function to find the largest
nodes in Doubly Linked List */
int LargestInDLL(struct Node** head_ref)
{
    struct Node *max, *temp;

    /* initialize two pointer temp
       and max on the head node */
    temp = max = *head_ref;

    // traverse the whole doubly linked list
    while (temp != NULL) {

        /* if temp's data is greater than
           max's data, then put max = temp
           and return max->data */
        if (temp->data > max->data)
```

```
    max = temp;
    temp = temp->next;
}
return max->data;
}

int main()
{
    // Start with the empty list
    struct Node* head = NULL;

    // Let us create a linked list
    push(&head, 20);
    push(&head, 14);
    push(&head, 181);
    push(&head, 100);

    printf("%d", LargestInDLL(&head));
    return 0;
}
```

Output: 181

**Time Complexity:** O(n)  
**Auxiliary Space :** O(1)

## Source

<https://www.geeksforgeeks.org/find-largest-node-doubly-linked-list/>

## Chapter 87

# Find the middle of a given linked list in C and Java

Find the middle of a given linked list in C and Java - GeeksforGeeks

Given a singly linked list, find middle of the linked list. For example, if given linked list is 1->2->3->4->5 then output should be 3.

If there are even nodes, then there would be two middle nodes, we need to print second middle element. For example, if given linked list is 1->2->3->4->5->6 then output should be 4.

### Method 1:

Traverse the whole linked list and count the no. of nodes. Now traverse the list again till count/2 and return the node at count/2.

### Method 2:

Traverse linked list using two pointers. Move one pointer by one and other pointer by two. When the fast pointer reaches end slow pointer will reach middle of the linked list.

C

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

/* Link list node */
struct Node
{
    int data;
    struct Node* next;
};

/* Function to get the middle of the linked list*/
void printMiddle(struct Node *head)
```

```
{  
    struct Node *slow_ptr = head;  
    struct Node *fast_ptr = head;  
  
    if (head!=NULL)  
    {  
        while (fast_ptr != NULL && fast_ptr->next != NULL)  
        {  
            fast_ptr = fast_ptr->next->next;  
            slow_ptr = slow_ptr->next;  
        }  
        printf("The middle element is [%d]\n\n", slow_ptr->data);  
    }  
}  
  
void push(struct Node** head_ref, int new_data)  
{  
    /* allocate node */  
    struct Node* new_node =  
        (struct Node*) malloc(sizeof(struct Node));  
  
    /* put in the data */  
    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;  
}  
  
// A utility function to print a given linked list  
void printList(struct Node *ptr)  
{  
    while (ptr != NULL)  
    {  
        printf("%d->", ptr->data);  
        ptr = ptr->next;  
    }  
    printf("NULL\n");  
}  
  
/* Drier program to test above function*/  
int main()  
{  
    /* Start with the empty list */  
    struct Node* head = NULL;  
    int i;
```

```
for (i=5; i>0; i--)
{
    push(&head, i);
    printList(head);
    printMiddle(head);
}

return 0;
}

Java

// Java program to find middle of linked list
class LinkedList
{
    Node head; // head of linked list

    /* Linked list node */
    class Node
    {
        int data;
        Node next;
        Node(int d)
        {
            data = d;
            next = null;
        }
    }

    /* Function to print middle of linked list */
    void printMiddle()
    {
        Node slow_ptr = head;
        Node fast_ptr = head;
        if (head != null)
        {
            while (fast_ptr != null && fast_ptr.next != null)
            {
                fast_ptr = fast_ptr.next.next;
                slow_ptr = slow_ptr.next;
            }
            System.out.println("The middle element is [" +
                               slow_ptr.data + "] \n");
        }
    }

    /* Inserts a new Node at front of the list. */
}
```

```
public void push(int new_data)
{
    /* 1 & 2: Allocate the Node &
       Put in the data*/
    Node new_node = new Node(new_data);

    /* 3. Make next of new Node as head */
    new_node.next = head;

    /* 4. Move the head to point to new Node */
    head = new_node;
}

/* This function prints contents of linked list
   starting from the given node */
public void printList()
{
    Node tnode = head;
    while (tnode != null)
    {
        System.out.print(tnode.data+"->");
        tnode = tnode.next;
    }
    System.out.println("NULL");
}

public static void main(String [] args)
{
    LinkedList llist = new LinkedList();
    for (int i=5; i>0; --i)
    {
        llist.push(i);
        llist.printList();
        llist.printMiddle();
    }
}
// This code is contributed by Rajat Mishra
```

Output:

```
5->NULL
The middle element is [5]

4->5->NULL
The middle element is [5]

3->4->5->NULL
```

The middle element is [4]

2->3->4->5->NULL

The middle element is [4]

1->2->3->4->5->NULL

The middle element is [3]

**Method 3:**

Initialize mid element as head and initialize a counter as 0. Traverse the list from head, while traversing increment the counter and change mid to mid->next whenever the counter is odd. So the mid will move only half of the total length of the list.

Thanks to Narendra Kangralkar for suggesting this method.

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

/* Link list node */
struct node
{
    int data;
    struct node* next;
};

/* Function to get the middle of the linked list*/
void printMiddle(struct node *head)
{
    int count = 0;
    struct node *mid = head;

    while (head != NULL)
    {
        /* update mid, when 'count' is odd number */
        if (count & 1)
            mid = mid->next;

        ++count;
        head = head->next;
    }

    /* if empty list is provided */
    if (mid != NULL)
        printf("The middle element is [%d]\n\n", mid->data);
}

void push(struct node** head_ref, int new_data)
{
```

```
/* allocate node */
struct node* new_node =
    (struct node*) malloc(sizeof(struct node));

/* put in the data */
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;
}

// A utility function to print a given linked list
void printList(struct node *ptr)
{
    while (ptr != NULL)
    {
        printf("%d->", ptr->data);
        ptr = ptr->next;
    }
    printf("NULL\n");
}

/* Drier program to test above function*/
int main()
{
    /* Start with the empty list */
    struct node* head = NULL;
    int i;

    for (i=5; i>0; i--)
    {
        push(&head, i);
        printList(head);
        printMiddle(head);
    }

    return 0;
}
```

Output:

```
5->NULL
The middle element is [5]
```

```
4->5->NULL  
The middle element is [5]
```

```
3->4->5->NULL  
The middle element is [4]
```

```
2->3->4->5->NULL  
The middle element is [4]
```

```
1->2->3->4->5->NULL  
The middle element is [3]
```

## Source

<https://www.geeksforgeeks.org/write-a-c-function-to-print-the-middle-of-the-linked-list/>

## Chapter 88

# Find the sum of last n nodes of the given Linked List

Find the sum of last n nodes of the given Linked List - GeeksforGeeks

Given a linked list and a number **n**. Find the sum of last **n** nodes of the linked list.

**Constraints:**  $0 \leq n \leq$  number of nodes in the linked list.

Examples:

Input : 10->6->8->4->12, n = 2

Output : 16

Sum of last two nodes:

12 + 4 = 16

Input : 15->7->9->5->16->14, n = 4

Output : 44

### Method 1: (Recursive approach using system call stack)

Recursively traverse the linked list up to the end. Now during return from the function calls, add up the last **n** nodes. The sum can be accumulated in some variable passed by reference to the function or to some global variable.

```
// C++ implementation to find the sum of
// last 'n' nodes of the Linked List
#include <bits/stdc++.h>
using namespace std;

/* A Linked list node */
struct Node {
    int data;
```

```
    struct Node* next;
};

// function to insert a node at the
// beginning of the linked list
void push(struct Node** head_ref, int new_data)
{
    /* allocate node */
    struct Node* new_node = new Node;

    /* put in the data */
    new_node->data = new_data;

    /* link the old list to the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref) = new_node;
}

// function to recursively find the sum of last
// 'n' nodes of the given linked list
void sumOfLastN_Nodes(struct Node* head, int* n,
                      int* sum)
{
    // if head = NULL
    if (!head)
        return;

    // recursively traverse the remaining nodes
    sumOfLastN_Nodes(head->next, n, sum);

    // if node count 'n' is greater than 0
    if (*n > 0) {

        // accumulate sum
        *sum = *sum + head->data;

        // reduce node count 'n' by 1
        --*n;
    }
}

// utility function to find the sum of last 'n' nodes
int sumOfLastN_NodesUtil(struct Node* head, int n)
{
    // if n == 0
    if (n <= 0)
```

```
    return 0;

    int sum = 0;

    // find the sum of last 'n' nodes
    sumOfLastN_Nodes(head, &n, &sum);

    // required sum
    return sum;
}

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

    // create linked list 10->6->8->4->12
    push(&head, 12);
    push(&head, 4);
    push(&head, 8);
    push(&head, 6);
    push(&head, 10);

    int n = 2;
    cout << "Sum of last " << n << " nodes = "
        << sumOfLastN_NodesUtil(head, n);
    return 0;
}
```

Output:

```
Sum of last 2 nodes = 16
```

Time Complexity: O(n), where n is the number of nodes in the linked list.  
Auxiliary Space: O(n), if system call stack is being considered.

#### **Method 2 (Iterative approach using user defined stack)**

It is an iterative procedure to the recursive approach explained in **Method 1** of this post. Traverses the nodes from left to right. While traversing pushes the nodes to a user defined stack. Then pops the top n values from the stack and adds them.

```
// C++ implementation to find the sum of last
// 'n' nodes of the Linked List
#include <bits/stdc++.h>
using namespace std;

/* A Linked list node */
```

```
struct Node {  
    int data;  
    struct Node* next;  
};  
  
// function to insert a node at the  
// beginning of the linked list  
void push(struct Node** head_ref, int new_data)  
{  
    /* allocate node */  
    struct Node* new_node = new Node;  
  
    /* put in the data */  
    new_node->data = new_data;  
  
    /* link the old list to the new node */  
    new_node->next = (*head_ref);  
  
    /* move the head to point to the new node */  
    (*head_ref) = new_node;  
}  
  
// utility function to find the sum of last 'n' nodes  
int sumOfLastN_NodesUtil(struct Node* head, int n)  
{  
    // if n == 0  
    if (n <= 0)  
        return 0;  
  
    stack<int> st;  
    int sum = 0;  
  
    // traverses the list from left to right  
    while (head != NULL) {  
  
        // push the node's data onto the stack 'st'  
        st.push(head->data);  
  
        // move to next node  
        head = head->next;  
    }  
  
    // pop 'n' nodes from 'st' and  
    // add them  
    while (n--) {  
        sum += st.top();  
        st.pop();  
    }  
}
```

```
// required sum
return sum;
}

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

    // create linked list 10->6->8->4->12
    push(&head, 12);
    push(&head, 4);
    push(&head, 8);
    push(&head, 6);
    push(&head, 10);

    int n = 2;
    cout << "Sum of last " << n << " nodes = "
        << sumOfLastN_NodesUtil(head, n);
    return 0;
}
```

Output:

```
Sum of last 2 nodes = 16
```

Time Complexity:  $O(n)$ , where  $n$  is the number of nodes in the linked list.

Auxiliary Space:  $O(n)$ , stack size

### Method 3 (Reversing the linked list)

Following are the steps:

1. Reverse the given linked list.
2. Traverse the first  $n$  nodes of the reversed linked list.
3. While traversing add them.
4. Reverse the linked list back to its original order.
5. Return the added sum.

```
// C++ implementation to find the sum of last
// 'n' nodes of the Linked List
#include <bits/stdc++.h>
using namespace std;

/* A Linked list node */
struct Node {
```

```
int data;
struct Node* next;
};

// function to insert a node at the
// beginning of the linked list
void push(struct Node** head_ref, int new_data)
{
    /* allocate node */
    struct Node* new_node = new Node;

    /* put in the data */
    new_node->data = new_data;

    /* link the old list to the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref) = new_node;
}

void reverseList(struct Node** head_ref)
{
    struct Node* current, *prev, *next;
    current = *head_ref;
    prev = NULL;

    while (current != NULL) {
        next = current->next;
        current->next = prev;
        prev = current;
        current = next;
    }

    *head_ref = prev;
}

// utility function to find the sum of last 'n' nodes
int sumOfLastN_NodesUtil(struct Node* head, int n)
{
    // if n == 0
    if (n <= 0)
        return 0;

    // reverse the linked list
    reverseList(&head);

    int sum = 0;
```

```
struct Node* current = head;

// traverse the 1st 'n' nodes of the reversed
// linked list and add them
while (current != NULL && & n--) {

    // accumulate node's data to 'sum'
    sum += current->data;

    // move to next node
    current = current->next;
}

// reverse back the linked list
reverseList(&head);

// required sum
return sum;
}

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

    // create linked list 10->6->8->4->12
    push(&head, 12);
    push(&head, 4);
    push(&head, 8);
    push(&head, 6);
    push(&head, 10);

    int n = 2;
    cout << "Sum of last " << n << " nodes = "
        << sumOfLastN_NodesUtil(head, n);
    return 0;
}
```

Output:

```
Sum of last 2 nodes = 16
```

Time Complexity: O(n), where n is the number of nodes in the linked list.  
Auxiliary Space: O(1)

#### Method 4 (Using length of linked list)

Following are the steps:

1. Calculate the length of the given Linked List. Let it be **len**.
2. First traverse the (**len - n**) nodes from the beginning.
3. Then traverse the remaining **n** nodes and while traversing add them.
4. Return the added sum.

```
// C++ implementation to find the sum of last
// 'n' nodes of the Linked List
#include <bits/stdc++.h>
using namespace std;

/* A Linked list node */
struct Node {
    int data;
    struct Node* next;
};

// function to insert a node at the
// beginning of the linked list
void push(struct Node** head_ref, int new_data)
{
    /* allocate node */
    struct Node* new_node = new Node;

    /* put in the data */
    new_node->data = new_data;

    /* link the old list to the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref) = new_node;
}

// utility function to find the sum of last 'n' nodes
int sumOfLastN_NodesUtil(struct Node* head, int n)
{
    // if n == 0
    if (n <= 0)
        return 0;

    int sum = 0, len = 0;
    struct Node* temp = head;

    // calculate the length of the linked list
    while (temp != NULL) {
        len++;
        temp = temp->next;
    }
}
```

```
// count of first (len - n) nodes
int c = len - n;
temp = head;

// just traverse the 1st 'c' nodes
while (temp != NULL && & c--)

    // move to next node
    temp = temp->next;

// now traverse the last 'n' nodes and add them
while (temp != NULL) {

    // accumulate node's data to sum
    sum += temp->data;

    // move to next node
    temp = temp->next;
}

// required sum
return sum;
}

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

    // create linked list 10->6->8->4->12
    push(&head, 12);
    push(&head, 4);
    push(&head, 8);
    push(&head, 6);
    push(&head, 10);

    int n = 2;
    cout << "Sum of last " << n << " nodes = "
        << sumOfLastN_NodesUtil(head, n);
    return 0;
}
```

Output:

```
Sum of last 2 nodes = 16
```

Time Complexity: O(n), where n is the number of nodes in the linked list.

Auxiliary Space: O(1)

**Method 5 (Use of two pointers requires single traversal)**

Maintain two pointers – reference pointer and main pointer. Initialize both reference and main pointers to head. First move reference pointer to **n** nodes from head and while traversing accumulate node's data to some variable, say **sum**. Now move both pointers simultaneously until reference pointer reaches to the end of the list and while traversing accumulate all node's data to **sum** pointed by the reference pointer and accumulate all node's data to some variable, say **temp**, pointed by the main pointer. Now, (**sum - temp**) is the required sum of the last **n** nodes.

```
// C++ implementation to find the sum of last
// 'n' nodes of the Linked List
#include <bits/stdc++.h>
using namespace std;

/* A Linked list node */
struct Node {
    int data;
    struct Node* next;
};

// function to insert a node at the
// beginning of the linked list
void push(struct Node** head_ref, int new_data)
{
    /* allocate node */
    struct Node* new_node = new Node;

    /* put in the data */
    new_node->data = new_data;

    /* link the old list to the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref) = new_node;
}

// utility function to find the sum of last 'n' nodes
int sumOfLastN_NodesUtil(struct Node* head, int n)
{
    // if n == 0
    if (n <= 0)
        return 0;

    int sum = 0, temp = 0;
    struct Node* ref_ptr, *main_ptr;
```

```
ref_ptr = main_ptr = head;

// traverse 1st 'n' nodes through 'ref_ptr' and
// accumulate all node's data to 'sum'
while (ref_ptr != NULL && & n--) {
    sum += ref_ptr->data;

    // move to next node
    ref_ptr = ref_ptr->next;
}

// traverse to the end of the linked list
while (ref_ptr != NULL) {

    // accumulate all node's data to 'temp' pointed
    // by the 'main_ptr'
    temp += main_ptr->data;

    // accumulate all node's data to 'sum' pointed by
    // the 'ref_ptr'
    sum += ref_ptr->data;

    // move both the pointers to their respective
    // next nodes
    main_ptr = main_ptr->next;
    ref_ptr = ref_ptr->next;
}

// required sum
return (sum - temp);
}

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

    // create linked list 10->6->8->4->12
    push(&head, 12);
    push(&head, 4);
    push(&head, 8);
    push(&head, 6);
    push(&head, 10);

    int n = 2;
    cout << "Sum of last " << n << " nodes = "
        << sumOfLastN_NodesUtil(head, n);
    return 0;
}
```

}

Output:

```
Sum of last 2 nodes = 16
```

Time Complexity: O(n), where n is the number of nodes in the linked list.

Auxiliary Space: O(1)

## Source

<https://www.geeksforgeeks.org/find-sum-last-n-nodes-given-linked-list/>

# Chapter 89

## Find unique elements in linked list

Find unique elements in linked list - GeeksforGeeks

Given a linked list. We need to find unique elements in the linked list i.e, those elements which are not repeated in the linked list or those elements whose frequency is 1. If No such elements are present in list so Print " No Unique Elements".

Examples:

```
Input : 1 -> 4 -> 4 -> 2 -> 3 -> 5 -> 3 -> 4 -> 5  
Output :1 2
```

```
Input :4 -> 5 -> 2 -> 5 -> 1 -> 4 -> 1 -> 2  
Output :No Unique Elements
```

**Method 1 (Using Two Loops)** This is the simple way where two loops are used. Outer loop is used to pick the elements one by one and inner loop compares the picked element with rest of the elements. If Element is not equal to other elements than Print that Element.  
Time Complexity :  $O(N * n)$

**Method 2 (Sorting)** : Sort the elements using Merge Sort.  $O(n \log n)$ . Now Traverse List in linear time and check if current element is not equal to previous element then Print  $O(N)$

Please note that this method doesn't preserve the original order of elements.

Time Complexity:  $O(N \log N)$

**Method 3 (Hashing)**

We use the concept of Hash table Here, We traverse the link list from head to end. For every newly encountered element, we put it in the hash table after that we again traverse list and Print those elements whose frequency is 1. Time Complexity :  $O(N)$

Below is the Implementation of this

```
// C++ Program to Find the Unique elements in
// linked lists
#include <bits/stdc++.h>
using namespace std;

/* Linked list node */
struct Node {
    int data;
    struct Node* next;
};

/* Function to insert a node at the beginning of
   the linked list */
void push(struct Node** head_ref, int new_data)
{
    struct Node* new_node = new Node;
    new_node->data = new_data;
    new_node->next = *head_ref;
    *head_ref = new_node;
}

// function to Find the unique elements in linked lists
void uniqueElements(struct Node* head)
{
    // Initialize hash array that store the
    // frequency of each element of list
    unordered_map<int, int> hash;

    for (Node *temp=head; temp!=NULL; temp=temp->next)
        hash[temp->data]++;

    int count = 0;
    for (Node *temp=head; temp!=NULL; temp=temp->next) {

        // Check whether the frequency of current
        // element is 1 or not
        if (hash[temp->data] == 1) {
            cout << temp->data << " ";
            count++;
        }
    }

    // If No unique element in list
    if (count == 0)
        cout << " No Unique Elements ";
}

// Driver program to test above
```

```
int main()
{
    struct Node* head = NULL;

    // creating linked list
    push(&head, 5);
    push(&head, 4);
    push(&head, 3);
    push(&head, 5);
    push(&head, 3);
    push(&head, 2);
    push(&head, 4);
    push(&head, 4);
    push(&head, 1);
    uniqueElements(head);
    return 0;
}
```

**Output:**

1 2

Time Complexity :  $O(N)$   
Auxiliary Space :  $O(N)$

**Source**

<https://www.geeksforgeeks.org/find-unique-elements-linked-list/>

## Chapter 90

# First common element in two linked lists

First common element in two linked lists - GeeksforGeeks

Given two Linked Lists, find the first common element between given linked list i.e., we need to find first node of first list which is also present in second list.

Examples:

```
Input :  
List1: 10->15->4->20  
Lsit2: 8->4->2->10  
Output : 10
```

```
Input :  
List1: 1->2->3->4  
Lsit2: 5->6->3->8  
Output : 3
```

We traverse first list and for every node, we search it in second list. As soon as we find an element in second list, we return it.

```
// C++ program to find first common element in  
// two unsorted linked list  
#include <bits/stdc++.h>  
using namespace std;  
  
/* Link list node */  
struct Node {  
    int data;  
    struct Node* next;
```

```
};

/* A utility function to insert a node at the
   beginning of a linked list*/
void push(struct Node** head_ref, int new_data)
{
    struct Node* new_node =
        (struct Node*)malloc(sizeof(struct Node));
    new_node->data = new_data;
    new_node->next = (*head_ref);
    (*head_ref) = new_node;
}

/* Returns the first repeating element in linked list*/
int firstCommon(struct Node* head1, struct Node* head2)
{
    // Traverse through every node of first list
    for (; head1 != NULL; head1=head1->next)

        // If current node is present in second list
        for (Node *p = head2; p != NULL; p = p->next)
            if (p->data == head1->data)
                return head1->data;

    // If no common node
    return 0;
}

// Driver code
int main()
{
    struct Node* head1 = NULL;
    push(&head1, 20);
    push(&head1, 5);
    push(&head1, 15);
    push(&head1, 10);

    struct Node* head2 = NULL;
    push(&head2, 10);
    push(&head2, 2);
    push(&head2, 15);
    push(&head2, 8);

    cout << firstCommon(head1, head2);
    return 0;
}
```

Output:

10

**Source**

<https://www.geeksforgeeks.org/first-common-element-two-linked-lists/>

## Chapter 91

# First non-repeating in a linked list

First non-repeating in a linked list - GeeksforGeeks

Given a linked list, find its first non-repeating integer element.

Examples:

Input : 10->20->30->10->20->40->30->NULL  
Output :First Non-repeating element is 40.

Input :1->1->2->2->3->4->3->4->5->NULL  
Output :First Non-repeating element is 5.

Input :1->1->2->2->3->4->3->4->NULL  
Output :No Non-repeating element is found.

- 1) Create a hash table and marked all element as zero.
- 2) Traverse the linked list and count the frequency of all the element in hashtable.
- 3) Traverse the linked list again and see the element who's frequency is 1 in hashtable.

```
// C++ program to find first non-repeating
// element in a linked list
#include<bits/stdc++.h>
using namespace std;

/* Link list node */
struct Node
{
    int data;
    struct Node* next;
}
```

```
};

/* Function to find the first non-repeating
   element in the linked list */
int firstNonRepeating(struct Node *head)
{
    // Create an empty map and insert all linked
    // list elements into hash table
    unordered_map<int, int> mp;
    for (Node *temp=head; temp!=NULL; temp=temp->next)
        mp[temp->data]++;
    
    // Traverse the linked list again and return
    // the first node whose count is 1
    for (Node *temp=head; temp!=NULL; temp=temp->next)
        if (mp[temp->data] == 1)
            return temp->data;

    return -1;
}

/* Function to push a node */
void push(struct Node** head_ref, int new_data)
{
    struct Node* new_node =
        (struct Node*) malloc(sizeof(struct Node));
    new_node->data = new_data;
    new_node->next = (*head_ref);
    (*head_ref) = new_node;
}

/* Driver program to test above function*/
int main()
{
    // Let us create below linked list.
    // 85->15->18->20->85->35->4->20->NULL
    struct Node* head = NULL;
    push(&head, 20);
    push(&head, 4);
    push(&head, 35);
    push(&head, 85);
    push(&head, 20);
    push(&head, 18);
    push(&head, 15);
    push(&head, 85);
    cout << firstNonRepeating(head);
    return 0;
}
```

Output:

15

**Further Optimizations:**

The above solution requires two traversals of linked list. In case we have many repeating elements, we can save one traversal by storing positions also in hash table. Please refer last method of [Given a string, find its first non-repeating character](#) for details.

**Source**

<https://www.geeksforgeeks.org/first-non-repeating-linked-list/>

## Chapter 92

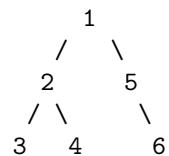
# Flatten a binary tree into linked list

Flatten a binary tree into linked list - GeeksforGeeks

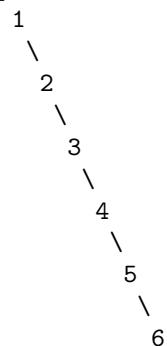
Given a binary tree, flatten it into linked list in-place. Usage of auxiliary data structure is not allowed. After flattening, left of each node should point to NULL and right should contain next node in level order.

Examples:

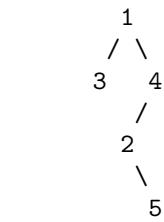
Input :



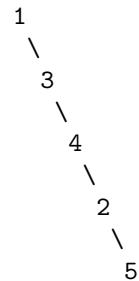
Output :



Input :



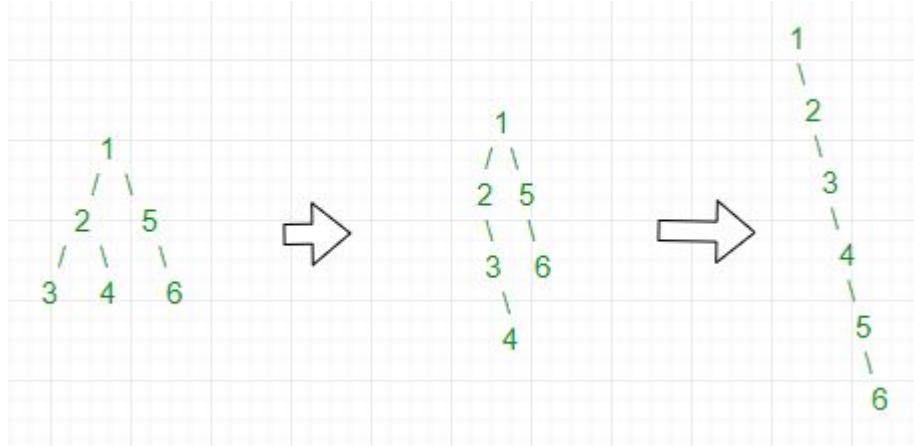
Output :



**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 93

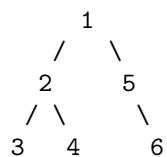
# Flatten a binary tree into linked list | Set-2

Flatten a binary tree into linked list | Set-2 - GeeksforGeeks

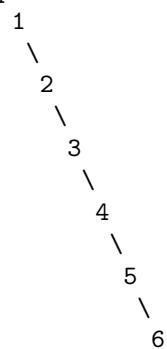
Given a binary tree, flatten it into a linked list. After flattening, the left of each node should point to NULL and right should contain next node in level order.

**Example:**

**Input:**

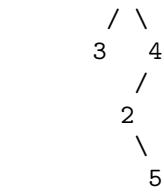


**Output:**

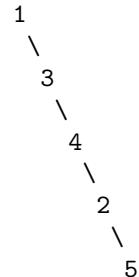


**Input:**

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 flattenning  
    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 94

# Flatten a multi-level linked list | Set 2 (Depth wise)

Flatten a multi-level linked list | Set 2 (Depth wise) - GeeksforGeeks

We have discussed [flattening of a multi-level linked list](#) where nodes have two pointers down and next. In the previous post, we flattened the linked list level wise. How to flatten a linked list when we always need to process down pointer before next at every node.

**Input:**

```
1 - 2 - 3 - 4
|
7 - 8 - 10 - 12
|   |
9   16   11
|   |
14  17 - 18 - 19 - 20
|           |
15 - 23           21
|
24
```

**Output:**

```
Linked List to be flattened to
1 - 2 - 7 - 9 - 14 - 15 - 23 - 24 - 8
- 16 - 17 - 18 - 19 - 20 - 21 - 10 -
11 - 12 - 3 - 4
Note : 9 appears before 8 (When we are
at a node, we process down pointer before
right pointer)
```

Source : Oracle Interview

If we take a closer look, we can notice that this problem is similar to [tree to linked list conversion](#). We recursively flatten a linked list with following steps.

- 1) If node is NULL, return NULL.
- 2) Store next node of current node (used in step 4).
- 3) Recursively flatten down list. While flattening, keep track of last visited node, so that the next list can be linked after it.
- 4) Recursively flatten next list (we get the next list from pointer stored in step 2) and attach it after last visited node.

Below is C++ implementation of above idea.

```
// C++ program to flatten a multilevel linked list
#include <bits/stdc++.h>
using namespace std;

// A Linked List Node
struct Node
{
    int data;
    struct Node *next;
    struct Node *down;
};

// Flattens a multi-level linked list depth wise
Node* flattenList(Node* node)
{
    // Base case
    if (node == NULL)
        return NULL;

    // To keep track of last visited node
    // (NOTE: This is static)
    static Node *last;
    last = node;

    // Store next pointer
    Node *next = node->next;

    // If down list exists, process it first
    // Add down list as next of current node
    if (node->down)
        node->next = flattenList(node->down);

    // If next exists, add it after the next
    // of last added node
    if (next)
        last->next = flattenList(next);
}
```

```
    return node;
}

// Utility method to print a linked list
void printFlattenNodes(Node* head)
{
    while (head)
    {
        printf("%d ", head->data);
        head = head->next;
    }
}

// Utility function to create a new node
Node* newNode(int new_data)
{
    Node* new_node = new Node;
    new_node->data = new_data;
    new_node->next = new_node->down = NULL;
    return new_node;
}

// Driver code
int main()
{
    // Creating above example list
    Node* head = newNode(1);
    head->next = newNode(2);
    head->next->next = newNode(3);
    head->next->next->next = newNode(4);
    head->next->next->down = newNode(7);
    head->next->next->down->down = newNode(9);
    head->next->next->down->down->down = newNode(14);
    head->next->next->down->down->down->down
                                = newNode(15);
    head->next->next->down->down->down->next
                                = newNode(23);
    head->next->next->down->down->down->next->down
                                = newNode(24);
    head->next->next->down->next = newNode(8);
    head->next->next->down->next->down = newNode(16);
    head->next->next->down->next->down->down = newNode(17);
    head->next->next->down->next->down->next
                                = newNode(18);
    head->next->next->down->next->down->next->next
                                = newNode(19);
    head->next->next->down->next->down->next->down->next
                                = newNode(20);
```

```
head->next->down->next->down->down->next->next->next->down  
= newNode(21);  
head->next->down->next->next = newNode(10);  
head->next->down->next->next->down = newNode(11);  
  
head->next->down->next->next->next = newNode(12);  
  
// Flatten list and print modified list  
head = flattenList(head);  
printFlattenNodes(head);  
  
return 0;  
}
```

Output:

```
1 2 7 9 14 15 23 24 8 16 17 18 19 20 21 10 11 12 3 4
```

## Source

<https://www.geeksforgeeks.org/flatten-a-multi-level-linked-list-set-2-depth-wise/>

# Chapter 95

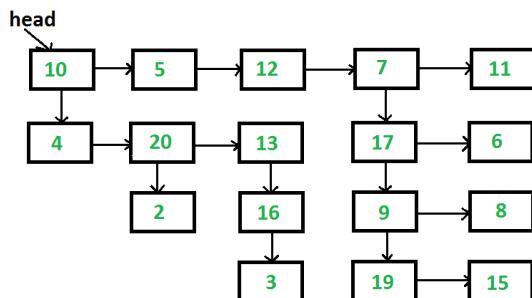
## Flatten a multilevel linked list

Flatten a multilevel linked list - GeeksforGeeks

Given a linked list where in addition to the next pointer, each node has a child pointer, which may or may not point to a separate list. These child lists may have one or more children of their own, and so on, to produce a multilevel data structure, as shown in below figure. You are given the head of the first level of the list. Flatten the list so that all the nodes appear in a single-level linked list. You need to flatten the list in way that all nodes at first level should come first, then nodes of second level, and so on.

Each node is a C struct with the following definition.

```
struct List
{
    int data;
    struct List *next;
    struct List *child;
};
```



The above list should be converted to 10->5->12->7->11->4->20->13->17->6->2->16->9->8->3->19->15

The problem clearly say that we need to flatten level by level. The idea of solution is, we start from first level, process all nodes one by one, if a node has a child, then we append the child at the end of list, otherwise we don't do anything. After the first level is processed, all next level nodes will be appended after first level. Same process is followed for the appended nodes.

- 1) Take "cur" pointer, which will point to head of the fist level of the list
- 2) Take "tail" pointer, which will point to end of the first level of the list
- 3) Repeat the below procedure while "curr" is not NULL.
  - I) if current node has a child then
    - a) append this new child list to the "tail"
 

```
tail->next = cur->child
```
    - b) find the last node of new child list and update "tail"
 

```
tmp = cur->child;
while (tmp->next != NULL)
    tmp = tmp->next;
tail = tmp;
```
  - II) move to the next node. i.e. cur = cur->next

Following is the implementation of the above algorithm.

## C

```
// Program to flatten list with next and child pointers
#include <stdio.h>
#include <stdlib.h>

// Macro to find number of elements in array
#define SIZE(arr) (sizeof(arr)/sizeof(arr[0]))

// A linked list node has data, next pointer and child pointer
struct Node
{
    int data;
    struct Node *next;
    struct Node *child;
};

// A utility function to create a linked list with n nodes. The data
// of nodes is taken from arr[]. All child pointers are set as NULL
struct Node *createList(int *arr, int n)
{
    struct Node *head = NULL;
    struct Node *p;

    int i;
```

```
for (i = 0; i < n; ++i) {
    if (head == NULL)
        head = p = (struct Node *)malloc(sizeof(*p));
    else {
        p->next = (struct Node *)malloc(sizeof(*p));
        p = p->next;
    }
    p->data = arr[i];
    p->next = p->child = NULL;
}
return head;
}

// A utility function to print all nodes of a linked list
void printList(struct Node *head)
{
    while (head != NULL) {
        printf("%d ", head->data);
        head = head->next;
    }
    printf("\n");
}

// This function creates the input list. The created list is same
// as shown in the above figure
struct Node *createList(void)
{
    int arr1[] = {10, 5, 12, 7, 11};
    int arr2[] = {4, 20, 13};
    int arr3[] = {17, 6};
    int arr4[] = {9, 8};
    int arr5[] = {19, 15};
    int arr6[] = {2};
    int arr7[] = {16};
    int arr8[] = {3};

    /* create 8 linked lists */
    struct Node *head1 = createList(arr1, SIZE(arr1));
    struct Node *head2 = createList(arr2, SIZE(arr2));
    struct Node *head3 = createList(arr3, SIZE(arr3));
    struct Node *head4 = createList(arr4, SIZE(arr4));
    struct Node *head5 = createList(arr5, SIZE(arr5));
    struct Node *head6 = createList(arr6, SIZE(arr6));
    struct Node *head7 = createList(arr7, SIZE(arr7));
    struct Node *head8 = createList(arr8, SIZE(arr8));

    /* modify child pointers to create the list shown above */
}
```

```
head1->child = head2;
head1->next->next->next->child = head3;
head3->child = head4;
head4->child = head5;
head2->next->child = head6;
head2->next->next->child = head7;
head7->child = head8;

/* Return head pointer of first linked list. Note that all nodes are
   reachable from head1 */
return head1;
}

/* The main function that flattens a multilevel linked list */
void flattenList(struct Node *head)
{
    /*Base case*/
    if (head == NULL)
        return;

    struct Node *tmp;

    /* Find tail node of first level linked list */
    struct Node *tail = head;
    while (tail->next != NULL)
        tail = tail->next;

    // One by one traverse through all nodes of first level
    // linked list till we reach the tail node
    struct Node *cur = head;
    while (cur != tail)
    {
        // If current node has a child
        if (cur->child)
        {
            // then append the child at the end of current list
            tail->next = cur->child;

            // and update the tail to new last node
            tmp = cur->child;
            while (tmp->next)
                tmp = tmp->next;
            tail = tmp;
        }

        // Change current node
        cur = cur->next;
    }
}
```

```
    }
}

// A driver program to test above functions
int main(void)
{
    struct Node *head = NULL;
    head = createList();
    flattenList(head);
    printList(head);
    return 0;
}
```

**Java**

```
// Java program to flatten linked list with next and child pointers

class LinkedList {

    static Node head;

    class Node {

        int data;
        Node next, child;

        Node(int d) {
            data = d;
            next = child = null;
        }
    }

    // A utility function to create a linked list with n nodes. The data
    // of nodes is taken from arr[]. All child pointers are set as NULL
    Node createList(int arr[], int n) {
        Node node = null;
        Node p = null;

        int i;
        for (i = 0; i < n; ++i) {
            if (node == null) {
                node = p = new Node(arr[i]);
            } else {
                p.next = new Node(arr[i]);
                p = p.next;
            }
            p.next = p.child = null;
        }
    }
}
```

```
        return node;
    }

// A utility function to print all nodes of a linked list
void printList(Node node) {
    while (node != null) {
        System.out.print(node.data + " ");
        node = node.next;
    }
    System.out.println("");
}

Node createList() {
    int arr1[] = new int[]{10, 5, 12, 7, 11};
    int arr2[] = new int[]{4, 20, 13};
    int arr3[] = new int[]{17, 6};
    int arr4[] = new int[]{9, 8};
    int arr5[] = new int[]{19, 15};
    int arr6[] = new int[]{2};
    int arr7[] = new int[]{16};
    int arr8[] = new int[]{3};

    /* create 8 linked lists */
    Node head1 = createList(arr1, arr1.length);
    Node head2 = createList(arr2, arr2.length);
    Node head3 = createList(arr3, arr3.length);
    Node head4 = createList(arr4, arr4.length);
    Node head5 = createList(arr5, arr5.length);
    Node head6 = createList(arr6, arr6.length);
    Node head7 = createList(arr7, arr7.length);
    Node head8 = createList(arr8, arr8.length);

    /* modify child pointers to create the list shown above */
    head1.child = head2;
    head1.next.next.next.child = head3;
    head3.child = head4;
    head4.child = head5;
    head2.next.child = head6;
    head2.next.next.child = head7;
    head7.child = head8;

    /* Return head pointer of first linked list. Note that all nodes are
       reachable from head1 */
    return head1;
}

/* The main function that flattens a multilevel linked list */
void flattenList(Node node) {
```

```
/*Base case*/
if (node == null) {
    return;
}

Node tmp = null;

/* Find tail node of first level linked list */
Node tail = node;
while (tail.next != null) {
    tail = tail.next;
}

// One by one traverse through all nodes of first level
// linked list till we reach the tail node
Node cur = node;
while (cur != tail) {

    // If current node has a child
    if (cur.child != null) {

        // then append the child at the end of current list
        tail.next = cur.child;

        // and update the tail to new last node
        tmp = cur.child;
        while (tmp.next != null) {
            tmp = tmp.next;
        }
        tail = tmp;
    }

    // Change current node
    cur = cur.next;
}
}

public static void main(String[] args) {
    LinkedList list = new LinkedList();
    head = list.createList();
    list.flattenList(head);
    list.printList(head);
}
}

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

Output:

```
10 5 12 7 11 4 20 13 17 6 2 16 9 8 3 19 15
```

Time Complexity: Since every node is visited at most twice, the time complexity is  $O(n)$  where  $n$  is the number of nodes in given linked list.

This article is compiled by **Narendra Kangalkar**. 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/flatten-a-linked-list-with-next-and-child-pointers/>

# Chapter 96

## Flattening a Linked List

Flattening a Linked List - GeeksforGeeks

Given a linked list where every node represents a linked list and contains two pointers of its type:

- (i) Pointer to next node in the main list (we call it ‘right’ pointer in below code)
  - (ii) Pointer to a linked list where this node is head (we call it ‘down’ pointer in below code).
- All linked lists are sorted. See the following example

```
5 -> 10 -> 19 -> 28
|   |   |
V   V   V
7   20   22   35
|           |
V           V
8           50   40
|
V           V
30          45
```

Write a function flatten() to flatten the lists into a single linked list. The flattened linked list should also be sorted. For example, for the above input list, output list should be 5->7->8->10->19->20->22->28->30->35->40->45->50.

The idea is to use Merge() process of [merge sort for linked lists](#). We use merge() to merge lists one by one. We recursively merge() the current list with already flattened list. The down pointer is used to link nodes of the flattened list.

Following are C and Java implementations.

C/C++

```
// C program for flattening a linked list
#include <stdio.h>
#include <stdlib.h>

// A Linked List Node
typedef struct Node
{
    int data;
    struct Node *right;
    struct Node *down;
} Node;

/* A utility function to insert a new node at the begining
   of linked list */
void push (Node** head_ref, int new_data)
{
    /* allocate node */
    Node* new_node = (Node *) malloc(sizeof(Node));
    new_node->right = NULL;

    /* put in the data */
    new_node->data = new_data;

    /* link the old list off the new node */
    new_node->down = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref) = new_node;
}

/* Function to print nodes in the flattened linked list */
void printList(Node *node)
{
    while (node != NULL)
    {
        printf("%d ", node->data);
        node = node->down;
    }
}

// A utility function to merge two sorted linked lists
Node* merge( Node* a, Node* b )
{
    // If first list is empty, the second list is result
    if (a == NULL)
        return b;

    // If second list is empty, the second list is result
    if (b == NULL)
```

```
if (b == NULL)
    return a;

// Compare the data members of head nodes of both lists
// and put the smaller one in result
Node* result;
if (a->data < b->data)
{
    result = a;
    result->down = merge( a->down, b );
}
else
{
    result = b;
    result->down = merge( a, b->down );
}

return result;
}

// The main function that flattens a given linked list
Node* flatten (Node* root)
{
    // Base cases
    if (root == NULL || root->right == NULL)
        return root;

    // Merge this list with the list on right side
    return merge( root, flatten(root->right) );
}

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

/* Let us create the following linked list
   5 -> 10 -> 19 -> 28
   |     |     |     |
   V     V     V     V
   7     20    22    35
   |             |     |
   V             V     V
   8             50    40
   |             |
   V             V
   30            45
*/
}
```

```
push( &root, 30 );
push( &root, 8 );
push( &root, 7 );
push( &root, 5 );

push( &( root->right ), 20 );
push( &( root->right ), 10 );

push( &( root->right->right ), 50 );
push( &( root->right->right ), 22 );
push( &( root->right->right ), 19 );

push( &( root->right->right->right ), 45 );
push( &( root->right->right->right ), 40 );
push( &( root->right->right->right ), 35 );
push( &( root->right->right->right ), 20 );

// Let us flatten the list
root = flatten(root);

// Let us print the flattened linked list
printList(root);

return 0;
}
```

### Java

```
// Java program for flattening a Linked List
class LinkedList
{
    Node head; // head of list

    /* Linked list Node*/
    class Node
    {
        int data;
        Node right, down;
        Node(int data)
        {
            this.data = data;
            right = null;
            down = null;
        }
    }

    // An utility function to merge two sorted linked lists
    Node merge(Node a, Node b)
```

```
{  
    // if first linked list is empty then second  
    // is the answer  
    if (a == null)      return b;  
  
    // if second linked list is empty then first  
    // is the result  
    if (b == null)      return a;  
  
    // compare the data members of the two linked lists  
    // and put the larger one in the result  
    Node result;  
  
    if (a.data < b.data)  
    {  
        result = a;  
        result.down = merge(a.down, b);  
    }  
  
    else  
    {  
        result = b;  
        result.down = merge(a, b.down);  
    }  
  
    return result;  
}  
  
Node flatten(Node root)  
{  
    // Base Cases  
    if (root == null || root.right == null)  
        return root;  
  
    // recur for list on right  
    root.right = flatten(root.right);  
  
    // now merge  
    root = merge(root, root.right);  
  
    // return the root  
    // it will be in turn merged with its left  
    return root;  
}  
  
/* Utility function to insert a node at begining of the  
linked list */  
Node push(Node head_ref, int data)
```

```

{
    /* 1 & 2: Allocate the Node &
       Put in the data*/
    Node new_node = new Node(data);

    /* 3. Make next of new Node as head */
    new_node.down = head_ref;

    /* 4. Move the head to point to new Node */
    head_ref = new_node;

    /*5. return to link it back */
    return head_ref;
}

void printList()
{
    Node temp = head;
    while (temp != null)
    {
        System.out.print(temp.data + " ");
        temp = temp.down;
    }
    System.out.println();
}

/* Drier program to test above functions */
public static void main(String args[])
{
    LinkedList L = new LinkedList();

    /* Let us create the following linked list
       5 -> 10 -> 19 -> 28
       |   |   |
       V   V   V   V
       7   20  22  35
       |           |
       V           V
       8           50  40
       |           |
       V           V
       30          45
    */
    L.head = L.push(L.head, 30);
    L.head = L.push(L.head, 8);
    L.head = L.push(L.head, 7);
    L.head = L.push(L.head, 5);
}

```

```
L.head.right = L.push(L.head.right, 20);
L.head.right = L.push(L.head.right, 10);

L.head.right.right = L.push(L.head.right.right, 50);
L.head.right.right = L.push(L.head.right.right, 22);
L.head.right.right = L.push(L.head.right.right, 19);

L.head.right.right.right = L.push(L.head.right.right.right, 45);
L.head.right.right.right = L.push(L.head.right.right.right, 40);
L.head.right.right.right = L.push(L.head.right.right.right, 35);
L.head.right.right.right = L.push(L.head.right.right.right, 20);

// flatten the list
L.head = L.flatten(L.head);

L.printList();
}

} /* This code is contributed by Rajat Mishra */
```

Output:

```
5 7 8 10 19 20 20 22 30 35 40 45 50
```

**Improved By :** [kaviarasu](#)

## Source

<https://www.geeksforgeeks.org/flattening-a-linked-list/>

## Chapter 97

# Function to check if a singly linked list is palindrome

Function to check if a singly linked list is palindrome - GeeksforGeeks

Given a singly linked list of characters, write a function that returns true if the given list is palindrome, else false.



### METHOD 1 (Use a Stack)

A simple solution is to use a stack of list nodes. This mainly involves three steps.

- 1) Traverse the given list from head to tail and push every visited node to stack.
- 2) Traverse the list again. For every visited node, pop a node from stack and compare data of popped node with currently visited node.
- 3) If all nodes matched, then return true, else false.

Time complexity of above method is  $O(n)$ , but it requires  $O(n)$  extra space. Following methods solve this with constant extra space.

### METHOD 2 (By reversing the list)

This method takes  $O(n)$  time and  $O(1)$  extra space.

- 1) Get the middle of the linked list.
- 2) Reverse the second half of the linked list.
- 3) Check if the first half and second half are identical.

- 4) Construct the original linked list by reversing the second half again and attaching it back to the first half

To divide the list in two halves, method 2 of [this post](#) is used.

When number of nodes are even, the first and second half contain exactly half nodes. The challenging thing in this method is to handle the case when number of nodes are odd. We don't want the middle node as part of any of the lists as we are going to compare them for equality. For odd case, we use a separate variable 'midnode'.

C

```
/* Program to check if a linked list is palindrome */
#include<stdio.h>
#include<stdlib.h>
#include<stdbool.h>

/* Link list node */
struct Node
{
    char data;
    struct Node* next;
};

void reverse(struct Node**);
bool compareLists(struct Node*, struct Node *);

/* Function to check if given linked list is
   palindrome or not */
bool isPalindrome(struct Node *head)
{
    struct Node *slow_ptr = head, *fast_ptr = head;
    struct Node *second_half, *prev_of_slow_ptr = head;
    struct Node *midnode = NULL; // To handle odd size list
    bool res = true; // initialize result

    if (head!=NULL && head->next!=NULL)
    {
        /* Get the middle of the list. Move slow_ptr by 1
           and fast_ptr by 2, slow_ptr will have the middle
           node */
        while (fast_ptr != NULL && fast_ptr->next != NULL)
        {
            fast_ptr = fast_ptr->next->next;

            /*We need previous of the slow_ptr for
               linked lists with odd elements */
            prev_of_slow_ptr = slow_ptr;
            slow_ptr = slow_ptr->next;
        }
    }
}
```

```

/* fast_ptr would become NULL when there are even elements in list.
   And not NULL for odd elements. We need to skip the middle node
   for odd case and store it somewhere so that we can restore the
   original list*/
if (fast_ptr != NULL)
{
    midnode = slow_ptr;
    slow_ptr = slow_ptr->next;
}

// Now reverse the second half and compare it with first half
second_half = slow_ptr;
prev_of_slow_ptr->next = NULL; // NULL terminate first half
reverse(&second_half); // Reverse the second half
res = compareLists(head, second_half); // compare

/* Construct the original list back */
reverse(&second_half); // Reverse the second half again

// If there was a mid node (odd size case) which
// was not part of either first half or second half.
if (midnode != NULL)
{
    prev_of_slow_ptr->next = midnode;
    midnode->next = second_half;
}
else prev_of_slow_ptr->next = second_half;
}
return res;
}

/* Function to reverse the linked list Note that this
   function may change the head */
void reverse(struct Node** head_ref)
{
    struct Node* prev    = NULL;
    struct Node* current = *head_ref;
    struct Node* next;
    while (current != NULL)
    {
        next  = current->next;
        current->next = prev;
        prev = current;
        current = next;
    }
    *head_ref = prev;
}

```

```
}

/* Function to check if two input lists have same data*/
bool compareLists(struct Node* head1, struct Node *head2)
{
    struct Node* temp1 = head1;
    struct Node* temp2 = head2;

    while (temp1 && temp2)
    {
        if (temp1->data == temp2->data)
        {
            temp1 = temp1->next;
            temp2 = temp2->next;
        }
        else return 0;
    }

    /* Both are empty return 1*/
    if (temp1 == NULL && temp2 == NULL)
        return 1;

    /* Will reach here when one is NULL
       and other is not */
    return 0;
}

/* Push a node to linked list. Note that this function
   changes the head */
void push(struct Node** head_ref, char new_data)
{
    /* allocate node */
    struct Node* new_node =
        (struct Node*) malloc(sizeof(struct Node));

    /* put in the data */
    new_node->data = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to pochar to the new node */
    (*head_ref) = new_node;
}

// A utility function to print a given linked list
void printList(struct node *ptr)
{
```

```
while (ptr != NULL)
{
    printf("%c->", ptr->data);
    ptr = ptr->next;
}
printf("NULL\n");

/* Drier program to test above function*/
int main()
{
    /* Start with the empty list */
    struct Node* head = NULL;
    char str[] = "abacaba";
    int i;

    for (i = 0; str[i] != '\0'; i++)
    {
        push(&head, str[i]);
        printList(head);
        isPalindrome(head)? printf("Is Palindrome\n\n"):
                           printf("Not Palindrome\n\n");
    }

    return 0;
}
```

### Java

```
/* Java program to check if linked list is palindrome */

class LinkedList
{
    Node head; // head of list
    Node slow_ptr, fast_ptr, second_half;

    /* Linked list Node*/
    class Node
    {
        char data;
        Node next;

        Node(char d)
        {
            data = d;
            next = null;
        }
    }
}
```

```
}

/* Function to check if given linked list is
   palindrome or not */
boolean isPalindrome(Node head)
{
    slow_ptr = head; fast_ptr = head;
    Node prev_of_slow_ptr = head;
    Node midnode = null; // To handle odd size list
    boolean res = true; // initialize result

    if (head != null && head.next != null)
    {
        /* Get the middle of the list. Move slow_ptr by 1
           and fast_ptr by 2, slow_ptr will have the middle
           node */
        while (fast_ptr != null && fast_ptr.next != null)
        {
            fast_ptr = fast_ptr.next.next;

            /*We need previous of the slow_ptr for
               linked lists with odd elements */
            prev_of_slow_ptr = slow_ptr;
            slow_ptr = slow_ptr.next;
        }

        /* fast_ptr would become NULL when there are even elements
           in the list and not NULL for odd elements. We need to skip
           the middle node for odd case and store it somewhere so that
           we can restore the original list */
        if (fast_ptr != null)
        {
            midnode = slow_ptr;
            slow_ptr = slow_ptr.next;
        }

        // Now reverse the second half and compare it with first half
        second_half = slow_ptr;
        prev_of_slow_ptr.next = null; // NULL terminate first half
        reverse(); // Reverse the second half
        res = compareLists(head, second_half); // compare

        /* Construct the original list back */
        reverse(); // Reverse the second half again

        if (midnode != null)
        {
            // If there was a mid node (odd size case) which
```

```
// was not part of either first half or second half.
prev_of_slow_ptr.next = midnode;
midnode.next = second_half;
} else
    prev_of_slow_ptr.next = second_half;
}
return res;
}

/* Function to reverse the linked list Note that this
function may change the head */
void reverse()
{
    Node prev = null;
    Node current = second_half;
    Node next;
    while (current != null)
    {
        next = current.next;
        current.next = prev;
        prev = current;
        current = next;
    }
    second_half = prev;
}

/* Function to check if two input lists have same data*/
boolean compareLists(Node head1, Node head2)
{
    Node temp1 = head1;
    Node temp2 = head2;

    while (temp1 != null && temp2 != null)
    {
        if (temp1.data == temp2.data)
        {
            temp1 = temp1.next;
            temp2 = temp2.next;
        } else
            return false;
    }

    /* Both are empty return 1*/
    if (temp1 == null && temp2 == null)
        return true;

    /* Will reach here when one is NULL
    and other is not */
}
```

```
        return false;
    }

/* Push a node to linked list. Note that this function
   changes the head */
public void push(char new_data)
{
    /* Allocate the Node &
       Put in the data */
    Node new_node = new Node(new_data);

    /* link the old list off the new one */
    new_node.next = head;

    /* Move the head to point to new Node */
    head = new_node;
}

// A utility function to print a given linked list
void printList(Node ptr)
{
    while (ptr != null)
    {
        System.out.print(ptr.data + "->");
        ptr = ptr.next;
    }
    System.out.println("NULL");
}

/* Driver program to test the above functions */
public static void main(String[] args) {

    /* Start with the empty list */
    LinkedList llist = new LinkedList();

    char str[] = {'a', 'b', 'a', 'c', 'a', 'b', 'a'};
    String string = new String(str);
    for (int i = 0; i< 7 ; i++) {
        llist.push(str[i]);
        llist.printList(llist.head);
        if (llist.isPalindrome(llist.head) != false)
        {
            System.out.println("Is Palindrome");
            System.out.println("");
        }
        else
        {
            System.out.println("Not Palindrome");
        }
    }
}
```

```
        System.out.println("");
    }
}
}
```

Output:

```
a->NULL
Palindrome
```

```
b->a->NULL
Not Palindrome
```

```
a->b->a->NULL
Is Palindrome
```

```
c->a->b->a->NULL
Not Palindrome
```

```
a->c->a->b->a->NULL
Not Palindrome
```

```
b->a->c->a->b->a->NULL
Not Palindrome
```

```
a->b->a->c->a->b->a->NULL
Is Palindrome
```

Time Complexity O(n)  
Auxiliary Space: O(1)

### METHOD 3 (Using Recursion)

Use two pointers left and right. Move right and left using recursion and check for following in each recursive call.

- 1) Sub-list is palindrome.
- 2) Value at current left and right are matching.

If both above conditions are true then return true.

The idea is to use function call stack as container. Recursively traverse till the end of list. When we return from last NULL, we will be at last node. The last node to be compared with first node of list.

In order to access first node of list, we need list head to be available in the last call of recursion. Hence we pass head also to the recursive function. If they both match we need to compare (2, n-2) nodes. Again when recursion falls back to (n-2)nd node, we need reference to 2nd node from head. We advance the head pointer in previous call, to refer to next node in the list.

However, the trick in identifying double pointer. Passing single pointer is as good as pass-by-value, and we will pass the same pointer again and again. We need to pass the address of head pointer for reflecting the changes in parent recursive calls.

Thanks to **Sharad Chandra** for suggesting this approach.

C

```
// Recursive program to check if a given linked list is palindrome
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

/* Link list node */
struct node
{
    char data;
    struct node* next;
};

// Initial parameters to this function are &head and head
bool isPalindromeUtil(struct node **left, struct node *right)
{
    /* stop recursion when right becomes NULL */
    if (right == NULL)
        return true;

    /* If sub-list is not palindrome then no need to
       check for current left and right, return false */
    bool isp = isPalindromeUtil(left, right->next);
    if (isp == false)
        return false;

    /* Check values at current left and right */
    bool isp1 = (right->data == (*left)->data);

    /* Move left to next node */
    *left = (*left)->next;

    return isp1;
}

// A wrapper over isPalindromeUtil()
bool isPalindrome(struct node *head)
{
    isPalindromeUtil(&head, head);
}

/* Push a node to linked list. Note that this function
```

```
changes the head */
void push(struct node** head_ref, char new_data)
{
    /* allocate node */
    struct node* new_node =
        (struct node*) malloc(sizeof(struct node));

    /* put in the data */
    new_node->data = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to pochar to the new node */
    (*head_ref) = new_node;
}

// A utility function to print a given linked list
void printList(struct node *ptr)
{
    while (ptr != NULL)
    {
        printf("%c->", ptr->data);
        ptr = ptr->next;
    }
    printf("NULL\n");
}

/* Drier program to test above function*/
int main()
{
    /* Start with the empty list */
    struct node* head = NULL;
    char str[] = "abacaba";
    int i;

    for (i = 0; str[i] != '\0'; i++)
    {
        push(&head, str[i]);
        printList(head);
        isPalindrome(head)? printf("Is Palindrome\n\n"):
                           printf("Not Palindrome\n\n");
    }

    return 0;
}
```

Java

```
/* Java program to check if linked list is palindrome recursively */

class LinkedList
{
    Node head; // head of list
    Node left;

    /* Linked list Node*/
    class Node
    {
        char data;
        Node next;

        Node(char d)
        {
            data = d;
            next = null;
        }
    }

    // Initial parameters to this function are &head and head
    boolean isPalindromeUtil(Node right)
    {
        left = head;

        /* stop recursion when right becomes NULL */
        if (right == null)
            return true;

        /* If sub-list is not palindrome then no need to
           check for current left and right, return false */
        boolean isp = isPalindromeUtil(right.next);
        if (isp == false)
            return false;

        /* Check values at current left and right */
        boolean isp1 = (right.data == (left).data);

        /* Move left to next node */
        left = left.next;

        return isp1;
    }

    // A wrapper over isPalindromeUtil()
    boolean isPalindrome(Node head)
    {
        boolean result = isPalindromeUtil(head);
```

```
        return result;
    }

/* Push a node to linked list. Note that this function
   changes the head */
public void push(char new_data)
{
    /* Allocate the Node &
       Put in the data */
    Node new_node = new Node(new_data);

    /* link the old list off the new one */
    new_node.next = head;

    /* Move the head to point to new Node */
    head = new_node;
}

// A utility function to print a given linked list
void printList(Node ptr)
{
    while (ptr != null)
    {
        System.out.print(ptr.data + "->");
        ptr = ptr.next;
    }
    System.out.println("NULL");
}

/* Driver program to test the above functions */
public static void main(String[] args)
{
    /* Start with the empty list */
    LinkedList llist = new LinkedList();

    char str[] = {'a', 'b', 'a', 'c', 'a', 'b', 'a'};
    String string = new String(str);
    for (int i = 0; i < 7; i++)
    {
        llist.push(str[i]);
        llist.printList(llist.head);
        if (llist.isPalindrome(llist.head) != false)
        {
            System.out.println("Is Palindrome");
            System.out.println("");
        }
        else
        {
```

```
        System.out.println("Not Palindrome");
        System.out.println("");
    }
}
}

// This code has been contributed by Mayank Jaiswal(mayank_24)
```

Output:

```
a->NULL
Not Palindrome
```

```
b->a->NULL
Not Palindrome
```

```
a->b->a->NULL
Is Palindrome
```

```
c->a->b->a->NULL
Not Palindrome
```

```
a->c->a->b->a->NULL
Not Palindrome
```

```
b->a->c->a->b->a->NULL
Not Palindrome
```

```
a->b->a->c->a->b->a->NULL
Is Palindrome
```

Time Complexity: O(n)

Auxiliary Space: O(n) if Function Call Stack size is considered, otherwise O(1).

Please comment if you find any bug in the programs/algorithms or a better way to do the same.

## Source

<https://www.geeksforgeeks.org/function-to-check-if-a-singly-linked-list-is-palindrome/>

## Chapter 98

# Generic Linked List in C

Generic Linked List in C - GeeksforGeeks

Unlike [C++](#) and [Java](#), [C](#) doesn't support generics. How to create a linked list in C that can be used for any data type? In C, we can use [void pointer](#) and function pointer to implement the same functionality. The great thing about void pointer is it can be used to point to any data type. Also, size of all types of pointers is always same, so we can always allocate a linked list node. Function pointer is needed process actual content stored at address pointed by void pointer.

Following is a sample C code to demonstrate working of generic linked list.

```
// C program for generic linked list
#include<stdio.h>
#include<stdlib.h>

/* A linked list node */
struct Node
{
    // Any data type can be stored in this node
    void *data;

    struct Node *next;
};

/* Function to add a node at the beginning of Linked List.
   This function expects a pointer to the data to be added
   and size of the data type */
void push(struct Node** head_ref, void *new_data, size_t data_size)
{
    // Allocate memory for node
    struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));

    new_node->data = malloc(data_size);
```

```

new_node->next = (*head_ref);

// Copy contents of new_data to newly allocated memory.
// Assumption: char takes 1 byte.
int i;
for (i=0; i<data_size; i++)
    *(char *)(new_node->data + i) = *(char *)(new_data + i);

// Change head pointer as new node is added at the beginning
(*head_ref) = new_node;
}

/* Function to print nodes in a given linked list. fpitr is used
   to access the function to be used for printing current node data.
   Note that different data types need different specifier in printf() */
void printList(struct Node *node, void (*fpitr)(void *))
{
    while (node != NULL)
    {
        (*fpitr)(node->data);
        node = node->next;
    }
}

// Function to print an integer
void printInt(void *n)
{
    printf(" %d", *(int *)n);
}

// Function to print a float
void printFloat(void *f)
{
    printf(" %f", *(float *)f);
}

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

    // Create and print an int linked list
    unsigned int_size = sizeof(int);
    int arr[] = {10, 20, 30, 40, 50}, i;
    for (i=4; i>=0; i--)
        push(&start, &arr[i], int_size);
    printf("Created integer linked list is \n");
    printList(start, printInt);
}

```

```
// Create and print a float linked list
unsigned float_size = sizeof(float);
start = NULL;
float arr2[] = {10.1, 20.2, 30.3, 40.4, 50.5};
for (i=4; i>=0; i--)
    push(&start, &arr2[i], float_size);
printf("\n\nCreated float linked list is \n");
printList(start, printFloat);

return 0;
}
```

Output:

```
Created integer linked list is
10 20 30 40 50

Created float linked list is
10.100000 20.200001 30.299999 40.400002 50.500000
```

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/generic-linked-list-in-c-2/>

## Chapter 99

# Given a linked list of line segments, remove middle points

Given a linked list of line segments, remove middle points - GeeksforGeeks

Given a linked list of co-ordinates where adjacent points either form a vertical line or a horizontal line. Delete points from the linked list which are in the middle of a horizontal or vertical line.

Examples:

Input:  $(0,10) \rightarrow (1,10) \rightarrow (5,10) \rightarrow (7,10)$   
                  |  
                   $(7,5) \rightarrow (20,5) \rightarrow (40,5)$   
Output: Linked List should be changed to following  
 $(0,10) \rightarrow (7,10)$   
                  |  
                   $(7,5) \rightarrow (40,5)$

The given linked list represents a horizontal line from  $(0,10)$  to  $(7, 10)$  followed by a vertical line from  $(7, 10)$  to  $(7, 5)$ , followed by a horizontal line from  $(7, 5)$  to  $(40, 5)$ .

Input:  $(2,3) \rightarrow (4,3) \rightarrow (6,3) \rightarrow (10,3) \rightarrow (12,3)$   
Output: Linked List should be changed to following  
 $(2,3) \rightarrow (12,3)$   
There is only one vertical line, so all middle points are removed.

Source: [Microsoft Interview Experience](#)

The idea is to keep track of current node, next node and next-next node. While the next node is same as next-next node, keep deleting the next node. In this complete procedure we need to keep an eye on shifting of pointers and checking for NULL values.

Following are C/C++ and Java implementations of above idea.

### C/C++

```
// C program to remove intermediate points in a linked list
// that represents horizontal and vertical line segments
#include <stdio.h>
#include <stdlib.h>

// Node has 3 fields including x, y coordinates and a pointer
// to next node
struct Node
{
    int x, y;
    struct Node *next;
};

/* Function to insert a node at the beginning */
void push(struct Node ** head_ref, int x,int y)
{
    struct Node* new_node =
        (struct Node*) malloc(sizeof(struct Node));
    new_node->x = x;
    new_node->y = y;
    new_node->next = (*head_ref);
    (*head_ref) = new_node;
}

/* Utility function to print a singly linked list */
void printList(struct Node *head)
{
    struct Node *temp = head;
    while (temp != NULL)
    {
        printf("(%d,%d)-> ", temp->x,temp->y);
        temp = temp->next;
    }
    printf("\n");
}

// Utility function to remove Next from linked list
// and link nodes after it to head
void deleteNode(struct Node *head, struct Node *Next)
{
    head->next = Next->next;
    Next->next = NULL;
    free(Next);
}
```

```
}  
  
// This function deletes middle nodes in a sequence of  
// horizontal and vertical line segments represented by  
// linked list.  
struct Node* deleteMiddle(struct Node *head)  
{  
    // If only one node or no node...Return back  
    if (head==NULL || head->next ==NULL || head->next->next==NULL)  
        return head;  
  
    struct Node* Next = head->next;  
    struct Node *NextNext = Next->next ;  
  
    // Check if this is a vertical line or horizontal line  
    if (head->x == Next->x)  
    {  
        // Find middle nodes with same x value, and delete them  
        while (NextNext !=NULL && Next->x==NextNext->x)  
        {  
            deleteNode(head, Next);  
  
            // Update Next and NextNext for next iteration  
            Next = NextNext;  
            NextNext = NextNext->next;  
        }  
    }  
    else if (head->y==Next->y) // If horizontal line  
    {  
        // Find middle nodes with same y value, and delete them  
        while (NextNext !=NULL && Next->y==NextNext->y)  
        {  
            deleteNode(head, Next);  
  
            // Update Next and NextNext for next iteration  
            Next = NextNext;  
            NextNext = NextNext->next;  
        }  
    }  
    else // Adjacent points must have either same x or same y  
    {  
        puts("Given linked list is not valid");  
        return NULL;  
    }  
  
    // Recur for next segment  
    deleteMiddle(head->next);  
}
```

```
    return head;
}

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

    push(&head, 40,5);
    push(&head, 20,5);
    push(&head, 10,5);
    push(&head, 10,8);
    push(&head, 10,10);
    push(&head, 3,10);
    push(&head, 1,10);
    push(&head, 0,10);
    printf("Given Linked List: \n");
    printList(head);

    if (deleteMiddle(head) != NULL);
    {
        printf("Modified Linked List: \n");
        printList(head);
    }
    return 0;
}
```

### Java

```
// Java program to remove middle points in a linked list of
// line segments,
class LinkedList
{
    Node head; // head of list

    /* Linked list Node*/
    class Node
    {
        int x,y;
        Node next;
        Node(int x, int y)
        {
            this.x = x;
            this.y = y;
            next = null;
        }
    }
}
```

```
// This function deletes middle nodes in a sequence of
// horizontal and vertical line segments represented
// by linked list.
Node deleteMiddle()
{
    // If only one node or no node...Return back
    if (head == null || head.next == null ||
        head.next.next == null)
        return head;

    Node Next = head.next;
    Node NextNext = Next.next;

    // check if this is vertical or horizontal line
    if (head.x == Next.x)
    {
        // Find middle nodes with same value as x and
        // delete them.
        while (NextNext != null && Next.x == NextNext.x)
        {
            head.next = Next.next;
            Next.next = null;

            // Update NextNext for the next iteration
            Next = NextNext;
            NextNext = NextNext.next;
        }
    }

    // if horizontal
    else if (head.y == Next.y)
    {
        // find middle nodes with same value as y and
        // delete them
        while (NextNext != null && Next.y == NextNext.y)
        {
            head.next = Next.next;
            Next.next = null;

            // Update NextNext for the next iteration
            Next = NextNext;
            NextNext = NextNext.next;
        }
    }

    // Adjacent points should have same x or same y
    else
    {
```

```
System.out.println("Given list is not valid");
    return null;
}

// recur for other segment

// temporarily store the head and move head forward.
Node temp = head;
head = head.next;

// call deleteMiddle() for next segment
this.deleteMiddle();

// restore head
head = temp;

// return the head
return head;
}

/* Given a reference (pointer to pointer) to the head
   of a list and an int, push a new node on the front
   of the list. */
void push(int x, int y)
{
    /* 1 & 2: Allocate the Node &
       Put in the data*/
    Node new_node = new Node(x,y);

    /* 3. Make next of new Node as head */
    new_node.next = head;

    /* 4. Move the head to point to new Node */
    head = new_node;
}

void printList()
{
    Node temp = head;
    while (temp != null)
    {
        System.out.print("(" + temp.x + ", " + temp.y + ") ->");
        temp = temp.next;
    }
    System.out.println();
}
```

```
/* Drier program to test above functions */
public static void main(String args[])
{
    LinkedList llist = new LinkedList();

    llist.push(40,5);
    llist.push(20,5);
    llist.push(10,5);
    llist.push(10,8);
    llist.push(10,10);
    llist.push(3,10);
    llist.push(1,10);
    llist.push(0,10);

    System.out.println("Given list");
    llist.printList();

    if (llist.deleteMiddle() != null)
    {
        System.out.println("Modified Linked List is");
        llist.printList();
    }
}
} /* This code is contributed by Rajat Mishra */
```

### Python

```
# Python program to remove middle points in a linked list of
# line segments,
class LinkedList(object):
    def __init__(self):
        self.head = None

    # Linked list Node
    class Node(object):
        def __init__(self, x, y):
            self.x = x
            self.y = y
            self.next = None

    # This function deletes middle nodes in a sequence of
    # horizontal and vertical line segments represented
    # by linked list.
    def deleteMiddle(self):
        # If only one node or no node...Return back
        if self.head == None or self.head.next == None or self.head.next.next == None:
            return self.head
```

```
Next = self.head.next
NextNext = Next.next
# check if this is vertical or horizontal line
if self.head.x == Next.x:
    # Find middle nodes with same value as x and
    # delete them.
    while NextNext != None and Next.x == NextNext.x:
        self.head.next = Next.next
        Next.next = None
        # Update NextNext for the next iteration
        Next = NextNext
        NextNext = NextNext.next
elif self.head.y == Next.y:
    # find middle nodes with same value as y and
    # delete them
    while NextNext != None and Next.y == NextNext.y:
        self.head.next = Next.next
        Next.next = None
        # Update NextNext for the next iteration
        Next = NextNext
        NextNext = NextNext.next
else:
    # Adjacent points should have same x or same y
    print "Given list is not valid"
    return None
# recur for other segment
# temporarily store the head and move head forward.
temp = self.head
self.head = self.head.next
# call deleteMiddle() for next segment
self.deleteMiddle()
# restore head
self.head = temp
# return the head
return self.head

# Given a reference (pointer to pointer) to the head
# of a list and an int, push a new node on the front
# of the list.
def push(self, x, y):
    # 1 & 2: Allocate the Node &
    # Put in the data
    new_node = self.Node(x, y)
    # 3. Make next of new Node as head
    new_node.next = self.head
    # 4. Move the head to point to new Node
    self.head = new_node
```

```
def printList(self):
    temp = self.head
    while temp != None:
        print "(" + str(temp.x) + "," + str(temp.y) + ")->",
        temp = temp.next
    print ''

# Driver program
llist = LinkedList()
llist.push(40,5)
llist.push(20,5)
llist.push(10,5)
llist.push(10,8)
llist.push(10,10)
llist.push(3,10)
llist.push(1,10)
llist.push(0,10)

print "Given list"
llist.printList()

if llist.deleteMiddle() != None:
    print "Modified Linked List is"
    llist.printList()

# This code is contributed by BHAVYA JAIN
```

Output:

```
Given Linked List:
(0,10)-> (1,10)-> (3,10)-> (10,10)-> (10,8)-> (10,5)-> (20,5)-> (40,5)->
Modified Linked List:
(0,10)-> (10,10)-> (10,5)-> (40,5)->
```

Time Complexity of the above solution is  $O(n)$  where  $n$  is number of nodes in given linked list.

**Exercise:**

The above code is recursive, write an iterative code for the same problem. Please see below for solution.

[Iterative approach for removing middle points in a linked list of line segments](#)

This article is contributed by Sanket Jain. 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-line-segments-remove-middle-points/>

## Chapter 100

**Given a linked list which is sorted, how will you insert in sorted way**

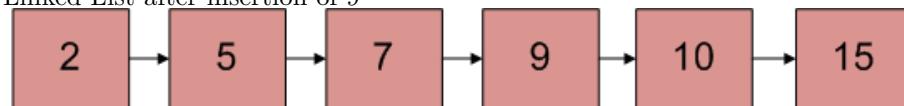
Given a linked list which is sorted, how will you insert in sorted way - GeeksforGeeks

Given a sorted linked list and a value to insert, write a function to insert the value in sorted way.

Initial Linked List



Linked List after insertion of 9



**Algorithm:**

Let input linked list is sorted in increasing order.

- 1) If Linked list is empty then make the node as head and return it.
- 2) If value of the node to be inserted is smaller than value of head node, then insert the node at start and make it head.
- 3) In a loop, find the appropriate node after which the input node (let 9) is to be inserted. To find the appropriate node start from head, keep moving until you reach a node GN (10 in

the below diagram) who's value is greater than the input node. The node just before GN is the appropriate node (7).

- 4) Insert the node (9) after the appropriate node (7) found in step 3.

**Implementation:**

C/C++

```
/* Program to insert in a sorted list */
#include<stdio.h>
#include<stdlib.h>

/* Link list node */
struct Node
{
    int data;
    struct Node* next;
};

/* function to insert a new_node in a list. Note that this
   function expects a pointer to head_ref as this can modify the
   head of the input linked list (similar to push())*/
void sortedInsert(struct Node** head_ref, struct Node* new_node)
{
    struct Node* current;
    /* Special case for the head end */
    if (*head_ref == NULL || (*head_ref)->data >= new_node->data)
    {
        new_node->next = *head_ref;
        *head_ref = new_node;
    }
    else
    {
        /* Locate the node before the point of insertion */
        current = *head_ref;
        while (current->next!=NULL &&
               current->next->data < new_node->data)
        {
            current = current->next;
        }
        new_node->next = current->next;
        current->next = new_node;
    }
}
```

```
/* BELOW FUNCTIONS ARE JUST UTILITY TO TEST sortedInsert */

/* A utility function to create a new node */
struct Node *newNode(int new_data)
{
    /* allocate node */
    struct Node* new_node =
        (struct Node*) malloc(sizeof(struct Node));

    /* put in the data */
    new_node->data = new_data;
    new_node->next = NULL;

    return new_node;
}

/* Function to print linked list */
void printList(struct Node *head)
{
    struct Node *temp = head;
    while(temp != NULL)
    {
        printf("%d ", temp->data);
        temp = temp->next;
    }
}

/* Drier program to test count function*/
int main()
{
    /* Start with the empty list */
    struct Node* head = NULL;
    struct Node *new_node = newNode(5);
    sortedInsert(&head, new_node);
    new_node = newNode(10);
    sortedInsert(&head, new_node);
    new_node = newNode(7);
    sortedInsert(&head, new_node);
    new_node = newNode(3);
    sortedInsert(&head, new_node);
    new_node = newNode(1);
    sortedInsert(&head, new_node);
    new_node = newNode(9);
    sortedInsert(&head, new_node);
    printf("\n Created Linked List\n");
    printList(head);
```

```
    return 0;
}
```

### Java

```
// Java Program to insert in a sorted list
class LinkedList
{
    Node head; // head of list

    /* Linked list Node*/
    class Node
    {
        int data;
        Node next;
        Node(int d) {data = d; next = null; }
    }

    /* function to insert a new_node in a list. */
    void sortedInsert(Node new_node)
    {
        Node current;

        /* Special case for head node */
        if (head == null || head.data >= new_node.data)
        {
            new_node.next = head;
            head = new_node;
        }
        else {

            /* Locate the node before point of insertion. */
            current = head;

            while (current.next != null &&
                   current.next.data < new_node.data)
                current = current.next;

            new_node.next = current.next;
            current.next = new_node;
        }
    }

    /*Utility functions*/

    /* Function to create a node */
    Node newNode(int data)
    {
```

```
Node x = new Node(data);
return x;
}

/* Function to print linked list */
void printList()
{
    Node temp = head;
    while (temp != null)
    {
        System.out.print(temp.data+" ");
        temp = temp.next;
    }
}

/* Driver function to test above methods */
public static void main(String args[])
{
    LinkedList llist = new LinkedList();
    Node new_node;
    new_node = llist.newNode(5);
    llist.sortedInsert(new_node);
    new_node = llist.newNode(10);
    llist.sortedInsert(new_node);
    new_node = llist.newNode(7);
    llist.sortedInsert(new_node);
    new_node = llist.newNode(3);
    llist.sortedInsert(new_node);
    new_node = llist.newNode(1);
    llist.sortedInsert(new_node);
    new_node = llist.newNode(9);
    llist.sortedInsert(new_node);
    System.out.println("Created Linked List");
    llist.printList();
}
}

/* This code is contributed by Rajat Mishra */
```

### Python

```
# Python program to insert in sorted list

# Node class
class Node:

    # Constructor to initialize the node object
    def __init__(self, data):
        self.data = data
```

```
        self.next = None

class LinkedList:

    # Function to initialize head
    def __init__(self):
        self.head = None

    def sortedInsert(self, new_node):

        # Special case for the empty linked list
        if self.head is None:
            new_node.next = self.head
            self.head = new_node

        # Special case for head at end
        elif self.head.data >= new_node.data:
            new_node.next = self.head
            self.head = new_node

        else :

            # Locate the node before the point of insertion
            current = self.head
            while(current.next is not None and
                  current.next.data < new_node.data):
                current = current.next

            new_node.next = current.next
            current.next = new_node

    # Function to insert a new node at the beginning
    def push(self, new_data):
        new_node = Node(new_data)
        new_node.next = self.head
        self.head = new_node

    # Utility function to print the linked LinkedList
    def printList(self):
        temp = self.head
        while(temp):
            print temp.data,
            temp = temp.next

# Driver program
llist = LinkedList()
new_node = Node(5)
```

```
llist.sortedInsert(new_node)
new_node = Node(10)
llist.sortedInsert(new_node)
new_node = Node(7)
llist.sortedInsert(new_node)
new_node = Node(3)
llist.sortedInsert(new_node)
new_node = Node(1)
llist.sortedInsert(new_node)
new_node = Node(9)
llist.sortedInsert(new_node)
print "Create Linked List"
llist.printList()

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

Output:

```
Created Linked List
1 3 5 7 9 10
```

#### Shorter Implementation using double pointers

Thanks to Murat M Ozturk for providing this solution. Please see Murat M Ozturk's comment below for complete function. The code uses double pointer to keep track of the next pointer of the previous node (after which new node is being inserted).

Note that below line in code changes *current* to have address of next pointer in a node.

```
current = &((*current)->next);
```

Also, note below comments.

```
/* Copies the value-at-address current to
   new_node's next pointer*/
new_node->next = *current;

/* Fix next pointer of the node (using it's address)
   after which new_node is being inserted */
*current = new_node;
```

**Time Complexity:**  $O(n)$

**References:**

<http://cslibrary.stanford.edu/105/LinkedListProblems.pdf>

#### Source

<https://www.geeksforgeeks.org/given-a-linked-list-which-is-sorted-how-will-you-insert-in-sorted-way/>

# Chapter 101

## Given a linked list, reverse alternate nodes and append at the end

Given a linked list, reverse alternate nodes and append at the end - GeeksforGeeks

Given a linked list, reverse alternate nodes and append them to end of list. Extra allowed space is O(1)

Examples

Input List: 1->2->3->4->5->6  
Output List: 1->3->5->6->4->2

Input List: 12->14->16->18->20  
Output List: 12->16->20->18->14

The idea is to maintain two linked lists, one list of all odd positioned nodes (1, 3, 5 in above example) and other list of all even positioned nodes (6, 4 and 2 in above example). Following are detailed steps.

1) Traverse the given linked list which is considered as odd list. Do following for every visited node.

.....a) If the node is even node, remove it from odd list and add it to the front of even node list. Nodes are added at front to keep the reverse order.

2) Append the even node list at the end of odd node list.

C

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

```
/* A linked list node */
struct Node
{
    int data;
    struct Node *next;
};

/* Function to reverse all even positioned node and append at the end
   odd is the head node of given linked list */
void rearrange(struct Node *odd)
{
    // If linked list has less than 3 nodes, no change is required
    if (odd == NULL || odd->next == NULL || odd->next->next == NULL)
        return;

    // even points to the beginning of even list
    struct Node *even = odd->next;

    // Remove the first even node
    odd->next = odd->next->next;

    // odd points to next node in odd list
    odd = odd->next;

    // Set terminator for even list
    even->next = NULL;

    // Traverse the list
    while (odd && odd->next)
    {
        // Store the next node in odd list
        struct Node *temp = odd->next->next;

        // Link the next even node at the beginning of even list
        odd->next->next = even;
        even = odd->next;

        // Remove the even node from middle
        odd->next = temp;

        // Move odd to the next odd node
        if (temp != NULL)
            odd = temp;
    }

    // Append the even list at the end of odd list
    odd->next = even;
}
```

```
/* Function to add a node at the beginning of Linked List */
void push(struct Node** head_ref, int new_data)
{
    struct Node* new_node = (struct Node*) malloc(sizeof(struct Node));
    new_node->data = new_data;
    new_node->next = (*head_ref);
    (*head_ref) = new_node;
}

/* Function to print nodes in a given linked list */
void printList(struct Node *node)
{
    while (node != NULL)
    {
        printf("%d ", node->data);
        node = node->next;
    }
}

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

    /* The constructed linked list is:
       1->2->3->4->5->6->7 */
    push(&start, 7);
    push(&start, 6);
    push(&start, 5);
    push(&start, 4);
    push(&start, 3);
    push(&start, 2);
    push(&start, 1);

    printf("\n Linked list before calling rearrange() ");
    printList(start);

    rearrange(start);

    printf("\n Linked list after calling rearrange() ");
    printList(start);

    return 0;
}
```

Java

```
// Java program to reverse alternate nodes of a linked list
// and append at the end

class LinkedList {

    static Node head;

    static class Node {

        int data;
        Node next;

        Node(int item) {
            data = item;
            next = null;
        }
    }

    /* Function to reverse all even positioned node and append at the end
     odd is the head node of given linked list */
    void rearrange(Node odd) {

        // If linked list has less than 3 nodes, no change is required
        if (odd == null || odd.next == null || odd.next.next == null) {
            return;
        }

        // even points to the beginning of even list
        Node even = odd.next;

        // Remove the first even node
        odd.next = odd.next.next;

        // odd points to next node in odd list
        odd = odd.next;

        // Set terminator for even list
        even.next = null;

        // Traverse the list
        while (odd != null && odd.next != null) {

            // Store the next node in odd list
            Node temp = odd.next.next;

            // Link the next even node at the beginning of even list
            odd.next.next = even;
            even = odd.next;
        }
    }
}
```

```
// Remove the even node from middle
odd.next = temp;

// Move odd to the next odd node
if (temp != null) {
    odd = temp;
}
}

// Append the even list at the end of odd list
odd.next = even;
}

/* Function to print nodes in a given linked list */
void printList(Node node) {
    while (node != null) {
        System.out.print(node.data + " ");
        node = node.next;
    }
}

public static void main(String[] args) {
    LinkedList list = new LinkedList();
    list.head = new Node(1);
    list.head.next = new Node(2);
    list.head.next.next = new Node(3);
    list.head.next.next.next = new Node(4);
    list.head.next.next.next.next = new Node(5);
    list.head.next.next.next.next.next = new Node(6);
    list.head.next.next.next.next.next.next = new Node(7);

    System.out.println("Linked list before calling rearrange : ");
    list.printList(head);

    System.out.println("");
    list.rearrange(head);

    System.out.println("Linked list after calling rearrange : ");
    list.printList(head);
}

}
```

### Python

```
# Python program to reverse alternate nodes and append
# at end
```

```
# Extra space allowed - O(1)

# Node Class
class Node:

    # Constructor to initialize the node object
    def __init__(self, data):
        self.data = data
        self.next = None

# Linked list class contains node object
class LinkedList:

    # Constructor to initialize head
    def __init__(self):
        self.head = None

    # Function to insert a new node at the beginning
    def push(self, new_data):
        new_node = Node(new_data)
        new_node.next = self.head
        self.head = new_node

    def printList(self):
        temp = self.head
        while(temp):
            print temp.data,
            temp = temp.next

    def rearrange(self):

        # If linked list has less than 3 nodes, no change
        # is required
        odd = self.head
        if (odd is None or odd.next is None or
            odd.next.next is None):
            return

        # Even points to the beginning of even list
        even = odd.next

        # Remove the first even node
        odd.next = odd.next.next

        # Odd points to next node in odd list
        odd = odd.next
```

```
# Set terminator for even list
even.next = None

# Traverse the list
while (odd and odd.next):
    # Store the next node in odd list
    temp = odd.next.next

    # Link the next even node at the beginning
    # of even list
    odd.next.next = even
    even = odd.next

    # Remove the even node from middle
    odd.next = temp

    # Move odd to the next odd node
    if temp is not None:
        odd = temp

# Append the even list at the end of odd list
odd.next = even

# Code execution starts here
if __name__ == '__main__':
    start = LinkedList()

#The constructed linked list is ;
# 1->2->3->4->5->6->7
start.push(7)
start.push(6)
start.push(5)
start.push(4)
start.push(3)
start.push(2)
start.push(1)

print "Linked list before calling  rearrange() "
start.printList()

start.rearrange()

print "\nLinked list after calling  rearrange()"
start.printList()

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

Output:

```
Linked list before calling rearrange() 1 2 3 4 5 6 7  
Linked list after calling rearrange() 1 3 5 7 6 4 2
```

Time Complexity: The above code simply traverses the given linked list. So time complexity is  $O(n)$

Auxiliary Space:  $O(1)$

This article is contributed 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/given-linked-list-reverse-alternate-nodes-append-end/>

## Chapter 102

# Given only a pointer to a node to be deleted in a singly linked list, how do you delete it?

Given only a pointer to a node to be deleted in a singly linked list, how do you delete it? - GeeksforGeeks

A **simple solution** is to traverse the linked list until you find the node you want to delete. But this solution requires pointer to the head node which contradicts the problem statement.

A **fast solution** is to copy the data from the next node to the node to be deleted and delete the next node. Something like following.

```
struct Node *temp = node_ptr->next;
node_ptr->data = temp->data;
node_ptr->next = temp->next;
free(temp);
```

**Program:**

C

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

/* Link list node */
struct Node
{
```

```
int data;
struct Node* next;
};

/* Given a reference (pointer to pointer) to the head
of a list and an int, push a new node on the front
of the list. */
void push(struct Node** head_ref, int new_data)
{
    /* allocate node */
    struct Node* new_node =
        (struct Node*) malloc(sizeof(struct Node));

    /* put in the data */
    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;
}

void printList(struct Node *head)
{
    struct Node *temp = head;
    while(temp != NULL)
    {
        printf("%d ", temp->data);
        temp = temp->next;
    }
}

void deleteNode(struct Node *node_ptr)
{
    struct Node *temp = node_ptr->next;
    node_ptr->data = temp->data;
    node_ptr->next = temp->next;
    free(temp);
}

/* Driver program to test above function*/
int main()
{
    /* Start with the empty list */
    struct Node* head = NULL;

    /* Use push() to construct below list
```

```
1->12->1->4->1  */
push(&head, 1);
push(&head, 4);
push(&head, 1);
push(&head, 12);
push(&head, 1);

printf("\n Before deleting \n");
printList(head);

/* I m deleting the head itself.
   You can check for more cases */
deleteNode(head);

printf("\n After deleting \n");
printList(head);
getchar();
}
```

### Java

```
// Java program to del the node in which only a single pointer
// is known pointing to that node

class LinkedList {

    static Node head;

    static class Node {

        int data;
        Node next;

        Node(int d) {
            data = d;
            next = null;
        }
    }

    void printList(Node node) {
        while (node != null) {
            System.out.print(node.data + " ");
            node = node.next;
        }
    }

    void deleteNode(Node node) {
        Node temp = node.next;
```

```
node.data = temp.data;
node.next = temp.next;
System.gc();

}

// Driver program to test above functions
public static void main(String[] args) {
    LinkedList list = new LinkedList();
    list.head = new Node(1);
    list.head.next = new Node(12);
    list.head.next.next = new Node(1);
    list.head.next.next.next = new Node(4);
    list.head.next.next.next.next = new Node(1);

    System.out.println("Before Deleting ");
    list.printList(head);

    /* I m deleting the head itself.
     You can check for more cases */
    list.deleteNode(head);
    System.out.println("");
    System.out.println("After deleting ");
    list.printList(head);
}
}

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

**This solution doesn't work if the node to be deleted is the last node of the list.** To make this solution work we can mark the end node as a dummy node. But the programs/functions that are using this function should also be modified.

Try this problem for doubly linked list.

## Source

<https://www.geeksforgeeks.org/in-a-linked-list-given-only-a-pointer-to-a-node-to-be-deleted-in-a-singly-linked-list/>

## Chapter 103

# Given only a pointer/reference to a node to be deleted in a singly linked list, how do you delete it?

Given only a pointer/reference to a node to be deleted in a singly linked list, how do you delete it? - GeeksforGeeks

Given a pointer to a node to be deleted, delete the node. Note that we don't have pointer to head node.

A **simple solution** is to traverse the linked list until you find the node you want to delete. But this solution requires pointer to the head node which contradicts the problem statement.

A **fast solution** is to copy the data from the next node to the node to be deleted and delete the next node. Something like following.

```
// Find next node using next pointer
struct Node *temp = node_ptr->next;

// Copy data of next node to this node
node_ptr->data = temp->data;

// Unlink next node
node_ptr->next = temp->next;

// Delete next node
free(temp);
```

**Program:**

C

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

/* Link list node */
struct Node
{
    int data;
    struct Node* next;
};

/* Given a reference (pointer to pointer) to the head
   of a list and an int, push a new node on the front
   of the list. */
void push(struct Node** head_ref, int new_data)
{
    /* allocate node */
    struct Node* new_node =
        (struct Node*) malloc(sizeof(struct Node));

    /* put in the data */
    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;
}

void printList(struct Node *head)
{
    struct Node *temp = head;
    while(temp != NULL)
    {
        printf("%d ", temp->data);
        temp = temp->next;
    }
}

void deleteNode(struct Node *node_ptr)
{
    struct Node *temp = node_ptr->next;
    node_ptr->data = temp->data;
```

```
    node_ptr->next    = temp->next;
    free(temp);
}

/* Drier program to test above function*/
int main()
{
    /* Start with the empty list */
    struct Node* head = NULL;

    /* Use push() to construct below list
     * 1->12->1->4->1  */
    push(&head, 1);
    push(&head, 4);
    push(&head, 1);
    push(&head, 12);
    push(&head, 1);

    printf("Before deleting \n");
    printList(head);

    /* I m deleting the head itself.
       You can check for more cases */
    deleteNode(head);

    printf("\nAfter deleting \n");
    printList(head);
    getchar();
    return 0;
}
```

### Java

```
class LinkedList
{
    Node head; // head of the list

    class Node
    {
        int data;
        Node next;
        Node(int d) {data = d; next = null; }
    }

    /* Given a reference to the head of a list and an int,
       inserts a new Node on the front of the list. */
    public void push(int new_data)
    {
```

```
/* 1. alloc the Node and put the data */
Node new_Node = new Node(new_data);

/* 2. Make next of new Node as head */
new_Node.next = head;

/* 3. Move the head to point to new Node */
head = new_Node;
}

/* This function prints contents of linked list
   starting from the given Node */
public void printList()
{
    Node tNode = head;
    while (tNode != null) {
        System.out.print(tNode.data+" ");
        tNode = tNode.next;
    }
}

public void deleteNode(Node Node_ptr)
{
    Node temp = Node_ptr.next;
    Node_ptr.data = temp.data;
    Node_ptr.next = temp.next;
    temp = null;
}

public static void main(String[] args)
{
    LinkedList llist = new LinkedList();

    /* Use push() to construct below list
       1->12->1->4->1 */
    llist.push(1);
    llist.push(4);
    llist.push(1);
    llist.push(12);
    llist.push(1);

    System.out.println("Before deleting");
    llist.printList();

    /* I m deleting the head itself.
       You can check for more cases */
    llist.deleteNode(llist.head);
```

```
        System.out.println("\nAfter Deleting");
        llist.printList();
    }
}
// This code is contributed by Rajat Mishra
```

Output:

```
Before deleting
1 12 1 4 1
After deleting
12 1 4 1
```

**This solution doesn't work if the node to be deleted is the last node of the list.** To make this solution work we can mark the end node as a dummy node. But the programs/functions that are using this function should also be modified.

Exercise: Try this problem for doubly linked list.

## Source

<https://www.geeksforgeeks.org/given-only-a-pointer-to-a-node-to-be-deleted-in-a-singly-linked-list-how-do-you-del>

## Chapter 104

# Hashtables Chaining with Doubly Linked Lists

Hashtables Chaining with Doubly Linked Lists - GeeksforGeeks

**Prerequisite** – [Hashing Introduction](#), [Hashtable using Singly Linked List](#) & [Implementing our Own Hash Table with Separate Chaining in Java](#)

Implementing hash table using Chaining through Doubly Linked List is similar to implementing [Hashtable using Singly Linked List](#). The only difference is that every node of Linked List has the address of both, the next and the previous node. This will speed up the process of adding and removing elements from the list, hence the time complexity will be reduced drastically.

**Example:**

If we have a Singly linked list:

1->2->3->4

If we are at 3 and there is a need to remove it, then 2 need to be linked with 4 and as from 3, 2 can't be accessed as it is singly linked list. So, the list has to be traversed again i.e  $O(n)$ , but if we have doubly linked list i.e.

1234

2 & 4 can be accessed from 3, hence in  $O(1)$ , 3 can be removed.

Below is the implementation of the above approach:

```
// C++ implementation of Hashtable
// using doubly linked list
#include <bits/stdc++.h>
using namespace std;

const int tablesiz = 25;

// declaration of node
struct hash_node {
    int val, key;
    hash_node* next;
    hash_node* prev;
};

// hashmap's declaration
class HashMap {
public:
    hash_node **hashtable, **top;

    // constructor
    HashMap()
    {
        // create a empty hashtable
        hashtable = new hash_node*[tablesiz];
        top = new hash_node*[tablesiz];
        for (int i = 0; i < tablesiz; i++) {
            hashtable[i] = NULL;
            top[i] = NULL;
        }
    }

    // destructor
    ~HashMap()
    {
        delete[] hashtable;
    }

    // hash function definition
    int HashFunc(int key)
    {
        return key % tablesiz;
    }

    // searching method
    void find(int key)
    {
        // Applying hashFunc to find
        // index for given key
```

```

int hash_val = HashFunc(key);
bool flag = false;
hash_node* entry = hashtable[hash_val];

// if hashtable at that index has some
// values stored
if (entry != NULL) {
    while (entry != NULL) {
        if (entry->key == key) {
            flag = true;
        }
        if (flag) {
            cout << "Element found at key "
                << key << ": ";
            cout << entry->val << endl;
        }
        entry = entry->next;
    }
}
if (!flag)
    cout << "No Element found at key "
        << key << endl;
}

// removing an element
void remove(int key)
{
    // Applying hashFunc to find
    // index for given key
    int hash_val = HashFunc(key);
    hash_node* entry = hashtable[hash_val];
    if (entry->key != key || entry == NULL) {
        cout << "Couldn't find any element at this key "
            << key << endl;
        return;
    }

    // if some values are present at that key &
    // traversing the list and removing all values
    while (entry != NULL) {
        if (entry->next == NULL) {
            if (entry->prev == NULL) {
                hashtable[hash_val] = NULL;
                top[hash_val] = NULL;
                delete entry;
                break;
            }
        else {

```

```
        top[hash_val] = entry->prev;
        top[hash_val]->next = NULL;
        delete entry;
        entry = top[hash_val];
    }
}
entry = entry->next;
}
cout << "Element was successfully removed at the key "
     << key << endl;
}

// inserting method
void add(int key, int value)
{
    // Applying hashFunc to find
    // index for given key
    int hash_val = HashFunc(key);
    hash_node* entry = hashtable[hash_val];

    // if key has no value stored
    if (entry == NULL) {
        // creating new node
        entry = new hash_node;
        entry->val = value;
        entry->key = key;
        entry->next = NULL;
        entry->prev = NULL;
        hashtable[hash_val] = entry;
        top[hash_val] = entry;
    }

    // if some values are present
    else {
        // traversing till the end of
        // the list
        while (entry != NULL)
            entry = entry->next;

        // creating the new node
        entry = new hash_node;
        entry->val = value;
        entry->key = key;
        entry->next = NULL;
        entry->prev = top[hash_val];
        top[hash_val]->next = entry;
        top[hash_val] = entry;
    }
}
```

```
        cout << "Value " << value << " was successfully"
              " added at key " << key << endl;
    }
};

// Driver Code
int main()
{
    HashMap hash;
    hash.add(4, 5);
    hash.find(4);
    hash.remove(4);
    return 0;
}
```

**Output:**

```
Value 5 was successfully added at key 4
Element found at key 4: 5
Element was successfully removed at the key 4
```

**Source**

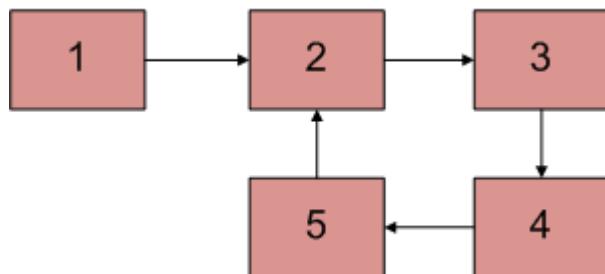
<https://www.geeksforgeeks.org/hashtables-chaining-with-doubly-linked-lists/>

## Chapter 105

# How does Floyd's slow and fast pointers approach work?

How does Floyd's slow and fast pointers approach work? - GeeksforGeeks

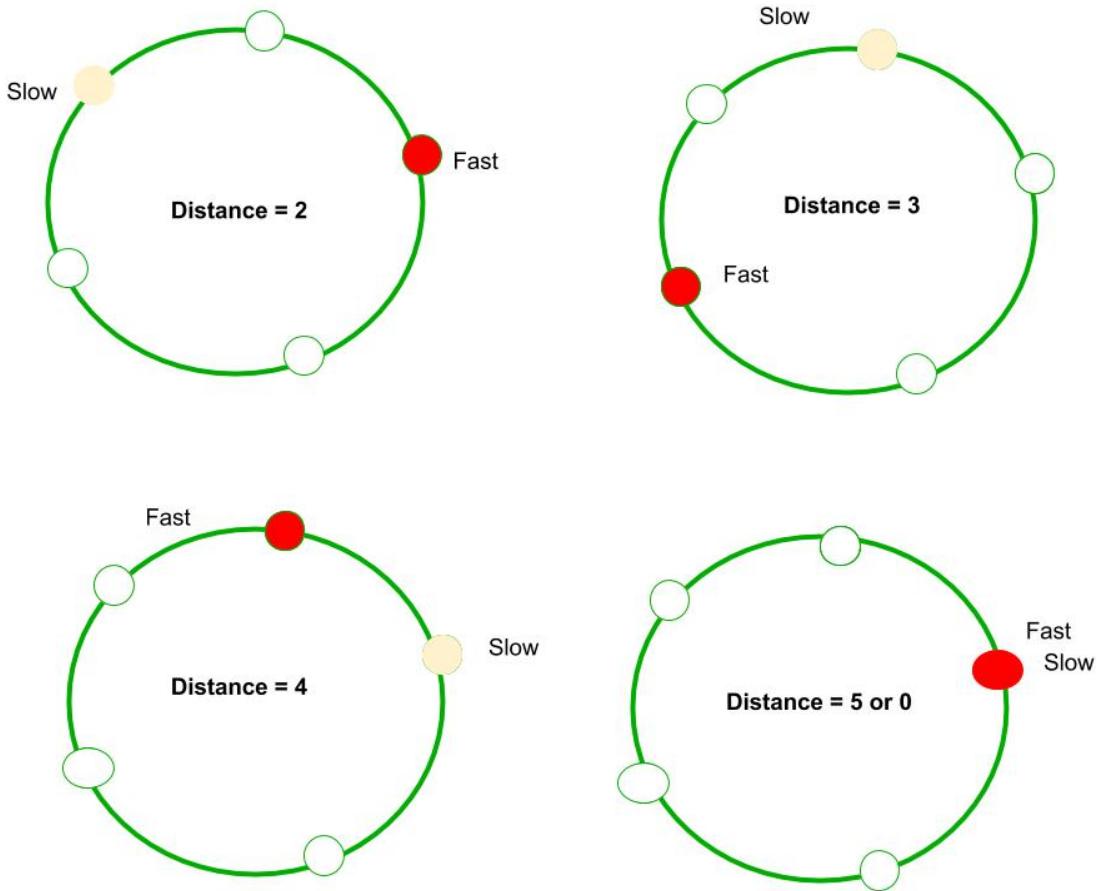
We have discussed Floyd's fast and slow pointer algorithms in [Detect loop in a linked list](#).



The algorithm is to start two pointers, slow and fast from head of linked list. We move slow one node at a time and fast two nodes at a time. If there is a loop, then they will definitely meet. This approach works because of the following facts.

- 1) When slow pointer enters the loop, the fast pointer must be inside the loop. Let fast pointer be distance  $k$  from slow.
- 2) Now if consider movements of slow and fast pointers, we can notice that distance between them (from slow to fast) increase by one after every iteration. After one iteration (of slow = next of slow and fast = next of next of fast), distance between slow and fast becomes  $k+1$ , after two iterations,  $k+2$ , and so on. When distance becomes  $n$ , they meet because they are moving in a cycle of length  $n$ .

For example, we can see in below diagram, initial distance is 2. After one iteration, distance becomes 3, after 2 iterations, it becomes 4. After 3 iterations, it becomes 5 which is distance 0. And they meet.



**How does cycle removal algorithm work?**

Please see method 3 of [Detect and Remove Loop in a Linked List](#)

## Source

<https://www.geeksforgeeks.org/how-does-floyds-slow-and-fast-pointers-approach-work/>

## Chapter 106

# How to write C functions that modify head pointer of a Linked List?

How to write C functions that modify head pointer of a Linked List? - GeeksforGeeks

Consider simple representation (without any dummy node) of Linked List. Functions that operate on such Linked lists can be divided in two categories:

**1) Functions that do not modify the head pointer:** Examples of such functions include, printing a linked list, updating data members of nodes like adding given a value to all nodes, or some other operation which access/update data of nodes

It is generally easy to decide prototype of functions of this category. We can always pass head pointer as an argument and traverse/update the list. For example, the following function that adds x to data members of all nodes.

```
void addXtoList(struct Node *node, int x)
{
    while(node != NULL)
    {
        node->data = node->data + x;
        node = node->next;
    }
}
```

**2) Functions that modify the head pointer:** Examples include, inserting a node at the beginning (head pointer is always modified in this function), inserting a node at the end (head pointer is modified only when the first node is being inserted), deleting a given node (head pointer is modified when the deleted node is first node). There may be different ways to update the head pointer in these functions. Let us discuss these ways using following simple problem:

*“Given a linked list, write a function deleteFirst() that deletes the first node of a given linked list. For example, if the list is 1->2->3->4, then it should be modified to 2->3->4”*

Algorithm to solve the problem is a simple 3 step process: (a) Store the head pointer (b) change the head pointer to point to next node (c) delete the previous head node.

Following are different ways to update head pointer in deleteFirst() so that the list is updated everywhere.

**2.1) Make head pointer global:** We can make the head pointer global so that it can be accessed and updated in our function. Following is C code that uses global head pointer.

```
// global head pointer
struct Node *head = NULL;

// function to delete first node: uses approach 2.1
// See http://ideone.com/ClfQB for complete program and output
void deleteFirst()
{
    if(head != NULL)
    {
        // store the old value of head pointer
        struct Node *temp = head;

        // Change head pointer to point to next node
        head = head->next;

        // delete memory allocated for the previous head node
        free(temp);
    }
}
```

See [this](#)for complete running program that uses above function.

This is not a recommended way as it has many problems like following:

- a) head is globally accessible, so it can be modified anywhere in your project and may lead to unpredictable results.
- b) If there are multiple linked lists, then multiple global head pointers with different names are needed.

See [this](#)to know all reasons why should we avoid global variables in our projects.

**2.2) Return head pointer:** We can write deleteFirst() in such a way that it returns the modified head pointer. Whoever is using this function, have to use the returned value to update the head node.

```
// function to delete first node: uses approach 2.2
// See http://ideone.com/P5oLe for complete program and output
struct Node *deleteFirst(struct Node *head)
{
    if(head != NULL)
```

```
{  
    // store the old value of head pointer  
    struct Node *temp = head;  
  
    // Change head pointer to point to next node  
    head = head->next;  
  
    // delete memory allocated for the previous head node  
    free(temp);  
}  
  
return head;  
}
```

See [this](#)for complete program and output.

This approach is much better than the previous 1. There is only one issue with this, if user misses to assign the returned value to head, then things become messy. C/C++ compilers allows to call a function without assigning the returned value.

```
head = deleteFirst(head); // proper use of deleteFirst()  
deleteFirst(head); // improper use of deleteFirst(), allowed by compiler
```

**2.3) Use Double Pointer:** This approach follows the simple C rule: *if you want to modify local variable of one function inside another function, pass pointer to that variable.* So we can pass pointer to the head pointer to modify the head pointer in our deleteFirst() function.

```
// function to delete first node: uses approach 2.3  
// See http://ideone.com/9GwTb for complete program and output  
void deleteFirst(struct Node **head_ref)  
{  
    if(*head_ref != NULL)  
    {  
        // store the old value of pointer to head pointer  
        struct Node *temp = *head_ref;  
  
        // Change head pointer to point to next node  
        *head_ref = (*head_ref)->next;  
  
        // delete memory allocated for the previous head node  
        free(temp);  
    }  
}
```

See [this](#) for complete program and output.

This approach seems to be the best among all three as there are less chances of having problems.

## Source

<https://www.geeksforgeeks.org/how-to-write-functions-that-modify-the-head-pointer-of-a-linked-list/>

## Chapter 107

# Identical Linked Lists

Identical Linked Lists - GeeksforGeeks

Two Linked Lists are identical when they have same data and arrangement of data is also same. For example Linked lists a (1->2->3) and b(1->2->3) are identical. . Write a function to check if the given two linked lists are identical.

### Method 1 (Iterative)

To identify if two lists are identical, we need to traverse both lists simultaneously, and while traversing we need to compare data.

C

```
// An iterative C program to check if two linked lists are
// identical or not
#include<stdio.h>
#include<stdlib.h>
#include<stdbool.h>

/* Structure for a linked list node */
struct Node
{
    int data;
    struct Node *next;
};

/* Returns true if linked lists a and b are identical,
otherwise false */
bool areIdentical(struct Node *a, struct Node *b)
{
    while (a != NULL && b != NULL)
    {
        if (a->data != b->data)
            return false;
        a = a->next;
        b = b->next;
    }
    return (a == b);
}
```

```

/* If we reach here, then a and b are not NULL and
   their data is same, so move to next nodes in both
   lists */
a = a->next;
b = b->next;
}

// If linked lists are identical, then 'a' and 'b' must
// be NULL at this point.
return (a == NULL && b == NULL);
}

/* UTILITY FUNCTIONS TO TEST fun1() and fun2() */
/* Given a reference (pointer to pointer) to the head
   of a list and an int, push a new node on the front
   of the list. */
void push(struct Node** head_ref, int new_data)
{
    /* allocate node */
    struct Node* new_node =
        (struct Node*) malloc(sizeof(struct Node));

    /* put in the data */
    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;
}

/* Driver program to test above function */
int main()
{
    /* The constructed linked lists are :
       a: 3->2->1
       b: 3->2->1 */
    struct Node *a = NULL;
    struct Node *b = NULL;
    push(&a, 1);
    push(&a, 2);
    push(&a, 3);
    push(&b, 1);
    push(&b, 2);
    push(&b, 3);
}

```

```
areIdentical(a, b)? printf("Identical"):
    printf("Not identical");

return 0;
}
```

**Java**

```
// An iterative Java program to check if two linked lists
// are identical or not
class LinkedList
{
    Node head; // head of list

    /* Linked list Node*/
    class Node
    {
        int data;
        Node next;
        Node(int d) { data = d; next = null; }
    }

    /* Returns true if linked lists a and b are identical,
       otherwise false */
    boolean areIdentical(LinkedList listb)
    {
        Node a = this.head, b = listb.head;
        while (a != null && b != null)
        {
            if (a.data != b.data)
                return false;

            /* If we reach here, then a and b are not null
               and their data is same, so move to next nodes
               in both lists */
            a = a.next;
            b = b.next;
        }

        // If linked lists are identical, then 'a' and 'b' must
        // be null at this point.
        return (a == null && b == null);
    }

    /* UTILITY FUNCTIONS TO TEST fun1() and fun2() */
    /* Given a reference (pointer to pointer) to the head
       of a list and an int, push a new node on the front
       of the list. */
}
```

```
void push(int new_data)
{
    /* 1 & 2: Allocate the Node &
       Put in the data*/
    Node new_node = new Node(new_data);

    /* 3. Make next of new Node as head */
    new_node.next = head;

    /* 4. Move the head to point to new Node */
    head = new_node;
}

/* Drier program to test above functions */
public static void main(String args[])
{
    LinkedList llist1 = new LinkedList();
    LinkedList llist2 = new LinkedList();

    /* The constructed linked lists are :
       llist1: 3->2->1
       llist2: 3->2->1 */

    llist1.push(1);
    llist1.push(2);
    llist1.push(3);

    llist2.push(1);
    llist2.push(2);
    llist2.push(3);

    if (llist1.areIdentical(llist2) == true)
        System.out.println("Identical ");
    else
        System.out.println("Not identical ");

}
} /* This code is contributed by Rajat Mishra */
```

Output:

Identical

#### **Method 2 (Recursive)**

Recursive solution code is much cleaner than the iterative code. You probably wouldn't

want to use the recursive version for production code however, because it will use stack space which is proportional to the length of the lists

**C**

```
// A recursive C function to check if two linked
// lists are identical or not
bool areIdentical(struct Node *a, struct Node *b)
{
    // If both lists are empty
    if (a == NULL && b == NULL)
        return true;

    // If both lists are not empty, then data of
    // current nodes must match, and same should
    // be recursively true for rest of the nodes.
    if (a != NULL && b != NULL)
        return (a->data == b->data) &&
               areIdentical(a->next, b->next);

    // If we reach here, then one of ths lists
    // is empty and other is not
    return false;
}
```

**Java**

```
// A recursive Java method to check if two linked
// lists are identical or not
boolean areIdenticalRecur(Node a, Node b)
{
    // If both lists are empty
    if (a == null && b == null)
        return true;

    // If both lists are not empty, then data of
    // current nodes must match, and same should
    // be recursively true for rest of the nodes.
    if (a != null && b != null)
        return (a.data == b.data) &&
               areIdenticalRecur(a.next, b.next);

    // If we reach here, then one of ths lists
    // is empty and other is not
    return false;
}

/* Returns true if linked lists a and b are identical,
```

```
    otherwise false */
boolean areIdentical(LinkedList listb)
{
    return areIdenticalRecur(this.head, listb.head);
}
```

Time Complexity:  $O(n)$  for both iterative and recursive versions.  $n$  is the length of the smaller list among a and b.

## Source

<https://www.geeksforgeeks.org/identical-linked-lists/>

## Chapter 108

# Implementation of Deque using doubly linked list

Implementation of Deque using doubly linked list - GeeksforGeeks

[Deque or Double Ended Queue](#) is a generalized version of [Queue data structure](#) that allows insert and delete at both ends. In [previous post](#) Implementation of Deque using circular array has been discussed. Now in this post we see how we implement Deque using [Doubly Linked List](#).

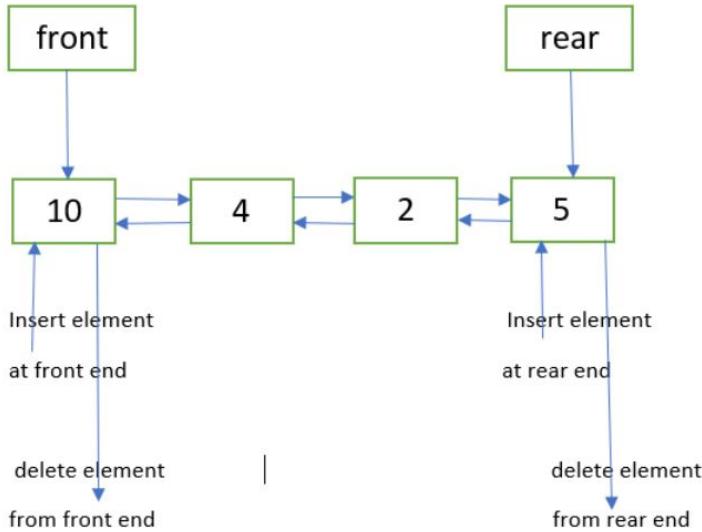
### Operations on Deque :

Mainly the following four basic operations are performed on queue :

```
insertFront() : Adds an item at the front of Deque.  
insertRear()  : Adds an item at the rear of Deque.  
deleteFront() : Deletes an item from front of Deque.  
deleteRear()  : Deletes an item from rear of Deque.
```

In addition to above operations, following operations are also supported :

```
getFront()   : Gets the front item from queue.  
getRear()    : Gets the last item from queue.  
isEmpty()    : Checks whether Deque is empty or not.  
size()       : Gets number of elements in Deque.  
erase()      : Deletes all the elements from Deque.
```



#### Doubly Linked List Representation of Deque :

For implementing deque, we need to keep track of two pointers, **front** and **rear**. We **enqueue (push)** an item at the rear or the front end of deque and **dequeue (pop)** an item from both rear and front end.

#### Working :

Declare two pointers **front** and **rear** of type **Node**, where **Node** represents the structure of a node of a doubly linked list. Initialize both of them with value **NUL**L.

#### Insertion at Front end :

1. Allocate space for a **newNode** of doubly linked list.
2. IF **newNode == NULL**, then
3.     print "Overflow"
4. ELSE
5.     IF **front == NULL**, then
6.         **rear = front = newNode**
7.     ELSE
8.         **newNode->next = front**
9.         **front->prev = newNode**
10.         **front = newNode**

#### Insertion at Rear end :

1. Allocate space for a **newNode** of doubly linked list.
2. IF **newNode == NULL**, then
3.     print "Overflow"

```
4. ELSE
5.     IF rear == NULL, then
6.         front = rear = newNode
7.     ELSE
8.         newNode->prev = rear
9.         rear->next = newNode
10.        rear = newNode
```

**Deletion from Front end :**

```
1. IF front == NULL
2.     print "Underflow"
3. ELSE
4.     Initialize temp = front
5.     front = front->next
6.     IF front == NULL
7.         rear = NULL
8.     ELSE
9.         front->prev = NULL
10.    Deallocate space for temp
```

**Deletion from Rear end :**

```
1. IF front == NULL
2.     print "Underflow"
3. ELSE
4.     Initialize temp = rear
5.     rear = rear->prev
6.     IF rear == NULL
7.         front = NULL
8.     ELSE
9.         rear->next = NULL
10.    Deallocate space for temp
```

```
// C++ implementation of Deque using
// doubly linked list
#include <bits/stdc++.h>
```

```
using namespace std;

// Node of a doubly linked list
struct Node
{
    int data;
    Node *prev, *next;
```

```
// Function to get a new node
static Node* getnode(int data)
{
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->data = data;
    newNode->prev = newNode->next = NULL;
    return newNode;
}

// A structure to represent a deque
class Deque
{
    Node* front;
    Node* rear;
    int Size;

public:
    Deque()
    {
        front = rear = NULL;
        Size = 0;
    }

    // Operations on Deque
    void insertFront(int data);
    void insertRear(int data);
    void deleteFront();
    void deleteRear();
    int getFront();
    int getRear();
    int size();
    bool isEmpty();
    void erase();
};

// Function to check whether deque
// is empty or not
bool Deque::isEmpty()
{
    return (front == NULL);
}

// Function to return the number of
// elements in the deque
int Deque::size()
{
    return Size;
```

```
}

// Function to insert an element
// at the front end
void Deque::insertFront(int data)
{
    Node* newNode = Node::getnode(data);
    // If true then new element cannot be added
    // and it is an 'Overflow' condition
    if (newNode == NULL)
        cout << "OverFlow\n";
    else
    {
        // If deque is empty
        if (front == NULL)
            rear = front = newNode;

        // Inserts node at the front end
        else
        {
            newNode->next = front;
            front->prev = newNode;
            front = newNode;
        }

        // Increments count of elements by 1
        Size++;
    }
}

// Function to insert an element
// at the rear end
void Deque::insertRear(int data)
{
    Node* newNode = Node::getnode(data);
    // If true then new element cannot be added
    // and it is an 'Overflow' condition
    if (newNode == NULL)
        cout << "OverFlow\n";
    else
    {
        // If deque is empty
        if (rear == NULL)
            front = rear = newNode;

        // Inserts node at the rear end
        else
        {
```

```
    newNode->prev = rear;
    rear->next = newNode;
    rear = newNode;
}

Size++;
}

// Function to delete the element
// from the front end
void Deque::deleteFront()
{
    // If deque is empty then
    // 'Underflow' condition
    if (isEmpty())
        cout << "UnderFlow\n";

    // Deletes the node from the front end and makes
    // the adjustment in the links
    else
    {
        Node* temp = front;
        front = front->next;

        // If only one element was present
        if (front == NULL)
            rear = NULL;
        else
            front->prev = NULL;
        free(temp);

        // Decrements count of elements by 1
        Size--;
    }
}

// Function to delete the element
// from the rear end
void Deque::deleteRear()
{
    // If deque is empty then
    // 'Underflow' condition
    if (isEmpty())
        cout << "UnderFlow\n";

    // Deletes the node from the rear end and makes
    // the adjustment in the links
```

```
else
{
    Node* temp = rear;
    rear = rear->prev;

    // If only one element was present
    if (rear == NULL)
        front = NULL;
    else
        rear->next = NULL;
    free(temp);

    // Decrements count of elements by 1
    Size--;
}
}

// Function to return the element
// at the front end
int Deque::getFront()
{
    // If deque is empty, then returns
    // garbage value
    if (isEmpty())
        return -1;
    return front->data;
}

// Function to return the element
// at the rear end
int Deque::getRear()
{
    // If deque is empty, then returns
    // garbage value
    if (isEmpty())
        return -1;
    return rear->data;
}

// Function to delete all the elements
// from Deque
void Deque::erase()
{
    rear = NULL;
    while (front != NULL)
    {
        Node* temp = front;
        front = front->next;
    }
}
```

```
        free(temp);
    }
    Size = 0;
}

// Driver program to test above
int main()
{
    Deque dq;
    cout << "Insert element '5' at rear end\n";
    dq.insertRear(5);

    cout << "Insert element '10' at rear end\n";
    dq.insertRear(10);

    cout << "Rear end element: "
        << dq.getRear() << endl;

    dq.deleteRear();
    cout << "After deleting rear element new rear"
        << " is: " << dq.getRear() << endl;

    cout << "Inserting element '15' at front end \n";
    dq.insertFront(15);

    cout << "Front end element: "
        << dq.getFront() << endl;

    cout << "Number of elements in Deque: "
        << dq.size() << endl;

    dq.deleteFront();
    cout << "After deleting front element new "
        << "front is: " << dq.getFront() << endl;

    return 0;
}
```

Output :

```
Insert element '5' at rear end
Insert element '10' at rear end
Rear end element: 10
After deleting rear element new rear is: 5
Inserting element '15' at front end
Front end element: 15
Number of elements in Deque: 2
```

After deleting front element new front is: 5

Time Complexity : Time complexity of operations like insertFront(), insertRear(), deleteFront(), deleteRear() is O(1). Time Complexity of erase() is O(n).

### Source

<https://www.geeksforgeeks.org/implementation-deque-using-doubly-linked-list/>

## Chapter 109

# Implementing Iterator pattern of a single Linked List

Implementing Iterator pattern of a single Linked List - GeeksforGeeks

STL is one of the pillars of C++. It makes life lot easier, especially when your focus is on problem solving and you don't want to spend time in implementing something that is already available which guarantees a robust solution. One of the key aspects of Software Engineering is to avoid reinventing the wheel. Reusability is **always** preferred.

While relying on library functions directly impacts our efficiency, without having a proper understanding of how it works sometimes loses meaning of the engineering efficiency we keep on talking. A wrongly chosen data structure may come back sometime in future to haunt us. The solution is simple. Use library methods, but know how does it handles operations under the hood.

Enough said! Today we will look on how we can implement our own **Iterator pattern of a single Linked List**. So, here is how an STL implementation of Linked List looks like:

```
#include <iostream>
#include <vector>
using namespace std;

int main()
{
    // creating a list
    vector<int> list;

    // elements to be added at the end.
    // in the above created list.
    list.push_back(1);
    list.push_back(2);
    list.push_back(3);
```

```
// elements of list are retrieved through iterator.  
for (vector<int>::iterator it = list.begin();  
     it != list.end(); ++it)  
    cout << *it << " ";  
  
return 0;  
}
```

Output

```
1 2 3
```

One of the beauty of **cin** and **cout** is that they don't demand format specifiers to work with the type of data. This combined with templates make the code much cleaner and readable. Although I prefer naming method in C++ start with caps, this implementation follows STL rules to mimic exact set of method calls, viz push\_back, begin, end.

Here is our own implementation of LinkedList and its Iterator pattern:

```
// C++ program to implement Custom Linked List and  
// iterator pattern.  
#include <iostream>  
using namespace std;  
  
// Custom class to handle Linked List operations  
// Operations like push_back, push_front, pop_back,  
// pop_front, erase, size can be added here  
template <typename T>  
class LinkedList  
{  
    // Forward declaration  
    class Node;  
  
public:  
    LinkedList<T>() noexcept  
    {  
        // caution: static members can't be  
        // initialized by initializer list  
        m_spRoot = nullptr;  
    }  
  
    // Forward declaration must be done  
    // in the same access scope  
    class Iterator;  
  
    // Root of LinkedList wrapped in Iterator type  
    Iterator begin()
```

```
{  
    return Iterator(m_spRoot);  
}  
  
// End of LInkedList wrapped in Iterator type  
Iterator end()  
{  
    return Iterator(nullptr);  
}  
  
// Adds data to the end of list  
void push_back(T data);  
  
void Traverse();  
  
// Iterator class can be used to  
// sequentially access nodes of linked list  
class Iterator  
{  
public:  
    Iterator() noexcept :  
        m_pCurrentNode (m_spRoot) { }  
  
    Iterator(const Node* pNode) noexcept :  
        m_pCurrentNode (pNode) { }  
  
    Iterator& operator=(Node* pNode)  
    {  
        this->m_pCurrentNode = pNode;  
        return *this;  
    }  
  
    // Prefix ++ overload  
    Iterator& operator++()  
    {  
        if (m_pCurrentNode)  
            m_pCurrentNode = m_pCurrentNode->pNext;  
        return *this;  
    }  
  
    // Postfix ++ overload  
    Iterator operator++(int)  
    {  
        Iterator iterator = *this;  
        ++*this;  
        return iterator;  
    }  
}
```

```
bool operator!=(const Iterator& iterator)
{
    return m_pCurrentNode != iterator.m_pCurrentNode;
}

int operator*()
{
    return m_pCurrentNode->data;
}

private:
    const Node* m_pCurrentNode;
};

private:

class Node
{
    T data;
    Node* pNext;

    // LinkedList class methods need
    // to access Node information
    friend class LinkedList;
};

// Create a new Node
Node* GetNode(T data)
{
    Node* pNewNode = new Node;
    pNewNode->data = data;
    pNewNode->pNext = nullptr;

    return pNewNode;
}

// Return by reference so that it can be used in
// left hand side of the assignment expression
Node*& GetRootNode()
{
    return m_spRoot;
}

static Node* m_spRoot;
};

template <typename T>
/*static*/ typename LinkedList<T>::Node* LinkedList<T>::m_spRoot = nullptr;
```

```
template <typename T>
void LinkedList<T>::push_back(T data)
{
    Node* pTemp = GetNode(data);
    if (!GetRootNode())
    {
        GetRootNode() = pTemp;
    }
    else
    {
        Node* pCrawler = GetRootNode();
        while (pCrawler->pNext)
        {
            pCrawler = pCrawler->pNext;
        }

        pCrawler->pNext = pTemp;
    }
}

template <typename T>
void LinkedList<T>::Traverse()
{
    Node* pCrawler = GetRootNode();

    while (pCrawler)
    {
        cout << pCrawler->data << " ";
        pCrawler = pCrawler->pNext;
    }

    cout << endl;
}

//Driver program
int main()
{
    LinkedList<int> list;

    // Add few items to the end of LinkedList
    list.push_back(1);
    list.push_back(2);
    list.push_back(3);

    cout << "Traversing LinkedList through method" << endl;
    list.Traverse();
```

```
cout << "Traversing LinkedList through Iterator" << endl;
for ( LinkedList<int>::Iterator iterator = list.begin();
      iterator != list.end(); iterator++)
{
    cout << *iterator << " ";
}

cout << endl;

return 0;
}
```

Output:

```
Traversing LinkedList through method
1 2 3
Traversing LinkedList through Iterator
1 2 3
```

**Exercise:**

The above implementation works well when we have one data. Extend this code to work for set of data wrapped in a class.

## Source

<https://www.geeksforgeeks.org/implementing-iterator-pattern-of-a-single-linked-list/>

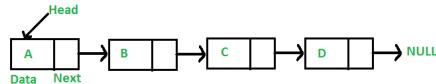
## Chapter 110

# Implementing a Linked List in Java using Class

Implementing a Linked List in Java using Class - GeeksforGeeks

**Pre-requisite:** [Linked List Data Structure](#)

Like arrays, Linked List is a linear data structure. Unlike arrays, linked list elements are not stored at the contiguous location, the elements are linked using pointers as shown below.



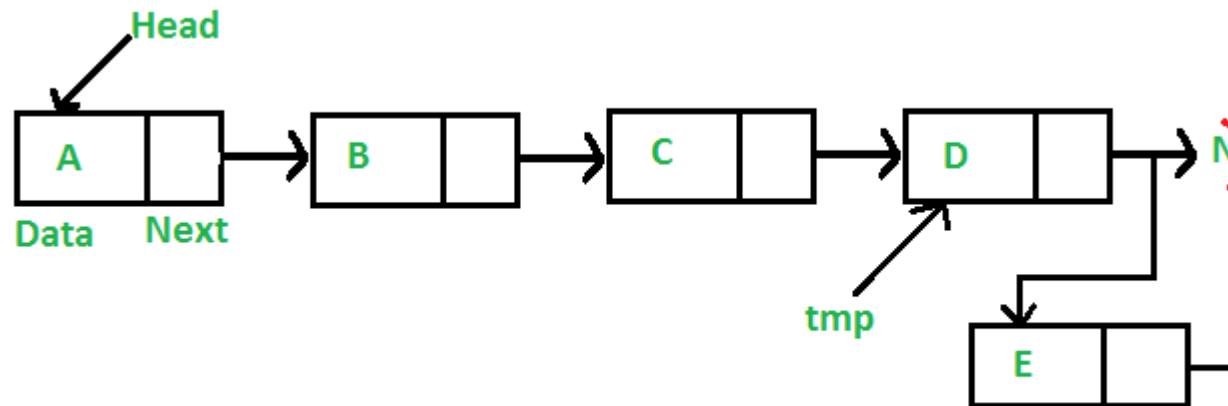
In Java, LinkedList can be represented as a class and a Node as a separate class. The LinkedList class contains a reference of Node class type.

```
class LinkedList {  
    Node head; // head of list  
  
    /* Linked list Node*/  
    class Node {  
        int data;  
        Node next;  
  
        // Constructor to create a new node  
        // Next is by default initialized  
        // as null  
        Node(int d) { data = d; }  
    }  
}
```

In this article, insertion in the list is done at the end, that is the new node is added after

the last node of the given Linked List. For example, if the given Linked List is 5->10->15->20->25 and 30 is to be inserted, then the Linked List becomes 5->10->15->20->25->30.

Since a Linked List is typically represented by the head pointer of it, it is required to traverse the list till the last node and then change the next of last node to new node.



```

import java.io.*;

// Java program to implement
// a Singly Linked List
public class LinkedList {

    Node head; // head of list

    // Linked list Node.
    // This inner class is made static
    // so that main() can access it
    static class Node {

        int data;
        Node next;

        // Constructor
        Node(int d)
        {
            data = d;
            next = null;
        }
    }

    // Method to insert a new node
    public static LinkedList insert(LinkedList list, int data)
    {
        Node newNode = new Node(data);
        Node temp = list.head;

        if (list.head == null)
            list.head = newNode;
        else
        {
            while (temp.next != null)
                temp = temp.next;
            temp.next = newNode;
        }
        return list;
    }
}

```

```
{  
    // Create a new node with given data  
    Node new_node = new Node(data);  
    new_node.next = null;  
  
    // If the Linked List is empty,  
    // then make the new node as head  
    if (list.head == null) {  
        list.head = new_node;  
    }  
    else {  
        // Else traverse till the last node  
        // and insert the new_node there  
        Node last = list.head;  
        while (last.next != null) {  
            last = last.next;  
        }  
  
        // Insert the new_node at last node  
        last.next = new_node;  
    }  
  
    // Return the list by head  
    return list;  
}  
  
// Method to print the LinkedList.  
public static void printList(LinkedList list)  
{  
    Node currNode = list.head;  
  
    System.out.print("LinkedList: ");  
  
    // Traverse through the LinkedList  
    while (currNode != null) {  
        // Print the data at current node  
        System.out.print(currNode.data + " ");  
  
        // Go to next node  
        currNode = currNode.next;  
    }  
}  
  
// Driver code  
public static void main(String[] args)  
{  
    /* Start with the empty list. */  
    LinkedList list = new LinkedList();
```

```
//  
// *****INSERTION*****  
  
// Insert the values  
list = insert(list, 1);  
list = insert(list, 2);  
list = insert(list, 3);  
list = insert(list, 4);  
list = insert(list, 5);  
list = insert(list, 6);  
list = insert(list, 7);  
list = insert(list, 8);  
  
// Print the LinkedList  
printList(list);  
}  
}
```

**Output:**

```
LinkedList: 1 2 3 4 5 6 7 8
```

For traversal, below is a general purpose function printList() that prints any given list by traversing the list from head node to the last.

```
import java.io.*;  
  
// Java program to implement  
// a Singly Linked List  
public class LinkedList {  
  
    Node head; // head of list  
  
    // Linked list Node.  
    // This inner class is made static  
    // so that main() can access it  
    static class Node {  
  
        int data;  
        Node next;  
  
        // Constructor  
        Node(int d)  
        {
```

```
        data = d;
        next = null;
    }
}

// Method to insert a new node
public static LinkedList insert(LinkedList list, int data)
{
    // Create a new node with given data
    Node new_node = new Node(data);
    new_node.next = null;

    // If the Linked List is empty,
    // then make the new node as head
    if (list.head == null) {
        list.head = new_node;
    }
    else {
        // Else traverse till the last node
        // and insert the new_node there
        Node last = list.head;
        while (last.next != null) {
            last = last.next;
        }

        // Insert the new_node at last node
        last.next = new_node;
    }

    // Return the list by head
    return list;
}

// Method to print the LinkedList.
public static void printList(LinkedList list)
{
    Node currNode = list.head;

    System.out.print("LinkedList: ");

    // Traverse through the LinkedList
    while (currNode != null) {
        // Print the data at current node
        System.out.print(currNode.data + " ");

        // Go to next node
        currNode = currNode.next;
    }
}
```

```
}

// *****MAIN METHOD*****

// method to create a Singly linked list with n nodes
public static void main(String[] args)
{
    /* Start with the empty list. */
    LinkedList list = new LinkedList();

    //

    // *****INSERTION*****
    //

    // Insert the values
    list = insert(list, 1);
    list = insert(list, 2);
    list = insert(list, 3);
    list = insert(list, 4);
    list = insert(list, 5);
    list = insert(list, 6);
    list = insert(list, 7);
    list = insert(list, 8);

    // Print the LinkedList
    printList(list);
}
}
```

**Output:**

```
LinkedList: 1 2 3 4 5 6 7 8
```

The deletion process can be understood as follows:

**To be done:**

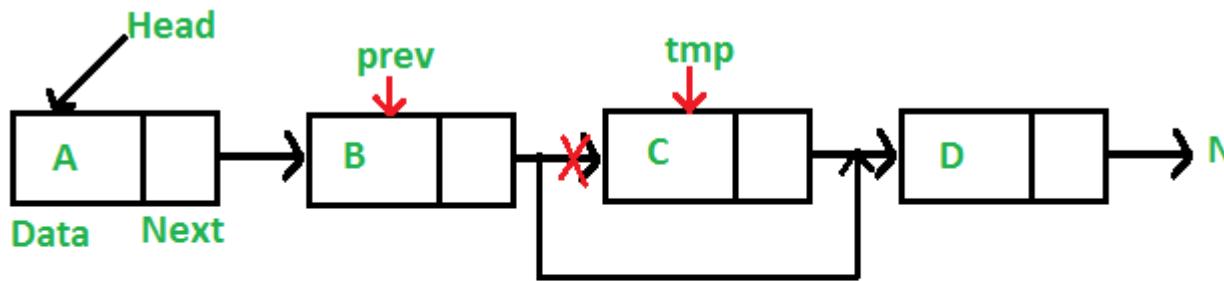
*Given a 'key', delete the first occurrence of this key in linked list.*

**How to do it:**

To delete a node from linked list, do following steps.

1. Search the key for its first occurrence in the list
2. Now, Any of the 3 conditions can be there:
  - **Case 1: The key is found at head**
    - (a) In this case, Change the head of the node to the next node of current head.
    - (b) Free the memory of replaced head node.

- **Case 2: The key is found at in the middle or last, except at head**
  - (a) In this case, Find previous node of the node to be deleted.
  - (b) Change the next of previous node to the next node of current node.
  - (c) Free the memory of replaced node.
- **Case 3: The key is not found in the list**
  - (a) In this case, No operation needs to be done.



```

import java.io.*;

// Java program to implement
// a Singly Linked List
public class LinkedList {

    Node head; // head of list

    // Linked list Node.
    // This inner class is made static
    // so that main() can access it
    static class Node {

        int data;
        Node next;

        // Constructor
        Node(int d)
        {
            data = d;
            next = null;
        }
    }
}

```

```
// Method to insert a new node
public static LinkedList insert(LinkedList list, int data)
{
    // Create a new node with given data
    Node new_node = new Node(data);
    new_node.next = null;

    // If the Linked List is empty,
    // then make the new node as head
    if (list.head == null) {
        list.head = new_node;
    }
    else {
        // Else traverse till the last node
        // and insert the new_node there
        Node last = list.head;
        while (last.next != null) {
            last = last.next;
        }

        // Insert the new_node at last node
        last.next = new_node;
    }

    // Return the list by head
    return list;
}

// Method to print the LinkedList.
public static void printList(LinkedList list)
{
    Node currNode = list.head;

    System.out.print("LinkedList: ");

    // Traverse through the LinkedList
    while (currNode != null) {
        // Print the data at current node
        System.out.print(currNode.data + " ");

        // Go to next node
        currNode = currNode.next;
    }

    System.out.println();
}
```

```
// *****DELETION BY KEY*****  
  
// Method to delete a node in the LinkedList by KEY  
public static LinkedList deleteByKey(LinkedList list, int key)  
{  
    // Store head node  
    Node currNode = list.head, prev = null;  
  
    //  
    // CASE 1:  
    // If head node itself holds the key to be deleted  
  
    if (currNode != null && currNode.data == key) {  
        list.head = currNode.next; // Changed head  
  
        // Display the message  
        System.out.println(key + " found and deleted");  
  
        // Return the updated List  
        return list;  
    }  
  
    //  
    // CASE 2:  
    // If the key is somewhere other than at head  
    //  
  
    // Search for the key to be deleted,  
    // keep track of the previous node  
    // as it is needed to change currNode.next  
    while (currNode != null && currNode.data != key) {  
        // If currNode does not hold key  
        // continue to next node  
        prev = currNode;  
        currNode = currNode.next;  
    }  
  
    // If the key was present, it should be at currNode  
    // Therefore the currNode shall not be null  
    if (currNode != null) {  
        // Since the key is at currNode  
        // Unlink currNode from linked list  
        prev.next = currNode.next;  
  
        // Display the message  
        System.out.println(key + " found and deleted");  
    }  
}
```

```
//  
// CASE 3: The key is not present  
  
// If key was not present in linked list  
// currNode should be null  
if (currNode == null) {  
    // Display the message  
    System.out.println(key + " not found");  
}  
  
// return the List  
return list;  
}  
  
// *****MAIN METHOD*****  
  
// method to create a Singly linked list with n nodes  
public static void main(String[] args)  
{  
    /* Start with the empty list. */  
    LinkedList list = new LinkedList();  
  
    //  
    // *****INSERTION*****  
    //  
  
    // Insert the values  
    list = insert(list, 1);  
    list = insert(list, 2);  
    list = insert(list, 3);  
    list = insert(list, 4);  
    list = insert(list, 5);  
    list = insert(list, 6);  
    list = insert(list, 7);  
    list = insert(list, 8);  
  
    // Print the LinkedList  
    printList(list);  
  
    //  
    // *****DELETION BY KEY*****  
    //  
  
    // Delete node with value 1  
    // In this case the key is ***at head***  
    deleteByKey(list, 1);
```

```
// Print the LinkedList  
printList(list);  
  
// Delete node with value 4  
// In this case the key is present ***in the middle***  
deleteByKey(list, 4);  
  
// Print the LinkedList  
printList(list);  
  
// Delete node with value 10  
// In this case the key is ***not present***  
deleteByKey(list, 10);  
  
// Print the LinkedList  
printList(list);  
}  
}
```

**Output:**

```
LinkedList: 1 2 3 4 5 6 7 8  
1 found and deleted  
LinkedList: 2 3 4 5 6 7 8  
4 found and deleted  
LinkedList: 2 3 5 6 7 8  
10 not found  
LinkedList: 2 3 5 6 7 8
```

This deletion process can be understood as follows:

**To be done:**

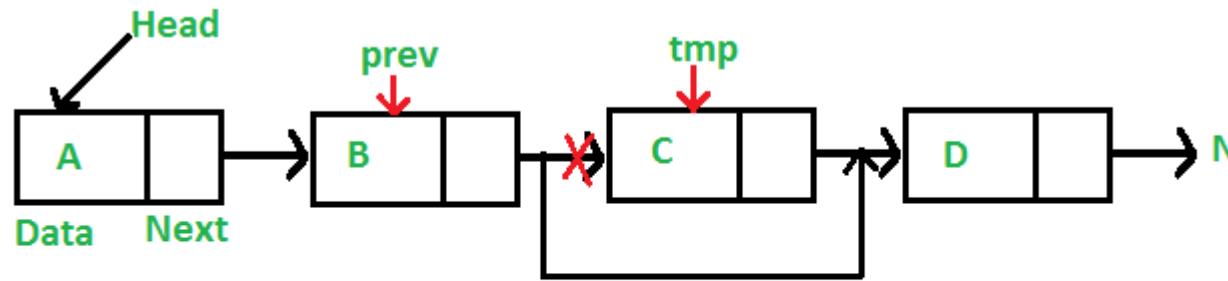
*Given a ‘position’, delete the node at this position from the linked list.*

**How to do it:**

The steps to do it are as follows:

1. Traverse the list by counting the index of the nodes
2. For each index, match the index to be same as position
3. Now, Any of the 3 conditions can be there:
  - **Case 1: The position is 0, i.e. the head is to be deleted**
    - (a) In this case, Change the head of the node to the next node of current head.
    - (b) Free the memory of replaced head node.
  - **Case 2: The position is greater than 0 but less than the size of the list, i.e. in the middle or last, except at head**
    - (a) In this case, Find previous node of the node to be deleted.

- (b) Change the next of previous node to the next node of current node.
- (c) Free the memory of replaced node.
- **Case 3: The position is greater than the size of the list, i.e. position not found in the list**
  - (a) In this case, No operation needs to be done.



```

import java.io.*;

// Java program to implement
// a Singly Linked List
public class LinkedList {

    Node head; // head of list

    // Linked list Node.
    // This inner class is made static
    // so that main() can access it
    static class Node {

        int data;
        Node next;

        // Constructor
        Node(int d)
        {
            data = d;
            next = null;
        }
    }

    // Method to insert a new node
}

```

```
public static LinkedList insert(LinkedList list, int data)
{
    // Create a new node with given data
    Node new_node = new Node(data);
    new_node.next = null;

    // If the Linked List is empty,
    // then make the new node as head
    if (list.head == null) {
        list.head = new_node;
    }
    else {
        // Else traverse till the last node
        // and insert the new_node there
        Node last = list.head;
        while (last.next != null) {
            last = last.next;
        }

        // Insert the new_node at last node
        last.next = new_node;
    }

    // Return the list by head
    return list;
}

// Method to print the LinkedList.
public static void printList(LinkedList list)
{
    Node currNode = list.head;

    System.out.print("LinkedList: ");

    // Traverse through the LinkedList
    while (currNode != null) {
        // Print the data at current node
        System.out.print(currNode.data + " ");

        // Go to next node
        currNode = currNode.next;
    }

    System.out.println();
}

// Method to delete a node in the LinkedList by POSITION
public static LinkedList deleteAtPosition(LinkedList list, int index)
```

```
{  
    // Store head node  
    Node currNode = list.head, prev = null;  
  
    //  
    // CASE 1:  
    // If index is 0, then head node itself is to be deleted  
  
    if (index == 0 && currNode != null) {  
        list.head = currNode.next; // Changed head  
  
        // Display the message  
        System.out.println(index + " position element deleted");  
  
        // Return the updated List  
        return list;  
    }  
  
    //  
    // CASE 2:  
    // If the index is greater than 0 but less than the size of LinkedList  
    //  
    // The counter  
    int counter = 0;  
  
    // Count for the index to be deleted,  
    // keep track of the previous node  
    // as it is needed to change currNode.next  
    while (currNode != null) {  
  
        if (counter == index) {  
            // Since the currNode is the required position  
            // Unlink currNode from linked list  
            prev.next = currNode.next;  
  
            // Display the message  
            System.out.println(index + " position element deleted");  
            break;  
        }  
        else {  
            // If current position is not the index  
            // continue to next node  
            prev = currNode;  
            currNode = currNode.next;  
            counter++;  
        }  
    }  
}
```

```
// If the position element was found, it should be at currNode
// Therefore the currNode shall not be null
//
// CASE 3: The index is greater than the size of the LinkedList
//
// In this case, the currNode should be null
if (currNode == null) {
    // Display the message
    System.out.println(index + " position element not found");
}

// return the List
return list;
}

// ****MAIN METHOD****

// method to create a Singly linked list with n nodes
public static void main(String[] args)
{
    /* Start with the empty list. */
    LinkedList list = new LinkedList();

    //
    // *****INSERTION*****
    //

    // Insert the values
    list = insert(list, 1);
    list = insert(list, 2);
    list = insert(list, 3);
    list = insert(list, 4);
    list = insert(list, 5);
    list = insert(list, 6);
    list = insert(list, 7);
    list = insert(list, 8);

    // Print the LinkedList
    printList(list);

    //
    // *****DELETION AT POSITION*****
    //

    // Delete node at position 0
    // In this case the key is ***at head***
    deleteAtPosition(list, 0);
```

```
// Print the LinkedList  
printList(list);  
  
// Delete node at position 2  
// In this case the key is present ***in the middle***  
deleteAtPosition(list, 2);  
  
// Print the LinkedList  
printList(list);  
  
// Delete node at position 10  
// In this case the key is ***not present***  
deleteAtPosition(list, 10);  
  
// Print the LinkedList  
printList(list);  
}  
}
```

**Output:**

```
LinkedList: 1 2 3 4 5 6 7 8  
0 position element deleted  
LinkedList: 2 3 4 5 6 7 8  
2 position element deleted  
LinkedList: 2 3 5 6 7 8  
10 position element not found  
LinkedList: 2 3 5 6 7 8
```

Below is the complete program that applies each operations together:

```
import java.io.*;  
  
// Java program to implement  
// a Singly Linked List  
public class LinkedList {  
  
    Node head; // head of list  
  
    // Linked list Node.  
    // This inner class is made static  
    // so that main() can access it  
    static class Node {  
  
        int data;  
        Node next;  
    }
```

```
// Constructor
Node(int d)
{
    data = d;
    next = null;
}
}

// *****INSERTION*****

// Method to insert a new node
public static LinkedList insert(LinkedList list, int data)
{
    // Create a new node with given data
    Node new_node = new Node(data);
    new_node.next = null;

    // If the Linked List is empty,
    // then make the new node as head
    if (list.head == null) {
        list.head = new_node;
    }
    else {
        // Else traverse till the last node
        // and insert the new_node there
        Node last = list.head;
        while (last.next != null) {
            last = last.next;
        }

        // Insert the new_node at last node
        last.next = new_node;
    }

    // Return the list by head
    return list;
}

// *****TRAVERSAL*****


// Method to print the LinkedList.
public static void printList(LinkedList list)
{
    Node currNode = list.head;

    System.out.print("\nLinkedList: ");
}
```

```
// Traverse through the LinkedList
while (currNode != null) {
    // Print the data at current node
    System.out.print(currNode.data + " ");

    // Go to next node
    currNode = currNode.next;
}
System.out.println("\n");
}

// *****DELETION BY KEY*****

// Method to delete a node in the LinkedList by KEY
public static LinkedList deleteByKey(LinkedList list, int key)
{
    // Store head node
    Node currNode = list.head, prev = null;

    //
    // CASE 1:
    // If head node itself holds the key to be deleted

    if (currNode != null && currNode.data == key) {
        list.head = currNode.next; // Changed head

        // Display the message
        System.out.println(key + " found and deleted");

        // Return the updated List
        return list;
    }

    //
    // CASE 2:
    // If the key is somewhere other than at head
    //

    // Search for the key to be deleted,
    // keep track of the previous node
    // as it is needed to change currNode.next
    while (currNode != null && currNode.data != key) {
        // If currNode does not hold key
        // continue to next node
        prev = currNode;
        currNode = currNode.next;
    }
}
```

```
// If the key was present, it should be at currNode
// Therefore the currNode shall not be null
if (currNode != null) {
    // Since the key is at currNode
    // Unlink currNode from linked list
    prev.next = currNode.next;

    // Display the message
    System.out.println(key + " found and deleted");
}

//
// CASE 3: The key is not present
//

// If key was not present in linked list
// currNode should be null
if (currNode == null) {
    // Display the message
    System.out.println(key + " not found");
}

// return the List
return list;
}

// *****DELETION AT A POSITION*****

// Method to delete a node in the LinkedList by POSITION
public static LinkedList deleteAtPosition(LinkedList list, int index)
{
    // Store head node
    Node currNode = list.head, prev = null;

    //
    // CASE 1:
    // If index is 0, then head node itself is to be deleted

    if (index == 0 && currNode != null) {
        list.head = currNode.next; // Changed head

        // Display the message
        System.out.println(index + " position element deleted");

        // Return the updated List
        return list;
    }
}
```

```
//  
// CASE 2:  
// If the index is greater than 0 but less than the size of LinkedList  
//  
// The counter  
int counter = 0;  
  
// Count for the index to be deleted,  
// keep track of the previous node  
// as it is needed to change currNode.next  
while (currNode != null) {  
  
    if (counter == index) {  
        // Since the currNode is the required position  
        // Unlink currNode from linked list  
        prev.next = currNode.next;  
  
        // Display the message  
        System.out.println(index + " position element deleted");  
        break;  
    }  
    else {  
        // If current position is not the index  
        // continue to next node  
        prev = currNode;  
        currNode = currNode.next;  
        counter++;  
    }  
}  
  
// If the position element was found, it should be at currNode  
// Therefore the currNode shall not be null  
//  
// CASE 3: The index is greater than the size of the LinkedList  
//  
// In this case, the currNode should be null  
if (currNode == null) {  
    // Display the message  
    System.out.println(index + " position element not found");  
}  
  
// return the List  
return list;  
}  
  
// *****MAIN METHOD*****  
  
// method to create a Singly linked list with n nodes
```

```
public static void main(String[] args)
{
    /* Start with the empty list. */
    LinkedList list = new LinkedList();

    //
    // *****INSERTION*****
    //

    // Insert the values
    list = insert(list, 1);
    list = insert(list, 2);
    list = insert(list, 3);
    list = insert(list, 4);
    list = insert(list, 5);
    list = insert(list, 6);
    list = insert(list, 7);
    list = insert(list, 8);

    // Print the LinkedList
    printList(list);

    //
    // *****DELETION BY KEY*****
    //

    // Delete node with value 1
    // In this case the key is ***at head***
    deleteByKey(list, 1);

    // Print the LinkedList
    printList(list);

    // Delete node with value 4
    // In this case the key is present ***in the middle***
    deleteByKey(list, 4);

    // Print the LinkedList
    printList(list);

    // Delete node with value 10
    // In this case the key is ***not present***
    deleteByKey(list, 10);

    // Print the LinkedList
    printList(list);

    //
}
```

```
// *****DELETION AT POSITION*****
//
// Delete node at position 0
// In this case the key is ***at head***
deleteAtPosition(list, 0);

// Print the LinkedList
printList(list);

// Delete node at position 2
// In this case the key is present ***in the middle***
deleteAtPosition(list, 2);

// Print the LinkedList
printList(list);

// Delete node at position 10
// In this case the key is ***not present***
deleteAtPosition(list, 10);

// Print the LinkedList
printList(list);
}
}
```

**Output:**

```
LinkedList: 1 2 3 4 5 6 7 8
```

```
1 found and deleted
```

```
LinkedList: 2 3 4 5 6 7 8
```

```
4 found and deleted
```

```
LinkedList: 2 3 5 6 7 8
```

```
10 not found
```

```
LinkedList: 2 3 5 6 7 8
```

```
0 position element deleted
```

```
LinkedList: 3 5 6 7 8
```

```
2 position element deleted
```

```
LinkedList: 3 5 7 8  
10 position element not found  
LinkedList: 3 5 7 8
```

## Source

<https://www.geeksforgeeks.org/implementing-a-linked-list-in-java-using-class/>

## Chapter 111

# In-place Merge two linked lists without changing links of first list

In-place Merge two linked lists without changing links of first list - GeeksforGeeks

Given two sorted singly linked lists having n and m elements each, merge them using constant space. First n smallest elements in both the lists should become part of first list and rest elements should be part of second list. Sorted order should be maintained. We are not allowed to change pointers of first linked list.

For example,

**Input:**

First List: 2->4->7->8->10  
Second List: 1->3->12

**Output:**

First List: 1->2->3->4->7  
Second List: 8->10->12

**We strongly recommend you to minimize your browser and try this yourself first.**

The problem becomes very simple if we're allowed to change pointers of first linked list. If we are allowed to change links, we can simply do something like merge of merge-sort algorithm. We assign first n smallest elements to the first linked list where n is the number of elements in first linked list and the rest to second linked list. We can achieve this in  $O(m + n)$  time and  $O(1)$  space, but this solution violates the requirement that we can't change links of first list.

The problem becomes a little tricky as we're not allowed to change pointers in first linked list. The idea is something similar to [this](#) post but as we are given singly linked list, we can't proceed backwards with the last element of LL2.

The idea is for each element of LL1, we compare it with first element of LL2. If LL1 has a greater element than first element of LL2, then we swap the two elements involved. To keep LL2 sorted, we need to place first element of LL2 at its correct position. We can find mismatch by traversing LL2 once and correcting the pointers.

Below is C++ implementation of this idea.

```
// Program to merge two sorted linked lists without
// using any extra space and without changing links
// of first list
#include <bits/stdc++.h>
using namespace std;

/* Structure for a linked list node */
struct Node
{
    int data;
    struct Node *next;
};

/* Given a reference (pointer to pointer) to the head
   of a list and an int, push a new node on the front
   of the list. */
void push(struct Node** head_ref, int new_data)
{
    /* allocate node */
    struct Node* new_node =
        (struct Node*) malloc(sizeof(struct Node));

    /* put in the data */
    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;
}

// Function to merge two sorted linked lists
// LL1 and LL2 without using any extra space.
void mergeLists(struct Node *a, struct Node * &b)
{
    // run till either one of a or b runs out
    while (a && b)
```

```
{  
    // for each element of LL1,  
    // compare it with first element of LL2.  
    if (a->data > b->data)  
    {  
        // swap the two elements involved  
        // if LL1 has a greater element  
        swap(a->data, b->data);  
  
        struct Node *temp = b;  
  
        // To keep LL2 sorted, place first  
        // element of LL2 at its correct place  
        if (b->next && b->data > b->next->data)  
        {  
            b = b->next;  
            struct Node *ptr= b, *prev = NULL;  
  
            // find mismatch by traversing the  
            // second linked list once  
            while (ptr && ptr->data < temp->data)  
            {  
                prev = ptr;  
                ptr = ptr -> next;  
            }  
  
            // correct the pointers  
            prev->next = temp;  
            temp->next = ptr;  
        }  
    }  
  
    // move LL1 pointer to next element  
    a = a->next;  
}  
}  
  
// Code to print the linked link  
void printList(struct Node *head)  
{  
    while (head)  
    {  
        cout << head->data << "->" ;  
        head = head->next;  
    }  
    cout << "NULL" << endl;  
}
```

```
// Driver code
int main()
{
    struct Node *a = NULL;
    push(&a, 10);
    push(&a, 8);
    push(&a, 7);
    push(&a, 4);
    push(&a, 2);

    struct Node *b = NULL;
    push(&b, 12);
    push(&b, 3);
    push(&b, 1);

    mergeLists(a, b);

    cout << "First List: ";
    printList(a);

    cout << "Second List: ";
    printList(b);

    return 0;
}
```

Output :

```
First List: 1->2->3->4->7->NULL
Second List: 8->10->12->NULL
```

Time Complexity : O( $m n$ )

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/in-place-merge-two-linked-list-without-changing-links-of-first-list/>

## Chapter 112

# In-place conversion of Sorted DLL to Balanced BST

In-place conversion of Sorted DLL to Balanced BST - GeeksforGeeks

Given a Doubly Linked List which has data members sorted in ascending order. Construct a [Balanced Binary Search Tree](#) which has same data members as the given Doubly Linked List. The tree must be constructed in-place (No new node should be allocated for tree conversion)

Examples:

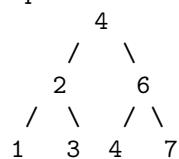
Input: Doubly Linked List 1 2 3

Output: A Balanced BST



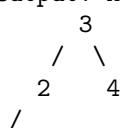
Input: Doubly Linked List 1 2 3 4 5 6 7

Output: A Balanced BST



Input: Doubly Linked List 1 2 3 4

Output: A Balanced BST



1

Input: Doubly Linked List 1 2 3 4 5 6  
Output: A Balanced BST

```
        4
       /   \
      2     6
     / \   /
    1   3  5
```

The Doubly Linked List conversion is very much similar to [this Singly Linked List problem](#) and the method 1 is exactly same as the method 1 of [previous post](#). Method 2 is also almost same. The only difference in method 2 is, instead of allocating new nodes for BST, we reuse same DLL nodes. We use prev pointer as left and next pointer as right.

### Method 1 (Simple)

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 linked list 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.

Time complexity:  $O(n\log n)$  where  $n$  is the number of nodes in Linked List.

### Method 2 (Tricky)

The method 1 constructs the tree from root to leaves. In this method, we construct from leaves to root. The idea is to insert nodes in BST in the same order as they appear in Doubly Linked List, so that the tree can be constructed in  $O(n)$  time complexity. We first count the number of nodes in the given Linked List. Let the count be  $n$ . After counting nodes, we take left  $n/2$  nodes and recursively construct the left subtree. After left subtree is constructed, we assign middle node to root and link the left subtree with root. Finally, we recursively construct the right subtree and link it with root.

While constructing the BST, we also keep moving the list head pointer to next so that we have the appropriate pointer in each recursive call.

Following is C implementation of method 2. The main code which creates Balanced BST is highlighted.

C++

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

/* A Doubly Linked List node that will also be used as a tree node */
```

```
struct Node
{
    int data;

    // For tree, next pointer can be used as right subtree pointer
    struct Node* next;

    // For tree, prev pointer can be used as left subtree pointer
    struct Node* prev;
};

// A utility function to count nodes in a Linked List
int countNodes(struct Node *head);

struct Node* sortedListToBSTRecur(struct Node **head_ref, int n);

/* This function counts the number of nodes in Linked List and then calls
   sortedListToBSTRecur() to construct BST */
struct Node* sortedListToBST(struct Node *head)
{
    /*Count the number of nodes in Linked List */
    int n = countNodes(head);

    /* Construct BST */
    return sortedListToBSTRecur(&head, n);
}

/* The main function that constructs balanced BST and returns root of it.
   head_ref --> Pointer to pointer to head node of Doubly linked list
   n --> No. of nodes in the Doubly Linked List */
struct Node* sortedListToBSTRecur(struct Node **head_ref, int n)
{
    /* Base Case */
    if (n <= 0)
        return NULL;

    /* Recursively construct the left subtree */
    struct Node *left = sortedListToBSTRecur(head_ref, n/2);

    /* head_ref now refers to middle node, make middle node as root of BST*/
    struct Node *root = *head_ref;

    // Set pointer to left subtree
    root->prev = left;

    /* Change head pointer of Linked List for parent recursive calls */
    *head_ref = (*head_ref)->next;
}
```

```

/* Recursively construct the right subtree and link it with root
   The number of nodes in right subtree is total nodes - nodes in
   left subtree - 1 (for root) */
root->next = sortedListToBSTRecur(head_ref, n-n/2-1);

return root;
}

/* UTILITY FUNCTIONS */
/* A utility function that returns count of nodes in a given Linked List */
int countNodes(struct Node *head)
{
    int count = 0;
    struct Node *temp = head;
    while(temp)
    {
        temp = temp->next;
        count++;
    }
    return count;
}

/* Function to insert a node at the beginning of the Doubly Linked List */
void push(struct Node** head_ref, int new_data)
{
    /* allocate node */
    struct Node* new_node =
        (struct Node*) malloc(sizeof(struct Node));

    /* put in the data */
    new_node->data = new_data;

    /* since we are adding at the begining,
       prev is always NULL */
    new_node->prev = NULL;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* change prev of head node to new node */
    if((*head_ref) != NULL)
        (*head_ref)->prev = new_node ;

    /* move the head to point to the new node */
    (*head_ref) = new_node;
}

/* Function to print nodes in a given linked list */

```

```
void printList(struct Node *node)
{
    while (node!=NULL)
    {
        printf("%d ", node->data);
        node = node->next;
    }
}

/* A utility function to print preorder traversal of BST */
void preOrder(struct Node* node)
{
    if (node == NULL)
        return;
    printf("%d ", node->data);
    preOrder(node->prev);
    preOrder(node->next);
}

/* Driver program to test above functions*/
int main()
{
    /* Start with the empty list */
    struct Node* head = NULL;

    /* Let us create a sorted linked list to test the functions
     * Created linked list will be 7->6->5->4->3->2->1 */
    push(&head, 7);
    push(&head, 6);
    push(&head, 5);
    push(&head, 4);
    push(&head, 3);
    push(&head, 2);
    push(&head, 1);

    printf("Given Linked List\n");
    printList(head);

    /* Convert List to BST */
    struct Node *root = sortedListToBST(head);
    printf("\n PreOrder Traversal of constructed BST \n ");
    preOrder(root);

    return 0;
}
```

Java

```
class Node
{
    int data;
    Node next, prev;

    Node(int d)
    {
        data = d;
        next = prev = null;
    }
}

class LinkedList
{
    Node head;

    /* This function counts the number of nodes in Linked List
       and then calls sortedListToBSTRecur() to construct BST */
    Node sortedListToBST()
    {
        /*Count the number of nodes in Linked List */
        int n = countNodes(head);

        /* Construct BST */
        return sortedListToBSTRecur(n);
    }

    /* The main function that constructs balanced BST and
       returns root of it.
       n --> No. of nodes in the Doubly Linked List */
    Node sortedListToBSTRecur(int n)
    {
        /* Base Case */
        if (n <= 0)
            return null;

        /* Recursively construct the left subtree */
        Node left = sortedListToBSTRecur(n / 2);

        /* head_ref now refers to middle node,
           make middle node as root of BST*/
        Node root = head;

        // Set pointer to left subtree
        root.prev = left;

        /* Change head pointer of Linked List for parent
           recursive calls */
    }
}
```

```
head = head.next;

/* Recursively construct the right subtree and link it
   with root. The number of nodes in right subtree is
   total nodes - nodes in left subtree - 1 (for root) */
root.next = sortedListToBSTRecur(n - n / 2 - 1);

return root;
}

/* UTILITY FUNCTIONS */
/* A utility function that returns count of nodes in a
   given Linked List */
int countNodes(Node head)
{
    int count = 0;
    Node temp = head;
    while (temp != null)
    {
        temp = temp.next;
        count++;
    }
    return count;
}

/* Function to insert a node at the beginning of
   the Doubly Linked List */
void push(int new_data)
{
    /* allocate node */
    Node new_node = new Node(new_data);

    /* since we are adding at the beginning,
       prev is always NULL */
    new_node.prev = null;

    /* link the old list off the new node */
    new_node.next = head;

    /* change prev of head node to new node */
    if (head != null)
        head.prev = new_node;

    /* move the head to point to the new node */
    head = new_node;
}

/* Function to print nodes in a given linked list */
```

```
void printList()
{
    Node node = head;
    while (node != null)
    {
        System.out.print(node.data + " ");
        node = node.next;
    }
}

/* A utility function to print preorder traversal of BST */
void preOrder(Node node)
{
    if (node == null)
        return;
    System.out.print(node.data + " ");
    preOrder(node.prev);
    preOrder(node.next);
}

/* Drier program to test above functions */
public static void main(String[] args)
{
    LinkedList llist = new LinkedList();

    /* Let us create a sorted linked list to test the functions
       Created linked list will be 7->6->5->4->3->2->1 */
    llist.push(7);
    llist.push(6);
    llist.push(5);
    llist.push(4);
    llist.push(3);
    llist.push(2);
    llist.push(1);

    System.out.println("Given Linked List ");
    llist.printList();

    /* Convert List to BST */
    Node root = llist.sortedListToBST();
    System.out.println("");
    System.out.println("Pre-Order Traversal of constructed BST ");
    llist.preOrder(root);
}
}

// This code has been contributed by Mayank Jaiswal(mayank_24)
```

Output:

Given Linked List

1 2 3 4 5 6 7

Pre-Order Traversal of constructed BST

4 2 1 3 6 5 7

Time Complexity: O(n)

### Source

<https://www.geeksforgeeks.org/in-place-conversion-of-sorted-dll-to-balanced-bst/>

## Chapter 113

# Insert a node after the n-th node from the end

Insert a node after the n-th node from the end - GeeksforGeeks

Insert a node **x** after the **nth** node from the end in the given singly linked list. It is guaranteed that the list contains the **nth** node from the end. Also  $1 \leq n$ .

Examples:

```
Input : list: 1->3->4->5
        n = 4, x = 2
Output : 1->2->3->4->5
4th node from the end is 1 and
insertion has been done after this node.

Input : list: 10->8->3->12->5->18
        n = 2, x = 11
Output : 10->8->3->12->5->11->18
```

### Method 1 (Using length of the list):

Find the length of the linked list, i.e, the number of nodes in the list. Let it be **len**. Now traverse the list from the 1st node upto the **(len-n+1)th** node from the beginning and insert the new node after this node. This method requires two traversals of the list.

```
// C++ implementation to insert a node after
// the n-th node from the end
#include <bits/stdc++.h>
using namespace std;

// structure of a node
struct Node {
```

```
int data;
Node* next;
};

// 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->next = NULL;
    return newNode;
}

// function to insert a node after the
// nth node from the end
void insertAfterNthNode(Node* head, int n, int x)
{
    // if list is empty
    if (head == NULL)
        return;

    // get a new node for the value 'x'
    Node* newNode = getNode(x);
    Node* ptr = head;
    int len = 0, i;

    // find length of the list, i.e, the
    // number of nodes in the list
    while (ptr != NULL) {
        len++;
        ptr = ptr->next;
    }

    // traverse up to the nth node from the end
    ptr = head;
    for (i = 1; i <= (len - n); i++)
        ptr = ptr->next;

    // insert the 'newNode' by making the
    // necessary adjustment in the links
    newNode->next = ptr->next;
    ptr->next = newNode;
}

// function to print the list
```

```
void printList(Node* head)
{
    while (head != NULL) {
        cout << head->data << " ";
        head = head->next;
    }
}

// Driver program to test above
int main()
{
    // Creating list 1->3->4->5
    Node* head = getNode(1);
    head->next = getNode(3);
    head->next->next = getNode(4);
    head->next->next->next = getNode(5);

    int n = 4, x = 2;

    cout << "Original Linked List: ";
    printList(head);

    insertAfterNthNode(head, n, x);

    cout << "\nLinked List After Insertion: ";
    printList(head);

    return 0;
}
```

Output:

```
Original Linked List: 1 3 4 5
Linked List After Insertion: 1 2 3 4 5
```

Time Complexity: O(n), where **n** is the number of nodes in the list.

#### Method 2 (Single traversal):

This method uses two pointers, one is **slow\_ptr** and the other is **fast\_ptr**. First move the **fast\_ptr** up to the **n<sup>th</sup>** node from the beginning. Make the **slow\_ptr** point to the 1st node of the list. Now, simultaneously move both the pointers until **fast\_ptr** points to the last node. At this point the **slow\_ptr** will be pointing to the **n<sup>th</sup>** node from the end. Insert the new node after this node. This method requires single traversal of the list.

```
// C++ implementation to insert a node after the
// nth node from the end
#include <bits/stdc++.h>
```

```
using namespace std;

// structure of a node
struct Node {
    int data;
    Node* next;
};

// 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->next = NULL;
    return newNode;
}

// function to insert a node after the
// nth node from the end
void insertAfterNthNode(Node* head, int n, int x)
{
    // if list is empty
    if (head == NULL)
        return;

    // get a new node for the value 'x'
    Node* newNode = getNode(x);

    // Initializing the slow and fast pointers
    Node* slow_ptr = head;
    Node* fast_ptr = head;

    // move 'fast_ptr' to point to the nth node
    // from the beginning
    for (int i = 1; i <= n - 1; i++)
        fast_ptr = fast_ptr->next;

    // iterate until 'fast_ptr' points to the
    // last node
    while (fast_ptr->next != NULL) {

        // move both the pointers to the
        // respective next nodes
        slow_ptr = slow_ptr->next;
        fast_ptr = fast_ptr->next;
    }

    // insert the new node after the nth node
    slow_ptr->next = newNode;
}
```

```
    fast_ptr = fast_ptr->next;
}

// insert the 'newNode' by making the
// necessary adjustment in the links
newNode->next = slow_ptr->next;
slow_ptr->next = newNode;
}

// function to print the list
void printList(Node* head)
{
    while (head != NULL) {
        cout << head->data << " ";
        head = head->next;
    }
}

// Driver program to test above
int main()
{
    // Creating list 1->3->4->5
    Node* head = getNode(1);
    head->next = getNode(3);
    head->next->next = getNode(4);
    head->next->next->next = getNode(5);

    int n = 4, x = 2;

    cout << "Original Linked List: ";
    printList(head);

    insertAfterNthNode(head, n, x);

    cout << "\nLinked List After Insertion: ";
    printList(head);

    return 0;
}
```

Output:

```
Original Linked List: 1 3 4 5
Linked List After Insertion: 1 2 3 4 5
```

Time Complexity: O(n), where **n** is the number of nodes in the list.

**Source**

<https://www.geeksforgeeks.org/insert-node-n-th-node-end/>

## Chapter 114

# Insert a node at a specific position in a linked list

Insert a node at a specific position in a linked list - GeeksforGeeks

Given a singly linked list, a position and an element, the task is to write a program to insert that element in a linked list at a given position.

### Examples:

Input: 3->5->8->10, data = 2, position = 2  
Output: 3->2->5->8->10

Input: 3->5->8->10, data = 11, position = 5  
Output: 3->5->8->10->11

**Approach:** To insert a given data at a specified position, the below algorithm is to be followed:

- Traverse the Linked list upto *position-1* nodes.
- Once all the *position-1* nodes are traversed, allocate memory and the given data to the new node.
- Point the next pointer of the new node to the next of current node.
- Point the next pointer of current node to the new node.

Below is the implementation of the above algorithm.

```
// C++ program for insertion in a single linked
// list at a specified position
#include <bits/stdc++.h>
using namespace std;
```

```
// A linked list Node
struct Node {
    int data;
    struct Node* next;
};

// Size of linked list
int size = 0;

// function to create and return a Node
Node* getNode(int data)
{
    // allocating space
    Node* newNode = new Node();

    // inserting the required data
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}

// function to insert a Node at required position
void insertPos(Node** current, int pos, int data)
{
    // This condition to check whether the
    // position given is valid or not.
    if (pos < 1 || pos > size + 1)
        cout << "Invalid position!" << endl;
    else {

        // Keep looping until the pos is zero
        while (pos--) {

            if (pos == 0) {

                // adding Node at required position
                Node* temp = getNode(data);

                // Making the new Node to point to
                // the old Node at the same position
                temp->next = *current;

                // Changing the pointer of the Node previous
                // to the old Node to point to the new Node
                *current = temp;
            }
        }
    }
}
```

```
// Assign double pointer variable to point to the
// pointer pointing to the address of next Node
current = &(*current)->next;
}
size++;
}
}

// This function prints contents
// of the linked list
void printList(struct Node* head)
{
    while (head != NULL) {
        cout << " " << head->data;
        head = head->next;
    }
    cout << endl;
}

// Driver Code
int main()
{
    // Creating the list 3->5->8->10
    Node* head = NULL;
    head = getNode(3);
    head->next = getNode(5);
    head->next->next = getNode(8);
    head->next->next->next = getNode(10);

    size = 4;

    cout << "Linked list before insertion: ";
    printList(head);

    int data = 12, pos = 3;
    insertPos(&head, pos, data);
    cout << "Linked list after insertion of 12 at position 3: ";
    printList(head);

    // front of the linked list
    data = 1, pos = 1;
    insertPos(&head, pos, data);
    cout << "Linked list after insertion of 1 at position 1: ";
    printList(head);

    // insertion at end of the linked list
    data = 15, pos = 7;
    insertPos(&head, pos, data);
```

```
cout << "Linked list after insertion of 15 at position 7: ";
printList(head);

return 0;
}
```

**Output:**

```
Linked list before insertion: 3 5 8 10
Linked list after insertion of 12 at position 3: 3 5 12 8 10
Linked list after insertion of 1 at position 1: 1 3 5 12 8 10
Linked list after insertion of 15 at position 7: 1 3 5 12 8 10 15
```

**Time Complexity:** O(N)

**Source**

<https://www.geeksforgeeks.org/insert-a-node-at-a-specific-position-in-a-linked-list/>

## Chapter 115

# Insert a whole linked list into other at k-th position

Insert a whole linked list into other at k-th position - GeeksforGeeks

Given two linked list and a number k. Insert second linked list in to first at k-th position

Examples:

```
Input : a : 1->2->3->4->5->NULL
        b : 7->8->9->10->11->NULL
        k = 2
Output :1->2->7->8->9->10->11->3->4->5->NULL

Input : a: 10->15->20->NULL
        b: 11->17->16->18->NULL
        k = 3
Output : 10->15->20->11->17->16->18->NULL
```

A pictorial representation of the problem

for  
ins  
at

1

- 1) Traverse the first linked list till k-th point
- 2) Join second linked list head node to k-th point of first linked list
- 3) Traverse the second linked list till end at
- 4) Add (k+1)th point of first linked list to the end of the second linked list

```
// A C++ program to insert a linked list in
// to another linked list at position k
#include <bits/stdc++.h>
using namespace std;

/* Structure for a linked list node */
struct Node {
    int data;
    struct Node* next;
};

// Function to insert whole linked list in
// to another linked list at position k
void insert(struct Node* head1, struct Node* head2,
            int k)
{
    // traverse the first linked list until k-th
    // point is reached
    int count = 1;
    struct Node* curr = head1;
    while (count < k)
    {
        curr = curr->next;
        count++;
    }

    // backup next node of the k-th point
    struct Node* temp = curr->next;

    // join second linked list at the kth point
    curr->next = head2;

    // traverse the second linked list till end
    while (head2->next != NULL)
        head2 = head2->next;

    // join the second part of the linked list
    // to the end
    head2->next = temp;
}

// Function to print linked list recursively
void printList(Node* head)
```

```
{  
    if (head == NULL)  
        return;  
  
    // If head is not NULL, print current node  
    // and recur for remaining list  
    cout << head->data << " ";  
    printList(head->next);  
}  
  
/* Given a reference (pointer to pointer) to the head  
   of a list and an int, insert a new node on the front  
   of the list. */  
void push(struct Node** head_ref, int new_data)  
{  
    struct Node* new_node = new Node;  
    new_node->data = new_data;  
    new_node->next = (*head_ref);  
    (*head_ref) = new_node;  
}  
  
/* Driven program to test above function */  
int main()  
{  
    /* The constructed linked lists are :  
       a: 1->2->3->4->5;  
       b: 7->8->9->10->11 */  
    struct Node* a = NULL;  
    struct Node* b = NULL;  
    int k = 2;  
  
    // first linked list  
    push(&a, 5);  
    push(&a, 4);  
    push(&a, 3);  
    push(&a, 2);  
    push(&a, 1);  
  
    // second linked list  
    push(&b, 11);  
    push(&b, 10);  
    push(&b, 9);  
    push(&b, 8);  
    push(&b, 7);  
  
    printList(a);  
    cout << "\n";
```

```
printList(b);  
  
insert(a, b, k);  
  
cout << "\nResulting linked list\t";  
printList(a);  
  
return 0;  
}
```

Output:

```
1 2 3 4 5  
7 8 9 10 11  
Resulting linked list      1 2 7 8 9 10 11 3 4 5
```

## Source

<https://www.geeksforgeeks.org/insert-whole-linked-list-k-th-position/>

## Chapter 116

# Insert node into the middle of the linked list

Insert node into the middle of the linked list - GeeksforGeeks

Given a linked list containing **n** nodes. The problem is to insert a new node with data **x** at the middle of the list. If **n** is even, then insert the new node after the  $(n/2)$ th node, else insert the new node after the  $(n+1)/2$ th node.

Examples:

```
Input : list: 1->2->4->5
        x = 3
Output : 1->2->3->4->5

Input : list: 5->10->4->32->16
        x = 41
Output : 5->10->4->41->32->16
```

### Method 1(Using length of the linked list):

Find the number of nodes or length of the linked using one traversal. Let it be **len**. Calculate **c** =  $(len/2)$ , if **len** is even, else **c** =  $(len+1)/2$ , if **len** is odd. Traverse again the first **c** nodes and insert the new node after the **c**th node.

C++

```
// C++ implementation to insert node at the middle
// of the linked list
#include <bits/stdc++.h>

using namespace std;
```

```
// structure of a node
struct Node {
    int data;
    Node* next;
};

// function to create and return a node
Node* getNode(int data)
{
    // allocating space
    Node* newNode = (Node*)malloc(sizeof(Node));

    // inserting the required data
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}

// function to insert node at the middle
// of the linked list
void insertAtMid(Node** head_ref, int x)
{
    // if list is empty
    if (*head_ref == NULL)
        *head_ref = getNode(x);
    else {

        // get a new node
        Node* newNode = getNode(x);

        Node* ptr = *head_ref;
        int len = 0;

        // calculate length of the linked list
        //, i.e, the number of nodes
        while (ptr != NULL) {
            len++;
            ptr = ptr->next;
        }

        // 'count' the number of nodes after which
        // the new node is to be inserted
        int count = ((len % 2) == 0) ? (len / 2) :
                                (len + 1) / 2;
        ptr = *head_ref;

        // 'ptr' points to the node after which
```

```
// the new node is to be inserted
while (count-- > 1)
    ptr = ptr->next;

// insert the 'newNode' and adjust the
// required links
newNode->next = ptr->next;
ptr->next = newNode;
}
}

// function to display the linked list
void display(Node* head)
{
    while (head != NULL) {
        cout << head->data << " ";
        head = head->next;
    }
}

// Driver program to test above
int main()
{
    // Creating the list 1->2->4->5
    Node* head = NULL;
    head = getNode(1);
    head->next = getNode(2);
    head->next->next = getNode(4);
    head->next->next->next = getNode(5);

    cout << "Linked list before insertion: ";
    display(head);

    int x = 3;
    insertAtMid(&head, x);

    cout << "\nLinked list after insertion: ";
    display(head);

    return 0;
}
```

**Java**

```
// Java implementation to insert node
// at the middle of the linked list
import java.util.*;
import java.lang.*;
```

```
import java.io.*;

class LinkedList
{
    static Node head; // head of list

    /* Node Class */
    static class Node {
        int data;
        Node next;

        // Constructor to create a new node
        Node(int d) {
            data = d;
            next = null;
        }
    }

    // function to insert node at the
    // middle of the linked list
    static void insertAtMid(int x)
    {
        // if list is empty
        if (head == null)
            head = new Node(x);
        else {
            // get a new node
            Node newNode = new Node(x);

            Node ptr = head;
            int len = 0;

            // calculate length of the linked list
            //, i.e, the number of nodes
            while (ptr != null) {
                len++;
                ptr = ptr.next;
            }

            // 'count' the number of nodes after which
            // the new node is to be inserted
            int count = ((len % 2) == 0) ? (len / 2) :
                           (len + 1) / 2;
            ptr = head;

            // 'ptr' points to the node after which
            // the new node is to be inserted
            while (count-- > 1)
```

```
        ptr = ptr.next;

        // insert the 'newNode' and adjust
        // the required links
        newNode.next = ptr.next;
        ptr.next = newNode;
    }
}

// function to display the linked list
static void display()
{
    Node temp = head;
    while (temp != null)
    {
        System.out.print(temp.data + " ");
        temp = temp.next;
    }
}

// Driver program to test above
public static void main (String[] args)
{
    // Creating the list 1.2.4.5
    head = null;
    head = new Node(1);
    head.next = new Node(2);
    head.next.next = new Node(4);
    head.next.next.next = new Node(5);

    System.out.println("Linked list before "+
                       "insertion: ");
    display();

    int x = 3;
    insertAtMid(x);

    System.out.println("\nLinked list after"+
                       " insertion: ");
    display();
}
}

// This article is contributed by Chhavi
```

Output:

```
Linked list before insertion: 1 2 4 5  
Linked list after insertion: 1 2 3 4 5
```

Time Complexity: O(n)

**Method 2(Using two pointers):**

Based on the tortoise and hare algorithm which uses two pointers, one known as **slow** and the other known as **fast**. This algorithm helps in finding the middle node of the linked list. It is explained in the front and black split procedure of [this](#) post. Now, you can insert the new node after the middle node obtained from the above process. This approach requires only a single traversal of the list.

C++

```
// C++ implementation to insert node at the middle  
// of the linked list  
#include <bits/stdc++.h>  
  
using namespace std;  
  
// structure of a node  
struct Node {  
    int data;  
    Node* next;  
};  
  
// function to create and return a node  
Node* getNode(int data)  
{  
    // allocating space  
    Node* newNode = (Node*)malloc(sizeof(Node));  
  
    // inserting the required data  
    newNode->data = data;  
    newNode->next = NULL;  
}  
  
// function to insert node at the middle  
// of the linked list  
void insertAtMid(Node** head_ref, int x)  
{  
    // if list is empty  
    if (*head_ref == NULL)  
        *head_ref = getNode(x);  
  
    else {  
        // get a new node  
        Node* newNode = getNode(x);  
        ...  
    }  
}
```

```
// assign values to the slow and fast
// pointers
Node* slow = *head_ref;
Node* fast = (*head_ref)->next;

while (fast && fast->next) {

    // move slow pointer to next node
    slow = slow->next;

    // move fast pointer two nodes at a time
    fast = fast->next->next;
}

// insert the 'newNode' and adjust the
// required links
newNode->next = slow->next;
slow->next = newNode;
}

// function to display the linked list
void display(Node* head)
{
    while (head != NULL) {
        cout << head->data << " ";
        head = head->next;
    }
}

// Driver program to test above
int main()
{
    // Creating the list 1->2->4->5
    Node* head = NULL;
    head = getNode(1);
    head->next = getNode(2);
    head->next->next = getNode(4);
    head->next->next->next = getNode(5);

    cout << "Linked list before insertion: ";
    display(head);

    int x = 3;
    insertAtMid(&head, x);

    cout << "\nLinked list after insertion: ";
    display(head);
}
```

```
    return 0;
}

Java

// Java implementation to insert node
// at the middle of the linked list
import java.util.*;
import java.lang.*;
import java.io.*;

class LinkedList
{
    static Node head; // head of list

    /* Node Class */
    static class Node {
        int data;
        Node next;

        // Constructor to create a new node
        Node(int d) {
            data = d;
            next = null;
        }
    }

    // function to insert node at the
    // middle of the linked list
    static void insertAtMid(int x)
    {
        // if list is empty
        if (head == null)
            head = new Node(x);

        else {
            // get a new node
            Node newNode = new Node(x);

            // assign values to the slow
            // and fast pointers
            Node slow = head;
            Node fast = head.next;

            while (fast != null && fast.next
                   != null)
            {

```

```
// move slow pointer to next node
slow = slow.next;

// move fast pointer two nodes
// at a time
fast = fast.next.next;
}

// insert the 'newNode' and adjust
// the required links
newNode.next = slow.next;
slow.next = newNode;
}

}

// function to display the linked list
static void display()
{
    Node temp = head;
    while (temp != null)
    {
        System.out.print(temp.data + " ");
        temp = temp.next;
    }
}

// Driver program to test above
public static void main (String[] args)
{
    // Creating the list 1.2.4.5
    head = null;
    head = new Node(1);
    head.next = new Node(2);
    head.next.next = new Node(4);
    head.next.next.next = new Node(5);

    System.out.println("Linked list before"+
                      " insertion: ");
    display();

    int x = 3;
    insertAtMid(x);

    System.out.println("\nLinked list after"+
                      " insertion: ");
    display();
}
```

```
// This article is contributed by Chhavi
```

Output:

```
Linked list before insertion: 1 2 4 5  
Linked list after insertion: 1 2 3 4 5
```

Time Complexity: O(n)

### **Source**

<https://www.geeksforgeeks.org/insert-node-middle-linked-list/>

## Chapter 117

# Insert value in sorted way in a sorted doubly linked list

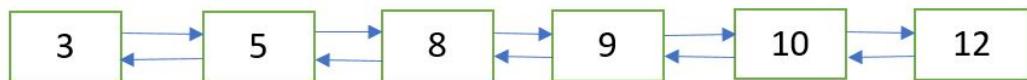
Insert value in sorted way in a sorted doubly linked list - GeeksforGeeks

Given a sorted doubly linked list and a value to insert, write a function to insert the value in sorted way.

Initial doubly linked list



Doubly Linked List after insertion of 9



### Algorithm:

Let input doubly linked list is sorted in increasing order.

New node passed to the function contains data in the data part and previous and next link are set to NULL.

```
sortedInsert(head_ref, newNode)
    if (head_ref == NULL)
        head_ref = newNode

    else if head_ref->data >= newNode->data
        newNode->next = head_ref
        newNode->next->prev = newNode
```

```
head_ref = newNode

else
    Initialize current = head_ref
    while (current->next != NULL and
           current->next->data < data)
        current = current->next

    newNode->next = current->next
    if current->next != NULL
        newNode->next->prev = newNode

    current->next = newNode
    newNode->prev = current

// C++ implementation to insert value in sorted way
// in a sorted doubly linked list
#include <bits/stdc++.h>

using namespace std;

// Node of a doubly linked list
struct Node {
    int data;
    struct Node* prev, *next;
};

// function to create and return a new node
// of a doubly linked list
struct Node* getNode(int data)
{
    // allocate node
    struct Node* newNode =
        (struct Node*)malloc(sizeof(struct Node));

    // put in the data
    newNode->data = data;
    newNode->prev = newNode->next = NULL;
    return newNode;
}

// function to insert a new node in sorted way in
// a sorted doubly linked list
void sortedInsert(struct Node** head_ref, struct Node* newNode)
{
    struct Node* current;

    // if list is empty
```

```
if (*head_ref == NULL)
    *head_ref = newNode;

// if the node is to be inserted at the beginning
// of the doubly linked list
else if ((*head_ref)->data >= newNode->data) {
    newNode->next = *head_ref;
    newNode->next->prev = newNode;
    *head_ref = newNode;
}

else {
    current = *head_ref;

    // locate the node after which the new node
    // is to be inserted
    while (current->next != NULL &&
           current->next->data < newNode->data)
        current = current->next;

    /* Make the appropriate links */
    newNode->next = current->next;

    // if the new node is not inserted
    // at the end of the list
    if (current->next != NULL)
        newNode->next->prev = newNode;

    current->next = newNode;
    newNode->prev = current;
}
}

// function to print the doubly linked list
void printList(struct Node* head)
{
    while (head != NULL) {
        cout << head->data << " ";
        head = head->next;
    }
}

// Driver program to test above
int main()
{
    /* start with the empty doubly linked list */
    struct Node* head = NULL;
```

```
// insert the following nodes in sorted way
struct Node* new_node = getNode(8);
sortedInsert(&head, new_node);
new_node = getNode(5);
sortedInsert(&head, new_node);
new_node = getNode(3);
sortedInsert(&head, new_node);
new_node = getNode(10);
sortedInsert(&head, new_node);
new_node = getNode(12);
sortedInsert(&head, new_node);
new_node = getNode(9);
sortedInsert(&head, new_node);

cout << "Created Doubly Linked Listn";
printList(head);
return 0;
}
```

Output:

```
Created Doubly Linked List
3 5 8 9 10 12
```

Time Complexity: O(n)

## Source

<https://www.geeksforgeeks.org/insert-value-sorted-way-sorted-doubly-linked-list/>

## Chapter 118

# Insertion Sort for Singly Linked List

Insertion Sort for Singly Linked List - GeeksforGeeks

We have discussed [Insertion Sort for arrays](#). In this article same for linked list is discussed.

Below is simple insertion sort algorithm for linked list.

- 1) Create an empty sorted (or result) list
- 2) Traverse the given list, do following for every node.
  - .....a) Insert current node in sorted way in sorted or result list.
  - 3) Change head of given linked list to head of sorted (or result) list.

The main step is (2.a) which has been covered in below post.

[Sorted Insert for Singly Linked List](#)

Below is implementation of above algorithm

C++

```
/* C program for insertion sort on a linked list */
#include<stdio.h>
#include<stdlib.h>

/* Link list node */
struct Node
{
    int data;
    struct Node* next;
};

// Function to insert a given node in a sorted linked list
```

```

void sortedInsert(struct Node**, struct Node*);

// function to sort a singly linked list using insertion sort
void insertionSort(struct Node **head_ref)
{
    // Initialize sorted linked list
    struct Node *sorted = NULL;

    // Traverse the given linked list and insert every
    // node to sorted
    struct Node *current = *head_ref;
    while (current != NULL)
    {
        // Store next for next iteration
        struct Node *next = current->next;

        // insert current in sorted linked list
        sortedInsert(&sorted, current);

        // Update current
        current = next;
    }

    // Update head_ref to point to sorted linked list
    *head_ref = sorted;
}

/* function to insert a new_node in a list. Note that this
   function expects a pointer to head_ref as this can modify the
   head of the input linked list (similar to push())*/
void sortedInsert(struct Node** head_ref, struct Node* new_node)
{
    struct Node* current;
    /* Special case for the head end */
    if (*head_ref == NULL || (*head_ref)->data >= new_node->data)
    {
        new_node->next = *head_ref;
        *head_ref = new_node;
    }
    else
    {
        /* Locate the node before the point of insertion */
        current = *head_ref;
        while (current->next!=NULL &&
               current->next->data < new_node->data)
        {
            current = current->next;
        }
    }
}

```

```
        new_node->next = current->next;
        current->next = new_node;
    }
}

/* BELOW FUNCTIONS ARE JUST UTILITY TO TEST sortedInsert */

/* Function to print linked list */
void printList(struct Node *head)
{
    struct Node *temp = head;
    while(temp != NULL)
    {
        printf("%d ", temp->data);
        temp = temp->next;
    }
}

/* A utility function to insert a node at the beginning of linked list */
void push(struct Node** head_ref, int new_data)
{
    /* allocate node */
    struct Node* new_node = new Node;

    /* put in the data */
    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;
}

// Driver program to test above functions
int main()
{
    struct Node *a = NULL;
    push(&a, 5);
    push(&a, 20);
    push(&a, 4);
    push(&a, 3);
    push(&a, 30);

    printf("Linked List before sorting \n");
    printList(a);
```

```
insertionSort(&a);

printf("\nLinked List after sorting \n");
printList(a);

return 0;
}
```

**Java**

```
// Java program to sort link list
// using insertion sort

public class LinkedlistIS
{
    node head;
    node sorted;

    class node
    {
        int val;
        node next;

        public node(int val)
        {
            this.val = val;
        }
    }

    void push(int val)
    {
        /* allocate node */
        node newnode = new node(val);
        /* link the old list off the new node */
        newnode.next = head;
        /* move the head to point to the new node */
        head = newnode;
    }

    // function to sort a singly linked list using insertion sort
    void insertionSort(node headref)
    {
        // Initialize sorted linked list
        sorted = null;
        node current = headref;
        // Traverse the given linked list and insert every
        // node to sorted
        while (current != null)
```

```
{  
    // Store next for next iteration  
    node next = current.next;  
    // insert current in sorted linked list  
    sortedInsert(current);  
    // Update current  
    current = next;  
}  
// Update head_ref to point to sorted linked list  
head = sorted;  
}  
  
/*  
 * function to insert a new_node in a list. Note that  
 * this function expects a pointer to head_ref as this  
 * can modify the head of the input linked list  
 * (similar to push())  
 */  
void sortedInsert(node newnode)  
{  
    /* Special case for the head end */  
    if (sorted == null || sorted.val >= newnode.val)  
    {  
        newnode.next = sorted;  
        sorted = newnode;  
    }  
    else  
    {  
        node current = sorted;  
        /* Locate the node before the point of insertion */  
        while (current.next != null && current.next.val < newnode.val)  
        {  
            current = current.next;  
        }  
        newnode.next = current.next;  
        current.next = newnode;  
    }  
}  
  
/* Function to print linked list */  
void printlist(node head)  
{  
    while (head != null)  
    {  
        System.out.print(head.val + " ");  
        head = head.next;  
    }  
}
```

```
// Driver program to test above functions
public static void main(String[] args)
{
    LinkedlistIS list = new LinkedlistIS();
    list.push(5);
    list.push(20);
    list.push(4);
    list.push(3);
    list.push(30);
    System.out.println("Linked List before Sorting..");
    list.printlist(list.head);
    list.insertionSort(list.head);
    System.out.println("\nLinkedList After sorting");
    list.printlist(list.head);
}
}

// This code is contributed by Rishabh Mahrsee
```

```
Linked List before sorting
30 3 4 20 5
Linked List after sorting
3 4 5 20 30
```

## Source

<https://www.geeksforgeeks.org/insertion-sort-for-singly-linked-list/>

## Chapter 119

# Insertion at Specific Position in a Circular Doubly Linked List

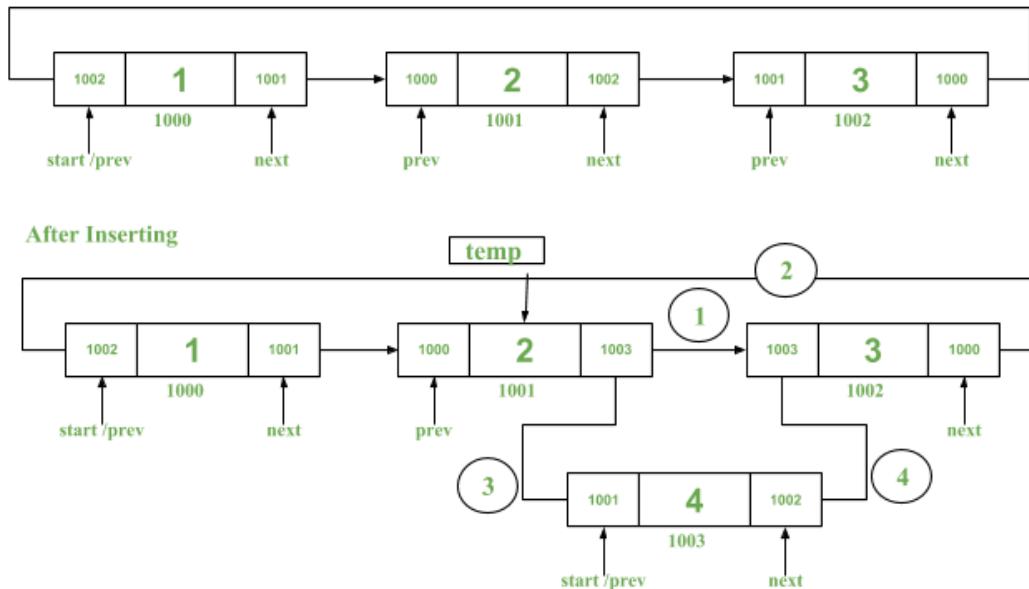
Insertion at Specific Position in a Circular Doubly Linked List - GeeksforGeeks

**Prerequisite:**

- [Insert Element Circular Doubly Linked List.](#)
- [Convert an Array to a Circular Doubly Linked List.](#)

Given the *start* pointer pointing to the start of a Circular Doubly Linked List, an *element* and a *position*. The task is to insert the *element* at the specified *position* in the given Circular Doubly Linked List.

### Linked List | Insert At Location 3:



The idea is to count the total number of elements in the list. Check whether the specified location is valid or not, i.e. location is within the count.

If location is valid:

1. Create a newNode in the memory.
2. Traverse in the list using a temporary pointer(**temp**) till node just before the given position at which new node is needed to be inserted.
3. Insert the new node by performing below operations:
  - Assign `newNode->next = temp->next`
  - Assign `newNode->prev as temp->next`
  - Assign `temp->next as newNode`
  - Assign `(temp->next)->prev as newNode->next`

Below is the implementation of the above idea:

```
// CPP program to convert insert an element at a specific
// position in a circular doubly linked list

#include <iostream>
using namespace std;

// Doubly linked list node
struct node {
    int data;
    node *next;
    node *prev;
}
```

```
    struct node* next;
    struct node* prev;
};

// Utility function to create a node in memory
struct node* getNode()
{
    return ((struct node*)malloc(sizeof(struct node)));
}

// Function to display the list
int displayList(struct node* temp)
{
    struct node* t = temp;
    if (temp == NULL)
        return 0;
    else {
        cout << "The list is: ";

        while (temp->next != t) {
            cout << temp->data << " ";
            temp = temp->next;
        }

        cout << temp->data << endl;

        return 1;
    }
}

// Function to count number of
// elements in the list
int countList(struct node* start)
{
    // Decalre temp pointer to
    // traverse the list
    struct node* temp = start;

    // Variable to store the count
    int count = 0;

    // Iterate the list and increment the count
    while (temp->next != start) {
        temp = temp->next;
        count++;
    }

    // As the list is circular, increment the
}
```

```
// counter at last
count++;

return count;
}

// Function to insert a node at a given position
// in the circular doubly linked list
bool insertAtLocation(struct node* start, int data, int loc)
{
    // Declare two pointers
    struct node *temp, *newNode;
    int i, count;

    // Create a new node in memory
    newNode = getNode();

    // Point temp to start
    temp = start;

    // count of total elements in the list
    count = countList(start);

    // If list is empty or the position is
    // not valid, return false
    if (temp == NULL || count < loc)
        return false;

    else {
        // Assign the data
        newNode->data = data;

        // Iterate till the loc
        for (i = 1; i < loc - 1; i++) {
            temp = temp->next;
        }

        // See in Image, circle 1
        newNode->next = temp->next;

        // See in Image, Circle 2
        (temp->next)->prev = newNode;

        // See in Image, Circle 3
        temp->next = newNode;

        // See in Image, Circle 4
        newNode->prev = temp;
    }
}
```

```
        return true;
    }

    return false;
}

// Function to create circular doubly linked list
// from array elements
void createList(int arr[], int n, struct node** start)
{
    // Declare newNode and temporary pointer
    struct node *newNode, *temp;
    int i;

    // Iterate the loop until array length
    for (i = 0; i < n; i++) {
        // Create new node
        newNode = getNode();

        // Assign the array data
        newNode->data = arr[i];

        // If it is first element
        // Put that node prev and next as start
        // as it is circular
        if (i == 0) {
            *start = newNode;
            newNode->prev = *start;
            newNode->next = *start;
        }
        else {
            // Find the last node
            temp = (*start)->prev;

            // Add the last node to make them
            // in circular fashion
            temp->next = newNode;
            newNode->next = *start;
            newNode->prev = temp;
            temp = *start;
            temp->prev = newNode;
        }
    }
}

// Driver Code
```

```
int main()
{
    // Array elements to create
    // circular doubly linked list
    int arr[] = { 1, 2, 3, 4, 5, 6 };
    int n = sizeof(arr) / sizeof(arr[0]);

    // Start Pointer
    struct node* start = NULL;

    // Create the List
    createList(arr, n, &start);

    // Display the list before insertion
    displayList(start);

    // Inserting 8 at 3rd position
    insertAtLocation(start, 8, 3);

    // Display the list after insertion
    displayList(start);

    return 0;
}
```

**Output:**

```
The list is: 1 2 3 4 5 6
The list is: 1 2 8 3 4 5 6
```

**Time Complexity:**  $O(n) \Rightarrow$  for counting the list,  $O(n) \Rightarrow$  Inserting the elements. So, total complexity is  $O(n + n) = O(n)$

**Source**

<https://www.geeksforgeeks.org/insertion-at-specific-position-in-a-circular-doubly-linked-list/>

## Chapter 120

# Insertion in Unrolled Linked List

Insertion in Unrolled Linked List - GeeksforGeeks

An unrolled linked list is a linked list of small arrays, all of the same size where each is so small that the insertion or deletion is fast and quick, but large enough to fill the cache line. An iterator pointing into the list consists of both a pointer to a node and an index into that node containing an array. It is also a data structure and is another variant of Linked List. It is related to B-Tree. It can store an array of elements at a node unlike a normal linked list which stores single element at a node. It is combination of arrays and linked list fusion-ed into one. It increases cache performance and decreases the memory overhead associated with storing reference for metadata. Other major advantages and disadvantages are already mentioned in the previous article.

**Prerequisite :** [Introduction to Unrolled Linked List](#)

Below is the insertion and display operation of Unrolled Linked List.

```
Input : 72 76 80 94 90 70
        capacity = 3
Output : Unrolled Linked List :
72 76
80 94
90 70
Explanation : The working is well shown in the
algorithm below. The nodes get broken at the
mentioned capacity i.e., 3 here, when 3rd element
is entered, the flow moves to another newly created
node. Every node contains an array of size
(int)[(capacity / 2) + 1]. Here it is 2.
```

```
Input : 49 47 62 51 77 17 71 71 35 76 36 54
        capacity = 5
Output :
Unrolled Linked List :
49 47 62
51 77 17
71 71 35
76 36 54
Explanation : The working is well shown in the
algorithm below. The nodes get broken at the
mentioned capacity i.e., 5 here, when 5th element
is entered, the flow moves to another newly
created node. Every node contains an array of
size (int)[(capacity / 2) + 1]. Here it is 3.
```

**Algorithm :**

```
Insert (ElementToBeInserted)
if start_pos == NULL
    Insert the first element into the first node
    start_pos.numElement ++
    end_pos = start_pos
If end_pos.numElements + 1 < node_size
    end_pos.numElements.push(newElement)
    end_pos.numElements ++
else
    create a new Node new_node
    move final half of end_pos.data into new_node.data
    new_node.data.push(newElement)
    end_pos.numElements = end_pos.data.size / 2 + 1
    end_pos.next = new_node
    end_pos = new_node
```

Following is the Java implementation of the insertion and display operation. In the below code, the capacity is 5 and random numbers are input.

**Java**

```
/* Java program to show the insertion operation
 * of Unrolled Linked List */
import java.util.Scanner;
import java.util.Random;

// class for each node
class UnrollNode {
    UnrollNode next;
```

```
int num_elements;
int array[];

// Constructor
public UnrollNode(int n)
{
    next = null;
    num_elements = 0;
    array = new int[n];
}

// Operation of Unrolled Function
class UnrollLinkedList {

    private UnrollNode start_pos;
    private UnrollNode end_pos;

    int size_node;
    int nNode;

    // Parameterized Constructor
    UnrollLinkedList(int capacity)
    {
        start_pos = null;
        end_pos = null;
        nNode = 0;
        size_node = capacity + 1;
    }

    // Insertion operation
    void Insert(int num)
    {
        nNode++;

        // Check if the list starts from NULL
        if (start_pos == null) {
            start_pos = new UnrollNode(size_node);
            start_pos.array[0] = num;
            start_pos.num_elements++;
            end_pos = start_pos;
            return;
        }

        // Attaching the elements into nodes
        if (end_pos.num_elements + 1 < size_node) {
            end_pos.array[end_pos.num_elements] = num;
            end_pos.num_elements++;
        }
    }
}
```

```
}

// Creation of new Node
else {
    UnrollNode node_pointer = new UnrollNode(size_node);
    int j = 0;
    for (int i = end_pos.num_elements / 2 + 1;
         i < end_pos.num_elements; i++)
        node_pointer.array[j++] = end_pos.array[i];

    node_pointer.array[j++] = num;
    node_pointer.num_elements = j;
    end_pos.num_elements = end_pos.num_elements / 2 + 1;
    end_pos.next = node_pointer;
    end_pos = node_pointer;
}
}

// Display the Linked List
void display()
{
    System.out.print("\nUnrolled Linked List = ");
    System.out.println();
    UnrollNode pointer = start_pos;
    while (pointer != null) {
        for (int i = 0; i < pointer.num_elements; i++)
            System.out.print(pointer.array[i] + " ");
        System.out.println();
        pointer = pointer.next;
    }
    System.out.println();
}
}

/* Main Class */
class UnrolledLinkedList_Check {

    // Driver code
    public static void main(String args[])
    {
        Scanner sc = new Scanner(System.in);

        // create instance of Random class
        Random rand = new Random();

        UnrollLinkList ull = new UnrollLinkList(5);

        // Perform Insertion Operation
    }
}
```

```
for (int i = 0; i < 12; i++) {  
  
    // Generate random integers in range 0 to 99  
    int rand_int1 = rand.nextInt(100);  
    System.out.println("Entered Element is " + rand_int1);  
    ull.Insert(rand_int1);  
    ull.display();  
}  
}  
}
```

**Output:**

Entered Element is 90

Unrolled Linked List =  
90

Entered Element is 3

Unrolled Linked List =  
90 3

Entered Element is 12

Unrolled Linked List =  
90 3 12

Entered Element is 43

Unrolled Linked List =  
90 3 12 43

Entered Element is 88

Unrolled Linked List =  
90 3 12 43 88

Entered Element is 94

Unrolled Linked List =  
90 3 12  
43 88 94

Entered Element is 15

Unrolled Linked List =

```
90 3 12  
43 88 94 15
```

```
Entered Element is 7
```

```
Unrolled Linked List =  
90 3 12  
43 88 94 15 7
```

```
Entered Element is 67
```

```
Unrolled Linked List =  
90 3 12  
43 88 94  
15 7 67
```

```
Entered Element is 74
```

```
Unrolled Linked List =  
90 3 12  
43 88 94  
15 7 67 74
```

```
Entered Element is 85
```

```
Unrolled Linked List =  
90 3 12  
43 88 94  
15 7 67 74 85
```

```
Entered Element is 48
```

```
Unrolled Linked List =  
90 3 12  
43 88 94  
15 7 67  
74 85 48
```

Time complexity : **O(n)**

Also, few real world applications :

- It is used in B-Tree and T-Tree
- Used in Hashed Array Tree
- Used in Skip List
- Used in CDR Coding

**Source**

<https://www.geeksforgeeks.org/insertion-unrolled-linked-list/>

## Chapter 121

# Intersection of two Sorted Linked Lists

Intersection of two Sorted Linked Lists - GeeksforGeeks

Given two lists sorted in increasing order, create and return a new list representing the intersection of the two lists. The new list should be made with its own memory — the original lists should not be changed.

For example, let the first linked list be 1->2->3->4->6 and second linked list be 2->4->6->8, then your function should create and return a third list as 2->4->6.

### Method 1 (Using Dummy Node)

The strategy here uses a temporary dummy node as the start of the result list. The pointer tail always points to the last node in the result list, so appending new nodes is easy. The dummy node gives tail something to point to initially when the result list is empty. This dummy node is efficient, since it is only temporary, and it is allocated in the stack. The loop proceeds, removing one node from either ‘a’ or ‘b’, and adding it to tail. When we are done, the result is in dummy.next.

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

/* Link list node */
struct Node
{
    int data;
    struct Node* next;
};

void push(struct Node** head_ref, int new_data);

/*This solution uses the temporary dummy to build up the result list */
struct Node* sortedIntersect(struct Node* a, struct Node* b)
```

```
{  
    struct Node dummy;  
    struct Node* tail = &dummy;  
    dummy.next = NULL;  
  
    /* Once one or the other list runs out -- we're done */  
    while (a != NULL && b != NULL)  
    {  
        if (a->data == b->data)  
        {  
            push((&tail->next), a->data);  
            tail = tail->next;  
            a = a->next;  
            b = b->next;  
        }  
        else if (a->data < b->data) /* advance the smaller list */  
            a = a->next;  
        else  
            b = b->next;  
    }  
    return(dummy.next);  
}  
  
/* UTILITY FUNCTIONS */  
/* Function to insert a node at the beginning of the linked list */  
void push(struct Node** head_ref, int new_data)  
{  
    /* allocate node */  
    struct Node* new_node =  
        (struct Node*) malloc(sizeof(struct Node));  
  
    /* put in the data */  
    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;  
}  
  
/* Function to print nodes in a given linked list */  
void printList(struct Node *node)  
{  
    while (node != NULL)  
    {  
        printf("%d ", node->data);  
        node = node->next;  
    }
```

```
    }

}

/* Drier program to test above functions*/
int main()
{
    /* Start with the empty lists */
    struct Node* a = NULL;
    struct Node* b = NULL;
    struct Node *intersect = NULL;

    /* Let us create the first sorted linked list to test the functions
     Created linked list will be 1->2->3->4->5->6 */
    push(&a, 6);
    push(&a, 5);
    push(&a, 4);
    push(&a, 3);
    push(&a, 2);
    push(&a, 1);

    /* Let us create the second sorted linked list
     Created linked list will be 2->4->6->8 */
    push(&b, 8);
    push(&b, 6);
    push(&b, 4);
    push(&b, 2);

    /* Find the intersection two linked lists */
    intersect = sortedIntersect(a, b);

    printf("\n Linked list containing common items of a & b \n ");
    printList(intersect);

    getchar();
}
```

Time Complexity:  $O(m+n)$  where m and n are number of nodes in first and second linked lists respectively.

#### **Method 2 (Using Local References)**

This solution is structurally very similar to the above, but it avoids using a dummy node. Instead, it maintains a `struct node**` pointer, `lastPtrRef`, that always points to the last pointer of the result list. This solves the same case that the dummy node did — dealing with the result list when it is empty. If you are trying to build up a list at its tail, either the dummy node or the `struct node**` “reference” strategy can be used.

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

```
/* Link list node */
struct Node
{
    int data;
    struct Node* next;
};

void push(struct Node** head_ref, int new_data);

/* This solution uses the local reference */
struct Node* sortedIntersect(struct Node* a, struct Node* b)
{
    struct Node* result = NULL;
    struct Node** lastPtrRef = &result;

    /* Advance comparing the first nodes in both lists.
       When one or the other list runs out, we're done. */
    while (a!=NULL && b!=NULL)
    {
        if (a->data == b->data)
        {
            /* found a node for the intersection */
            push(lastPtrRef, a->data);
            lastPtrRef = &((*lastPtrRef)->next);
            a = a->next;
            b = b->next;
        }
        else if (a->data < b->data)
            a=a->next;           /* advance the smaller list */
        else
            b=b->next;
    }
    return(result);
}

/* UTILITY FUNCTIONS */
/* Function to insert a node at the beginning of the linked list */
void push(struct Node** head_ref, int new_data)
{
    /* allocate node */
    struct Node* new_node =
        (struct Node*) malloc(sizeof(struct Node));

    /* put in the data */
    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;
}

/* Function to print nodes in a given linked list */
void printList(struct Node *node)
{
    while(node != NULL)
    {
        printf("%d ", node->data);
        node = node->next;
    }
}

/* Driver program to test above functions*/
int main()
{
    /* Start with the empty lists */
    struct Node* a = NULL;
    struct Node* b = NULL;
    struct Node *intersect = NULL;

    /* Let us create the first sorted linked list to test the functions
     * Created linked list will be 1->2->3->4->5->6 */
    push(&a, 6);
    push(&a, 5);
    push(&a, 4);
    push(&a, 3);
    push(&a, 2);
    push(&a, 1);

    /* Let us create the second sorted linked list
     * Created linked list will be 2->4->6->8 */
    push(&b, 8);
    push(&b, 6);
    push(&b, 4);
    push(&b, 2);

    /* Find the intersection two linked lists */
    intersect = sortedIntersect(a, b);

    printf("\n Linked list containing common items of a & b \n ");
    printList(intersect);

    getchar();
}
```

Time Complexity:  $O(m+n)$  where m and n are number of nodes in first and second linked lists respectively.

**Method 3 (Recursive)**

Below is the recursive implementation of sortedIntersect().

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

/* Link list node */
struct Node
{
    int data;
    struct Node* next;
};

struct Node *sortedIntersect(struct Node *a, struct Node *b)
{
    /* base case */
    if (a == NULL || b == NULL)
        return NULL;

    /* If both lists are non-empty */

    /* advance the smaller list and call recursively */
    if (a->data < b->data)
        return sortedIntersect(a->next, b);

    if (a->data > b->data)
        return sortedIntersect(a, b->next);

    // Below lines are executed only when a->data == b->data
    struct Node *temp = (struct Node *)malloc(sizeof(struct Node));
    temp->data = a->data;

    /* advance both lists and call recursively */
    temp->next = sortedIntersect(a->next, b->next);
    return temp;
}

/* UTILITY FUNCTIONS */
/* Function to insert a node at the beginning of the linked list */
void push(struct Node** head_ref, int new_data)
{
    /* allocate node */
    struct Node* new_node = (struct Node*) malloc(sizeof(struct Node));

    /* put in the data */
    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;
}

/* Function to print nodes in a given linked list */
void printList(struct Node *node)
{
    while (node != NULL)
    {
        printf("%d ", node->data);
        node = node->next;
    }
}

/* Drier program to test above functions*/
int main()
{
    /* Start with the empty lists */
    struct Node* a = NULL;
    struct Node* b = NULL;
    struct Node *intersect = NULL;

    /* Let us create the first sorted linked list to test the functions
     * Created linked list will be 1->2->3->4->5->6 */
    push(&a, 6);
    push(&a, 5);
    push(&a, 4);
    push(&a, 3);
    push(&a, 2);
    push(&a, 1);

    /* Let us create the second sorted linked list
     * Created linked list will be 2->4->6->8 */
    push(&b, 8);
    push(&b, 6);
    push(&b, 4);
    push(&b, 2);

    /* Find the intersection two linked lists */
    intersect = sortedIntersect(a, b);

    printf("\n Linked list containing common items of a & b \n ");
    printList(intersect);
```

```
    return 0;  
}
```

Time Complexity:  $O(m+n)$  where m and n are number of nodes in first and second linked lists respectively.

References:

[cslibrary.stanford.edu/105/LinkedListProblems.pdf](http://cslibrary.stanford.edu/105/LinkedListProblems.pdf)

## Source

<https://www.geeksforgeeks.org/intersection-of-two-sorted-linked-lists/>

## Chapter 122

# Iterative approach for removing middle points in a linked list of line segments

Iterative approach for removing middle points in a linked list of line segments - Geeks-forGeeks

This post explains the iterative approach of [this](#) problem.

We maintain two pointers, prev and temp. If these two have either x or y same, we move forward till the equality holds and keep deleting the nodes in between. The node from which the equality started, we adjust the next pointer of that node.

```
// C++ program to remove intermediate
// points in a linked list that represents
// horizontal and vertical line segments
#include <iostream>
using namespace std;

// Node has 3 fields including x, y
// coordinates and a pointer to next node
struct Node {
    int x, y;
    struct Node *next;
};

/* Function to insert a node at the beginning */
void push(struct Node **head_ref, int x, int y)
{
    struct Node *new_node = new Node;
    new_node->x = x;
    new_node->y = y;
```

```
new_node->next = (*head_ref);
(*head_ref) = new_node;
}

/* Utility function to print a singly linked list */
void printList(struct Node *head)
{
    struct Node *temp = head;
    while (temp != NULL) {
        printf("(%d, %d)-> ", temp->x, temp->y);
        temp = temp->next;
    }
    printf("\n");
}

// This function deletes middle nodes in a
// sequence of horizontal and vertical line
// segments represented by linked list.
void delete_Middle_Nodes(Node *head)
{
    Node *temp = head->next, *prev = head;

    while (temp) {

        // checking equality of point x
        if (temp->x == prev->x)
        {
            Node *curr = prev;
            prev = temp;
            temp = temp->next;

            // removing vertical points of line
            // segment from linked list
            while (temp && temp->x == prev->x)
            {
                curr->next = temp;
                free(prev);
                prev = temp;
                temp = temp->next;
            }
        }

        // checking equality of point y
        else if (temp->y == prev->y)
        {
            Node *curr = prev;
            prev = temp;
            temp = temp->next;
        }
    }
}
```

```
// removing horizontal points of line
// segment from linked list
while (temp && temp->y == prev->y)
{
    curr->next = temp;
    free(prev);
    prev = temp;
    temp = temp->next;
}
} else {
    prev = temp;
    temp = temp->next;
}
}

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

    push(&head, 40,5);
    push(&head, 20,5);
    push(&head, 10,5);
    push(&head, 10,8);
    push(&head, 10,10);
    push(&head, 3,10);
    push(&head, 1,10);
    push(&head, 0,10);

    printf("Given Linked List: \n");
    printList(head);

    delete_Middle_Nodes(head);

    printf("Modified Linked List: \n");
    printList(head);

    return 0;
}
```

Output:

```
Given Linked List:
(2, 3)-> (4, 3)-> (6, 3)-> (10, 3)-> (12, 3)->
Modified Linked List:
(2, 3)-> (12, 3)->
```

## Source

<https://www.geeksforgeeks.org/iterative-approach-for-removing-middle-points-in-a-linked-list-of-line-segements/>

## Chapter 123

# Iteratively Reverse a linked list using only 2 pointers (An Interesting Method)

Iteratively Reverse a linked list using only 2 pointers (An Interesting Method) - GeeksforGeeks

Given pointer to the head node of a linked list, the task is to reverse the linked list.

Examples:

Input : Head of following linked list  
1->2->3->4->NULL  
Output : Linked list should be changed to,  
4->3->2->1->NULL

Input : Head of following linked list  
1->2->3->4->5->NULL  
Output : Linked list should be changed to,  
5->4->3->2->1->NULL

We have seen how to reverse a linked list in article [Reverse a linked list](#). In **iterative method** we had used 3 pointers **prev**, **cur** and **next**. Below is an interesting approach that uses only two pointers. The idea is to use XOR to swap pointers.

C/C++

```
// C++ program to reverse a linked list using two pointers.  
#include <bits/stdc++.h>
```

```
using namespace std;
typedef uintptr_t ut;

/* Link list node */
struct Node {
    int data;
    struct Node* next;
};

/* Function to reverse the linked list using 2 pointers */
void reverse(struct Node** head_ref)
{
    struct Node* prev = NULL;
    struct Node* current = *head_ref;

    // at last prev points to new head
    while (current != NULL) {
        // This expression evaluates from left to right
        // current->next = prev, changes the link from
        // next to prev node
        // prev = current, moves prev to current node for
        // next reversal of node
        // This example of list will clear it more 1->2->3->4
        // initially prev = 1, current = 2
        // Final expression will be current = 1^2^3^2^1,
        // as we know that bitwise XOR of two same
        // numbers will always be 0 i.e; 1^1 = 2^2 = 0
        // After the evaluation of expression current = 3 that
        // means it has been moved by one node from its
        // previous position
        current = (struct Node*)((ut)prev ^ (ut)current ^ (ut)(current->next) ^ (ut)(current->next));
    }

    *head_ref = prev;
}

/* Function to push a node */
void push(struct Node** head_ref, int new_data)
{
    /* allocate node */
    struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));

    /* put in the data */
    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;
}

/* Function to print linked list */
void printList(struct Node* head)
{
    struct Node* temp = head;
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
}

/* Driver program to test above function*/
int main()
{
    /* Start with the empty list */
    struct Node* head = NULL;

    push(&head, 20);
    push(&head, 4);
    push(&head, 15);
    push(&head, 85);

    printf("Given linked list\n");
    printList(head);
    reverse(&head);
    printf("\nReversed Linked list \n");
    printList(head);
    return 0;
}
```

### Python

```
# Iteratively Reverse a linked list using only 2 pointers (An Interesting Method)
# Python program to reverse a linked list
# Link list node
# node class
class node:

    # Constructor to initialize the node object
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:
```

```
# Function to initialize head
def __init__(self):
    self.head = None

# Function to reverse the linked list
def reverse(self):
    prev = None
    current = self.head
# Described here https://www.geeksforgeeks.org/
# how-to-swap-two-variables-in-one-line / while(current is not None):
    # This expression evaluates from left to right
    # current->next = prev, changes the link from
    # next to prev node
    # prev = current, moves prev to current node for
    # next reversal of node
    # This example of list will clear it more 1->2
    # initially prev = 1, current = 2
    # Final expression will be current = 1, prev = 2
    next, current.next = current.next, prev
    prev, current = current, next
    self.head = prev

# Function to push a new node
def push(self, new_data):
    # allocate node and put in the data
    new_node = node(new_data)
    # link the old list off the new node
    new_node.next = self.head
    # move the head to point to the new node
    self.head = new_node

# Function to print the linked list
def printList(self):
    temp = self.head
    while(temp):
        print temp.data,
        temp = temp.next

# Driver program to test above functions
llist = LinkedList()
llist.push(20)
llist.push(4)
llist.push(15)
llist.push(85)

print "Given Linked List"
llist.printList()
```

```
llist.reverse()
print "\nReversed Linked List"
llist.printList()

# This code is contributed by Afzal Ansari
```

**Output:**

```
Given linked list
85 15 4 20
Reversed Linked list
20 4 15 85
```

Time Complexity: O(n)

Reference :

<http://discuss.joelonsoftware.com/default.asp?interview.11.564944.16>

**Alternate Solution :**

```
// C++ program to reverse a linked list using two pointers.
#include <bits/stdc++.h>
using namespace std;
typedef uintptr_t ut;

/* Link list node */
struct Node {
    int data;
    struct Node* next;
};

/* Function to reverse the linked list using 2 pointers */
void reverse(struct Node** head_ref)
{
    struct Node* current = *head_ref;
    struct Node* next;
    while (current->next != NULL) {
        next = current->next;
        current->next = next->next;
        next->next = (*head_ref);
        *head_ref = next;
    }
}

/* Function to push a node */
void push(struct Node** head_ref, int new_data)
{
```

```
struct Node* new_node = new Node;
new_node->data = new_data;
new_node->next = (*head_ref);
(*head_ref) = new_node;
}

/* Function to print linked list */
void printList(struct Node* head)
{
    struct Node* temp = head;
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
}

/* Driver program to test above function*/
int main()
{
    /* Start with the empty list */
    struct Node* head = NULL;

    push(&head, 20);
    push(&head, 4);
    push(&head, 15);
    push(&head, 85);

    printf("Given linked list\n");
    printList(head);
    reverse(&head);
    printf("\nReversed Linked list \n");
    printList(head);
    return 0;
}
```

**Output:**

```
Given linked list
85 15 4 20
Reversed Linked list
20 4 15 85
```

Thanks to **Abhay Yadav** for suggesting this approach.

**Source**

<https://www.geeksforgeeks.org/iteratively-reverse-a-linked-list-using-only-2-pointers/>

## Chapter 124

# Java Program for Reverse a linked list

Java Program for Reverse a linked list - GeeksforGeeks

Given pointer to the head node of a linked list, the task is to reverse the linked list. We need to reverse the list by changing links between nodes.

Examples:

```
Input : Head of following linked list  
        1->2->3->4->NULL  
Output : Linked list should be changed to,  
        4->3->2->1->NULL
```

```
Input : Head of following linked list  
        1->2->3->4->5->NULL  
Output : Linked list should be changed to,  
        5->4->3->2->1->NULL
```

```
Input : NULL  
Output : NULL
```

```
Input : 1->NULL  
Output : 1->NULL
```

### Iterative Method

#### Source

<https://www.geeksforgeeks.org/java-program-for-reverse-a-linked-list/>

**Java**

```
// Java program for reversing the linked list

class LinkedList {

    static Node head;

    static class Node {

        int data;
        Node next;

        Node(int d) {
            data = d;
            next = null;
        }
    }

    /* Function to reverse the linked list */
    Node reverse(Node node) {
        Node prev = null;
        Node current = node;
        Node next = null;
        while (current != null) {
            next = current.next;
            current.next = prev;
            prev = current;
            current = next;
        }
        node = prev;
        return node;
    }

    // prints content of double linked list
    void printList(Node node) {
        while (node != null) {
            System.out.print(node.data + " ");
            node = node.next;
        }
    }

    public static void main(String[] args) {
        LinkedList list = new LinkedList();
        list.head = new Node(85);
        list.head.next = new Node(15);
        list.head.next.next = new Node(4);
        list.head.next.next.next = new Node(20);
    }
}
```

```
        System.out.println("Given Linked list");
        list.printList(head);
        head = list.reverse(head);
        System.out.println("");
        System.out.println("Reversed linked list ");
        list.printList(head);
    }
}

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

**Recursive Method:**

```
void recursiveReverse(struct Node** head_ref)
{
    struct Node* first;
    struct Node* rest;

    /* empty list */
    if (*head_ref == NULL)
        return;

    /* suppose first = {1, 2, 3}, rest = {2, 3} */
    first = *head_ref;
    rest = first->next;

    /* List has only one node */
    if (rest == NULL)
        return;

    /* reverse the rest list and put the first element at the end */
    recursiveReverse(&rest);
    first->next->next = first;

    /* tricky step -- see the diagram */
    first->next = NULL;

    /* fix the head pointer */
    *head_ref = rest;
}
```

**A Simpler and Tail Recursive Method**

**Java**

```
// Java program for reversing the Linked list

class LinkedList {

    static Node head;

    static class Node {

        int data;
        Node next;

        Node(int d) {
            data = d;
            next = null;
        }
    }

    // A simple and tail recursive function to reverse
    // a linked list. prev is passed as NULL initially.
    Node reverseUtil(Node curr, Node prev) {

        /* If last node mark it head*/
        if (curr.next == null) {
            head = curr;

            /* Update next to prev node */
            curr.next = prev;
            return null;
        }

        /* Save curr->next node for recursive call */
        Node next1 = curr.next;

        /* and update next ...*/
        curr.next = prev;

        reverseUtil(next1, curr);
        return head;
    }

    // prints content of double linked list
    void printList(Node node) {
        while (node != null) {
            System.out.print(node.data + " ");
            node = node.next;
        }
    }
}
```

```
public static void main(String[] args) {
    LinkedList list = new LinkedList();
    list.head = new Node(1);
    list.head.next = new Node(2);
    list.head.next.next = new Node(3);
    list.head.next.next.next = new Node(4);
    list.head.next.next.next.next = new Node(5);
    list.head.next.next.next.next.next = new Node(6);
    list.head.next.next.next.next.next.next = new Node(7);
    list.head.next.next.next.next.next.next.next = new Node(8);

    System.out.println("Original Linked list ");
    list.printList(head);
    Node res = list.reverseUtil(head, null);
    System.out.println("");
    System.out.println("");
    System.out.println("Reversed linked list ");
    list.printList(res);
}
}

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

Please refer complete article on [Reverse a linked list](#) for more details!

## Chapter 125

# Josephus Circle using circular linked list

Josephus Circle using circular linked list - GeeksforGeeks

There are n people standing in a circle waiting to be executed. The counting out begins at some point in the circle and proceeds around the circle in a fixed direction. In each step, a certain number of people are skipped and the next person is executed. The elimination proceeds around the circle (which is becoming smaller and smaller as the executed people are removed), until only the last person remains, who is given freedom. Given the total number of persons n and a number m which indicates that m-1 persons are skipped and m-th person is killed in circle. The task is to choose the place in the initial circle so that you are the last one remaining and so survive.

**Examples :**

```
Input : Length of circle : n = 4
        Count to choose next : m = 2
Output : 1

Input : n = 5
        m = 3
Output : 4
```

We have discussed different solutions of this problem ([here](#) and [here](#)). In this post a simple [circular linked list](#) based solution is discussed.

- 1) Create a circular linked list of size n.
- 2) Traverse through linked list and one by one delete every m-th node until there is one node left.
- 3) Return value of the only left node.

**C++**

```
// CPP program to find last man standing
#include<bits/stdc++.h>
using namespace std;

/* structure for a node in circular
   linked list */
struct Node
{
    int data;
    struct Node *next;
};

// To create a new node of circular
// linked list
Node *newNode(int data)
{
    Node *temp = new Node;
    temp->next = temp;
    temp->data = data;
}

/* Function to find the only person left
   after one in every m-th node is killed
   in a circle of n nodes */
void getJosephusPosition(int m, int n)
{
    // Create a circular linked list of
    // size N.
    Node *head = newNode(1);
    Node *prev = head;
    for (int i = 2; i <= n; i++)
    {
        prev->next = newNode(i);
        prev = prev->next;
    }
    prev->next = head; // Connect last
                       // node to first

    /* while only one node is left in the
       linked list*/
    Node *ptr1 = head, *ptr2 = head;
    while (ptr1->next != ptr1)
    {
        // Find m-th node
        int count = 1;
        while (count != m)
        {
            ptr2 = ptr1;
            count++;
        }
        ptr1 = ptr1->next;
    }
}
```

```
        ptr1 = ptr1->next;
        count++;
    }

    /* Remove the m-th node */
    ptr2->next = ptr1->next;
    ptr1 = ptr2->next;
}

printf ("Last person left standing "
        "(Josephus Position) is %d\n ",
        ptr1->data);
}

/* Driver program to test above functions */
int main()
{
    int n = 14, m = 2;
    getJosephusPosition(m, n);
    return 0;
}
```

### Java

```
// Java Code to find the last man Standing
public class GFG {

    // Node class to store data
    static class Node
    {
        public int data ;
        public Node next;
        public Node( int data )
        {
            this.data = data;
        }
    }

    /* Function to find the only person left
    after one in every m-th node is killed
    in a circle of n nodes */
    static void getJosephusPosition(int m, int n)
    {
        // Create a circular linked list of
        // size N.
        Node head = new Node(1);
        Node prev = head;
        for(int i = 2; i <= n; i++)
```

```
{  
    prev.next = new Node(i);  
    prev = prev.next;  
}  
  
// Connect last node to first  
prev.next = head;  
  
/* while only one node is left in the  
linked list*/  
Node ptr1 = head, ptr2 = head;  
  
while(ptr1.next != ptr1)  
{  
  
    // Find m-th node  
    int count = 1;  
    while(count != m)  
    {  
        ptr2 = ptr1;  
        ptr1 = ptr1.next;  
        count++;  
    }  
  
    /* Remove the m-th node */  
    ptr2.next = ptr1.next;  
    ptr1 = ptr2.next;  
}  
System.out.println ("Last person left standing " +  
                    "(Josephus Position) is " + ptr1.data);  
}  
  
/* Driver program to test above functions */  
public static void main(String args[])  
{  
    int n = 14, m = 2;  
    getJosephusPosition(m, n);  
}  
}
```

Output :

Last person left standing (Josephus Position) is 13

Time complexity:  $O(m * n)$

Improved By : [Sargam Modak](#), [Devansh Upadhyay](#)

**Source**

<https://www.geeksforgeeks.org/josephus-circle-using-circular-linked-list/>

# Chapter 126

# Large number arithmetic using doubly linked list

Large number arithmetic using doubly linked list - GeeksforGeeks

Given two very large numbers in form of strings. Your task is to apply different arithmetic operations on these strings.

**Prerequisite :** Doubly Linked List.

Examples:

```
Input :  
m : 55  
n : 2  
Output :  
Product : 110  
Sum : 57  
Difference : 53  
Quotient : 27  
Remainder(%) : 1
```

**Approach :** Split the number into digits in a doubly linked list. Using basic addition principles that goes digit by digit, with a carry are implemented in add and subtract functions. These functions are now used to carry out multiplication and division operations using the basic approach of multiplying last digit to all and then shifting and adding or finding the closest large multiple to divisor to divide the dividend. Finally the results are displayed using the display function.

**Below is the implementation of above approach :**

```
// CPP problem to illustrate arithmetic operations of
// very large numbers using Doubly Linked List
#include <bits/stdc++.h>
using namespace std;

// Structure of Double Linked List
struct node {

    // To store a single digit
    int data;

    // Pointers to the previous and next digit
    struct node* next;
    struct node* prev;
    node(int);
};

// To initialize the structure with a single digit
node::node(int val)
{
    data = val;
    next = prev = NULL;
}

class HugeInt {
public:
    HugeInt();
    ~HugeInt();

    // To insert a digit in front
    void insertInFront(int);

    // To insert a digit at the end
    void insertInEnd(int);

    // To display the large number
    void display();

    int length();
    void add(HugeInt*, HugeInt*);
}
```

```
void mul(HugeInt*, HugeInt*);  
void dif(HugeInt*, HugeInt*);  
void quo(HugeInt*, HugeInt*);  
int cmp(HugeInt*, HugeInt*);  
node* head;  
node* tail;  
int size;  
};  
  
// Constructor of the Class  
HugeInt::HugeInt()  
{  
    head = tail = NULL;  
    size = 0;  
}  
  
// To insert at the beginning of the list  
void HugeInt::insertInFront(int value)  
{  
    node* temp = new node(value);  
  
    if (head == NULL)  
        head = tail = temp;  
    else {  
        head->prev = temp;  
        temp->next = head;  
        head = temp;  
    }  
    size++;  
}  
  
// To insert in the end  
void HugeInt::insertInEnd(int value)  
{  
    node* temp = new node(value);  
  
    if (tail == NULL)  
        head = tail = temp;  
    else {  
        tail->next = temp;  
        temp->prev = tail;  
        tail = temp;  
    }  
    size++;  
}  
  
/*  
To display the number can be
```

```
modified to remove leading zeros*/
void HugeInt::display()
{
    node* temp = head;

    while (temp != NULL) {
        cout << temp->data;
        temp = temp->next;
    }
}

// Returns the number of digits
int HugeInt::length()
{
    return size;
}

/*
Uses simple addition method that we
follow using carry*/
void HugeInt::add(HugeInt* a, HugeInt* b)
{
    int c = 0, s;
    HugeInt* a1 = new HugeInt(*a);
    HugeInt* b1 = new HugeInt(*b);

    // default copy constructor
    // Copy Constructor - used to copy objects
    this->head = NULL;
    this->tail = NULL;
    this->size = 0;

    while (a1->tail != NULL || b1->tail != NULL) {
        if (a1->tail != NULL && b1->tail != NULL) {
            s = ((a1->tail->data) + (b1->tail->data) + c) % 10;
            c = ((a1->tail->data) + (b1->tail->data) + c) / 10;
            a1->tail = a1->tail->prev;
            b1->tail = b1->tail->prev;
        }
        else if (a1->tail == NULL && b1->tail != NULL) {
            s = ((b1->tail->data) + c) % 10;
            c = ((b1->tail->data) + c) / 10;
            b1->tail = b1->tail->prev;
        }
        else if (a1->tail != NULL && b1->tail == NULL) {
            s = ((a1->tail->data) + c) % 10;
            c = ((a1->tail->data) + c) / 10;
            a1->tail = a1->tail->prev;
        }
    }
}
```

```

    }

    // Inserting the sum digit
    insertInFront(s);
}

// Inserting last carry
if (c != 0)
    insertInFront(c);
}

// Normal subtraction is done by borrowing
void HugeInt::dif(HugeInt* a, HugeInt* b)
{
    int c = 0, s;
    HugeInt* a1 = new HugeInt(*a);
    HugeInt* b1 = new HugeInt(*b);

    this->head = NULL;
    this->tail = NULL;
    this->size = 0;

    while (a1->tail != NULL || b1->tail != NULL) {
        if (a1->tail != NULL && b1->tail != NULL) {
            if ((a1->tail->data) + c >= (b1->tail->data)) {
                s = ((a1->tail->data) + c - (b1->tail->data));
                c = 0;
            }
            else {
                s = ((a1->tail->data) + c + 10 - (b1->tail->data));
                c = -1;
            }
            a1->tail = a1->tail->prev;
            b1->tail = b1->tail->prev;
        }
        else if (a1->tail != NULL && b1->tail == NULL) {
            if (a1->tail->data >= 1) {
                s = ((a1->tail->data) + c);
                c = 0;
            }
            else {
                if (c != 0) {
                    s = ((a1->tail->data) + 10 + c);
                    c = -1;
                }
                else
                    s = a1->tail->data;
            }
        }
    }
}

```

```

        a1->tail = a1->tail->prev;
    }
    insertInFront(s);
}
}

// This compares the two numbers and returns
// true or 1 when a is greater
int HugeInt::cmp(HugeInt* a, HugeInt* b)
{
    if (a->size != b->size)
        return ((a->size > b->size) ? 1 : 0);
    else {
        HugeInt* a1 = new HugeInt(*a);
        HugeInt* b1 = new HugeInt(*b);
        while (a1->head != NULL && b1->head != NULL) {
            if (a1->head->data > b1->head->data)
                return 1;
            else if (a1->head->data < b1->head->data)
                return 0;
            else {
                a1->head = a1->head->next;
                b1->head = b1->head->next;
            }
        }
        return 2;
    }
}

// Returns the quotient using Normal Division
// Multiplication is used to find what factor
// is to be multiplied
void HugeInt::quo(HugeInt* a, HugeInt* b)
{
    HugeInt* a1 = new HugeInt(*a);
    HugeInt* b1 = new HugeInt(*b);
    HugeInt* ex = new HugeInt();
    HugeInt* mp = new HugeInt();
    HugeInt* pr = new HugeInt();
    int i = 0;
    for (i = 0; i < b1->size; i++) {
        ex->insertInEnd(a1->head->data);
        a1->head = a1->head->next;
    }

    for (i = 0; i < 10; i++) {
        HugeInt* b2 = new HugeInt(*b);
        mp->insertInEnd(i);
    }
}
```

```

pr->mul(b2, mp);
if (!cmp(ex, pr))
    break;
mp->head = mp->tail = NULL;
pr->head = pr->tail = NULL;
mp->size = pr->size = 0;
}

mp->head = mp->tail = NULL;
pr->head = pr->tail = NULL;
mp->size = pr->size = 0;

mp->insertInEnd(i - 1);
pr->mul(b1, mp);
ex->dif(ex, pr);
insertInEnd(i - 1);
mp->head = mp->tail = NULL;
pr->head = pr->tail = NULL;
mp->size = pr->size = 0;

while (a1->head != NULL) {
    ex->insertInEnd(a1->head->data);
    while (ex->head->data == 0) {
        ex->head = ex->head->next;
        ex->size--;
    }
    for (i = 0; i < 10; i++) {
        HugeInt* b2 = new HugeInt(*b);
        mp->insertInEnd(i);
        pr->mul(b2, mp);
        if (!cmp(ex, pr))
            break;
        mp->head = mp->tail = NULL;
        pr->head = pr->tail = NULL;
        mp->size = pr->size = 0;
    }
}

mp->head = mp->tail = NULL;
pr->head = pr->tail = NULL;
mp->size = pr->size = 0;

mp->insertInEnd(i - 1);
pr->mul(b1, mp);
ex->dif(ex, pr);

insertInEnd(i - 1);

mp->head = mp->tail = NULL;

```

```
pr->head = pr->tail = NULL;
mp->size = pr->size = 0;

a1->head = a1->head->next;
}

cout << endl
<< "\nModulus :" << endl;
ex->display();
}

// Normal multiplication is used i.e. in one to all way
void HugeInt::mul(HugeInt* a, HugeInt* b)
{
    int k = 0, i;
    HugeInt* tpro = new HugeInt();
    while (b->tail != NULL) {
        int c = 0, s = 0;
        HugeInt* temp = new HugeInt(*a);
        HugeInt* pro = new HugeInt();
        while (temp->tail != NULL) {
            s = ((temp->tail->data) * (b->tail->data) + c) % 10;
            c = ((temp->tail->data) * (b->tail->data) + c) / 10;
            pro->insertInFront(s);
            temp->tail = temp->tail->prev;
        }
        if (c != 0)
            pro->insertInFront(c);

        for (i = 0; i < k; i++)
            pro->insertInEnd(0);

        add(this, pro);
        k++;
        b->tail = b->tail->prev;
        pro->head = pro->tail = NULL;
        pro->size = 0;
    }
}

// CPP problem to illustrate arithmetic operations of
// very large numbers using Doubly Linked List
#include <bits/stdc++.h>
using namespace std;

// Structure of Double Linked List
struct Node
{
```

```
// To store a single digit
int data;

// Pointers to the previous and next digit
struct Node* next;
struct Node* prev;
Node(int);
};

// To initialize the structure with a single digit
Node::Node(int val)
{
    data = val;
    next = prev = NULL;
}

class HugeIntLL
{
public:
    HugeIntLL();
    ~HugeIntLL();

    // To insert a digit in front
    void insertInFront(int);

    // To insert a digit at the end
    void insertInEnd(int);

    // To display the large number
    void display();

    int length();
    void add(HugeIntLL*, HugeIntLL*);
    void mul(HugeIntLL*, HugeIntLL*);
    void dif(HugeIntLL*, HugeIntLL*);
    void quo(HugeIntLL*, HugeIntLL*);
    int cmp(HugeIntLL*, HugeIntLL*);

    Node* head;
    Node* tail;
    int size;
};

// Constructor of the Class
HugeIntLL::HugeIntLL()
{
    head = tail = NULL;
    size = 0;
}
```

```
// To insert at the beginning of the list
void HugeIntLL::insertInFront(int value)
{
    Node* temp = new Node(value);

    if (head == NULL)
        head = tail = temp;
    else
    {
        head->prev = temp;
        temp->next = head;
        head = temp;
    }
    size++;
}

// To insert in the end
void HugeIntLL::insertInEnd(int value)
{
    Node* temp = new Node(value);

    if (tail == NULL)
        head = tail = temp;
    else
    {
        tail->next = temp;
        temp->prev = tail;
        tail = temp;
    }
    size++;
}

/*
To display the number can be
modified to remove leading zeros*/
void HugeIntLL::display()
{
    Node* temp = head;

    while (temp != NULL)
    {
        cout << temp->data;
        temp = temp->next;
    }
}

// Returns the number of digits
```

```

int HugeIntLL::length()
{
    return size;
}

/* Uses simple addition method that we
   follow using carry*/
void HugeIntLL::add(HugeIntLL* a, HugeIntLL* b)
{
    int c = 0, s;
    HugeIntLL* a1 = new HugeIntLL(*a);
    HugeIntLL* b1 = new HugeIntLL(*b);

    // default copy constructor
    // Copy Constructor - used to copy objects
    this->head = NULL;
    this->tail = NULL;
    this->size = 0;

    while (a1->tail != NULL || b1->tail != NULL)
    {
        if (a1->tail != NULL && b1->tail != NULL)
        {
            s = ((a1->tail->data) + (b1->tail->data) + c) % 10;
            c = ((a1->tail->data) + (b1->tail->data) + c) / 10;
            a1->tail = a1->tail->prev;
            b1->tail = b1->tail->prev;
        }
        else if (a1->tail == NULL && b1->tail != NULL)
        {
            s = ((b1->tail->data) + c) % 10;
            c = ((b1->tail->data) + c) / 10;
            b1->tail = b1->tail->prev;
        }
        else if (a1->tail != NULL && b1->tail == NULL)
        {
            s = ((a1->tail->data) + c) % 10;
            c = ((a1->tail->data) + c) / 10;
            a1->tail = a1->tail->prev;
        }

        // Inserting the sum digit
        insertInFront(s);
    }

    // Inserting last carry
    if (c != 0)
        insertInFront(c);
}

```

```
}

// Normal subtraction is done by borrowing
void HugeIntLL::dif(HugeIntLL* a, HugeIntLL* b)
{
    int c = 0, s;
    HugeIntLL* a1 = new HugeIntLL(*a);
    HugeIntLL* b1 = new HugeIntLL(*b);

    this->head = NULL;
    this->tail = NULL;
    this->size = 0;

    while (a1->tail != NULL || b1->tail != NULL)
    {
        if (a1->tail != NULL && b1->tail != NULL)
        {
            if ((a1->tail->data) + c >= (b1->tail->data))
            {
                s = ((a1->tail->data) + c - (b1->tail->data));
                c = 0;
            }
            else
            {
                s = ((a1->tail->data) + c + 10 - (b1->tail->data));
                c = -1;
            }
            a1->tail = a1->tail->prev;
            b1->tail = b1->tail->prev;
        }
        else if (a1->tail != NULL && b1->tail == NULL)
        {
            if (a1->tail->data >= 1)
            {
                s = ((a1->tail->data) + c);
                c = 0;
            }
            else
            {
                if (c != 0)
                {
                    s = ((a1->tail->data) + 10 + c);
                    c = -1;
                }
                else
                    s = a1->tail->data;
            }
            a1->tail = a1->tail->prev;
        }
    }
}
```

```

        }
        insertInFront(s);
    }
}

// This compares the two numbers and returns
// true or 1 when a is greater
int HugeIntLL::cmp(HugeIntLL* a, HugeIntLL* b)
{
    if (a->size != b->size)
        return ((a->size > b->size) ? 1 : 0);

    HugeIntLL* a1 = new HugeIntLL(*a);
    HugeIntLL* b1 = new HugeIntLL(*b);
    while (a1->head != NULL && b1->head != NULL)
    {
        if (a1->head->data > b1->head->data)
            return 1;
        else if (a1->head->data < b1->head->data)
            return 0;
        else
        {
            a1->head = a1->head->next;
            b1->head = b1->head->next;
        }
    }
    return 2;
}

// Returns the quotient using Normal Division
// Multiplication is used to find what factor
// is to be multiplied
void HugeIntLL::quo(HugeIntLL* a, HugeIntLL* b)
{
    HugeIntLL* a1 = new HugeIntLL(*a);
    HugeIntLL* b1 = new HugeIntLL(*b);
    HugeIntLL* ex = new HugeIntLL();
    HugeIntLL* mp = new HugeIntLL();
    HugeIntLL* pr = new HugeIntLL();
    int i = 0;
    for (i = 0; i < b1->size; i++)
    {
        ex->insertInEnd(a1->head->data);
        a1->head = a1->head->next;
    }

    for (i = 0; i < 10; i++)
    {

```

```

HugeIntLL* b2 = new HugeIntLL(*b);
mp->insertInEnd(i);
pr->mul(b2, mp);
if (!cmp(ex, pr))
    break;
mp->head = mp->tail = NULL;
pr->head = pr->tail = NULL;
mp->size = pr->size = 0;
}

mp->head = mp->tail = NULL;
pr->head = pr->tail = NULL;
mp->size = pr->size = 0;

mp->insertInEnd(i - 1);
pr->mul(b1, mp);
ex->dif(ex, pr);
insertInEnd(i - 1);
mp->head = mp->tail = NULL;
pr->head = pr->tail = NULL;
mp->size = pr->size = 0;

while (a1->head != NULL)
{
    ex->insertInEnd(a1->head->data);
    while (ex->head->data == 0)
    {
        ex->head = ex->head->next;
        ex->size--;
    }
    for (i = 0; i < 10; i++)
    {
        HugeIntLL* b2 = new HugeIntLL(*b);
        mp->insertInEnd(i);
        pr->mul(b2, mp);
        if (!cmp(ex, pr))
            break;
        mp->head = mp->tail = NULL;
        pr->head = pr->tail = NULL;
        mp->size = pr->size = 0;
    }

    mp->head = mp->tail = NULL;
    pr->head = pr->tail = NULL;
    mp->size = pr->size = 0;

    mp->insertInEnd(i - 1);
    pr->mul(b1, mp);
}

```

```

ex->dif(ex, pr);

insertInEnd(i - 1);

mp->head = mp->tail = NULL;
pr->head = pr->tail = NULL;
mp->size = pr->size = 0;

a1->head = a1->head->next;
}

cout << endl
    << "\nModulus :" << endl;
ex->display();
}

// Normal multiplication is used i.e. in one to all way
void HugeIntLL::mul(HugeIntLL* a, HugeIntLL* b)
{
    int k = 0, i;
    HugeIntLL* tpro = new HugeIntLL();
    while (b->tail != NULL)
    {
        int c = 0, s = 0;
        HugeIntLL* temp = new HugeIntLL(*a);
        HugeIntLL* pro = new HugeIntLL();
        while (temp->tail != NULL)
        {
            s = ((temp->tail->data) * (b->tail->data) + c) % 10;
            c = ((temp->tail->data) * (b->tail->data) + c) / 10;
            pro->insertInFront(s);
            temp->tail = temp->tail->prev;
        }
        if (c != 0)
            pro->insertInFront(c);

        for (i = 0; i < k; i++)
            pro->insertInEnd(0);

        add(this, pro);
        k++;
        b->tail = b->tail->prev;
        pro->head = pro->tail = NULL;
        pro->size = 0;
    }
}

// Driver code

```

```
int main()
{
    HugeIntLL* m = new HugeIntLL();
    HugeIntLL* n = new HugeIntLL();
    HugeIntLL* s = new HugeIntLL();
    HugeIntLL* p = new HugeIntLL();
    HugeIntLL* d = new HugeIntLL();
    HugeIntLL* q = new HugeIntLL();

    string s1 = "12345678912345678912345678"
                "9123456789123456789123456789";
    string s2 = "45678913456789123456789123456"
                "789123456789123456789";

    for (int i = 0; i < s1.length(); i++)
        m->insertInEnd(s1.at(i) - '0');

    for (int i = 0; i < s2.length(); i++)
        n->insertInEnd(s2.at(i) - '0');

    // Creating copies of m and n
    HugeIntLL* m1 = new HugeIntLL(*m);
    HugeIntLL* n1 = new HugeIntLL(*n);
    HugeIntLL* m2 = new HugeIntLL(*m);
    HugeIntLL* n2 = new HugeIntLL(*n);
    HugeIntLL* m3 = new HugeIntLL(*m);
    HugeIntLL* n3 = new HugeIntLL(*n);

    cout << "Product :" << endl;
    s->mul(m, n);
    s->display();
    cout << endl;

    cout << "Sum :" << endl;
    p->add(m1, n1);
    p->display();
    cout << endl;

    cout << "Difference (m-n) : m>n:" << endl;

    d->dif(m2, n2);
    d->display();
    q->quo(m3, n3);
    cout << endl;

    cout << "Quotient :" << endl;
    q->display();
    return 0;
}
```

}

## Output:

**Time Complexity :**  $O(N)$ , where  $N$  is the size of string.

## Source

<https://www.geeksforgeeks.org/large-number-arithmetic-using-doubly-linked-list/>

## Chapter 127

# Length of longest palindrome list in a linked list using O(1) extra space

Length of longest palindrome list in a linked list using O(1) extra space - GeeksforGeeks

Given a linked list, find length of the longest palindrome list that exist in that linked list.

Examples:

```
Input  : List = 2->3->7->3->2->12->24
Output : 5
The longest palindrome list is 2->3->7->3->2

Input  : List = 12->4->4->3->14
Output : 2
The longest palindrome list is 4->4
```

A simple solution could be to copy linked list content to array and then find longest palindromic subarray in array, but this solution is not allowed as it requires extra space.

The idea is based on [iterative linked list reverse process](#). We iterate through given linked list and one by one reverse every prefix of linked list from left. After reversing a prefix, we find the longest common list beginning from reversed prefix and list after the reversed prefix.

Below is C++ implementation of above idea.

```
// C++ program to find longest palindrome
// sublist in a list in O(1) time.
#include<bits/stdc++.h>
using namespace std;
```

```
//structure of the linked list
struct Node
{
    int data;
    struct Node* next;
};

// function for counting the common elements
int countCommon(Node *a, Node *b)
{
    int count = 0;

    // loop to count coomon in the list starting
    // from node a and b
    for (; a && b; a = a->next, b = b->next)

        // increment the count for same values
        if (a->data == b->data)
            ++count;
        else
            break;

    return count;
}

// Returns length of the longest palindrome
// sublist in given list
int maxPalindrome(Node *head)
{
    int result = 0;
    Node *prev = NULL, *curr = head;

    // loop till the end of the linked list
    while (curr)
    {
        // The sublist from head to current
        // reversed.
        Node *next = curr->next;
        curr->next = prev;

        // check for odd length palindrome
        // by finding longest common list elements
        // beginning from prev and from next (We
        // exclude curr)
        result = max(result,
                    2*countCommon(prev, next)+1);
    }
}
```

```
// check for even length palindrome
// by finding longest common list elements
// beginning from curr and from next
result = max(result,
              2*countCommon(curr, next));

// update prev and curr for next iteration
prev = curr;
curr = next;
}
return result;
}

// Utility function to create a new list node
Node *newNode(int key)
{
    Node *temp = new Node;
    temp->data = key;
    temp->next = NULL;
    return temp;
}

/* Drier program to test above functions*/
int main()
{
    /* Let us create a linked lists to test
       the functions
    Created list is a: 2->4->3->4->2->15 */
    Node *head = newNode(2);
    head->next = newNode(4);
    head->next->next = newNode(3);
    head->next->next->next = newNode(4);
    head->next->next->next->next = newNode(2);
    head->next->next->next->next->next = newNode(15);

    cout << maxPalindrome(head) << endl;
    return 0;
}
```

Output :

5

Time Complexity :  $O(n^2)$

Note that the above code modifies the given linked list and may not work if modifications to linked list are not allowed. However we can finally do one more reverse to get original list back.

## Source

<https://www.geeksforgeeks.org/length-longest-palindrome-list-linked-list-using-o1-extra-space/>

## Chapter 128

# Linked List Pair Sum

Linked List Pair Sum - GeeksforGeeks

Given a linked list, and a number, check if their exist two numbers whose sum is equal to given number. If there exist two numbers, print them. If there are multiple answer, print any of them.

Examples:

```
Input : 1 -> 2 -> 3 -> 4 -> 5 -> NULL
        sum = 3
Output : Pair is (1, 2)

Input : 10 -> 12 -> 31 -> 42 -> 53 -> NULL
        sum = 15
Output : NO PAIR EXIST
```

### Method(Brute force)

Iteratively check if their exist any pair or not

C++

```
// CPP code to find the pair with given sum
#include <bits/stdc++.h>
using namespace std;

/* Link list node */
struct Node {
    int data;
    struct Node* next;
};

/* Given a reference (pointer to pointer)
```

```

to the head of a list and an int,
push a new node on the front
of the list. */
void push(struct Node** head_ref, int new_data)
{
    /* allocate node */
    struct Node* new_node =
        (struct Node*)malloc(sizeof(struct Node));

    /* put in the data */
    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;
}

/* Takes head pointer of the linked list and sum*/
int check_pair_sum(struct Node* head, int sum)
{
    struct Node* p = head, *q;
    while (p != NULL) {

        q = p->next;
        while (q != NULL) {

            // check if both sum is equal to
            // given sum
            if ((p->data) + (q->data) == sum) {
                cout << p->data << " " << q->data;
                return true;
            }
            q = q->next;
        }

        p = p->next;
    }

    return 0;
}

/* Driver program to test above function */
int main()
{
    /* Start with the empty list */
    struct Node* head = NULL;
}

```

```
/* Use push() to construct linked list*/
push(&head, 1);
push(&head, 4);
push(&head, 1);
push(&head, 12);
push(&head, 1);
push(&head, 18);
push(&head, 47);
push(&head, 16);
push(&head, 12);
push(&head, 14);

/* function to print the result*/
bool res = check_pair_sum(head, 26);
if (res == false)
    cout << "NO PAIR EXIST";

return 0;
}
```

**Output:**

14 12

Time complexity: $O(n^2)$

**Method 2 (using hashing)**

1. Take a hashtable and mark all element with zero
2. Iteratively mark all the element as 1 in hashtable which are present in linked list
3. Iteratively find sum-current element of linked list is present in hashtable or not

C++

```
// CPP program to for finding the pair with given sum
#include <bits/stdc++.h>
#define MAX 100000
using namespace std;

/* Link list node */
struct Node {
    int data;
    struct Node* next;
};

/* Given a reference (pointer to pointer) to the head
   of a list and an int, push a new node on the front
```

```

        of the list. */
void push(struct Node** head_ref, int new_data)
{
    /* allocate node */
    struct Node* new_node =
        (struct Node*)malloc(sizeof(struct Node));

    /* put in the data */
    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;
}

/* Takes head pointer of the linked list and sum*/
bool check_pair_sum(struct Node* head, int sum)
{
    unordered_set<int> s;

    struct Node* p = head;
    while (p != NULL) {
        int curr = p->data;
        if (s.find(sum - curr) != s.end())
        {
            cout << curr << " " << sum - curr;
            return true;
        }
        s.insert(p->data);
        p = p->next;
    }

    return false;
}

/* Drier program to test above function*/
int main()
{
    /* Start with the empty list */
    struct Node* head = NULL;

    /* Use push() to construct linked list */
    push(&head, 1);
    push(&head, 4);
    push(&head, 1);
    push(&head, 12);
}

```

```
push(&head, 1);
push(&head, 18);
push(&head, 47);
push(&head, 16);
push(&head, 12);
push(&head, 14);

/* function to print the result*/
bool res = check_pair_sum(head, 26);
if (res == false)
    cout << "NO PAIR EXIST";

return 0;
}
```

**Output:**

14 12

Time complexity :  $O(n)$

Auxiliary Space :  $O(n)$

**Source**

<https://www.geeksforgeeks.org/linked-list-pair-sum/>

## Chapter 129

# Linked List Sum of Nodes Between 0s

Linked List Sum of Nodes Between 0s - GeeksforGeeks

Given a linked list which contains series of numbers separated by “0”. Add them and store in the linked list in-place.

**Note :** There will not be continuous zeros in input.

Examples:

Input : 1->2->3->0->5->4->0->3->2->0  
Output : 6->9->5

Input : 1->2->3->4  
Output : 1->2->3->4

1. Start iterating over nodes of linked list.
2. Iterate while temp.data !=0, and add these data into a variable ‘sum’.
3. When you encounter 0 as the node’s data, change pointers of previous nodes.

```
// Java program to in-place add linked list
// nodes between 0s.
class Node {
    int data;
    Node next;

    public Node(int data)
    {
        this.data = data;
        this.next = null;
    }
}
```

```
}  
  
public class inPlaceStoreLL {  
  
    // Function to store numbers till 0  
    static void inPlaceStore(Node head)  
    {  
        if(head.data == 0){  
            head = head.next;  
        }  
  
        // To store modified list  
        Node res = head;  
  
        // Traverse linked list and keep  
        // adding nodes between 0s.  
        Node temp = head;  
        int sum = 0;  
        while (temp != null) {  
  
            // loop to sum the data of nodes till  
            // it encounters 0  
            if (temp.data != 0) {  
                sum += temp.data;  
                temp = temp.next;  
            }  
  
            // If we encounters 0, we need  
            // to update next pointers  
            else {  
                res.data = sum;  
                res.next = temp.next;  
                temp = res.next;  
                res = res.next;  
                sum = 0;  
            }  
        }  
        printList(head);  
    }  
  
    // Function to traverse and print Linked List  
    static void printList(Node head)  
    {  
        while (head.next != null) {  
            System.out.print(head.data + "-> ");  
            head = head.next;  
        }  
    }  
}
```

```
        }
        System.out.println(head.data);
    }

// Driver Code
public static void main(String[] args)
{
    Node head = new Node(3);
    head.next = new Node(2);
    head.next.next = new Node(0);
    head.next.next.next = new Node(4);
    head.next.next.next.next = new Node(5);
    head.next.next.next.next.next = new Node(0);
    head.next.next.next.next.next.next = new Node(6);
    head.next.next.next.next.next.next.next = new Node(7);
    inPlaceStore(head);

}
}
```

Output:

5-> 9-> 6-> 7

## Source

<https://www.geeksforgeeks.org/linked-list-sum-nodes-0s/>

## Chapter 130

# Linked List representation of Disjoint Set Data Structures

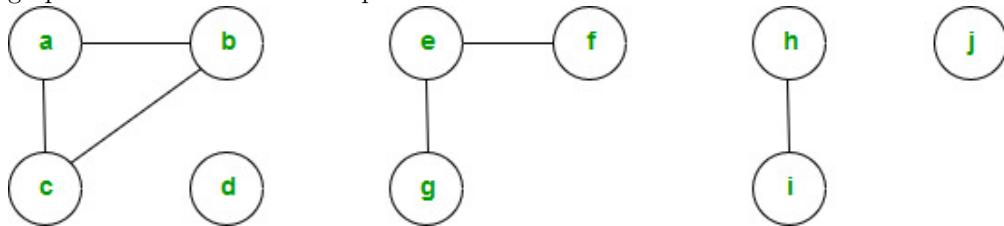
Linked List representation of Disjoint Set Data Structures - GeeksforGeeks

Prerequisites : [Union Find \(or Disjoint Set\)](#), [Disjoint Set Data Structures \(Java Implementation\)](#)

A *disjoint-set data structure* maintains a collection  $S = \{S_1, S_2, \dots, S_k\}$  of disjoint dynamic sets. We identify each set by a *representative*, which is some member of the set. In some applications, it doesn't matter which member is used as the representative; we care only that if we ask for the representative of a dynamic set twice without modifying the set between the requests, we get the same answer both times. Other applications may require a pre-specified rule for choosing the representative, such as choosing the smallest member in the set.

### Example:

Determining the connected components of an Undirected graph. Below figure, shows a graph with fourconnected components.



Solution : One procedure X that follows uses the disjoint-set operations to compute the connected components of a graph. Once X has pre-processed the graph, the procedure Y answers queries about whether two vertices are in the same connected component. Below figure shows the collection of disjoint sets after processing each edge.

Edge processed	Collection of disjoint sets									
initial sets	{a}	{b}	{c}	{d}	{e}	{f}	{g}	{h}	{i}	{j}
(b,d)	{a}	{b,d}	{c}		{e}	{f}	{g}	{h}	{i}	{j}
(a,g)	{a}	{b,d}	{c}		{e,g}	{f}		{h}	{i}	{j}
(a,c)	{a,c}	{b,d}			{e,g}	{f}		{h}	{i}	{j}
(h,i)	{a,c}	{b,d}			{e,g}	{f}		{h,i}		{j}
(a,b)	{a,b,c,d}				{e,g}	{f}		{h,i}		{j}
(e,f)	{a,b,c,d}				{e,f,g}			{h,i}		{j}
(b,c)	{a,b,c,d}				{e,f,g}			{h,i}		{j}

See [here](#) as the above example was discussed earlier.

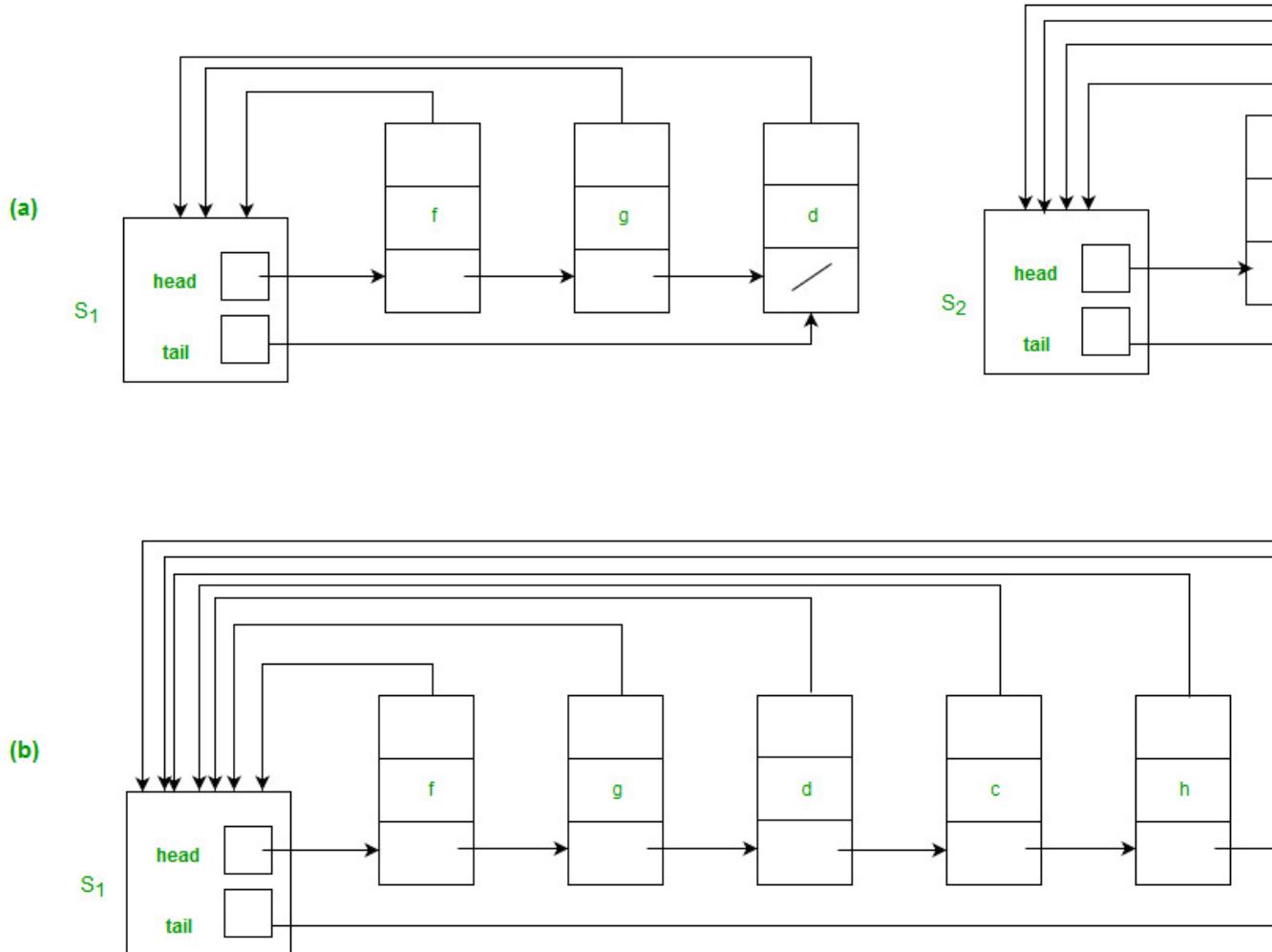


Figure (a) Linked-list representations of two sets. Set S1 contains members d, f, and g,

with representative f, and set S2 contains members b, c, e, and h, with representative c. Each object in the list contains a set member, a pointer to the next object in the list, and a pointer back to the set object. Each set object has pointers head and tail to the first and last objects, respectively. **(b)** The result of UNION(e, g), which appends the linked list containing e to the linked list containing g. The representative of the resulting set is f . The set object for e's list, S2, is destroyed.

Above three figures are taken from the Cormen book. Above Figure shows a simple way to implement a disjoint-set data structure: each set is represented by its own linked list. The object for each set has attributes head, pointing to the 1st object in the list, and tail, pointing to the last object.

Each object in the list contains a set member, a pointer to the next object in the list, and a pointer back to the set object. Within each linked list, the objects may appear in any order. The representative is the set member in the 1st object in the list.

To carry out MAKE-SET (x), we create a new linked list whose only object is x. For FIND-SET(x), we just follow the pointer from x back to its set object and then return the member in the object that head points to. For example, in the Figure, the call FIND-SET(g) would return f.

**Algorithm:**

Letting x denote an object, we wish to support the following operations:

**MAKE-SET(x)** creates a new set whose only member (and thus representative) is x. Since the sets are disjoint, we require that x not already be in some other set.

**UNION (x, y)** unites the dynamic sets that contain x and y, say  $S_x$  and  $S_y$ , into a new set that is the union of these two sets. We assume that the two sets are disjoint prior to the operation. The representative of the resulting set is any member of  $S_x \cup S_y$ , although many implementations of UNION specifically choose the representative of either  $S_x$  or  $S_y$  as the new representative. Since we require the sets in the collection to be disjoint, conceptually we destroy sets  $S_x$  and  $S_y$ , removing them from the collection S. In practice, we often absorb the elements of one of the sets into the other set.

**FIND-SET(x)** returns a pointer to the representative of the (unique) set containing x.

Based on the above explanation, below are implementations:

```
// C++ program for implementation of disjoint
// set data structure using linked list
#include <bits/stdc++.h>
using namespace std;

// to represent linked list which is a set
struct Item;

// to represent Node of linked list. Every
// node has a pointer to representative
struct Node
{
    int val;
```

```
    Node *next;
    Item *itemPtr;
};

// A list has a pointer to head and tail
struct Item
{
    Node *hd, *tl;
};

// To represent union set
class ListSet
{
private:

    // Hash to store addresses of set representatives
    // for given values. It is made global for ease of
    // implementation. And second part of hash is actually
    // address of Nodes. We typecast addresses to long
    // before storing them.
    unordered_map<int, Node *> nodeAddress;

public:
    void makeset(int a);
    Item* find(int key);
    void Union(Item *i1, Item *i2);
};

// To make a set with one object
// with its representative
void ListSet::makeset(int a)
{
    // Create a new Set
    Item *newSet = new Item;

    // Create a new linked list node
    // to store given key
    newSet->hd = new Node;

    // Initialize head and tail
    newSet->tl = newSet->hd;
    nodeAddress[a] = newSet->hd;

    // Create a new set
    newSet->hd->val = a;
    newSet->hd->itemPtr = newSet;
    newSet->hd->next = NULL;
}
```

```
// To find representative address of a
// key
Item *ListSet::find(int key)
{
    Node *ptr = nodeAddress[key];
    return (ptr->itemPtr);
}

// union function for joining two subsets
// of a universe. Mergese set2 into set1
// and deletes set1.
void ListSet::Union(Item *set1, Item *set2)
{
    Node *cur = set2->hd;
    while (cur != 0)
    {
        cur->itemPtr = set1;
        cur = cur->next;
    }

    // Join the tail of the set to head
    // of the input set
    (set1->tl)->next = set2->hd;
    set1->tl = set2->tl;

    delete set2;
}

// Driver code
int main()
{
    ListSet a;
    a.makeset(13); //a new set is made with one object only
    a.makeset(25);
    a.makeset(45);
    a.makeset(65);

    cout << "find(13): " << a.find(13) << endl;
    cout << "find(25): "
        << a.find(25) << endl;
    cout << "find(65): "
        << a.find(65) << endl;
    cout << "find(45): "
        << a.find(45) << endl << endl;
    cout << "Union(find(65), find(45)) \n";

    a.Union(a.find(65), a.find(45));
}
```

```
    cout << "find(65]): "
        << a.find(65) << endl;
    cout << "find(45]): "
        << a.find(45) << endl;
    return 0;
}
```

Output:

```
find(13): 0x1aa3c20
find(25): 0x1aa3ca0
find(65): 0x1aa3d70
find(45): 0x1aa3c80

Union(find(65), find(45))
find(65]): 0x1aa3d70
find(45]): 0x1aa3d70
```

**Note:** The node address will change every time, we run the program.

Time complexities of MAKE-SET and FIND-SET are O(1). Time complexity for UNION is O(n).

## Source

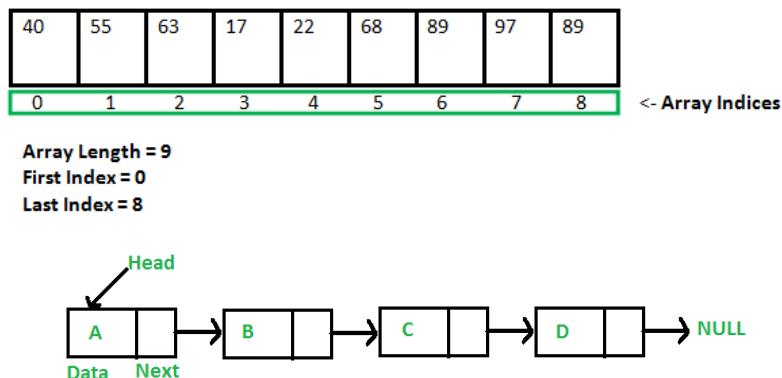
<https://www.geeksforgeeks.org/linked-list-representation-disjoint-set-data-structures/>

# Chapter 131

## Linked List vs Array

Linked List vs Array - GeeksforGeeks

Both [Arrays](#) and [Linked List](#) can be used to store linear data of similar types, but they both have some advantages and disadvantages over each other.



Following are the points in favour of Linked Lists.

- (1) The size of the arrays is fixed: So we must know the upper limit on the number of elements in advance. Also, generally, the allocated memory is equal to the upper limit irrespective of the usage, and in practical uses, upper limit is rarely reached.
- (2) Inserting a new element in an array of elements is expensive, because room has to be created for the new elements and to create room existing elements have to shifted.

For example, suppose we maintain a sorted list of IDs in an array `id[]`.

`id[] = [1000, 1010, 1050, 2000, 2040, ....].`

And if we want to insert a new ID 1005, then to maintain the sorted order, we have to move all the elements after 1000 (excluding 1000).

Deletion is also expensive with arrays until unless some special techniques are used. For example, to delete 1010 in id[], everything after 1010 has to be moved.

So Linked list provides following two advantages over arrays

- 1) Dynamic size
- 2) Ease of insertion/deletion

Linked lists have following drawbacks:

- 1) Random access is not allowed. We have to access elements sequentially starting from the first node. So we cannot do binary search with linked lists.
- 2) Extra memory space for a pointer is required with each element of the list.
- 3) Arrays have better cache locality that can make a pretty big difference in performance.

References:

<http://cslibrary.stanford.edu/103/LinkedListBasics.pdf>

## Source

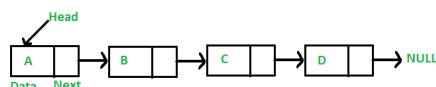
<https://www.geeksforgeeks.org/linked-list-vs-array/>

## Chapter 132

# Linked List | Set 1 (Introduction)

[Linked List | Set 1 \(Introduction\) - GeeksforGeeks](#)

Like arrays, Linked List is a linear data structure. Unlike arrays, linked list elements are not stored at contiguous location; the elements are linked using pointers.



### Why Linked List?

Arrays can be used to store linear data of similar types, but arrays have following limitations.

- 1) The size of the arrays is fixed: So we must know the upper limit on the number of elements in advance. Also, generally, the allocated memory is equal to the upper limit irrespective of the usage.
- 2) Inserting a new element in an array of elements is expensive, because room has to be created for the new elements and to create room existing elements have to shifted.

For example, in a system if we maintain a sorted list of IDs in an array id[].

$$\text{id}[] = [1000, 1010, 1050, 2000, 2040].$$

And if we want to insert a new ID 1005, then to maintain the sorted order, we have to move all the elements after 1000 (excluding 1000).

Deletion is also expensive with arrays until unless some special techniques are used. For example, to delete 1010 in id[], everything after 1010 has to be moved.

### Advantages over arrays

- 1) Dynamic size
- 2) Ease of insertion/deletion

### Drawbacks:

- 1) Random access is not allowed. We have to access elements sequentially starting from

the first node. So we cannot do binary search with linked lists efficiently with its default implementation. Read about it [here](#).

**2)** Extra memory space for a pointer is required with each element of the list.

**3)** Not cache friendly. Since array elements are contiguous locations, there is locality of reference which is not there in case of linked lists.

**Representation:**

A linked list is represented by a pointer to the first node of the linked list. The first node is called head. If the linked list is empty, then value of head is NULL.

Each node in a list consists of at least two parts:

1) data

2) Pointer (Or Reference) to the next node

In C, we can represent a node using structures. Below is an example of a linked list node with an integer data.

In Java, LinkedList can be represented as a class and a Node as a separate class. The LinkedList class contains a reference of Node class type.

**C**

```
// A linked list node
struct Node
{
    int data;
    struct Node *next;
};
```

**Java**

```
class LinkedList
{
    Node head; // head of list

    /* Linked list Node*/
    class Node
    {
        int data;
        Node next;

        // Constructor to create a new node
        // Next is by default initialized
        // as null
        Node(int d) {data = d;}
    }
}
```

**Python**

```
# Node class
class Node:

    # Function to initialize the node object
    def __init__(self, data):
        self.data = data  # Assign data
        self.next = None  # Initialize
                          # next as null

# Linked List class
class LinkedList:

    # Function to initialize the Linked
    # List object
    def __init__(self):
        self.head = None
```

**First Simple Linked List in C** Let us create a simple linked list with 3 nodes.

C

```
// A simple C program to introduce
// a linked list
#include<stdio.h>
#include<stdlib.h>

struct Node
{
    int data;
    struct Node *next;
};

// Program to create a simple linked
// list with 3 nodes
int main()
{
    struct Node* head = NULL;
    struct Node* second = NULL;
    struct Node* third = NULL;

    // allocate 3 nodes in the heap
    head = (struct Node*)malloc(sizeof(struct Node));
    second = (struct Node*)malloc(sizeof(struct Node));
    third = (struct Node*)malloc(sizeof(struct Node));

    /* Three blocks have been allocated dynamically.
       We have pointers to these three blocks as first,
       second and third
```

```

head          second        third
|             |             |
|             |             |
+---+----+    +---+----+    +---+----+
| # | # |    | # | # |    | # | # |
+---+----+    +---+----+    +---+----+
# represents any random value.
Data is random because we haven't assigned
anything yet */

head->data = 1; //assign data in first node
head->next = second; // Link first node with
                      // the second node

/* data has been assigned to data part of first
   block (block pointed by head). And next
   pointer of first block points to second.
   So they both are linked.

head          second        third
|             |             |
|             |             |
+---+----+    +---+----+    +---+----+
| 1 | o----->| # | # |    | # | # |
+---+----+    +---+----+    +---+----+
*/
// assign data to second node
second->data = 2;

// Link second node with the third node
second->next = third;

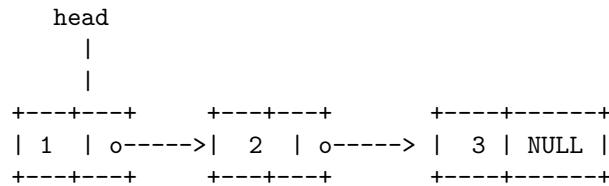
/* data has been assigned to data part of second
   block (block pointed by second). And next
   pointer of the second block points to third
   block. So all three blocks are linked.

head          second        third
|             |             |
|             |             |
+---+----+    +---+----+    +---+----+
| 1 | o----->| 2 | o----->| # | # |
+---+----+    +---+----+    +---+----+      */
third->data = 3; //assign data to third node
third->next = NULL;

```

```
/* data has been assigned to data part of third
   block (block pointed by third). And next pointer
   of the third block is made NULL to indicate
   that the linked list is terminated here.
```

We have the linked list ready.



Note that only head is sufficient to represent  
the whole list. We can traverse the complete  
list by following next pointers. \*/

```
return 0;
}
```

### Java

```
// A simple Java program to introduce a linked list
class LinkedList
{
    Node head; // head of list

    /* Linked list Node. This inner class is made static so that
       main() can access it */
    static class Node {
        int data;
        Node next;
        Node(int d) { data = d; next=null; } // Constructor
    }

    /* method to create a simple linked list with 3 nodes*/
    public static void main(String[] args)
    {
        /* Start with the empty list. */
        LinkedList llist = new LinkedList();

        llist.head = new Node(1);
        Node second = new Node(2);
        Node third = new Node(3);
```

```

/* Three nodes have been allocated dynamically.
We have references to these three blocks as first,
second and third

    llist.head      second      third
    |              |          |
    |              |          |
+---+-----+      +---+-----+      +---+-----+
| 1 | null |      | 2 | null |      | 3 | null |
+---+-----+      +---+-----+      +---+-----+ */

llist.head.next = second; // Link first node with the second node

/* Now next of first Node refers to second. So they
both are linked.

    llist.head      second      third
    |              |          |
    |              |          |
+---+-----+      +---+-----+      +---+-----+
| 1 | o----->| 2 | null |      | 3 | null |
+---+-----+      +---+-----+      +---+-----+ */

second.next = third; // Link second node with the third node

/* Now next of second Node refers to third. So all three
nodes are linked.

    llist.head      second      third
    |              |          |
    |              |          |
+---+-----+      +---+-----+      +---+-----+
| 1 | o----->| 2 | o----->| 3 | null |
+---+-----+      +---+-----+      +---+-----+ */
}

}

```

### Python

```

# A simple Python program to introduce a linked list

# Node class
class Node:

    # Function to initialise the node object
    def __init__(self, data):
        self.data = data  # Assign data
        self.next = None  # Initialize next as null

```

```
# Linked List class contains a Node object
class LinkedList:

    # Function to initialize head
    def __init__(self):
        self.head = None

# Code execution starts here
if __name__=='__main__':

    # Start with the empty list
    llist = LinkedList()

    llist.head = Node(1)
    second = Node(2)
    third = Node(3)

    """
    Three nodes have been created.
    We have references to these three blocks as first,
    second and third

    llist.head      second       third
    |              |             |
    |              |             |
    +---+-----+    +---+-----+    +---+-----+
    | 1  | None |    | 2  | None |    | 3  | None |
    +---+-----+    +---+-----+    +---+-----+
    """

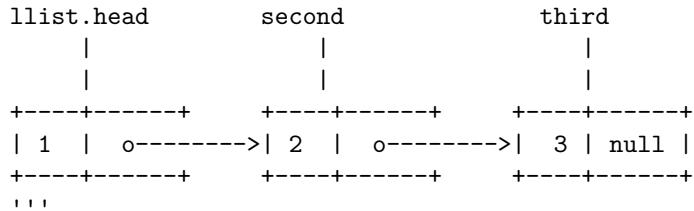
    llist.head.next = second; # Link first node with second

    """
    Now next of first Node refers to second. So they
    both are linked.

    llist.head      second       third
    |              |             |
    |              |             |
    +---+-----+    +---+-----+    +---+-----+
    | 1  | o----->| 2  | null |    | 3  | null |
    +---+-----+    +---+-----+    +---+-----+
    """

    second.next = third; # Link second node with the third node
```

```
'''  
Now next of second Node refers to third. So all three  
nodes are linked.
```



### Linked List Traversal

In the previous program, we have created a simple linked list with three nodes. Let us traverse the created list and print the data of each node. For traversal, let us write a general purpose function `printList()` that prints any given list.

C

```
// A simple C program for traversal of a linked list  
#include<stdio.h>  
#include<stdlib.h>  
  
struct Node  
{  
    int data;  
    struct Node *next;  
};  
  
// This function prints contents of linked list starting from  
// the given node  
void printList(struct Node *n)  
{  
    while (n != NULL)  
    {  
        printf(" %d ", n->data);  
        n = n->next;  
    }  
}  
  
int main()  
{  
    struct Node* head = NULL;  
    struct Node* second = NULL;  
    struct Node* third = NULL;  
  
    // allocate 3 nodes in the heap
```

```
head  = (struct Node*)malloc(sizeof(struct Node));
second = (struct Node*)malloc(sizeof(struct Node));
third  = (struct Node*)malloc(sizeof(struct Node));

head->data = 1; //assign data in first node
head->next = second; // Link first node with second

second->data = 2; //assign data to second node
second->next = third;

third->data = 3; //assign data to third node
third->next = NULL;

printList(head);

return 0;
}
```

### Java

```
// A simple Java program for traversal of a linked list
class LinkedList
{
    Node head; // head of list

    /* Linked list Node. This inner class is made static so that
       main() can access it */
    static class Node {
        int data;
        Node next;
        Node(int d) { data = d; next=null; } // Constructor
    }

    /* This function prints contents of linked list starting from head */
    public void printList()
    {
        Node n = head;
        while (n != null)
        {
            System.out.print(n.data+" ");
            n = n.next;
        }
    }

    /* method to create a simple linked list with 3 nodes*/
    public static void main(String[] args)
    {
        /* Start with the empty list. */

```

```
LinkedList llist = new LinkedList();

llist.head      = new Node(1);
Node second    = new Node(2);
Node third     = new Node(3);

llist.head.next = second; // Link first node with the second node
second.next = third; // Link first node with the second node

llist.printList();
}
}
```

### Python

```
# A simple Python program for traversal of a linked list

# Node class
class Node:

    # Function to initialise the node object
    def __init__(self, data):
        self.data = data    # Assign data
        self.next = None   # Initialize next as null

# Linked List class contains a Node object
class LinkedList:

    # Function to initialize head
    def __init__(self):
        self.head = None

    # This function prints contents of linked list
    # starting from head
    def printList(self):
        temp = self.head
        while (temp):
            print temp.data,
            temp = temp.next

# Code execution starts here
if __name__=='__main__':

    # Start with the empty list
    llist = LinkedList()
```

```
llist.head = Node(1)
second = Node(2)
third = Node(3)

llist.head.next = second; # Link first node with second
second.next = third; # Link second node with the third node

llist.printList()
```

Output:

1 2 3

**Important Links :**

- Practice MCQ Questions on Linked List
- Linked List Data Structure Page
- Coding Practice Questions on Linked List.

**Improved By :** ashwani khemani

**Source**

<https://www.geeksforgeeks.org/linked-list-set-1-introduction/>

## Chapter 133

# Linked List | Set 2 (Inserting a node)

Linked List | Set 2 (Inserting a node) - GeeksforGeeks

We have introduced Linked Lists in the [previous post](#). We also created a simple linked list with 3 nodes and discussed linked list traversal.

All programs discussed in this post consider following representations of linked list .

### C

```
// A linked list node
struct Node
{
    int data;
    struct Node *next;
};
```

### Java

```
// Linked List Class
class LinkedList
{
    Node head; // head of list

    /* Node Class */
    class Node
    {
        int data;
        Node next;
```

```
// Constructor to create a new node
Node(int d) {data = d; next = null; }
}
}
```

### Python

```
# Node class
class Node:

    # Function to initialize the node object
    def __init__(self, data):
        self.data = data    # Assign data
        self.next = None   # Initialize next as null

# Linked List class
class LinkedList:

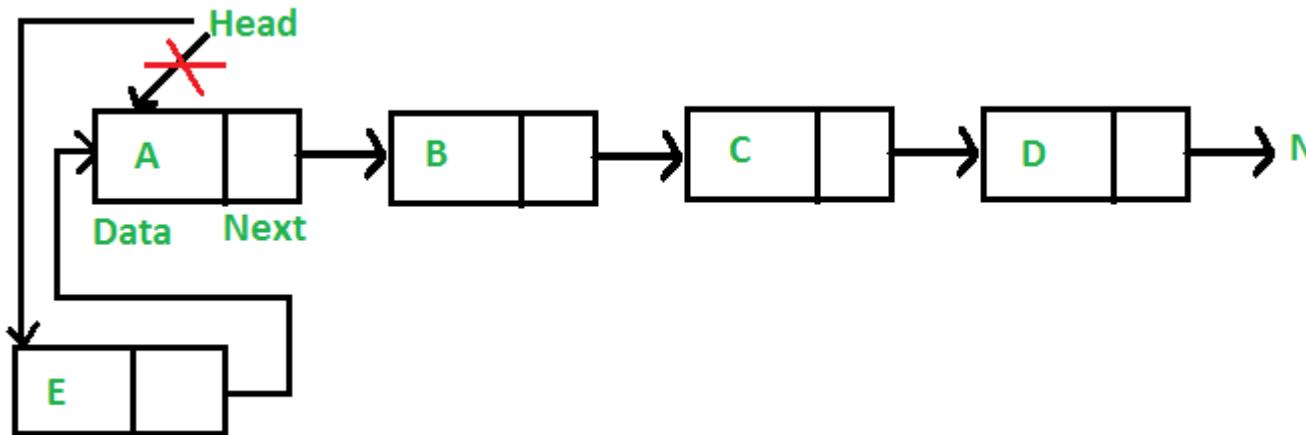
    # Function to initialize the Linked List object
    def __init__(self):
        self.head = None
```

In this post, methods to insert a new node in linked list are discussed. A node can be added in three ways

- 1) At the front of the linked list
- 2) After a given node.
- 3) At the end of the linked list.

#### Add a node at the front: (A 4 steps process)

The new node is always added before the head of the given Linked List. And newly added node becomes the new head of the Linked List. For example if the given Linked List is 10->15->20->25 and we add an item 5 at the front, then the Linked List becomes 5->10->15->20->25. Let us call the function that adds at the front of the list is push(). The push() must receive a pointer to the head pointer, because push must change the head pointer to point to the new node (See [this](#))



Following are the 4 steps to add node at the front.

C

```
/* Given a reference (pointer to pointer) to the head of a list
   and an int, inserts a new node on the front of the list. */
void push(struct Node** head_ref, int new_data)
{
    /* 1. allocate node */
    struct Node* new_node = (struct Node*) malloc(sizeof(struct Node));

    /* 2. put in the data */
    new_node->data = new_data;

    /* 3. Make next of new node as head */
    new_node->next = (*head_ref);

    /* 4. move the head to point to the new node */
    (*head_ref) = new_node;
}
```

Java

```
/* This function is in LinkedList class. Inserts a
   new Node at front of the list. This method is
   defined inside LinkedList class shown above */
public void push(int new_data)
{
    /* 1 & 2: Allocate the Node &
       Put in the data*/

```

```

Node new_node = new Node(new_data);

/* 3. Make next of new Node as head */
new_node.next = head;

/* 4. Move the head to point to new Node */
head = new_node;
}

```

### Python

```

# This function is in LinkedList class
# Function to insert a new node at the beginning
def push(self, new_data):

    # 1 & 2: Allocate the Node &
    #          Put in the data
    new_node = Node(new_data)

    # 3. Make next of new Node as head
    new_node.next = self.head

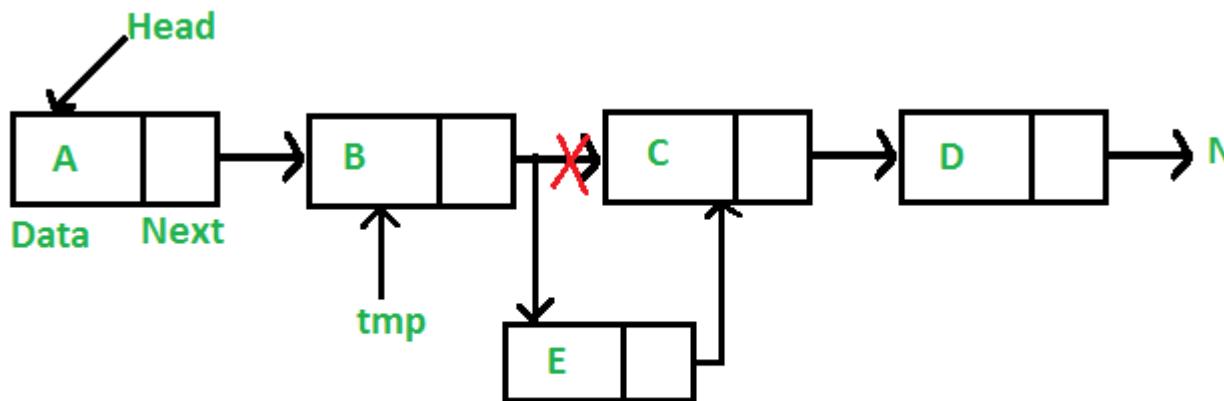
    # 4. Move the head to point to new Node
    self.head = new_node

```

Time complexity of push() is O(1) as it does constant amount of work.

### Add a node after a given node: (5 steps process)

We are given pointer to a node, and the new node is inserted after the given node.



```
/* Given a node prev_node, insert a new node after the given
   prev_node */
void insertAfter(struct Node* prev_node, int new_data)
{
    /*1. check if the given prev_node is NULL */
    if (prev_node == NULL)
    {
        printf("the given previous node cannot be NULL");
        return;
    }

    /* 2. allocate new node */
    struct Node* new_node = (struct Node*) malloc(sizeof(struct Node));

    /* 3. put in the data */
    new_node->data = new_data;

    /* 4. Make next of new node as next of prev_node */
    new_node->next = prev_node->next;

    /* 5. move the next of prev_node as new_node */
    prev_node->next = new_node;
}
```

### Java

```
/* This function is in LinkedList class.
   Inserts a new node after the given prev_node. This method is
   defined inside LinkedList class shown above */
public void insertAfter(Node prev_node, int new_data)
{
    /* 1. Check if the given Node is null */
    if (prev_node == null)
    {
        System.out.println("The given previous node cannot be null");
        return;
    }

    /* 2. Allocate the Node &
       3. Put in the data*/
    Node new_node = new Node(new_data);

    /* 4. Make next of new Node as next of prev_node */
    new_node.next = prev_node.next;

    /* 5. make next of prev_node as new_node */
    prev_node.next = new_node;
}
```

## Python

```

# This function is in LinkedList class.
# Inserts a new node after the given prev_node. This method is
# defined inside LinkedList class shown above */
def insertAfter(self, prev_node, new_data):

    # 1. check if the given prev_node exists
    if prev_node is None:
        print "The given previous node must inLinkedList."
        return

    # 2. Create new node &
    # 3. Put in the data
    new_node = Node(new_data)

    # 4. Make next of new Node as next of prev_node
    new_node.next = prev_node.next

    # 5. make next of prev_node as new_node
    prev_node.next = new_node

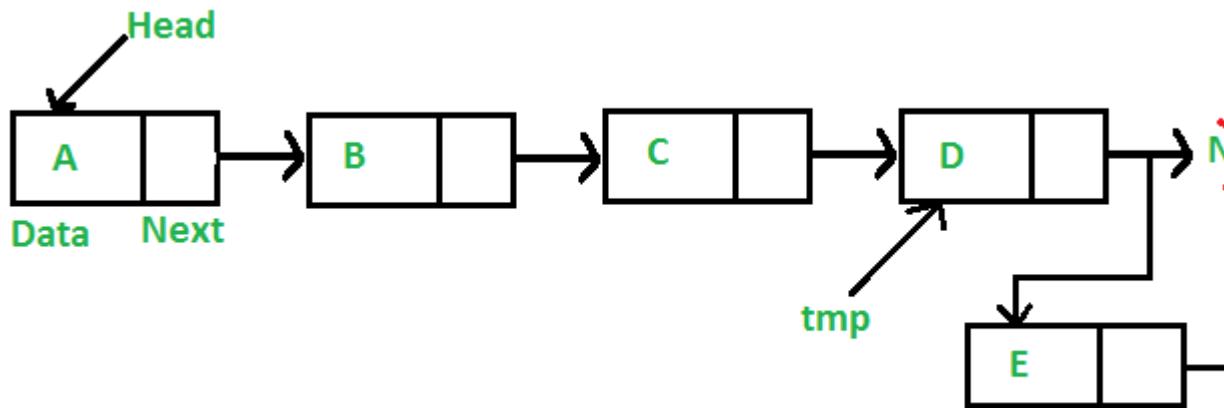
```

Time complexity of insertAfter() is O(1) as it does constant amount of work.

### Add a node at the end: (6 steps process)

The new node is always added after the last node of the given Linked List. For example if the given Linked List is 5->10->15->20->25 and we add an item 30 at the end, then the Linked List becomes 5->10->15->20->25->30.

Since a Linked List is typically represented by the head of it, we have to traverse the list till end and then change the next of last node to new node.



Following are the 6 steps to add node at the end.

C

```
/* Given a reference (pointer to pointer) to the head
   of a list and an int, appends a new node at the end */
void append(struct Node** head_ref, int new_data)
{
    /* 1. allocate node */
    struct Node* new_node = (struct Node*) malloc(sizeof(struct Node));

    struct Node *last = *head_ref; /* used in step 5*/

    /* 2. put in the data */
    new_node->data = new_data;

    /* 3. This new node is going to be the last node, so make next
       of it as NULL*/
    new_node->next = NULL;

    /* 4. If the Linked List is empty, then make the new node as head */
    if (*head_ref == NULL)
    {
        *head_ref = new_node;
        return;
    }

    /* 5. Else traverse till the last node */
    while (last->next != NULL)
        last = last->next;

    /* 6. Change the next of last node */
    last->next = new_node;
    return;
}
```

### Java

```
/* Appends a new node at the end. This method is
   defined inside LinkedList class shown above */
public void append(int new_data)
{
    /* 1. Allocate the Node &
       2. Put in the data
       3. Set next as null */
    Node new_node = new Node(new_data);

    /* 4. If the Linked List is empty, then make the
       new node as head */
    if (head == null)
    {
        head = new Node(new_data);
```

```
        return;
    }

/* 4. This new node is going to be the last node, so
   make next of it as null */
new_node.next = null;

/* 5. Else traverse till the last node */
Node last = head;
while (last.next != null)
    last = last.next;

/* 6. Change the next of last node */
last.next = new_node;
return;
}
```

### Python

```
# This function is defined in Linked List class
# Appends a new node at the end. This method is
# defined inside LinkedList class shown above */
def append(self, new_data):

    # 1. Create a new node
    # 2. Put in the data
    # 3. Set next as None
    new_node = Node(new_data)

    # 4. If the Linked List is empty, then make the
    #     new node as head
    if self.head is None:
        self.head = new_node
        return

    # 5. Else traverse till the last node
    last = self.head
    while (last.next):
        last = last.next

    # 6. Change the next of last node
    last.next = new_node
```

Time complexity of append is  $O(n)$  where  $n$  is the number of nodes in linked list. Since there is a loop from head to end, the function does  $O(n)$  work.

This method can also be optimized to work in  $O(1)$  by keeping an extra pointer to tail of linked list/

Following is a complete program that uses all of the above methods to create a linked list.

C

```
// A complete working C program to demonstrate all insertion methods
// on Linked List
#include <stdio.h>
#include <stdlib.h>

// A linked list node
struct Node
{
    int data;
    struct Node *next;
};

/* Given a reference (pointer to pointer) to the head of a list and
   an int, inserts a new node on the front of the list. */
void push(struct Node** head_ref, int new_data)
{
    /* 1. allocate node */
    struct Node* new_node = (struct Node*) malloc(sizeof(struct Node));

    /* 2. put in the data */
    new_node->data = new_data;

    /* 3. Make next of new node as head */
    new_node->next = (*head_ref);

    /* 4. move the head to point to the new node */
    (*head_ref) = new_node;
}

/* Given a node prev_node, insert a new node after the given
   prev_node */
void insertAfter(struct Node* prev_node, int new_data)
{
    /*1. check if the given prev_node is NULL */
    if (prev_node == NULL)
    {
        printf("the given previous node cannot be NULL");
        return;
    }

    /* 2. allocate new node */
    struct Node* new_node =(struct Node*) malloc(sizeof(struct Node));

    /* 3. put in the data */

```

```
new_node->data = new_data;

/* 4. Make next of new node as next of prev_node */
new_node->next = prev_node->next;

/* 5. move the next of prev_node as new_node */
prev_node->next = new_node;
}

/* Given a reference (pointer to pointer) to the head
   of a list and an int, appends a new node at the end */
void append(struct Node** head_ref, int new_data)
{
    /* 1. allocate node */
    struct Node* new_node = (struct Node*) malloc(sizeof(struct Node));

    struct Node *last = *head_ref; /* used in step 5*/

    /* 2. put in the data */
    new_node->data = new_data;

    /* 3. This new node is going to be the last node, so make next of
       it as NULL*/
    new_node->next = NULL;

    /* 4. If the Linked List is empty, then make the new node as head */
    if (*head_ref == NULL)
    {
        *head_ref = new_node;
        return;
    }

    /* 5. Else traverse till the last node */
    while (last->next != NULL)
        last = last->next;

    /* 6. Change the next of last node */
    last->next = new_node;
    return;
}

// This function prints contents of linked list starting from head
void printList(struct Node *node)
{
    while (node != NULL)
    {
        printf(" %d ", node->data);
        node = node->next;
    }
}
```

```
    }
}

/* Driver program to test above functions*/
int main()
{
    /* Start with the empty list */
    struct Node* head = NULL;

    // Insert 6. So linked list becomes 6->NULL
    append(&head, 6);

    // Insert 7 at the beginning. So linked list becomes 7->6->NULL
    push(&head, 7);

    // Insert 1 at the beginning. So linked list becomes 1->7->6->NULL
    push(&head, 1);

    // Insert 4 at the end. So linked list becomes 1->7->6->4->NULL
    append(&head, 4);

    // Insert 8, after 7. So linked list becomes 1->7->8->6->4->NULL
    insertAfter(head->next, 8);

    printf("\n Created Linked list is: ");
    printList(head);

    return 0;
}
```

### Java

```
// A complete working Java program to demonstrate all insertion methods
// on linked list
class LinkedList
{
    Node head; // head of list

    /* Linked list Node*/
    class Node
    {
        int data;
        Node next;
        Node(int d) {data = d; next = null; }
    }

    /* Inserts a new Node at front of the list. */
    public void push(int new_data)
```

```
{  
    /* 1 & 2: Allocate the Node &  
       Put in the data*/  
    Node new_node = new Node(new_data);  
  
    /* 3. Make next of new Node as head */  
    new_node.next = head;  
  
    /* 4. Move the head to point to new Node */  
    head = new_node;  
}  
  
/* Inserts a new node after the given prev_node. */  
public void insertAfter(Node prev_node, int new_data)  
{  
    /* 1. Check if the given Node is null */  
    if (prev_node == null)  
    {  
        System.out.println("The given previous node cannot be null");  
        return;  
    }  
  
    /* 2 & 3: Allocate the Node &  
       Put in the data*/  
    Node new_node = new Node(new_data);  
  
    /* 4. Make next of new Node as next of prev_node */  
    new_node.next = prev_node.next;  
  
    /* 5. make next of prev_node as new_node */  
    prev_node.next = new_node;  
}  
  
/* Appends a new node at the end. This method is  
   defined inside LinkedList class shown above */  
public void append(int new_data)  
{  
    /* 1. Allocate the Node &  
       2. Put in the data  
       3. Set next as null */  
    Node new_node = new Node(new_data);  
  
    /* 4. If the Linked List is empty, then make the  
       new node as head */  
    if (head == null)  
    {  
        head = new Node(new_data);  
        return;  
    }
```

```
}

/* 4. This new node is going to be the last node, so
   make next of it as null */
new_node.next = null;

/* 5. Else traverse till the last node */
Node last = head;
while (last.next != null)
    last = last.next;

/* 6. Change the next of last node */
last.next = new_node;
return;
}

/* This function prints contents of linked list starting from
   the given node */
public void printList()
{
    Node tnode = head;
    while (tnode != null)
    {
        System.out.print(tnode.data+" ");
        tnode = tnode.next;
    }
}

/* Driver program to test above functions. Ideally this function
   should be in a separate user class. It is kept here to keep
   code compact */
public static void main(String[] args)
{
    /* Start with the empty list */
    LinkedList llist = new LinkedList();

    // Insert 6. So linked list becomes 6->Nulllist
    llist.append(6);

    // Insert 7 at the beginning. So linked list becomes
    // 7->6->Nulllist
    llist.push(7);

    // Insert 1 at the beginning. So linked list becomes
    // 1->7->6->Nulllist
    llist.push(1);

    // Insert 4 at the end. So linked list becomes
```

```
// 1->7->6->4->NULLlist
llist.append(4);

// Insert 8, after 7. So linked list becomes
// 1->7->8->6->4->NULLlist
llist.insertAfter(llist.head.next, 8);

System.out.println("\nCreated Linked list is: ");
llist.printList();
}

}

// This code is contributed by Rajat Mishra
```

### Python

```
# A complete working Python program to demonstrate all
# insertion methods of linked list

# Node class
class Node:

    # Function to initialise the node object
    def __init__(self, data):
        self.data = data  # Assign data
        self.next = None  # Initialize next as null

# Linked List class contains a Node object
class LinkedList:

    # Function to initialize head
    def __init__(self):
        self.head = None

    # Function to insert a new node at the beginning
    def push(self, new_data):

        # 1 & 2: Allocate the Node &
        #         Put in the data
        new_node = Node(new_data)

        # 3. Make next of new Node as head
        new_node.next = self.head

        # 4. Move the head to point to new Node
        self.head = new_node
```

```
# This function is in LinkedList class. Inserts a
# new node after the given prev_node. This method is
# defined inside LinkedList class shown above */
def insertAfter(self, prev_node, new_data):

    # 1. check if the given prev_node exists
    if prev_node is None:
        print "The given previous node must inLinkedList."
        return

    # 2. create new node &
    #     Put in the data
    new_node = Node(new_data)

    # 4. Make next of new Node as next of prev_node
    new_node.next = prev_node.next

    # 5. make next of prev_node as new_node
    prev_node.next = new_node

# This function is defined in Linked List class
# Appends a new node at the end. This method is
# defined inside LinkedList class shown above */
def append(self, new_data):

    # 1. Create a new node
    # 2. Put in the data
    # 3. Set next as None
    new_node = Node(new_data)

    # 4. If the Linked List is empty, then make the
    #     new node as head
    if self.head is None:
        self.head = new_node
        return

    # 5. Else traverse till the last node
    last = self.head
    while (last.next):
        last = last.next

    # 6. Change the next of last node
    last.next = new_node

# Utility function to print the linked list
```

```
def printList(self):
    temp = self.head
    while (temp):
        print temp.data,
        temp = temp.next

# Code execution starts here
if __name__=='__main__':
    # Start with the empty list
    llist = LinkedList()

    # Insert 6. So linked list becomes 6->None
    llist.append(6)

    # Insert 7 at the beginning. So linked list becomes 7->6->None
    llist.push(7);

    # Insert 1 at the beginning. So linked list becomes 1->7->6->None
    llist.push(1);

    # Insert 4 at the end. So linked list becomes 1->7->6->4->None
    llist.append(4)

    # Insert 8, after 7. So linked list becomes 1 -> 7-> 8-> 6-> 4-> None
    llist.insertAfter(llist.head.next, 8)

    print 'Created linked list is:',
    llist.printList()

# This code is contributed by Manikantan Narasimhan
```

Output:

```
Created Linked list is: 1 7 8 6 4
```

You may like to try [Practice MCQ Questions on Linked List](#)

## Source

<https://www.geeksforgeeks.org/linked-list-set-2-inserting-a-node/>

## Chapter 134

# Linked List | Set 3 (Deleting a node)

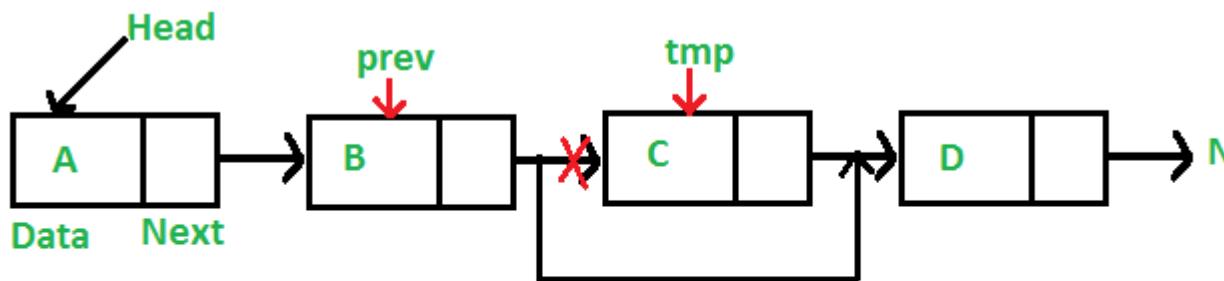
[Linked List | Set 3 \(Deleting a node\) - GeeksforGeeks](#)

We have discussed [Linked List Introduction](#) and [Linked List Insertion](#) in previous posts on singly linked list.

Let us formulate the problem statement to understand the deletion process. *Given a 'key', delete the first occurrence of this key in linked list.*

To delete a node from linked list, we need to do following steps.

- 1) Find previous node of the node to be deleted.
- 2) Change the next of previous node.
- 3) Free memory for the node to be deleted.



Since every node of linked list is dynamically allocated using `malloc()` in C, we need to call `free()` for freeing memory allocated for the node to be deleted.

C/C++

```
// A complete working C program to demonstrate deletion in singly
// linked list
#include <stdio.h>
#include <stdlib.h>

// A linked list node
struct Node
{
    int data;
    struct Node *next;
};

/* Given a reference (pointer to pointer) to the head of a list
   and an int, inserts a new node on the front of the list. */
void push(struct Node** head_ref, int new_data)
{
    struct Node* new_node = (struct Node*) malloc(sizeof(struct Node));
    new_node->data = new_data;
    new_node->next = (*head_ref);
    (*head_ref) = new_node;
}

/* Given a reference (pointer to pointer) to the head of a list
   and a key, deletes the first occurrence of key in linked list */
void deleteNode(struct Node **head_ref, int key)
{
    // Store head node
    struct Node* temp = *head_ref, *prev;

    // If head node itself holds the key to be deleted
    if (temp != NULL && temp->data == key)
    {
        *head_ref = temp->next; // Changed head
        free(temp); // free old head
        return;
    }

    // Search for the key to be deleted, keep track of the
    // previous node as we need to change 'prev->next'
    while (temp != NULL && temp->data != key)
    {
        prev = temp;
        temp = temp->next;
    }

    // If key was not present in linked list
```

```
if (temp == NULL) return;

// Unlink the node from linked list
prev->next = temp->next;

free(temp); // Free memory
}

// This function prints contents of linked list starting from
// the given node
void printList(struct Node *node)
{
    while (node != NULL)
    {
        printf(" %d ", node->data);
        node = node->next;
    }
}

/* Drier program to test above functions*/
int main()
{
    /* Start with the empty list */
    struct Node* head = NULL;

    push(&head, 7);
    push(&head, 1);
    push(&head, 3);
    push(&head, 2);

    puts("Created Linked List: ");
    printList(head);
    deleteNode(&head, 1);
    puts("\nLinked List after Deletion of 1: ");
    printList(head);
    return 0;
}
```

### Java

```
// A complete working Java program to demonstrate deletion in singly
// linked list
class LinkedList
{
    Node head; // head of list

    /* Linked list Node*/
    class Node
```

```
{  
    int data;  
    Node next;  
    Node(int d)  
    {  
        data = d;  
        next = null;  
    }  
}  
  
/* Given a key, deletes the first occurrence of key in linked list */  
void deleteNode(int key)  
{  
    // Store head node  
    Node temp = head, prev = null;  
  
    // If head node itself holds the key to be deleted  
    if (temp != null && temp.data == key)  
    {  
        head = temp.next; // Changed head  
        return;  
    }  
  
    // Search for the key to be deleted, keep track of the  
    // previous node as we need to change temp.next  
    while (temp != null && temp.data != key)  
    {  
        prev = temp;  
        temp = temp.next;  
    }  
  
    // If key was not present in linked list  
    if (temp == null) return;  
  
    // Unlink the node from linked list  
    prev.next = temp.next;  
}  
  
/* Inserts a new Node at front of the list. */  
public void push(int new_data)  
{  
    Node new_node = new Node(new_data);  
    new_node.next = head;  
    head = new_node;  
}  
  
/* This function prints contents of linked list starting from  
the given node */
```

```
public void printList()
{
    Node tnode = head;
    while (tnode != null)
    {
        System.out.print(tnode.data+" ");
        tnode = tnode.next;
    }
}

/* Drier program to test above functions. Ideally this function
should be in a separate user class. It is kept here to keep
code compact */
public static void main(String[] args)
{
    LinkedList llist = new LinkedList();

    llist.push(7);
    llist.push(1);
    llist.push(3);
    llist.push(2);

    System.out.println("\nCreated Linked list is:");
    llist.printList();

    llist.deleteNode(1); // Delete node at position 4

    System.out.println("\nLinked List after Deletion at position 4:");
    llist.printList();
}
}
```

### Python

```
# Python program to delete a node from linked list

# Node class
class Node:

    # Constructor to initialize the node object
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:

    # Function to initialize head
    def __init__(self):
```

```
    self.head = None

# Function to insert a new node at the beginning
def push(self, new_data):
    new_node = Node(new_data)
    new_node.next = self.head
    self.head = new_node

# Given a reference to the head of a list and a key,
# delete the first occurrence of key in linked list
def deleteNode(self, key):

    # Store head node
    temp = self.head

    # If head node itself holds the key to be deleted
    if (temp is not None):
        if (temp.data == key):
            self.head = temp.next
            temp = None
            return

    # Search for the key to be deleted, keep track of the
    # previous node as we need to change 'prev.next'
    while(temp is not None):
        if temp.data == key:
            break
        prev = temp
        temp = temp.next

    # if key was not present in linked list
    if(temp == None):
        return

    # Unlink the node from linked list
    prev.next = temp.next

    temp = None

# Utility function to print the linked LinkedList
def printList(self):
    temp = self.head
    while(temp):
        print " %d" %(temp.data),
        temp = temp.next
```

```
# Driver program
llist = LinkedList()
llist.push(7)
llist.push(1)
llist.push(3)
llist.push(2)

print "Created Linked List: "
llist.printList()
llist.deleteNode(1)
print "\nLinked List after Deletion of 1:"
llist.printList()

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

Output:

```
Created Linked List:
2 3 1 7
Linked List after Deletion of 1:
2 3 7
```

Improved By : [mlv](#)

## Source

<https://www.geeksforgeeks.org/linked-list-set-3-deleting-node/>

# Chapter 135

## LinkedList in Java

LinkedList in Java - GeeksforGeeks

LinkedList are linear data structures where the elements are not stored in contiguous locations and every element is a separate object with a data part and address part. The elements are linked using pointers and addresses. Each element is known as a node. Due to the dynamicity and ease of insertions and deletions, they are preferred over the arrays. It also has few disadvantages like the nodes cannot be accessed directly instead we need to start from the head and follow through the link to reach to a node we wish to access. To store the elements in a linked list we use a doubly linked list which provides a linear data structure and also used to inherit an abstract class and implement list and deque interfaces.

In Java, LinkedList class implements the [list interface](#). The LinkedList class also consists of various constructors and methods like other java collections.

### Constructors for Java LinkedList:

1. LinkedList(): Used to create an empty linked list.
2. LinkedList(Collection C): Used to create a ordered list which contains all the elements of a specified collection, as returned by the collection's iterator.

```
// Java code for Linked List implementation

import java.util.*;

public class Test
{
    public static void main(String args[])
    {
        // Creating object of class linked list
        LinkedList<String> object = new LinkedList<String>();

        // Adding elements to the linked list
        object.add("A");
    }
}
```

```
object.add("B");
object.addLast("C");
object.addFirst("D");
object.add(2, "E");
object.add("F");
object.add("G");
System.out.println("Linked list : " + object);

// Removing elements from the linked list
object.remove("B");
object.remove(3);
object.removeFirst();
object.removeLast();
System.out.println("Linked list after deletion: " + object);

// Finding elements in the linked list
boolean status = object.contains("E");

if(status)
    System.out.println("List contains the element 'E' ");
else
    System.out.println("List doesn't contain the element 'E'"');

// Number of elements in the linked list
int size = object.size();
System.out.println("Size of linked list = " + size);

// Get and set elements from linked list
Object element = object.get(2);
System.out.println("Element returned by get() : " + element);
object.set(2, "Y");
System.out.println("Linked list after change : " + object);
}

}
```

**Output:**

```
Linked list : [D, A, E, B, C, F, G]
Linked list after deletion: [A, E, F]
List contains the element 'E'
Size of linked list = 3
Element returned by get() : F
Linked list after change : [A, E, Y]
```

**Methods for Java *LinkedList*:**

1. **int size():** It returns the number of elements in this list.

2. `void clear()`: It removes all of the elements from the list.
3. `Object clone()`: It is used to make the copy of an existing linked list.
4. `Object set(int index, Object element)`: It is used to replace an existing element in the list with a new element.
5. `boolean contains(Object element)`: It returns true if the element is present in the list.
6. `boolean add(Object element)`: It appends the element to the end of the list.
7. `void add(int index, Object element)`: It inserts the element at the position ‘index’ in the list.
8. `boolean addAll(Collection C)`: It appends a collection to a Linked List.
9. `boolean addAll(int index, Collection C)`: It appends a collection to a linked list at a specified position.
10. `void addFirst(Object element)`: It inserts the element at the beginning of the list.
11. `void addLast(Object element)`: It appends the element at the end of the list.
12. `Object get(int index)`: It returns the element at the position ‘index’ in the list. It throws ‘IndexOutOfBoundsException’ if the index is out of range of the list.
13. `Object getFirst()`: It returns the first element of the Linked List.
14. `Object getLast()`: It returns the last element of the Linked List.
15. `int indexOf(Object element)`: If element is found, it returns the index of the first occurrence of the element. Else, it returns -1.
16. `int lastIndexOf(Object element)`: If element is found, it returns the index of the last occurrence of the element. Else, it returns -1.
17. `Object remove()`: It is used to remove and return the element from the head of the list.
18. `Object remove(int index)`: It removes the element at the position ‘index’ in this list. It throws ‘NoSuchElementException’ if the list is empty.
19. `boolean remove(Object O)`: It is used to remove a particular element from the linked list and returns a boolean value.
20. `Object removeLast()`: It is used to remove and return the last element of the Linked List.

This article is contributed by Mehak Narang.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

**Improved By :** [Chinmoy Lenka](#)

## Source

<https://www.geeksforgeeks.org/linked-list-in-java/>

## Chapter 136

# Longest Common Prefix using Linked List

Longest Common Prefix using Linked List - GeeksforGeeks

Given a set of strings, find the longest common prefix.

**Examples:**

```
Input : {"geeksforgeeks", "geeks", "geek", "geezer"}  
Output : "gee"
```

```
Input : {"apple", "ape", "april"}  
Output : "ap"
```

**Previous Approaches:** Word by Word Matching, Character by Character Matching, Divide and Conquer, Binary Search, Using Trie Data Structure

Below is an algorithm to solve above problem using **Linked List**.

- Create a linked list using the characters of the first string as the data in the linked list.
- Then one by one using all the remaining strings, iterate over the linked list deleting all the nodes after the point where the string gets exhausted or linked list gets exhausted or the characters do not match.
- The remaining data in linked list is the required longest common prefix.

Below is the implementation in C++

```
// C++ Program to find the longest common prefix  
// in an array of strings  
#include <bits/stdc++.h>
```

```
using namespace std;

// Structure for a node in the linked list
struct Node {
    char data;
    Node* next;
};

// Function to print the data in the linked list
// Remaining nodes represent the longest common prefix
void printLongestCommonPrefix(Node* head)
{
    // If the linked list is empty there is
    // no common prefix
    if (head == NULL) {
        cout << "There is no common prefix\n";
        return;
    }

    // Printing the longest common prefix
    cout << "The longest common prefix is ";
    while (head != NULL) {
        cout << head->data;
        head = head->next;
    }
}

// Function that creates a linked list of characters
// for the first word in the array of strings
void Initialise(Node** head, string str)
{
    // We calculate the length of the string
    // as we will insert from the last to the
    // first character
    int l = str.length();

    // Inserting all the nodes with the characters
    // using insert at the beginning technique
    for (int i = l - 1; i >= 0; i--) {
        Node* temp = new Node;
        temp->data = str[i];
        temp->next = *head;
        *head = temp;
    }

    // Since we have passed the address of the
    // head node it is not required to return
    // anything
}
```

```
}  
  
// Function to delete all the nodes  
// from the unmatched node till the end of the  
// linked list  
void deleteNodes(Node* head)  
{  
    // temp is used to facilitate the deletion of nodes  
    Node* current = head;  
    Node* next;  
    while (current != NULL) {  
        next = current->next;  
        free(current);  
        current = next;  
    }  
}  
  
// Function that compares the character of the string with  
// the nodes of the linked list and deletes all nodes after  
// the characters that do not match  
void longestCommonPrefix(Node** head, string str)  
{  
    int i = 0;  
  
    // Use the pointer to the previous node to  
    // delete the link between the unmatched node  
    // and its prev node  
    Node* temp = *head;  
    Node* prev = *head;  
    while (temp != NULL) {  
  
        // If the current string finishes or if the  
        // the characters in the linked list do not match  
        // with the character at the corresponding position  
        // delete all the nodes after that.  
        if (str[i] == '\0' || temp->data != str[i]) {  
  
            // If the first node does not match then there  
            // is no common prefix  
            if (temp == *head) {  
                free(temp);  
                *head = NULL;  
            }  
  
            // Delete all the nodes starting from the  
            // unmatched node  
            else {  
                prev->next = NULL;  
            }  
        }  
    }  
}
```

```
        deleteNodes(temp);
    }
    break;
}

// If the character matches, move to next
// node and store the address of the current
// node in prev
prev = temp;
temp = temp->next;
i++;
}
}

int main()
{
    string arr[] = { "geeksforgeeks", "geeks", "geek", "geezer",
                     "geekathon" };
    int n = sizeof(arr) / sizeof(arr[0]);

    struct Node* head = NULL;

    // A linked list is created with all the characters
    // of the first string
    Initialise(&head, arr[0]);

    // Process all the remaining strings to find the
    // longest common prefix
    for (int i = 1; i < n; i++)
        longestCommonPrefix(&head, arr[i]);

    printLongestCommonPrefix(head);
}
```

**Output:**

The longest common prefix is gee

**Source**

<https://www.geeksforgeeks.org/longest-common-prefix-using-linked-list/>

## Chapter 137

# Longest common suffix of two linked lists

Longest common suffix of two linked lists - GeeksforGeeks

Given two singly linked lists, find the Longest common suffix of two linked lists. If there are no common characters which are suffixes, return the minimum length of the two linked lists.

Examples:

```
Input : list1 = w -> a -> l -> k -> i -> n -> g
        list2 = l -> i -> s -> t -> e -> n -> i -> n -> g
Output :i -> n -> g

Input : list1 = p -> a -> r -> t -> y
        list2 = p -> a -> r -> t -> y -> i -> n -> g
Output :p -> a -> r -> t -> y
```

A simple solution is to use auxiliary arrays to store linked lists. Then print longest commons suffix of two arrays.

The above solution requires extra space. We can save space by first doing reverse of both linked lists. After reversing, we can easily find length of longest common prefix. Reversing again to get the original lists back.

One important point here is, order of elements. We need to print nodes from n-th to end. We use the above found count and print nodes in required order using two pointer approach.

```
// C++ program to find the Longest Common
// suffix in linked lists
#include <bits/stdc++.h>
using namespace std;
```

```
/* Linked list node */
struct Node
{
    char data;
    struct Node* next;
};

/* Function to insert a node at the beginning of
   the linked list */
void push(struct Node **head_ref, int new_data)
{
    struct Node* new_node = new Node;
    new_node->data = new_data;
    new_node->next = *head_ref;
    *head_ref = new_node;
}

/* Function to reverse the linked list */
struct Node *reverseList(struct Node *head_ref)
{
    struct Node *current, *prev, *next;
    current = head_ref;
    prev = NULL;

    while (current != NULL)
    {
        next = current->next;
        current->next = prev;
        prev = current;
        current = next;
    }
    return prev;
}

// Utility function to print last n nodes
void printLastNNode(struct Node* head, int n)
{
    // if n == 0
    if (n <= 0)
        return;

    // Move reference pointer n positions ahead
    struct Node* ref_ptr = head;
    while (ref_ptr != NULL && n--)
        ref_ptr = ref_ptr->next;
```

```
// Now move main and reference pointers at
// same speed. By the end of this loop,
// reference pointer would point to end and
// main pointer would point to n-th node
// from end.
Node *main_ptr = head;
while (ref_ptr != NULL) {
    main_ptr = main_ptr->next;
    ref_ptr = ref_ptr->next;
}

// Print last n nodes.
while (main_ptr != NULL)
{
    cout << main_ptr->data;
    main_ptr = main_ptr->next;
}
}

// Prints the Longest Common suffix in
// linked lists
void longestCommSuffix(Node *h1, Node *h2)
{
    // Reverse Both Linked list
    h1 = reverseList(h1);
    h2 = reverseList(h2);

    // Now we print common nodes from head
    Node *temp1 = h1, *temp2 = h2;
    int count = 0;
    while (temp1!=NULL&&temp2!=NULL)
    {
        // If a node is not common, break
        if (temp1 -> data != temp2 -> data)
            break;

        // Keep printing while there are
        // common nodes.
        count++;
        temp1 = temp1 -> next;
        temp2 = temp2 -> next;
    }

    // Reversing linked lists to retain
    // original lists.
    h1 = reverseList(h1);
    h2 = reverseList(h2);
```

```
    printLastNNode(h1, count);
}

// Driver program to test above
int main()
{
    struct Node *h1 = NULL, *h2 = NULL;

    // creating the 1 linked list
    push(&h1, 'g');
    push(&h1, 'n');
    push(&h1, 'i');
    push(&h1, 'k');
    push(&h1, 'l');
    push(&h1, 'a');
    push(&h1, 'w');

    // creating the 2 linked list
    push(&h2, 'g');
    push(&h2, 'n');
    push(&h2, 'i');
    push(&h2, 'n');
    push(&h2, 'e');
    push(&h2, 't');
    push(&h2, 's');
    push(&h2, 'i');
    push(&h2, 'l');

    longestCommSuffix(h1, h2);

    return 0;
}
```

**Output:**

gni

Time Complexity : O(N)

**Source**

<https://www.geeksforgeeks.org/longest-common-suffix-two-linked-lists/>

## Chapter 138

# Longest increasing sublist in a linked list

Longest increasing sublist in a linked list - GeeksforGeeks

Given a singly linked list and we want to count the elements that are continuously increasing and print the increasing linked list.

Examples:

```
Input : 8 -> 5 -> 7 -> 10 -> 9 -> 11 -> 12 -> 13 -> NULL
Output : Number of continuously increasing elements = 4
         Increasing linked list : 9 11 12 13
```

```
Input : 5 -> 12 -> 18 -> 7 -> 12 -> 15 -> NULL
Output : Number of continuously increasing elements = 3
         Increasing linked list = 5 12 18
```

The idea is to traverse singly linked list and compare curr->data with curr->next->data where curr is current node being traversed. If curr->data is smaller then curr->next->data then curr pointer point to curr->next and increment the length (continuous increasing element) by one. If the condition is false then compare the length with max and if max is less than len then assign the len value to max. Continue this process until head not equal to NULL. Also find the starting index of continuous increasing element. Next traverse the linked list and display the continuous increasing element in linked list.

```
// Program to count maximum number of continuous
// increasing element in linked list and display
// the elements of linked list.
#include <bits/stdc++.h>
using namespace std;
```

```
struct Node {
    int data;
    struct Node* next;
};

// Function that count maximum number of continuous
// increasing elements in linked list and display
// the list.
void countIncreasingElements(struct Node *head)
{
    // Traverse the list and keep track of max increasing
    // and current increasing lengths
    int curr_len = 1, max_len = 1;
    int total_count = 1, res_index = 0;
    for (Node *curr=head; curr->next!=NULL; curr=curr->next)
    {
        // Compare head->data with head->next->data
        if (curr->data < curr->next->data)
            curr_len++;
        else
        {
            // compare maximum length with len.
            if (max_len < curr_len)
            {
                max_len = curr_len;
                res_index = total_count - curr_len;
            }

            curr_len = 1;
        }
        total_count++;
    }

    if (max_len < curr_len)
    {
        max_len = curr_len;
        res_index = total_count - max_len;
    }

    // Print the maximum number of continuous elements
    // in linked list.
    cout << "Number of continuously increasing element"
        " in list : ";
    cout << max_len << endl;

    // Traverse the list again to print longest increasing
    // sublist
    int i = 0;
```

```
cout << "Increasing linked list" << endl;
for (Node* curr=head; curr!=NULL; curr=curr->next)
{
    // compare with starting index and index of
    // maximum increasing elements if both are
    // equals then execute it.
    if (i == res_index)
    {
        // loop until max greater then 0.
        while (max_len > 0)
        {
            // Display the list and temp point
            // to the next element.
            cout << curr->data << " ";
            curr = curr->next;
            max_len--;
        }
        break;
    }

    i++;
}
}

// Function to insert an element at the beginning
void push(struct Node** head, int data)
{
    struct Node* newNode = new Node;
    newNode->data = data;
    newNode->next = (*head);
    (*head) = newNode;
}

// Display linked list.
void printList(struct Node* node)
{
    while (node != NULL) {
        cout << node->data << " ";
        node = node->next;
    }
    cout << endl;
}

// Drier functions
int main()
{
    // Create a node and initialize with NULL
    struct Node* head = NULL;
```

```
// push() insert node in linked list.  
// 15 -> 18 -> 5 -> 8 -> 11 -> 12  
push(&head, 12);  
push(&head, 11);  
push(&head, 8);  
push(&head, 5);  
push(&head, 18);  
push(&head, 15);  
cout << "Linked list:" << endl;  
printList(head);  
  
// Function call countIncreasingElements(head)  
// cout << countIncreasingElements(head) << endl;  
countIncreasingElements(head);  
return 0;  
}
```

Output:

```
Linked list:  
15 18 5 8 11 12  
Number of continuously increasing element in list :4  
Increasing linked list  
5 8 11 12
```

## Source

<https://www.geeksforgeeks.org/longest-increasing-sublist-linked-list/>

## Chapter 139

# Lucky alive person in a circle | Code Solution to sword puzzle

[Lucky alive person in a circle | Code Solution to sword puzzle - GeeksforGeeks](#)

Given n people standing in a circle where 1st is having sword, find the luckiest person in the circle, if from 1st soldier who is having a sword each have to kill the next soldier and handover the sword to next soldier, in turn the soldier will kill the adjacent soldier and handover the sword to next soldier such that one soldier remain in this war who is not killed by anyone.

Prerequisite : [Puzzle 81 | 100 people in a circle with gun puzzle](#)

Examples :

Input : 5

Output : 3

Explanation :

N = 5

Soldier 1 2 3 4 5 (5 soldiers)

In first go 1 3 5 (remains) as 2 and 4 killed by 1 and 3.

In second go 3 as 5 killed 1 and 3rd kill 5 soldier 3 remains alive.

Input : 100

Output : 73

Explanation :

N = 10

Soldiers 1 2 3 4 5 6 7 8 9 10 (10 soldiers)

In first 1 3 5 7 9 as 2 4 6 8 10 were killed by 1 3 5 7 and 9.

In second 1 5 9 as 9 kill 1 and in turn 5 kill 9th soldier.

In third 5 5th soldiers remain alive

**Approach :** The idea is to use [circular linked list](#). A circular linked list is made based on number of soldier N. As rule state you have to kill your adjacent soldier and handover

the sword to the next soldier who in turn kill his adjacent soldier and handover sword to the next soldier. So in circular linked list the adjacent soldier are killed and the remaining soldier fights against each other in a circular way and a single soldier survive who is not killed by anyone.

```
// CPP code to find the luckiest person
#include <bits/stdc++.h>
using namespace std;

// Node structure
struct Node {
    int data;
    struct Node* next;
};

Node *newNode(int data)
{
    Node *node = new Node;
    node->data = data;
    node->next = NULL;
    return node;
}

// Function to find the luckiest person
int alivesol(int Num)
{
    if (Num == 1)
        return 1;

    // Create a single node circular
    // linked list.
    Node *last = newNode(1);
    last->next = last;

    for (int i = 2; i <= Num; i++) {
        Node *temp = newNode(i);
        temp->next = last->next;
        last->next = temp;
        last = temp;
    }

    // Starting from first soldier.
    Node *curr = last->next;

    // condition for evaluating the existence
    // of single soldier who is not killed.
    Node *temp;
    while (curr->next != curr) {
```

```
temp = curr;
curr = curr->next;
temp->next = curr->next;

// deleting soldier from the circular
// list who is killed in the fight.
delete curr;
temp = temp->next;
curr = temp;
}

// Returning the Luckiest soldier who
// remains alive.
int res = temp->data;
delete temp;

return res;
}

// Driver code
int main()
{
    int N = 100;
    cout << alivesol(N) << endl;
    return 0;
}
```

**Output:**

73

**Source**

<https://www.geeksforgeeks.org/lucky-alive-person-circle/>

## Chapter 140

# Majority element in a linked list

Majority element in a linked list - GeeksforGeeks

Given a linked list, find majority element. An element is called **Majority element** if it appears more than or equal to  $n/2$  times where n is total number of nodes in the linked list.

Examples:

```
Input  : 1->2->3->4->5->1->1->1->NULL
Output : 1
Explanation 1 occurs 4 times
Input :10->23->11->9->54->NULL
Output :NO majority element
```

### Method 1(simple)

Run two loops starting from head and count frequency of each element iteratively. Print the element whose frequency is greater than or equal to  $n/2$ . In this approach time complexity will be  $O(n*n)$  where n is the number of nodes in the linked list.

C++

```
// C++ program to find majority element in
// a linked list
#include <bits/stdc++.h>
using namespace std;

/* Link list node */
struct Node {
    int data;
    struct Node* next;
};


```

```

/* Function to get the nth node from the
last of a linked list*/
int majority(struct Node* head)
{
    struct Node* p = head;

    int total_count = 0, max_count = 0, res = -1;
    while (p != NULL) {

        // Count all occurrences of p->data
        int count = 1;
        struct Node* q = p->next;
        while (q != NULL) {
            if (p->data == q->data)
                count++;
            q = q->next;
        }

        // Update max_count and res if count
        // is more than max_count
        if (count > max_count)
        {
            max_count = count;
            res = p->data;
        }

        p = p->next;
        total_count++;
    }

    if (max_count >= total_count/2)
        return res;

    // if we reach here it means no
    // majority element is present.
    // and we assume that all the
    // elements are positive
    return -1;
}

void push(struct Node** head_ref, int new_data)
{
    struct Node* new_node =
        (struct Node*)malloc(sizeof(struct Node));
    new_node->data = new_data;
    new_node->next = (*head_ref);
    (*head_ref) = new_node;
}

```

```
/* Driver program to test above function*/
int main()
{
    /* Start with the empty list */
    struct Node* head = NULL;

    // create linked
    push(&head, 1);
    push(&head, 1);
    push(&head, 1);
    push(&head, 5);
    push(&head, 4);
    push(&head, 3);
    push(&head, 2);
    push(&head, 1);

    int res = majority(head);

    if (res != (-1))
        cout << "Majority element is " << res;
    else
        cout << "No majority element";
    return 0;
}
```

Time Complexity  $O(n^*n)$

**Method 2** Use hashing technique. We count frequency of each element and then we print the element whose frequency is  $n/2$ ;

C++

```
// CPP program to find majority element
// in the linked list using hashing
#include <bits/stdc++.h>
using namespace std;

/* Link list node */
struct Node {
    int data;
    struct Node* next;
};

/* Function to get the nth node from the last
   of a linked list*/
int majority(struct Node* head)
{
```

```
struct Node* p = head;

// Storing elements and their frequencies
// in a hash table.
unordered_map<int, int> hash;

int total_count = 0;
while (p != NULL) {

    // increase every element
    // frequency by 1
    hash[p->data]++;
}

p = p->next;
total_count++;
}

// Check if frequency of any element
// is more than or equal to total_count/2
for (auto x : hash)
    if (x.second >= total_count/2)
        return x.first;

// If we reach here means no majority element
// is present. We assume that all the element
// are positive
return -1;
}

void push(struct Node** head_ref, int new_data)
{
    struct Node* new_node =
        (struct Node*)malloc(sizeof(struct Node));
    new_node->data = new_data;
    new_node->next = (*head_ref);
    (*head_ref) = new_node;
}

// Driver program to test above function
int main()
{

    /* Start with the empty list */
    struct Node* head = NULL;
    push(&head, 1);
    push(&head, 1);
    push(&head, 1);
```

```
push(&head, 5);
push(&head, 4);
push(&head, 3);
push(&head, 2);
push(&head, 1);

int res = majority(head);

if (res != (-1))
    cout << "majority element is " << res;
else
    cout << "NO majority elemenet";
return 0;
}
```

Time Complexity O(n)

```
majority element is 1
```

## Source

<https://www.geeksforgeeks.org/majority-element-in-a-linked-list/>

## Chapter 141

# Make a loop at k-th position in a linked list

Make a loop at k-th position in a linked list - GeeksforGeeks

Given a linked list and a position k. Make a loop at k-th position

Examples:

Input : Given linked list

Output : Modified linked list

### Algorithm

- 1) Traverse the first linked list till k-th point
- 2) Make backup of the k-th node
- 3) Traverse the linked linked list till end
- 4) Attach the last node with k-th node

```
// CPP program for making loop at k-th index
// of given linked list
#include <bits/stdc++.h>
using namespace std;

/* Link list node */
struct Node {
    int data;
    struct Node* next;
};


```

```
/* Function to make loop at k-th elements of
linked list */
void makeloop(struct Node** head_ref, int k)
{
    // traverse the linked list until loop
    // point not found
    struct Node* temp = *head_ref;
    int count = 1;
    while (count < k) {
        temp = temp->next;
        count++;
    }

    // backup the joint point
    struct Node* joint_point = temp;

    // traverse remaining nodes
    while (temp->next != NULL)
        temp = temp->next;

    // joint the last node to k-th element
    temp->next = joint_point;
}

/* Function to push a node */
void push(struct Node** head_ref, int new_data)
{
    struct Node* new_node =
        (struct Node*)malloc(sizeof(struct Node));
    new_node->data = new_data;
    new_node->next = (*head_ref);
    (*head_ref) = new_node;
}

/* Function to print linked list */
void printList(struct Node* head, int total_nodes)
{
    struct Node* curr = head;
    int count = 0;
    while (count < total_nodes) {
        count++;
        cout << curr->data << " ";
        curr = curr->next;
    }
}

int countNodes(Node *ptr)
{
```

```
int count = 0;
while (ptr != NULL)
{
    ptr = ptr->next;
    count++;
}
return count;
}

/* Driver program to test above function*/
int main()
{
    // Create a linked list 1->2->3->4->5->6->7
    struct Node* head = NULL;
    push(&head, 7);
    push(&head, 6);
    push(&head, 5);
    push(&head, 4);
    push(&head, 3);
    push(&head, 2);
    push(&head, 1);

    // k should be less than the
    // numbers of nodes
    int k = 4;
    int total_nodes = countNodes(head);

    cout << "\nGiven list\n";
    printList(head, total_nodes);

    makeloop(&head, k);

    cout << "\nModified list\n";
    printList(head, total_nodes);
    return 0;
}
```

Output:

```
Given list
1 2 3 4 5 6 7
Modified list
1 2 3 4 5 6 7
```

## Source

<https://www.geeksforgeeks.org/make-loop-k-th-position-linked-list/>

## Chapter 142

# Make middle node head in a linked list

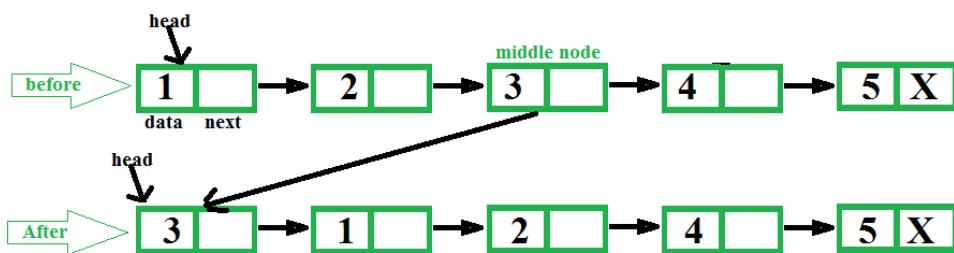
Make middle node head in a linked list - GeeksforGeeks

Given a singly linked list, find middle of the linked list and set middle node of the linked list at beginning of the linked list.

Examples:

Input : 1 2 3 4 5  
Output : 3 1 2 4 5

Input : 1 2 3 4 5 6  
Output : 4 1 2 3 5 6



The idea is to first [find middle of a linked list using two pointers](#), first one moves one at a time and second one moves two at a time. When second pointer reaches end, first reaches

middle. We also keep track of previous of first pointer so that we can remove middle node from its current position and can make it head.

## C

```
// C program to make middle node as head of
// linked list.
#include <stdio.h>
#include <stdlib.h>

/* Link list node */
struct Node {
    int data;
    struct Node* next;
};

/* Function to get the middle and set at
   beginning of the linked list*/
void setMiddleHead(struct Node** head)
{
    if (*head == NULL)
        return;

    // To traverse list nodes one by one
    struct Node* one_node = (*head);

    // To traverse list nodes by skipping
    // one.
    struct Node* two_node = (*head);

    // To keep track of previous of middle
    struct Node* prev = NULL;
    while (two_node != NULL && two_node->next != NULL) {

        /* for previous node of middle node */
        prev = one_node;

        /* move one node each time*/
        two_node = two_node->next->next;

        /* move two node each time*/
        one_node = one_node->next;
    }

    /* set middle node at head */
    prev->next = prev->next->next;
    one_node->next = (*head);
    (*head) = one_node;
}
```

```
}

// To insert a node at the beginning of linked
// list.
void push(struct Node** head_ref, int new_data)
{
    /* allocate node */
    struct Node* new_node =
        (struct Node*)malloc(sizeof(struct Node));

    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;
}

// A function to print a given linked list
void printList(struct Node* ptr)
{
    while (ptr != NULL) {
        printf("%d ", ptr->data);
        ptr = ptr->next;
    }
    printf("\n");
}

/* Driver function*/
int main()
{
    // Create a list of 5 nodes
    struct Node* head = NULL;
    int i;
    for (i = 5; i > 0; i--)
        push(&head, i);

    printf(" list before: ");
    printList(head);

    setMiddleHead(&head);

    printf(" list After:   ");
    printList(head);

    return 0;
}
```

**Java**

```
// Java program to make middle node
// as head of Linked list
public class GFG
{
    /* Link list node */
    static class Node {
        int data;
        Node next;
        Node(int data){
            this.data = data;
            next = null;
        }
    }

    static Node head;

    /* Function to get the middle and
    set at beginning of the linked list*/
    static void setMiddleHead()
    {
        if (head == null)
            return;

        // To traverse list nodes one
        // by one
        Node one_node = head;

        // To traverse list nodes by
        // skipping one.
        Node two_node = head;

        // To keep track of previous of middle
        Node prev = null;
        while (two_node != null &&
               two_node.next != null) {

            /* for previous node of middle node */
            prev = one_node;

            /* move one node each time*/
            two_node = two_node.next.next;

            /* move two node each time*/
            one_node = one_node.next;
        }
    }
}
```

```
/* set middle node at head */
prev.next = prev.next.next;
one_node.next = head;
head = one_node;
}

// To insert a node at the beginning of
// linked list.
static void push(int new_data)
{
    /* allocate node */
    Node new_node = new Node(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;
}

// A function to print a given linked list
static void printList(Node ptr)
{
    while (ptr != null) {
        System.out.print(ptr.data+" ");
        ptr = ptr.next;
    }
    System.out.println();
}

/* Driver function*/
public static void main(String args[])
{
    // Create a list of 5 nodes
    head = null;
    int i;
    for (i = 5; i > 0; i--)
        push(i);

    System.out.print(" list before: ");
    printList(head);

    setMiddleHead();

    System.out.print(" list After:   ");
    printList(head);
}
```

```
}
```

```
// This code is contributed by Sumit Ghosh
```

Output:

```
list before: 1 2 3 4 5
list After : 3 1 2 4 5
```

### Source

<https://www.geeksforgeeks.org/make-middle-node-head-linked-list/>

## Chapter 143

# Maximum occurring character in a linked list

Maximum occurring character in a linked list - GeeksforGeeks

Given a linked list of characters. The task is to find the maximum occurring character in the linked list. If there are multiple answers return the first maximum occurring character.

Examples:

Input : g → e → e → k → s  
Output : e

Input : a → a → b → b → c → c → d → d  
Output : d

### Method 1:

Iteratively count the frequency of each character in a string and return the one which has maximum occurrence.

```
// CPP program to count the maximum
// occurring character in linked list
#include <bits/stdc++.h>
using namespace std;

/* Link list node */
struct Node {
    char data;
    struct Node *next;
};

char maxChar(struct Node *head) {
```

```
struct Node *p = head;

int max = -1;
char res;

while (p != NULL) {

    // counting the frequency of current element
    // p->data
    struct Node *q = p->next;
    int count = 1;
    while (q != NULL) {
        if (p->data == q->data)
            count++;

        q = q->next;
    }

    // if current counting is greater than max
    if (max < count) {
        res = p->data;
        max = count;
    }

    p = p->next;
}

return res;
}

/* Push a node to linked list. Note that
   this function changes the head */
void push(struct Node **head_ref, char new_data) {
    struct Node *new_node = new Node;
    new_node->data = new_data;
    new_node->next = (*head_ref);
    (*head_ref) = new_node;
}

/* Drier program to test above function*/
int main() {
    /* Start with the empty list */
    struct Node *head = NULL;
    char str[] = "skeegforskeeg";
    int i;

    // this will create a linked list of
    // character "geeksforgeeks"
```

```
for (i = 0; str[i] != '\0'; i++)
    push(&head, str[i]);

cout << maxChar(head);

return 0;
}
```

**Output:**

e

**Time complexity** $O(N^2)$

**Method 2: (use count array)**

Create a count array and count each character frequency return the maximum occurring character.

```
// CPP program to count the maximum
// occurring character in linked list
#include <bits/stdc++.h>
using namespace std;

/* Link list node */
struct Node {
    char data;
    struct Node *next;
};

char maxChar(struct Node *head) {
    struct Node *p = head;
    int hash[256] = {0};

    // Storing element's frequencies
    // in a hash table.
    while (p != NULL) {
        hash[p->data]++;
        p = p->next;
    }

    p = head;

    int max = -1;
    char res;

    // calculating the first maximum element
    while (p != NULL) {
```

```
if (max < hash[p->data]) {  
    res = p->data;  
    max = hash[p->data];  
}  
p = p->next;  
}  
return res;  
}  
  
/* Push a node to linked list. Note that  
   this function changes the head */  
void push(struct Node **head_ref, char new_data) {  
    struct Node *new_node = new Node;  
    new_node->data = new_data;  
    new_node->next = (*head_ref);  
    (*head_ref) = new_node;  
}  
  
/* Driver program to test above function*/  
int main() {  
    struct Node *head = NULL;  
    char str[] = "skeegforskeeg";  
    for (int i = 0; str[i] != '\0'; i++)  
        push(&head, str[i]);  
    cout << maxChar(head);  
    return 0;  
}
```

**Output:**

e

**Time complexity**  $O(N)$

**Source**

<https://www.geeksforgeeks.org/maximum-occurring-character-linked-list/>

## Chapter 144

# Memory efficient doubly linked list

Memory efficient doubly linked list - GeeksforGeeks

Asked by Varun Bhatia.

**Question:**

Write a code for implementation of doubly linked list with use of single pointer in each node.

**Solution:**

This question is solved and very well explained at <http://www.linuxjournal.com/article/6828>.

We also recommend to read [http://en.wikipedia.org/wiki/XOR\\_linked\\_list](http://en.wikipedia.org/wiki/XOR_linked_list)

### Source

<https://www.geeksforgeeks.org/memory-efficient-doubly-linked-list/>

## Chapter 145

# Merge K sorted linked lists | Set 1

Merge K sorted linked lists | Set 1 - GeeksforGeeks

Given K sorted linked lists of size N each, merge them and print the sorted output.

Example:

```
Input: k = 3, n = 4
list1 = 1->3->5->7->NULL
list2 = 2->4->6->8->NULL
list3 = 0->9->10->11
```

Output:

```
0->1->2->3->4->5->6->7->8->9->10->11
```

### Method 1 (Simple)

A Simple Solution is to initialize result as first list. Now traverse all lists starting from second list. Insert every node of currently traversed list into result in a sorted way. Time complexity of this solution is  $O(N^2)$  where N is total number of nodes, i.e.,  $N = kn$ .

### Method 2 (Using Min Heap)

A **Better solution** is to use Min Heap based solution which is discussed [here](#) for arrays. Time complexity of this solution would be  $O(nk \log k)$

### Method 3 (Using Divide and Conquer))

In this post, **Divide and Conquer** approach is discussed. This approach doesn't require extra space for heap and works in  $O(nk \log k)$

We already know that [merging of two linked lists](#) can be done in  $O(n)$  time and  $O(1)$  space (For arrays  $O(n)$  space is required). The idea is to pair up  $K$  lists and merge each pair in linear time using  $O(1)$  space. After first cycle,  $K/2$  lists are left each of size  $2^*N$ . After second cycle,  $K/4$  lists are left each of size  $4^*N$  and so on. We repeat the procedure until we have only one list left.

Below is C++ implementation of the above idea.

```
// C++ program to merge k sorted arrays of size n each
#include <bits/stdc++.h>
using namespace std;

// A Linked List node
struct Node
{
    int data;
    Node* next;
};

/* Function to print nodes in a given linked list */
void printList(Node* node)
{
    while (node != NULL)
    {
        printf("%d ", node->data);
        node = node->next;
    }
}

/* Takes two lists sorted in increasing order, and merge
   their nodes together to make one big sorted list. Below
   function takes  $O(\log n)$  extra space for recursive calls,
   but it can be easily modified to work with same time and
    $O(1)$  extra space */
Node* SortedMerge(Node* a, Node* b)
{
    Node* result = NULL;

    /* Base cases */
    if (a == NULL)
        return (b);
    else if(b == NULL)
        return (a);

    /* Pick either a or b, and recur */
    if (a->data <= b->data)
    {
        result = a;
        result->next = SortedMerge(a->next, b);
    }
    else
    {
        result = b;
        result->next = SortedMerge(a, b->next);
    }
    return (result);
}
```

```

/* Pick either a or b, and recur */
if(a->data <= b->data)
{
    result = a;
    result->next = SortedMerge(a->next, b);
}
else
{
    result = b;
    result->next = SortedMerge(a, b->next);
}

return result;
}

// The main function that takes an array of lists
// arr[0..last] and generates the sorted output
Node* mergeKLists(Node* arr[], int last)
{
    // repeat until only one list is left
    while (last != 0)
    {
        int i = 0, j = last;

        // (i, j) forms a pair
        while (i < j)
        {
            // merge List i with List j and store
            // merged list in List i
            arr[i] = SortedMerge(arr[i], arr[j]);

            // consider next pair
            i++, j--;

            // If all pairs are merged, update last
            if (i >= j)
                last = j;
        }
    }

    return arr[0];
}

// Utility function to create a new node.
Node *newNode(int data)
{
    struct Node *temp = new Node;
    temp->data = data;
}

```

```
temp->next = NULL;
return temp;
}

// Driver program to test above functions
int main()
{
    int k = 3; // Number of linked lists
    int n = 4; // Number of elements in each list

    // an array of pointers storing the head nodes
    // of the linked lists
    Node* arr[k];

    arr[0] = newNode(1);
    arr[0]->next = newNode(3);
    arr[0]->next->next = newNode(5);
    arr[0]->next->next->next = newNode(7);

    arr[1] = newNode(2);
    arr[1]->next = newNode(4);
    arr[1]->next->next = newNode(6);
    arr[1]->next->next->next = newNode(8);

    arr[2] = newNode(0);
    arr[2]->next = newNode(9);
    arr[2]->next->next = newNode(10);
    arr[2]->next->next->next = newNode(11);

    // Merge all lists
    Node* head = mergeKLists(arr, k - 1);

    printList(head);

    return 0;
}
```

Output :

0 1 2 3 4 5 6 7 8 9 10 11

Time Complexity of above algorithm is  $O(nk \log k)$  as outer while loop in function `mergeKLists()` runs  $\log k$  times and every time we are processing  $nk$  elements.

[Merge k sorted linked lists | Set 2 \(Using Min Heap\)](#)

**Source**

<https://www.geeksforgeeks.org/merge-k-sorted-linked-lists/>

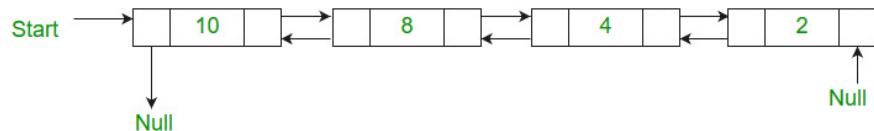
## Chapter 146

# Merge Sort for Doubly Linked List

Merge Sort for Doubly Linked List - GeeksforGeeks

Given a doubly linked list, write a function to sort the doubly linked list in increasing order using merge sort.

For example, the following doubly linked list should be changed to 24810



[Merge sort for singly linked list](#) is already discussed. The important change here is to modify the previous pointers also when merging two lists.

Below is the implementation of merge sort for doubly linked list.

C

```
// C program for merge sort on doubly linked list
#include<stdio.h>
#include<stdlib.h>
struct Node
{
    int data;
    struct Node *next, *prev;
};

struct Node *split(struct Node *head);

// Function to merge two linked lists
struct Node *merge(struct Node *first, struct 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->next = merge(first->next,second);  
        first->next->prev = first;  
        first->prev = NULL;  
        return first;  
    }  
    else  
    {  
        second->next = merge(first,second->next);  
        second->next->prev = second;  
        second->prev = NULL;  
        return second;  
    }  
}  
  
// Function to do merge sort  
struct Node *mergeSort(struct Node *head)  
{  
    if (!head || !head->next)  
        return head;  
    struct 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);  
}  
  
// A utility function to insert a new node at the  
// beginning of doubly linked list  
void insert(struct Node **head, int data)  
{  
    struct Node *temp =  
        (struct Node *)malloc(sizeof(struct Node));  
    temp->data = data;
```

```

temp->next = temp->prev = NULL;
if (!(*head))
    (*head) = temp;
else
{
    temp->next = *head;
    (*head)->prev = temp;
    (*head) = temp;
}
}

// A utility function to print a doubly linked list in
// both forward and backward directions
void print(struct Node *head)
{
    struct Node *temp = head;
    printf("Forward Traversal using next pointer\n");
    while (head)
    {
        printf("%d ", head->data);
        temp = head;
        head = head->next;
    }
    printf("\nBackward Traversal using prev pointer\n");
    while (temp)
    {
        printf("%d ", temp->data);
        temp = temp->prev;
    }
}

// Utility function to swap two integers
void swap(int *A, int *B)
{
    int temp = *A;
    *A = *B;
    *B = temp;
}

// Split a doubly linked list (DLL) into 2 DLLs of
// half sizes
struct Node *split(struct Node *head)
{
    struct Node *fast = head,*slow = head;
    while (fast->next && fast->next->next)
    {
        fast = fast->next->next;
        slow = slow->next;
    }
}

```

```
}

struct Node *temp = slow->next;
slow->next = NULL;
return temp;
}

// Driver program
int main(void)
{
    struct Node *head = NULL;
    insert(&head,5);
    insert(&head,20);
    insert(&head,4);
    insert(&head,3);
    insert(&head,30);
    insert(&head,10);
    head = mergeSort(head);
    printf("\n\nLinked List after sorting\n");
    print(head);
    return 0;
}
```

### Java

```
// Java program to implement merge sort in singly linked list

// Linked List Class
class LinkedList {

    static Node head; // head of list

    /* Node Class */
    static class Node {

        int data;
        Node next, prev;

        // Constructor to create a new node
        Node(int d) {
            data = d;
            next = prev = null;
        }
    }

    void print(Node node) {
        Node temp = node;
        System.out.println("Forward Traversal using next pointer");
        while (node != null) {
```

```
        System.out.print(node.data + " ");
        temp = node;
        node = node.next;
    }
    System.out.println("\nBackward Traversal using prev pointer");
    while (temp != null) {
        System.out.print(temp.data + " ");
        temp = temp.prev;
    }
}

// Split a doubly linked list (DLL) into 2 DLLs of
// half sizes
Node split(Node head) {
    Node fast = head, slow = head;
    while (fast.next != null && fast.next.next != null) {
        fast = fast.next.next;
        slow = slow.next;
    }
    Node temp = slow.next;
    slow.next = null;
    return temp;
}

Node mergeSort(Node node) {
    if (node == null || node.next == null) {
        return node;
    }
    Node second = split(node);

    // Recur for left and right halves
    node = mergeSort(node);
    second = mergeSort(second);

    // Merge the two sorted halves
    return merge(node, second);
}

// Function to merge two linked lists
Node merge(Node first, Node second) {
    // If first linked list is empty
    if (first == null) {
        return second;
    }

    // If second linked list is empty
    if (second == null) {
        return first;
```

```
}

// Pick the smaller value
if (first.data < second.data) {
    first.next = merge(first.next, second);
    first.next.prev = first;
    first.prev = null;
    return first;
} else {
    second.next = merge(first, second.next);
    second.next.prev = second;
    second.prev = null;
    return second;
}
}

// Driver program to test above functions
public static void main(String[] args) {

    LinkedList list = new LinkedList();
    list.head = new Node(10);
    list.head.next = new Node(30);
    list.head.next.next = new Node(3);
    list.head.next.next.next = new Node(4);
    list.head.next.next.next.next = new Node(20);
    list.head.next.next.next.next.next = new Node(5);

    Node node = null;
    node = list.mergeSort(head);
    System.out.println("Linked list after sorting :");
    list.print(node);
}

}

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

### Python

```
# Program for merge sort on doubly linked list

# A node of the doubly linked list
class Node:

    # Constructor to create a new node
    def __init__(self, data):
        self.data = data
```

```
        self.next = None
        self.prev = None

class DoublyLinkedList:

    # Constructor for empty Doubly Linked List
    def __init__(self):
        self.head = None

    # Function to merge two linked list
    def merge(self, first, second):

        # If first linked list is empty
        if first is None:
            return second

        # If secon linked list is empty
        if second is None:
            return first

        # Pick the smaller value
        if first.data < second.data:
            first.next = self.merge(first.next, second)
            first.next.prev = first
            first.prev = None
            return first
        else:
            second.next = self.merge(first, second.next)
            second.next.prev = second
            second.prev = None
            return second

    # Function to do merge sort
    def mergeSort(self, tempHead):
        if tempHead is None:
            return tempHead
        if tempHead.next is None:
            return tempHead

        second = self.split(tempHead)

        # Recur for left and righ halves
        tempHead = self.mergeSort(tempHead)
        second = self.mergeSort(second)

        # Merge the two sorted halves
        return self.merge(tempHead, second)
```

```
# Split the doubly linked list (DLL) into two DLLs
# of half sizes
def split(self, tempHead):
    fast = slow = tempHead
    while(True):
        if fast.next is None:
            break
        if fast.next.next is None:
            break
        fast = fast.next.next
        slow = slow.next

    temp = slow.next
    slow.next = None
    return temp

# Given a reference to the head of a list and an
# integer, inserts a new node on the front of list
def push(self, new_data):

    # 1. Allocates node
    # 2. Put the data in it
    new_node = Node(new_data)

    # 3. Make next of new node as head and
    # previous as None (already None)
    new_node.next = self.head

    # 4. change prev of head node to new_node
    if self.head is not None:
        self.head.prev = new_node

    # 5. move the head to point to the new node
    self.head = new_node

def printList(self, node):
    temp = node
    print "Forward Traversal using next pointer"
    while(node is not None):
        print node.data,
        temp = node
        node = node.next
    print "\nBackward Traversal using prev pointer"
    while(temp):
        print temp.data,
        temp = temp.prev
```

```
# Driver program to test the above functions
dll = DoublyLinkedList()
dll.push(5)
dll.push(20);
dll.push(4);
dll.push(3);
dll.push(30)
dll.push(10);
dll.head = dll.mergeSort(dll.head)
print "Linked List after sorting"
dll.printList(dll.head)

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

Output:

```
Linked List after sorting
Forward Traversal using next pointer
3 4 5 10 20 30
Backward Traversal using prev pointer
30 20 10 5 4 3
```

Thanks to Goku for providing above implementation in a comment [here](#).

**Time Complexity:** Time complexity of the above implementation is same as time complexity of [MergeSort for arrays](#). It takes  $\Theta(n\log n)$  time.

You may also like to see [QuickSort for doubly linked list](#)

## Source

<https://www.geeksforgeeks.org/merge-sort-for-doubly-linked-list/>

## Chapter 147

# Merge Sort for Linked Lists

Merge Sort for Linked Lists - GeeksforGeeks

[Merge sort](#) is often preferred for sorting a linked list. The slow random-access performance of a linked list makes some other algorithms (such as quicksort) perform poorly, and others (such as heapsort) completely impossible.

Let head be the first node of the linked list to be sorted and headRef be the pointer to head. Note that we need a reference to head in MergeSort() as the below implementation changes next links to sort the linked lists (not data at the nodes), so head node has to be changed if the data at original head is not the smallest value in linked list.

```
MergeSort(headRef)
1) If head is NULL or there is only one element in the Linked List
   then return.
2) Else divide the linked list into two halves.
   FrontBackSplit(head, &a, &b); /* a and b are two halves */
3) Sort the two halves a and b.
   MergeSort(a);
   MergeSort(b);
4) Merge the sorted a and b (using SortedMerge() discussed here)
   and update the head pointer using headRef.
   *headRef = SortedMerge(a, b);
```

C

```
// C code for linked list merged sort
#include<stdio.h>
#include<stdlib.h>

/* Link list node */
struct Node
```

```
{  
    int data;  
    struct Node* next;  
};  
  
/* function prototypes */  
struct Node* SortedMerge(struct Node* a, struct Node* b);  
void FrontBackSplit(struct Node* source,  
                     struct Node** frontRef, struct Node** backRef);  
  
/* sorts the linked list by changing next pointers (not data) */  
void MergeSort(struct Node** headRef)  
{  
    struct Node* head = *headRef;  
    struct Node* a;  
    struct Node* b;  
  
    /* Base case -- length 0 or 1 */  
    if ((head == NULL) || (head->next == NULL))  
    {  
        return;  
    }  
  
    /* Split head into 'a' and 'b' sublists */  
    FrontBackSplit(head, &a, &b);  
  
    /* Recursively sort the sublists */  
    MergeSort(&a);  
    MergeSort(&b);  
  
    /* answer = merge the two sorted lists together */  
    *headRef = SortedMerge(a, b);  
}  
  
/* See https://www.geeksforgeeks.org/?p=3622 for details of this  
function */  
struct Node* SortedMerge(struct Node* a, struct Node* b)  
{  
    struct Node* result = NULL;  
  
    /* Base cases */  
    if (a == NULL)  
        return(b);  
    else if (b==NULL)  
        return(a);  
  
    /* Pick either a or b, and recur */  
    if (a->data <= b->data)
```

```

{
    result = a;
    result->next = SortedMerge(a->next, b);
}
else
{
    result = b;
    result->next = SortedMerge(a, b->next);
}
return(result);
}

/* UTILITY FUNCTIONS */
/* Split the nodes of the given list into front and back halves,
   and return the two lists using the reference parameters.
   If the length is odd, the extra node should go in the front list.
   Uses the fast/slow pointer strategy. */
void FrontBackSplit(struct Node* source,
                     struct Node** frontRef, struct Node** backRef)
{
    struct Node* fast;
    struct Node* slow;
    slow = source;
    fast = source->next;

    /* Advance 'fast' two nodes, and advance 'slow' one node */
    while (fast != NULL)
    {
        fast = fast->next;
        if (fast != NULL)
        {
            slow = slow->next;
            fast = fast->next;
        }
    }

    /* 'slow' is before the midpoint in the list, so split it in two
       at that point. */
    *frontRef = source;
    *backRef = slow->next;
    slow->next = NULL;
}

/* Function to print nodes in a given linked list */
void printList(struct Node *node)
{
while(node!=NULL)
{

```

```
printf("%d ", node->data);
node = node->next;
}
}

/* Function to insert a node at the beginning of the linked list */
void push(struct Node** head_ref, int new_data)
{
/* allocate node */
struct Node* new_node =
    (struct Node*) malloc(sizeof(struct Node));

/* put in the data */
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;
}

/* Driver program to test above functions*/
int main()
{
/* Start with the empty list */
struct Node* res = NULL;
struct Node* a = NULL;

/* Let us create a unsorted linked lists to test the functions
Created lists shall be a: 2->3->20->5->10->15 */
push(&a, 15);
push(&a, 10);
push(&a, 5);
push(&a, 20);
push(&a, 3);
push(&a, 2);

/* Sort the above created Linked List */
MergeSort(&a);

printf("Sorted Linked List is: \n");
printList(a);

getchar();
return 0;
}
```

**Java**

```
// Java program to illustrate merge sorted
// of linkedList

public class linkedList
{
    node head = null;
    // node a,b;
    static class node
    {
        int val;
        node next;

        public node(int val)
        {
            this.val = val;
        }
    }

    node sortedMerge(node a, node b)
    {
        node result = null;
        /* Base cases */
        if (a == null)
            return b;
        if (b == null)
            return a;

        /* Pick either a or b, and recur */
        if (a.val <= b.val)
        {
            result = a;
            result.next = sortedMerge(a.next, b);
        }
        else
        {
            result = b;
            result.next = sortedMerge(a, b.next);
        }
        return result;
    }

    node mergeSort(node h)
    {
        // Base case : if head is null
        if (h == null || h.next == null)
```

```
{  
    return h;  
}  
  
// get the middle of the list  
node middle = getMiddle(h);  
node nextofmiddle = middle.next;  
  
// set the next of middle node to null  
middle.next = null;  
  
// Apply mergeSort on left list  
node left = mergeSort(h);  
  
// Apply mergeSort on right list  
node right = mergeSort(nextofmiddle);  
  
// Merge the left and right lists  
node sortedlist = sortedMerge(left, right);  
return sortedlist;  
}  
  
// Utility function to get the middle of the linked list  
node getMiddle(node h)  
{  
    //Base case  
    if (h == null)  
        return h;  
    node fastptr = h.next;  
    node slowptr = h;  
  
    // Move fastptr by two and slow ptr by one  
    // Finally slowptr will point to middle node  
    while (fastptr != null)  
    {  
        fastptr = fastptr.next;  
        if(fastptr!=null)  
        {  
            slowptr = slowptr.next;  
            fastptr=fastptr.next;  
        }  
    }  
    return slowptr;  
}  
  
void push(int new_data)  
{  
    /* allocate node */
```

```
node new_node = new node(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;
}

// Utility function to print the linked list
void printList(node headref)
{
    while (headref != null)
    {
        System.out.print(headref.val + " ");
        headref = headref.next;
    }
}

public static void main(String[] args)
{

linkedList li = new linkedList();
/*
 * Let us create a unsorted linked lists to test the functions Created
 * lists shall be a: 2->3->20->5->10->15
 */
li.push(15);
li.push(10);
li.push(5);
li.push(20);
li.push(3);
li.push(2);
System.out.println("Linked List without sorting is :");
li.printList(li.head);

// Apply merge Sort
li.head = li.mergeSort(li.head);
System.out.print("\n Sorted Linked List is: \n");
li.printList(li.head);
}

// This code is contributed by Rishabh Mahrsee
```

Time Complexity:  $O(n \log n)$

Sources:

[http://en.wikipedia.org/wiki/Merge\\_sort](http://en.wikipedia.org/wiki/Merge_sort)

<http://cslibrary.stanford.edu/105/LinkedListProblems.pdf>

## **Source**

<https://www.geeksforgeeks.org/merge-sort-for-linked-list/>

## Chapter 148

# Merge Sort for Linked Lists in JavaScript

Merge Sort for Linked Lists in JavaScript - GeeksforGeeks

Prerequisite:[Merge Sort for Linked Lists](#)

Merge sort is often preferred for sorting a linked list. The slow random-access performance of a linked list makes some other algorithms (such as quicksort) perform poorly, and others (such as heapsort) completely impossible.

In this post, Merge sort for linked list is implemented using JavaScript.

Examples:

```
Input : 5 -> 4 -> 3 -> 2 -> 1  
Output :1 -> 2 -> 3 -> 4 -> 5  
  
Input : 10 -> 20 -> 3 -> 2 -> 1  
Output : 1 -> 2 -> 3 -> 10 -> 20
```

```
<script>  
  
// Create Node of LinkedList  
function Node(data) {  
    this.node = data;  
    this.next = null;  
}  
  
// To initialize a linkedlist  
function LinkedList(list) {  
    this.head = list || null  
}  
  
// Function to insert The new Node into the linkedList
```

```
LinkedList.prototype.insert = function(data) {

    // Check if the linked list is empty
    // so insert first node and lead head
    // points to generic node
    if (this.head === null)
        this.head = new Node(data);

    else {

        // If linked list is not empty, insert the node
        // at the end of the linked list
        let list = this.head;
        while (list.next) {
            list = list.next;
        }

        // Now here list pointer points to last
        // node let's insert out new node in it
        list.next = new Node(data)
    }
}

// Function to print linkedList
LinkedList.prototype.iterate = function() {

    // First we will check whether our
    // linked list is empty or not
    if (this.head === null)
        return null;

    // If linked list is not empty we will
    // iterate from each Node and prints
    // its value stored in "data" property

    let list = this.head;

    // we will iterate until our list variable
    // contains the "Next" value of the last Node
    // i.e-> null
    while (list) {
        document.write(list.node)
        if (list.next)
            document.write(' -> ')
        list = list.next
    }
}
```

```
// Function to mergesort a linked list
LinkedList.prototype.mergeSort = function(list) {

    if (list.next === null)
        return list;

    let count = 0;
    let countList = list
    let leftPart = list;
    let leftPointer = list;
    let rightPart = null;
    let rightPointer = null;

    // Counting the nodes in the received linkedlist
    while (countList.next !== null) {
        count++;
        countList = countList.next;
    }

    // counting the mid of the linked list
    let mid = Math.floor(count / 2)
    let count2 = 0;

    // separating the left and right part with
    // respect to mid node in the linked list
    while (count2 < mid) {
        count2++;
        leftPointer = leftPointer.next;
    }

    rightPart = new LinkedList(leftPointer.next);
    leftPointer.next = null;

    // Here are two linked list which
    // contains the left most nodes and right
    // most nodes of the mid node
    return this._mergeSort(this.mergeSort(leftPart),
                           this.mergeSort(rightPart.head))
}

// Merging both lists in sorted manner
LinkedList.prototype._mergeSort = function(left, right) {

    // Create a new empty linked list
    let result = new LinkedList()

    let resultPointer = result.head;
    let pointerLeft = left;
```

```
let pointerRight = right;

// If true then add left most node value in result,
// increment left pointer else do the same in
// right linked list.
// This loop will be executed until pointer's of
// a left node or right node reached null
while (pointerLeft && pointerRight) {
    let tempNode = null;

    // Check if the right node's value is greater than
    // left node's value
    if (pointerLeft.node > pointerRight.node) {
        tempNode = pointerRight.node
        pointerRight = pointerRight.next;
    }
    else {
        tempNode = pointerLeft.node
        pointerLeft = pointerLeft.next;
    }

    if (result.head == null) {
        result.head = new Node(tempNode)
        resultPointer = result.head
    }
    else {
        resultPointer.next = new Node(tempNode)
        resultPointer = resultPointer.next
    }
}

// Add the remaining elements in the last of resultant
// linked list
resultPointer.next = pointerLeft;
while (resultPointer.next)
    resultPointer = resultPointer.next

resultPointer.next = pointerRight

// Result is the new sorted linked list
return result.head;
}

// Initialize the object
let l = new LinkedList();
l.insert(10)
l.insert(20)
```

```
l.insert(3)
l.insert(2)
l.insert(1)
// Print the linked list
l.iterate()

// Sort the linked list
l.head = LinkedList.prototype.mergeSort(l.head)

document.write('<br> After sorting : ');

// Print the sorted linked list
l.iterate()
</script>
```

### Output

```
10 -> 20 -> 3 -> 2 -> 1
After sorting : 1 -> 2 -> 3 -> 10 -> 20
```

### Source

<https://www.geeksforgeeks.org/merge-sort-linked-lists-javascript/>

## Chapter 149

# Merge a linked list into another linked list at alternate positions

Merge a linked list into another linked list at alternate positions - GeeksforGeeks

Given two linked lists, insert nodes of second list into first list at alternate positions of first list.

For example, if first list is 5->7->17->13->11 and second is 12->10->2->4->6, the first list should become 5->12->7->10->17->2->13->4->11->6 and second list should become empty. The nodes of second list should only be inserted when there are positions available. For example, if the first list is 1->2->3 and second list is 4->5->6->7->8, then first list should become 1->4->2->5->3->6 and second list to 7->8.

Use of extra space is not allowed (Not allowed to create additional nodes), i.e., insertion must be done in-place. Expected time complexity is O(n) where n is number of nodes in first list.

The idea is to run a loop while there are available positions in first loop and insert nodes of second list by changing pointers. Following are C and Java implementations of this approach.

C/C++

```
// C program to merge a linked list into another at
// alternate positions
#include <stdio.h>
#include <stdlib.h>

// A nexted list node
struct Node
{
    int data;
    struct Node *next;
};

// Function to print a linked list
void printList(struct Node *node)
{
    while (node != NULL)
    {
        printf("%d ", node->data);
        node = node->next;
    }
}
```

```
/* Function to insert a node at the beginning */
void push(struct Node ** head_ref, int new_data)
{
    struct Node* new_node =
        (struct Node*) malloc(sizeof(struct Node));
    new_node->data = new_data;
    new_node->next = (*head_ref);
    (*head_ref) = new_node;
}

/* Utility function to print a singly linked list */
void printList(struct Node *head)
{
    struct Node *temp = head;
    while (temp != NULL)
    {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}

// Main function that inserts nodes of linked list q into p at
// alternate positions. Since head of first list never changes
// and head of second list may change, we need single pointer
// for first list and double pointer for second list.
void merge(struct Node *p, struct Node **q)
{
    struct Node *p_curr = p, *q_curr = *q;
    struct Node *p_next, *q_next;

    // While there are available positions in p
    while (p_curr != NULL && q_curr != NULL)
    {
        // Save next pointers
        p_next = p_curr->next;
        q_next = q_curr->next;

        // Make q_curr as next of p_curr
        q_curr->next = p_next; // Change next pointer of q_curr
        p_curr->next = q_curr; // Change next pointer of p_curr

        // Update current pointers for next iteration
        p_curr = p_next;
        q_curr = q_next;
    }
}
```

```
*q = q_curr; // Update head pointer of second list
}

// Driver program to test above functions
int main()
{
    struct Node *p = NULL, *q = NULL;
    push(&p, 3);
    push(&p, 2);
    push(&p, 1);
    printf("First Linked List:\n");
    printList(p);

    push(&q, 8);
    push(&q, 7);
    push(&q, 6);
    push(&q, 5);
    push(&q, 4);
    printf("Second Linked List:\n");
    printList(q);

    merge(p, &q);

    printf("Modified First Linked List:\n");
    printList(p);

    printf("Modified Second Linked List:\n");
    printList(q);

    getchar();
    return 0;
}
```

### Java

```
// Java program to merge a linked list into another at
// alternate positions
class LinkedList
{
    Node head; // head of list

    /* Linked list Node*/
    class Node
    {
        int data;
        Node next;
        Node(int d) {data = d; next = null; }
    }
```

```
/* Inserts a new Node at front of the list. */
void push(int new_data)
{
    /* 1 & 2: Allocate the Node &
       Put in the data*/
    Node new_node = new Node(new_data);

    /* 3. Make next of new Node as head */
    new_node.next = head;

    /* 4. Move the head to point to new Node */
    head = new_node;
}

// Main function that inserts nodes of linked list q into p at
// alternate positions. Since head of first list never changes
// and head of second list/ may change, we need single pointer
// for first list and double pointer for second list.
void merge(LinkedList q)
{
    Node p_curr = head, q_curr = q.head;
    Node p_next, q_next;

    // While there are available positions in p;
    while (p_curr != null && q_curr != null) {

        // Save next pointers
        p_next = p_curr.next;
        q_next = q_curr.next;

        // make q_curr as next of p_curr
        q_curr.next = p_next; // change next pointer of q_curr
        p_curr.next = q_curr; // change next pointer of p_curr

        // update current pointers for next iteration
        p_curr = p_next;
        q_curr = q_next;
    }
    q.head = q_curr;
}

/* Function to print linked list */
void printList()
{
    Node temp = head;
    while (temp != null)
    {
```

```
        System.out.print(temp.data+" ");
        temp = temp.next;
    }
    System.out.println();
}

/* Drier program to test above functions */
public static void main(String args[])
{
    LinkedList llist1 = new LinkedList();
    LinkedList llist2 = new LinkedList();
    llist1.push(3);
    llist1.push(2);
    llist1.push(1);

    System.out.println("First Linked List:");
    llist1.printList();

    llist2.push(8);
    llist2.push(7);
    llist2.push(6);
    llist2.push(5);
    llist2.push(4);

    System.out.println("Second Linked List:");

    llist1.merge(llist2);

    System.out.println("Modified first linked list:");
    llist1.printList();

    System.out.println("Modified second linked list:");
    llist2.printList();
}
} /* This code is contributed by Rajat Mishra */
```

### Python

```
# Python program to merge a linked list into another at
# alternate positions
class LinkedList(object):
    def __init__(self):
        # head of list
        self.head = None

    # Linked list Node
    class Node(object):
        def __init__(self, d):
```

```
    self.data = d
    self.next = None

# Inserts a new Node at front of the list.
def push(self, new_data):

    # 1 & 2: Allocate the Node &
    # Put in the data
    new_node = self.Node(new_data)

    # 3. Make next of new Node as head
    new_node.next = self.head

    # 4. Move the head to point to new Node
    self.head = new_node

# Main function that inserts nodes of linked list q into p at
# alternate positions. Since head of first list never changes
# and head of second list/ may change, we need single pointer
# for first list and double pointer for second list.
def merge(self, q):
    p_curr = self.head
    q_curr = q.head

    # While there are available positions in p;
    while p_curr != None and q_curr != None:

        # Save next pointers
        p_next = p_curr.next
        q_next = q_curr.next

        # make q_curr as next of p_curr
        q_curr.next = p_next # change next pointer of q_curr
        p_curr.next = q_curr # change next pointer of p_curr

        # update current pointers for next iteration
        p_curr = p_next
        q_curr = q_next
    q.head = q_curr

# Function to print linked list
def printList(self):
    temp = self.head
    while temp != None:
        print str(temp.data),
        temp = temp.next
    print ''
```

```
# Driver program to test above functions
llist1 = LinkedList()
llist2 = LinkedList()
llist1.push(3)
llist1.push(2)
llist1.push(1)

print "First Linked List:"
llist1.printList()

llist2.push(8)
llist2.push(7)
llist2.push(6)
llist2.push(5)
llist2.push(4)

print "Second Linked List:"

llist2.printList()
llist1.merge(llist2)

print "Modified first linked list:"
llist1.printList()

print "Modified second linked list:"
llist2.printList()

# This code is contributed by BHAVYA JAIN
```

Output:

```
First Linked List:
1 2 3
Second Linked List:
4 5 6 7 8
Modified First Linked List:
1 4 2 5 3 6
Modified Second Linked List:
7 8
```

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/merge-a-linked-list-into-another-linked-list-at-alternate-positions/>

## Chapter 150

# Merge first half and reversed second half of the linked list alternatively

Merge first half and reversed second half of the linked list alternatively - GeeksforGeeks

Given a linked list, the task is to rearrange the linked list in the following manner:

1. Reverse the second half of given linked list.
2.
  - First element of the linked list is the first element of first half.
  - Second element of the linked list is the first element of second half.

### Examples:

Input: 1->2->3->4->5

Output: 1->5->2->4->3

Input: 1->2->3->4->5->6

Output: 1->6->2->5->3->4

**Approach:** Initially find the mid node of the linked list. The approach has been discussed [here](#). Reverse the linked list from mid to end. Once the linked list is reversed, traverse from the start and insert a node from the first half of the list and another node from the back half of the linked list simultaneously. Continue this process until the middle node is reached. Once the middle node is reached, point the node just before the middle node to NULL.

Below is the implementation of the above approach:

C++

```
// C++ program to sandwich the last part of
// linked list in between the first part of
// the linked list
#include<bits/stdc++.h>
#include <stdio.h>
using namespace std;

struct node {
    int data;
    struct node* next;
};

// Function to reverse Linked List
struct node* reverseLL(struct node* root)
{
    struct node *prev = NULL, *current = root, *next;
    while (current) {
        next = current->next;
        current->next = prev;
        prev = current;
        current = next;
    }

    // root needs to be updated after reversing
    root = prev;
    return root;
}

// Function to modify Linked List
void modifyLL(struct node* root)
{
    // Find the mid node
    struct node *slow_ptr = root, *fast_ptr = root;
    while (fast_ptr && fast_ptr->next) {
        fast_ptr = fast_ptr->next->next;
        slow_ptr = slow_ptr->next;
    }

    // Reverse the second half of the list
    struct node* root2 = reverseLL(slow_ptr->next);

    // partition the list
    slow_ptr->next = NULL;

    struct node *current1 = root, *current2 = root2;

    // insert the elements in between
    while (current1 && current2) {
```

```
// next node to be traversed in the first list
struct node* dnext1 = current1->next;

// next node to be traversed in the first list
struct node* dnext2 = current2->next;
current1->next = current2;
current2->next = dnext1;
current1 = dnext1;
current2 = dnext2;
}

}

// Function to insert node after the end
void insertNode(struct node** start, int val)
{

    // allocate memory
    struct node* temp = (struct node*)malloc(sizeof(struct node));
    temp->data = val;

    // if first node
    if (*start == NULL)
        *start = temp;
    else {

        // move to the end pointer of node
        struct node* dstart = *start;
        while (dstart->next != NULL)
            dstart = dstart->next;
        dstart->next = temp;
    }
}

// function to print the linked list
void display(struct node** start)
{
    struct node* temp = *start;

    // traverse till the entire linked
    // list is printed
    while (temp->next != NULL) {
        printf("%d->", temp->data);
        temp = temp->next;
    }
    printf("%d\n", temp->data);
}
```

```
// Driver Code
int main()
{
    // Odd Length Linked List
    struct node* start = NULL;
    insertNode(&start, 1);
    insertNode(&start, 2);
    insertNode(&start, 3);
    insertNode(&start, 4);
    insertNode(&start, 5);

    printf("Before Modifying: ");
    display(&start);
    modifyLL(start);
    printf("After Modifying: ");
    display(&start);

    // Even Length Linked List
    start = NULL;
    insertNode(&start, 1);
    insertNode(&start, 2);
    insertNode(&start, 3);
    insertNode(&start, 4);
    insertNode(&start, 5);
    insertNode(&start, 6);

    printf("\nBefore Modifying: ");
    display(&start);
    modifyLL(start);
    printf("After Modifying: ");
    display(&start);

    return 0;
}
```

### Java

```
// Java program to sandwich the
// last part of linked list in
// between the first part of
// the linked list
import java.util.*;

class node
{
    int data;
    node next;
    node(int key)
```

```
{  
    data = key;  
    next = null;  
}  
}  
  
class GFG  
{  
  
    // Function to reverse  
    // Linked List  
    public static node reverseLL(node root)  
{  
  
        node prev = null,  
        current = root, next;  
        while (current!= null)  
        {  
            next = current.next;  
            current.next = prev;  
            prev = current;  
            current = next;  
        }  
  
        // root needs to be  
        // upated after reversing  
        root = prev;  
        return root;  
    }  
  
    // Function to modify  
    // Linked List  
    public static void modifyLL( node root)  
{  
        // Find the mid node  
        node slow_ptr = root, fast_ptr = root;  
        while (fast_ptr != null &&  
              fast_ptr.next != null)  
        {  
            fast_ptr = fast_ptr.next.next;  
            slow_ptr = slow_ptr.next;  
        }  
  
        // Reverse the second  
        // half of the list  
        node root2 = reverseLL(slow_ptr.next);  
  
        // partition the list
```

```
slow_ptr.next = null;

node current1 = root,
    current2 = root2;

// insert the elements in between
while (current1 != null &&
       current2 != null)
{
    // next node to be traversed
    // in the first list
    node dnext1 = current1.next;

    // next node to be traversed
    // in the first list
    node dnext2 = current2.next;
    current1.next = current2;
    current2.next = dnext1;
    current1 = dnext1;
    current2 = dnext2;
}
}

// Function to insert
// node after the end
public static node insertNode(node start,
                               int val)
{
    // allocate memory
    node temp = new node(val);

    // if first node
    if (start == null)
        start = temp;
    else
    {

        // move to the end
        // pointer of node
        node dstart = start;
        while (dstart.next != null)
            dstart = dstart.next;
        dstart.next = temp;
    }

    return start;
}
```

```
}  
  
// function to print  
// the linked list  
public static void display(node start)  
{  
    node temp = start;  
  
    // traverse till the  
    // entire linked  
    // list is printed  
    while (temp.next != null) {  
        System.out.print(temp.data + "->");  
        temp = temp.next;  
    }  
    System.out.println(temp.data);  
}  
  
// Driver Code  
public static void main(String args[])  
{  
    // Odd Length Linked List  
    node start = null;  
    start = insertNode(start, 1);  
    start = insertNode(start, 2);  
    start = insertNode(start, 3);  
    start = insertNode(start, 4);  
    start = insertNode(start, 5);  
  
    System.out.print("Before Modifying: ");  
    display(start);  
    modifyLL(start);  
    System.out.print("After Modifying: ");  
    display(start);  
  
    // Even Length Linked List  
    start = null;  
    start = insertNode(start, 1);  
    start = insertNode(start, 2);  
    start = insertNode(start, 3);  
    start = insertNode(start, 4);  
    start = insertNode(start, 5);  
    start = insertNode(start, 6);  
  
    System.out.print("Before Modifying: ");  
    display(start);  
    modifyLL(start);
```

```
        System.out.print("After Modifying: ");
        display(start);
    }
}
```

**Output:**

```
Before Modifying: 1->2->3->4->5
After Modifying: 1->5->2->4->3
```

```
Before Modifying: 1->2->3->4->5->6
After Modifying: 1->6->2->5->3->4
```

**Time Complexity:** O(N)

**Exercise:** Solve the question without reversing the Linked List from the middle node.

**Source**

<https://www.geeksforgeeks.org/merge-first-half-and-reversed-second-half-of-the-linked-list-alternatively/>

## Chapter 151

# Merge k sorted linked lists | Set 2 (Using Min Heap)

Merge k sorted linked lists | Set 2 (Using Min Heap) - GeeksforGeeks

Given **k** sorted linked lists each of size **n**, merge them and print the sorted output.

Examples:

Input: **k** = 3, **n** = 4  
list1 = 1->3->5->7->NULL  
list2 = 2->4->6->8->NULL  
list3 = 0->9->10->11

Output:  
0->1->2->3->4->5->6->7->8->9->10->11

Source: [Merge K sorted Linked Lists | Method 2](#)

Approach: An efficient solution for the problem has been dicussed in **Method 3** of [this](#) post. Here another solution has been provided which the uses the **MIN HEAP** data structure. This solution is based on the min heap approach used to solve the problem ‘merge k sorted arrays’ which is discussed [here](#).

```
// C++ implementation to merge k sorted linked lists
// | Using MIN HEAP method
#include <bits/stdc++.h>
using namespace std;

struct Node {
    int data;
    struct Node* next;
}
```

```
};

// 'compare' function used to build up the
// priority queue
struct compare {
    bool operator()(struct Node* a, struct Node* b)
    {
        return a->data > b->data;
    }
};

// function to merge k sorted linked lists
struct Node* mergeKSortedLists(struct Node* arr[], int k)
{
    struct Node* head = NULL, *last;

    // priority_queue 'pq' implemented as min heap with the
    // help of 'compare' function
    priority_queue<Node*, vector<Node*>, compare> pq;

    // push the head nodes of all the k lists in 'pq'
    for (int i = 0; i < k; i++)
        pq.push(arr[i]);

    // loop till 'pq' is not empty
    while (!pq.empty()) {

        // get the top element of 'pq'
        struct Node* top = pq.top();
        pq.pop();

        // check if there is a node next to the 'top' node
        // in the list of which 'top' node is a member
        if (top->next != NULL)
            // push the next node in 'pq'
            pq.push(top->next);

        // if final merged list is empty
        if (head == NULL) {
            head = top;

            // points to the last node so far of
            // the final merged list
            last = top;
        }

        else {
            // insert 'top' at the end of the merged list so far
```

```
last->next = top;

        // update the 'last' pointer
        last = top;
    }
}

// head node of the required merged list
return head;
}

// function to print the singly linked list
void printList(struct Node* head)
{
    while (head != NULL) {
        cout << head->data << " ";
        head = head->next;
    }
}

// Utility function to create a new node
struct Node* newNode(int data)
{
    // allocate node
    struct Node* new_node = new Node();

    // put in the data
    new_node->data = data;
    new_node->next = NULL;

    return new_node;
}

// Driver program to test above
int main()
{
    int k = 3; // Number of linked lists
    int n = 4; // Number of elements in each list

    // an array of pointers storing the head nodes
    // of the linked lists
    Node* arr[k];

    // creating k = 3 sorted lists
    arr[0] = newNode(1);
    arr[0]->next = newNode(3);
    arr[0]->next->next = newNode(5);
    arr[0]->next->next->next = newNode(7);
```

```
arr[1] = newNode(2);
arr[1]->next = newNode(4);
arr[1]->next->next = newNode(6);
arr[1]->next->next->next = newNode(8);

arr[2] = newNode(0);
arr[2]->next = newNode(9);
arr[2]->next->next = newNode(10);
arr[2]->next->next->next = newNode(11);

// merge the k sorted lists
struct Node* head = mergeKSortedLists(arr, k);

// print the merged list
printList(head);

return 0;
}
```

Output:

0 1 2 3 4 5 6 7 8 9 10 11

Time Complexity:  $O(nk \log k)$   
Auxiliary Space:  $O(k)$

## Source

<https://www.geeksforgeeks.org/merge-k-sorted-linked-lists-set-2-using-min-heap/>

## Chapter 152

# Merge two sorted linked list without duplicates

Merge two sorted linked list without duplicates - GeeksforGeeks

Merge two sorted linked list of size **n1** and **n2**. The duplicates in two linked list should be present only once in the final sorted linked list.

Examples:

```
Input : list1: 1->1->4->5->7  
        list2: 2->4->7->9  
Output : 1 2 4 5 7 9
```

**Source:** Microsoft on Campus Placement and Interview Questions

**Approach:** Following are the steps:

1. Merge the two sorted linked list in sorted manner. Refer recursive approach of [this post](#). Let the final obtained list be **head**.
2. Remove duplicates from sorted linked list **head**.

```
// C++ implementation to merge two sorted linked list  
// without duplicates  
#include <bits/stdc++.h>  
  
using namespace std;  
  
// structure of a node  
struct Node {  
    int data;  
    Node* next;
```

```
};

// function to get a new node
Node* getNode(int data)
{
    // allocate space
    Node* temp = (Node*)malloc(sizeof(Node));

    // put in data
    temp->data = data;
    temp->next = NULL;
    return temp;
}

// function to merge two sorted linked list
// in a sorted manner
Node* sortedMerge(struct Node* a, struct Node* b)
{
    Node* result = NULL;

    /* Base cases */
    if (a == NULL)
        return (b);
    else if (b == NULL)
        return (a);

    /* Pick either a or b, and recur */
    if (a->data <= b->data) {
        result = a;
        result->next = sortedMerge(a->next, b);
    }
    else {
        result = b;
        result->next = sortedMerge(a, b->next);
    }
    return (result);
}

/* The function removes duplicates from a sorted list */
void removeDuplicates(Node* head)
{
    /* Pointer to traverse the linked list */
    Node* current = head;

    /* Pointer to store the next pointer of a node to be deleted*/
    Node* next_next;

    /* do nothing if the list is empty */
}
```

```
if (current == NULL)
    return;

/* Traverse the list till last node */
while (current->next != NULL) {

    /* Compare current node with next node */
    if (current->data == current->next->data) {

        /* The sequence of steps is important*/
        next_next = current->next->next;
        free(current->next);
        current->next = next_next;
    }
    else /* This is tricky: only advance if no deletion */
    {
        current = current->next;
    }
}

// function to merge two sorted linked list
// without duplicates
Node* sortedMergeWithoutDuplicates(Node* head1, Node* head2)
{
    // merge two linked list in sorted manner
    Node* head = sortedMerge(head1, head2);

    // remove duplicates from the list 'head'
    removeDuplicates(head);

    return head;
}

// function to print the linked list
void printList(Node* head)
{
    while (head != NULL) {
        cout << head->data << " ";
        head = head->next;
    }
}

// Driver program to test above
int main()
{
    // head1: 1->1->4->5->7
    Node* head1 = getNode(1);
```

```
head1->next = getNode(1);
head1->next->next = getNode(4);
head1->next->next->next = getNode(5);
head1->next->next->next->next = getNode(7);

// head2: 2->4->7->9
Node* head2 = getNode(2);
head2->next = getNode(4);
head2->next->next = getNode(7);
head2->next->next->next = getNode(9);

Node* head3;

head3 = sortedMergeWithoutDuplicates(head1, head2);

printList(head3);

return 0;
}
```

Output:

1 2 4 5 7 9

Time complexity:  $O(n_1 + n_2)$ .

Auxiliary Space:  $O(1)$ .

**Exercise:** Get the final sorted linked list without duplicates in a single traversal of the two lists.

## Source

<https://www.geeksforgeeks.org/merge-two-sorted-linked-list-without-duplicates/>

## Chapter 153

# Merge two sorted linked lists

Merge two sorted linked lists - GeeksforGeeks

Write a SortedMerge() function that takes two lists, each of which is sorted in increasing order, and merges the two together into one list which is in increasing order. SortedMerge() should return the new list. The new list should be made by splicing together the nodes of the first two lists.

For example if the first linked list a is 5->10->15 and the other linked list b is 2->3->20, then SortedMerge() should return a pointer to the head node of the merged list 2->3->5->10->15->20.

There are many cases to deal with: either ‘a’ or ‘b’ may be empty, during processing either ‘a’ or ‘b’ may run out first, and finally there’s the problem of starting the result list empty, and building it up while going through ‘a’ and ‘b’.

### Method 1 (Using Dummy Nodes)

The strategy here uses a temporary dummy node as the start of the result list. The pointer Tail always points to the last node in the result list, so appending new nodes is easy. The dummy node gives tail something to point to initially when the result list is empty. This dummy node is efficient, since it is only temporary, and it is allocated in the stack. The loop proceeds, removing one node from either ‘a’ or ‘b’, and adding it to tail. When we are done, the result is in dummy.next.

```
/* C/C++ program to merge two sorted linked lists */
#include<stdio.h>
#include<stdlib.h>
#include<assert.h>

/* Link list node */
struct Node
{
    int data;
    struct Node* next;
};

/* Function to merge two sorted linked lists */
struct Node* SortedMerge(struct Node* a, struct Node* b)
{
    struct Node* dummy = (struct Node*) malloc(sizeof(struct Node));
    struct Node* tail = dummy;
    while(a != NULL && b != NULL)
    {
        if(a->data < b->data)
        {
            tail->next = a;
            a = a->next;
        }
        else
        {
            tail->next = b;
            b = b->next;
        }
        tail = tail->next;
    }
    if(a == NULL)
        tail->next = b;
    else
        tail->next = a;
    return dummy->next;
}
```

```
/* pull off the front node of the source and put it in dest */
void MoveNode(struct Node** destRef, struct Node** sourceRef);

/* Takes two lists sorted in increasing order, and splices
   their nodes together to make one big sorted list which
   is returned. */
struct Node* SortedMerge(struct Node* a, struct Node* b)
{
    /* a dummy first node to hang the result on */
    struct Node dummy;

    /* tail points to the last result node */
    struct Node* tail = &dummy;

    /* so tail->next is the place to add new nodes
       to the result. */
    dummy.next = NULL;
    while (1)
    {
        if (a == NULL)
        {
            /* if either list runs out, use the
               other list */
            tail->next = b;
            break;
        }
        else if (b == NULL)
        {
            tail->next = a;
            break;
        }
        if (a->data <= b->data)
            MoveNode(&(tail->next), &a);
        else
            MoveNode(&(tail->next), &b);

        tail = tail->next;
    }
    return(dummy.next);
}

/* UTILITY FUNCTIONS */
/* MoveNode() function takes the node from the front of the
   source, and move it to the front of the dest.
   It is an error to call this with the source list empty.
```

Before calling MoveNode():

```
source == {1, 2, 3}
dest == {1, 2, 3}

Afffter calling MoveNode():
source == {2, 3}
dest == {1, 1, 2, 3} */

void MoveNode(struct Node** destRef, struct Node** sourceRef)
{
    /* the front source node */
    struct Node* newNode = *sourceRef;
    assert(newNode != NULL);

    /* Advance the source pointer */
    *sourceRef = newNode->next;

    /* Link the old dest off the new node */
    newNode->next = *destRef;

    /* Move dest to point to the new node */
    *destRef = newNode;
}

/* Function to insert a node at the begining of the
   linked list */
void push(struct Node** head_ref, int new_data)
{
    /* allocate node */
    struct Node* new_node =
        (struct Node*) malloc(sizeof(struct Node));

    /* put in the data */
    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;
}

/* Function to print nodes in a given linked list */
void printList(struct Node *node)
{
    while (node!=NULL)
    {
        printf("%d ", node->data);
        node = node->next;
    }
}
```

```
    }
}

/* Drier program to test above functions*/
int main()
{
    /* Start with the empty list */
    struct Node* res = NULL;
    struct Node* a = NULL;
    struct Node* b = NULL;

    /* Let us create two sorted linked lists to test
       the functions
       Created lists, a: 5->10->15,  b: 2->3->20 */
    push(&a, 15);
    push(&a, 10);
    push(&a, 5);

    push(&b, 20);
    push(&b, 3);
    push(&b, 2);

    /* Remove duplicates from linked list */
    res = SortedMerge(a, b);

    printf("Merged Linked List is: \n");
    printList(res);

    return 0;
}
```

Output :

```
Merged Linked List is:
2 3 5 10 15 20
```

### Method 2 (Using Local References)

This solution is structurally very similar to the above, but it avoids using a dummy node. Instead, it maintains a `struct node**` pointer, `lastPtrRef`, that always points to the last pointer of the result list. This solves the same case that the dummy node did — dealing with the result list when it is empty. If you are trying to build up a list at its tail, either the dummy node or the `struct node**` “reference” strategy can be used (see Section 1 for details).

```
struct Node* SortedMerge(struct Node* a, struct Node* b)
{
```

```
struct Node* result = NULL;

/* point to the last result pointer */
struct Node** lastPtrRef = &result;

while(1)
{
    if (a == NULL)
    {
        *lastPtrRef = b;
        break;
    }
    else if (b==NULL)
    {
        *lastPtrRef = a;
        break;
    }
    if(a->data <= b->data)
    {
        MoveNode(lastPtrRef, &a);
    }
    else
    {
        MoveNode(lastPtrRef, &b);
    }

    /* tricky: advance to point to the next ".next" field */
    lastPtrRef = &((*lastPtrRef)->next);
}
return(result);
}
```

### Method 3 (Using Recursion)

Merge is one of those nice recursive problems where the recursive solution code is much cleaner than the iterative code. You probably wouldn't want to use the recursive version for production code however, because it will use stack space which is proportional to the length of the lists.

### C/C++

```
struct Node* SortedMerge(struct Node* a, struct Node* b)
{
    struct Node* result = NULL;

    /* Base cases */
    if (a == NULL)
        return(b);
    if (b == NULL)
        return(a);

    if (a->data < b->data)
    {
        result = a;
        result->next = SortedMerge(a->next, b);
    }
    else
    {
        result = b;
        result->next = SortedMerge(a, b->next);
    }

    return(result);
}
```

```
else if (b==NULL)
    return(a);

/* Pick either a or b, and recur */
if (a->data <= b->data)
{
    result = a;
    result->next = SortedMerge(a->next, b);
}
else
{
    result = b;
    result->next = SortedMerge(a, b->next);
}
return(result);
}
```

### Python3

```
# Python3 program merge two sorted linked
# in third linked list using recursive.

# Node class
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

# Constructor to initialize the node object
class LinkedList:

    # Function to initialize head
    def __init__(self):
        self.head = None

    # Method to print linked list
    def printList(self):
        temp = self.head

        while temp :
            print(temp.data, end="->")
            temp = temp.next

    # Function to add of node at the end.
    def append(self, new_data):
        new_node = Node(new_data)
```

```
if self.head is None:
    self.head = new_node
    return
last = self.head

while last.next:
    last = last.next
last.next = new_node

# Function to merge two sorted linked list.
def mergeLists(head1, head2):

    # create a temp node NULL
    temp = None

    # List1 is empty then return List2
    if head1 is None:
        return head2

    # if List2 is empty then return List1
    if head2 is None:
        return head1

    # If List1's data is smaller or
    # equal to List2's data
    if head1.data <= head2.data:

        # assign temp to List1's data
        temp = head1

        # Again check List1's data is smaller or equal List2's
        # data and call mergeLists function.
        temp.next = mergeLists(head1.next, head2)

    else:
        # If List2's data is greater than or equal List1's
        # data assign temp to head2
        temp = head2

        # Again check List2's data is greater or equal List1's
        # data and call mergeLists function.
        temp.next = mergeLists(head1, head2.next)

    # return the temp list.
    return temp
```

```
# Driver Function
if __name__ == '__main__':
    # Create linked list :
    # 10->20->30->40->50
    list1 = LinkedList()
    list1.append(10)
    list1.append(20)
    list1.append(30)
    list1.append(40)
    list1.append(50)

    # Create linked list 2 :
    # 5->15->18->35->60
    list2 = LinkedList()
    list2.append(5)
    list2.append(15)
    list2.append(18)
    list2.append(35)
    list2.append(60)

    # Create linked list 3
    list3 = LinkedList()

    # Merging linked list 1 and linked list 2
    # in linked list 3
    list3.head = mergeLists(list1.head, list2.head)

    print(" Merged Linked List is : ", end="")
    list3.printList()

# This code is contributed by 'Shriaknt13'.
```

Please refer below post for simpler implementations :

**Merge two sorted lists (in-place)**

Source: <http://cslibrary.stanford.edu/105/LinkedListProblems.pdf>

## Source

<https://www.geeksforgeeks.org/merge-two-sorted-linked-lists/>

## Chapter 154

# Merge two sorted linked lists such that merged list is in reverse order

Merge two sorted linked lists such that merged list is in reverse order - GeeksforGeeks

Given two linked lists sorted in increasing order. Merge them such a way that the result list is in decreasing order (reverse order).

Examples:

```
Input: a: 5->10->15->40
      b: 2->3->20
Output: res: 40->20->15->10->5->3->2
```

```
Input: a: NULL
      b: 2->3->20
Output: res: 20->3->2
```

A Simple Solution is to do following.

- 1) Reverse first list 'a'.
- 2) Reverse second list 'b'.
- 3) Merge two reversed lists.

Another Simple Solution is first Merge both lists, then reverse the merged list.

Both of the above solutions require two traversals of linked list.

**How to solve without reverse, O(1) auxiliary space (in-place) and only one traversal of both lists?**

The idea is to follow merge style process. Initialize result list as empty. Traverse both lists from beginning to end. Compare current nodes of both lists and insert smaller of two at the beginning of the result list.

- 1) Initialize result list as empty: res = NULL.
- 2) Let 'a' and 'b' be heads first and second lists respectively.
- 3) While (a != NULL and b != NULL)
  - a) Find the smaller of two (Current 'a' and 'b')
  - b) Insert the smaller value node at the front of result.
  - c) Move ahead in the list of smaller node.
- 4) If 'b' becomes NULL before 'a', insert all nodes of 'a' into result list at the beginning.
- 5) If 'a' becomes NULL before 'b', insert all nodes of 'a' into result list at the beginning.

Below is the implementation of above solution.

#### C/C++

```
/* Given two sorted non-empty linked lists. Merge them in
   such a way that the result list will be in reverse
   order. Reversing of linked list is not allowed. Also,
   extra space should be O(1) */
#include<iostream>
using namespace std;

/* Link list Node */
struct Node
{
    int key;
    struct Node* next;
};

// Given two non-empty linked lists 'a' and 'b'
Node* SortedMerge(Node *a, Node *b)
{
    // If both lists are empty
    if (a==NULL && b==NULL) return NULL;

    // Initialize head of resultant list
    Node *res = NULL;

    // Traverse both lists while both of them
    // have nodes.
    while (a != NULL && b != NULL)
    {
        // If a's current value is smaller or equal to
        // b's current value.
        if (a->key <= b->key)
        {
            // Store next of current Node in first list
            a->next = b;
            res = a;
            a = a->next;
        }
        else
        {
            b->next = a;
            res = b;
            b = b->next;
        }
    }

    // If one of them has more elements
    if (a == NULL)
        return res;
    else
        return b;
}
```

```
Node *temp = a->next;

// Add 'a' at the front of resultant list
a->next = res;
res = a;

// Move ahead in first list
a = temp;
}

// If a's value is greater. Below steps are similar
// to above (Only 'a' is replaced with 'b')
else
{
    Node *temp = b->next;
    b->next = res;
    res = b;
    b = temp;
}
}

// If second list reached end, but first list has
// nodes. Add remaining nodes of first list at the
// front of result list
while (a != NULL)
{
    Node *temp = a->next;
    a->next = res;
    res = a;
    a = temp;
}

// If first list reached end, but second list has
// node. Add remaining nodes of first list at the
// front of result list
while (b != NULL)
{
    Node *temp = b->next;
    b->next = res;
    res = b;
    b = temp;
}

return res;
}

/* Function to print Nodes in a given linked list */
void printList(struct Node *Node)
```

```
{  
    while (Node!=NULL)  
    {  
        cout << Node->key << " ";  
        Node = Node->next;  
    }  
}  
  
/* Utility function to create a new node with  
   given key */  
Node *newNode(int key)  
{  
    Node *temp = new Node;  
    temp->key = key;  
    temp->next = NULL;  
    return temp;  
}  
  
/* Drier program to test above functions*/  
int main()  
{  
    /* Start with the empty list */  
    struct Node* res = NULL;  
  
    /* Let us create two sorted linked lists to test  
       the above functions. Created lists shall be  
       a: 5->10->15  
       b: 2->3->20 */  
    Node *a = newNode(5);  
    a->next = newNode(10);  
    a->next->next = newNode(15);  
  
    Node *b = newNode(2);  
    b->next = newNode(3);  
    b->next->next = newNode(20);  
  
    cout << "List A before merge: \n";  
    printList(a);  
  
    cout << "\nList B before merge: \n";  
    printList(b);  
  
    /* merge 2 increasing order LLs in descresing order */  
    res = SortedMerge(a, b);  
  
    cout << "\nMerged Linked List is: \n";  
    printList(res);
```

```
    return 0;
}
```

**Java**

```
// Java program to merge two sorted linked list such that merged
// list is in reverse order

// Linked List Class
class LinkedList {

    Node head; // head of list
    static Node a, b;

    /* Node Class */
    static class Node {

        int data;
        Node next;

        // Constructor to create a new node
        Node(int d) {
            data = d;
            next = null;
        }
    }

    void printlist(Node node) {
        while (node != null) {
            System.out.print(node.data + " ");
            node = node.next;
        }
    }

    Node sortedmerge(Node node1, Node node2) {

        // if both the nodes are null
        if (node1 == null && node2 == null) {
            return null;
        }

        // resultant node
        Node res = null;

        // if both of them have nodes present traverse them
        while (node1 != null && node2 != null) {

            // Now compare both nodes current data

```

```
        if (node1.data <= node2.data) {
            Node temp = node1.next;
            node1.next = res;
            res = node1;
            node1 = temp;
        } else {
            Node temp = node2.next;
            node2.next = res;
            res = node2;
            node2 = temp;
        }
    }

    // If second list reached end, but first list has
    // nodes. Add remaining nodes of first list at the
    // front of result list
    while (node1 != null) {
        Node temp = node1.next;
        node1.next = res;
        res = node1;
        node1 = temp;
    }

    // If first list reached end, but second list has
    // node. Add remaining nodes of first list at the
    // front of result list
    while (node2 != null) {
        Node temp = node2.next;
        node2.next = res;
        res = node2;
        node2 = temp;
    }
}

return res;
}

public static void main(String[] args) {

    LinkedList list = new LinkedList();
    Node result = null;

    /*Let us create two sorted linked lists to test
     *the above functions. Created lists shall be
     *a: 5->10->15
     *b: 2->3->20*/
    list.a = new Node(5);
    list.a.next = new Node(10);
```

```
list.a.next.next = new Node(15);

list.b = new Node(2);
list.b.next = new Node(3);
list.b.next.next = new Node(20);

System.out.println("List a before merge :");
list.printlist(a);
System.out.println("");
System.out.println("List b before merge :");
list.printlist(b);

// merge two sorted linkedlist in decreasing order
result = list.sortedmerge(a, b);
System.out.println("");
System.out.println("Merged linked list : ");
list.printlist(result);

}

}

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

Output:

```
List A before merge:
5 10 15
List B before merge:
2 3 20
Merged Linked List is:
20 15 10 5 3 2
```

This solution traverses both lists only once, doesn't require reverse and works in-place.

This article is contributed by **Mohammed Rafeeb**. 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/merge-two-sorted-linked-lists-such-that-merged-list-is-in-reverse-order/>

## Chapter 155

# Merge two sorted lists (in-place)

Merge two sorted lists (in-place) - GeeksforGeeks

Given two sorted lists, merge them so as to produce a combined sorted list (without using extra space).

Examples:

```
Input : head1: 5->7->9  
        head2: 4->6->8  
Output : 4->5->6->7->8->9
```

```
Input : head1: 1->3->5->7  
        head2: 2->4  
Output : 1->2->3->4->5->7
```

We have discussed different solutions in below post.

[Merge two sorted linked lists](#)

In this post, a new simpler solutions are discussed. The idea is to one by one compare nodes and form the result list.

Method 1 (Recursive)

```
// C program to merge two sorted linked lists  
// in-place.  
#include<bits/stdc++.h>  
using namespace std;  
  
struct Node  
{
```

```
int data;
struct Node *next;
};

// Function to create newNode in a linkedlist
Node *newNode(int key)
{
    struct Node *temp = new Node;
    temp->data = key;
    temp->next = NULL;
    return temp;
}

// A utility function to print linked list
void printList(Node *node)
{
    while (node != NULL)
    {
        printf("%d ", node->data);
        node = node->next;
    }
}

// Merges two given lists in-place. This function
// mainly compares head nodes and calls mergeUtil()
Node *merge(Node *h1, Node *h2)
{
    if (!h1)
        return h2;
    if (!h2)
        return h1;

    // start with the linked list
    // whose head data is the least
    if (h1->data < h2->data)
    {
        h1->next = merge(h1->next, h2);
        return h1;
    }
    else
    {
        h2->next = merge(h1, h2->next);
        return h2;
    }
}

// Driver program
int main()
```

```
{  
    Node *head1 = newNode(1);  
    head1->next = newNode(3);  
    head1->next->next = newNode(5);  
  
    // 1->3->5 LinkedList created  
  
    Node *head2 = newNode(0);  
    head2->next = newNode(2);  
    head2->next->next = newNode(4);  
  
    // 0->2->4 LinkedList created  
  
    Node *mergedhead = merge(head1, head2);  
  
    printList(mergedhead);  
    return 0;  
}
```

Output:

```
0 1 2 3 4 5
```

Method 2 (Iterative)

```
// C program to merge two sorted linked lists  
// in-place.  
#include<bits/stdc++.h>  
using namespace std;  
  
struct Node  
{  
    int data;  
    struct Node *next;  
};  
  
// Function to create newNode in a linkedlist  
struct Node *newNode(int key)  
{  
    struct Node *temp = new Node;  
    temp->data = key;  
    temp->next = NULL;  
    return temp;
```

```
}  
  
// A utility function to print linked list  
void printList(struct Node *node)  
{  
    while (node != NULL)  
    {  
        printf("%d ", node->data);  
        node = node->next;  
    }  
}  
  
// Merges two lists with headers as h1 and h2.  
// It assumes that h1's data is smaller than  
// or equal to h2's data.  
struct Node *mergeUtil(struct Node *h1,  
                      struct Node *h2)  
{  
    // if only one node in first list  
    // simply point its head to second list  
    if (!h1->next)  
    {  
        h1->next = h2;  
        return h1;  
    }  
  
    // Initialize current and next pointers of  
    // both lists  
    struct Node *curr1 = h1, *next1 = h1->next;  
    struct Node *curr2 = h2, *next2 = h2->next;  
  
    while (next1 && next2)  
    {  
        // if curr2 lies in between curr1 and next1  
        // then do curr1->curr2->next1  
        if ((curr2->data) > (curr1->data) &&  
            (curr2->data) < (next1->data))  
        {  
            next2 = curr2->next;  
            curr1->next = curr2;  
            curr2->next = next1;  
  
            // now let curr1 and curr2 to point  
            // to their immediate next pointers  
            curr1 = curr2;  
            curr2 = next2;  
        }  
        else
```

```
{  
    // if more nodes in first list  
    if (next1->next)  
    {  
        next1 = next1->next;  
        curr1 = curr1->next;  
    }  
  
    // else point the last node of first list  
    // to the remaining nodes of second list  
    else  
    {  
        next1->next = curr2;  
        return h1;  
    }  
}  
}  
return h1;  
}  
  
// Merges two given lists in-place. This function  
// mainly compares head nodes and calls mergeUtil()  
struct Node *merge(struct Node *h1,  
                   struct Node *h2)  
{  
    if (!h1)  
        return h2;  
    if (!h2)  
        return h1;  
  
    // start with the linked list  
    // whose head data is the least  
    if (h1->data < h2->data)  
        return mergeUtil(h1, h2);  
    else  
        return mergeUtil(h2, h1);  
}  
  
// Driver program  
int main()  
{  
    struct Node *head1 = newNode(1);  
    head1->next = newNode(3);  
    head1->next->next = newNode(5);  
  
    // 1->3->5 LinkedList created  
  
    struct Node *head2 = newNode(0);
```

```
head2->next = newNode(2);
head2->next->next = newNode(4);

// 0->2->4 LinkedList created

struct Node *mergedhead =
merge(head1, head2);

printList(mergedhead);
return 0;
}
```

Output:

0 1 2 3 4 5

## Source

<https://www.geeksforgeeks.org/merge-two-sorted-lists-place/>

## Chapter 156

# Modify and Rearrange List

Modify and Rearrange List - GeeksforGeeks

Given a singly linked list, with some positive numbers (valid numbers) and zeros (invalid numbers). Convert the linked list in such a way that if next valid number is same as current number, double its value and replace the next number with 0. After the modification, rearrange the linked list such that all 0's are shifted to the end.

Examples:

Input : 2->2->0->4->0->8  
Output : 4->4->8->0->0->0

Input : 0->2->2->2->0->6->6->0->0->8  
Output : 4->2->12->8->0->0->0->0->0->0

Source: [Microsoft IDC Interview Experience | Set 156.](#)

**Approach:** First modify the linked list as mentioned, i.e., if next valid number is same as current number, double its value and replace the next number with 0.

**Algorithm for Modification:**

```
1. ptr = head
2. while (ptr && ptr->next)
3. if (ptr->data == 0) || (ptr->data != ptr->next->data)
4.     ptr = ptr->next
5. else
6.     ptr->data = 2 * ptr->data
7.     ptr->next->data = 0
8.     ptr = ptr->next->next
```

After modifying the list segregate the valid (non-zero) and invalid (zero) elements. It is same as [Segregating Even and Odd nodes in a Linked list](#).

```
// C++ implementation to modify and
// rearrange list
#include <bits/stdc++.h>
using namespace std;

// structure of a node
struct Node {
    int data;
    Node* next;
};

// 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->next = NULL;
    return newNode;
}

// function to segregate valid (non-zero) numbers and
// invalid (zero) numbers in a list
Node* segValidInvalidNum(Node* head)
{
    // valid (non-zero numbers) list start and
    // end pointers
    Node *validStart = NULL, *validEnd = NULL;

    // invalid (zero numbers) list start and
    // end pointers
    Node *invalidStart = NULL, *invalidEnd = NULL;

    Node* currentNode = head;

    // traverse the given list
    while (currentNode != NULL) {
        // current node's data
        int element = currentNode->data;

        // if it is a non-zero element, then
        // add it to valid list
        if (element != 0) {


```

```

        if (validStart == NULL) {
            validStart = currentNode;
            validEnd = validStart;
        }
        else {
            validEnd->next = currentNode;
            validEnd = validEnd->next;
        }
    }

    // else it is a zero element, so
    // add it to invalid list
    else {
        if (invalidStart == NULL) {
            invalidStart = currentNode;
            invalidEnd = invalidStart;
        }
        else {
            invalidEnd->next = currentNode;
            invalidEnd = invalidEnd->next;
        }
    }

    // Move to the next node
    currentNode = currentNode->next;
}

// if true then return the original list as it is
if (validStart == NULL || invalidStart == NULL)
    return head;

// add the invalid list to the end of valid list
validEnd->next = invalidStart;
invalidEnd->next = NULL;
head = validStart;

// required head of the new list
return head;
}

// function to modify and
// rearrange list
Node* modifyAndRearrangeList(Node* head)
{
    // if list is empty or contains a single
    // element only
    if (head == NULL || head->next == NULL)
        return head;
}

```

```
// traverse the list
Node* ptr = head;
while (ptr && ptr->next) {

    // if current node's data is '0' or it is not equal
    // to next nodes data, them move one node ahead
    if ((ptr->data == 0) || (ptr->data != ptr->next->data))
        ptr = ptr->next;

    else {

        // double current node's data
        ptr->data = 2 * ptr->data;

        // put '0' in next node's data
        ptr->next->data = 0;

        // move two nodes ahead
        ptr = ptr->next->next;
    }
}

// segregate valid (non-zero) and
// invalid (non-zero) numbers
return segValidInvalidNum(head);
}

// function to print a linked list
void printList(Node* head)
{
    while (head != NULL) {
        cout << head->data << " ";
        head = head->next;
    }
}

// Driver program to test above
int main()
{
    Node* head = getNode(2);
    head->next = getNode(2);
    head->next->next = getNode(0);
    head->next->next->next = getNode(4);
    head->next->next->next->next = getNode(0);
    head->next->next->next->next = getNode(8);

    cout << "Original List: ";
```

```
printList(head);

head = modifyAndRearrangeList(head);

cout << "\nModified List: ";
printList(head);

return 0;
}
```

Output:

```
Original List: 2 2 0 4 0 8
Modified List: 4 4 8 0 0 0
```

Time Complexity: O(n).

## Source

<https://www.geeksforgeeks.org/modify-rearrange-list/>

## Chapter 157

# Modify contents of Linked List

Modify contents of Linked List - GeeksforGeeks

Given a singly linked list containing  $n$  nodes. Modify the value of first half nodes such that 1st node's new value is equal to the last node's value minus first node's current value, 2nd node's new value is equal to the second last node's value minus 2nd node's current value, likewise for first half nodes. If  $n$  is odd then the value of the middle node remains unchanged.

(No extra memory to be used).

Examples:

Input : 10 → 4 → 5 → 3 → 6  
Output : 4 → 1 → 5 → 3 → 6

Input : 2 → 9 → 8 → 12 → 7 → 10  
Output : -8 → 2 → -4 → 12 → 7 → 10

Asked in Amazon Interview

**Approach :** The following steps are:

1. Split the list from the middle. Perform **front and back split**. If the number of elements is odd, the extra element should go in the 1st(front) list.
2. **Reverse the 2nd(back) list.**
3. Perform the required subtraction while traversing both list simultaneously.
4. Again reverse the 2nd list.
5. Concatenate the 2nd list back to the end of the 1st list.

```
// C++ implementation to modify the contents of
// the linked list
#include <bits/stdc++.h>
```

```
using namespace std;

/* Linked list node */
struct Node
{
    int data;
    struct Node* next;
};

/* function prototype for printing the list */
void printList(struct Node*);

/* Function to insert a node at the beginning of
   the linked list */
void push(struct Node **head_ref, int new_data)
{
    /* allocate node */
    struct Node* new_node =
        (struct Node*) malloc(sizeof(struct Node));

    /* put in the data */
    new_node->data = new_data;

    /* link the old list at the end of the new node */
    new_node->next = *head_ref;

    /* move the head to point to the new node */
    *head_ref = new_node;
}

/* Split the nodes of the given list
   into front and back halves,
   and return the two lists
   using the reference parameters.
   Uses the fast/slow pointer strategy. */
void frontAndBackSplit(struct Node *head,
                      struct Node **front_ref, struct Node **back_ref)
{
    Node *slow, *fast;

    slow = head;
    fast = head->next;

    /* Advance 'fast' two nodes, and
       advance 'slow' one node */
    while (fast != NULL)
    {
        fast = fast->next;

```

```
if (fast != NULL)
{
    slow = slow->next;
    fast = fast->next;
}
}

/* 'slow' is before the midpoint in the list,
   so split it in two at that point. */
*front_ref = head;
*back_ref = slow->next;
slow->next = NULL;
}

/* Function to reverse the linked list */
void reverseList(struct Node **head_ref)
{
    struct Node *current, *prev, *next;
    current = *head_ref;
    prev = NULL;
    while (current != NULL)
    {
        next = current->next;
        current->next = prev;
        prev = current;
        current = next;
    }
    *head_ref = prev;
}

// perform the required subtraction operation on
// the 1st half of the linked list
void modifyTheContentsOf1stHalf(struct Node *front,
                               struct Node *back)
{
    // traversing both the lists simultaneously
    while (back != NULL)
    {
        // subtraction operation and node data
        // modification
        front->data = front->data - back->data;

        front = front->next;
        back = back->next;
    }
}

// function to concatenate the 2nd(back) list at the end of
```

```
// the 1st(front) list and returns the head of the new list
struct Node* concatFrontAndBackList(struct Node *front,
                                     struct Node *back)
{
    struct Node *head = front;

    while (front->next != NULL)
        front = front->next;

    front->next     = back;

    return head;
}

// function to modify the contents of the linked list
struct Node* modifyTheList(struct Node *head)
{
    // if list is empty or contains only single node
    if (!head || head->next == NULL)
        return head;

    struct Node *front, *back;

    // split the list into two halves
    // front and back lists
    frontAndBackSplit(head, &front, &back);

    // reverse the 2nd(back) list
    reverseList(&back);

    // modify the contents of 1st half
    modifyTheContentsOf1stHalf(front, back);

    // agains reverse the 2nd(back) list
    reverseList(&back);

    // concatenating the 2nd list back to the
    // end of the 1st list
    head = concatFrontAndBackList(front, back);

    // pointer to the modified list
    return head;
}

// function to print the linked list
void printList(struct Node *head)
{
    if (!head)
```

```
    return;

    while (head->next != NULL)
    {
        cout << head->data << " -> ";
        head = head->next;
    }
    cout << head->data << endl;
}

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

    // creating the linked list
    push(&head, 10);
    push(&head, 7);
    push(&head, 12);
    push(&head, 8);
    push(&head, 9);
    push(&head, 2);

    // modify the linked list
    head = modifyTheList(head);

    // print the modified linked list
    cout << "Modified List:" << endl;
    printList(head);
    return 0;
}
```

Output:

```
Modified List:
-8 -> 2 -> -4 -> 12 -> 7 -> 10
```

Time Complexity: O(n), where **n** in the number of nodes.

**Another approach (Using Stack) :**

1. Find the starting point of second half Linked List.
2. Push all elements of second half list into stack s.
3. Traverse list starting from head using temp until stack is not empty and do Modify temp->data by subtracting the top element of stack for every node.

Below is the implementation using stack.

C++

```
// C++ implementation to modify the
// contents of the linked list
#include <bits/stdc++.h>
using namespace std;

// Linked list node
struct Node
{
    int data;
    struct Node* next;
};

// function prototype for printing the list
void printList(struct Node*);

// Function to insert a node at the
// beginning of the linked list
void push(struct Node **head_ref, int new_data)
{
    // allocate node
    struct Node* new_node =
        (struct Node*) malloc(sizeof(struct Node));

    // put in the data
    new_node->data = new_data;

    // link the old list at the end of the new node
    new_node->next = *head_ref;

    // move the head to point to the new node
    *head_ref = new_node;
}

// function to print the linked list
void printList(struct Node *head)
{
    if (!head)
        return;

    while (head->next != NULL)
    {
        cout << head->data << " -> ";
        head = head->next;
    }
    cout << head->data << endl;
}
```

```
// Function to middle node of list.
Node* find_mid(Node *head)
{
    Node *temp = head, *slow = head, *fast = head ;

    while(fast && fast->next)
    {

        // Advance 'fast' two nodes, and
        // advance 'slow' one node
        slow = slow->next ;
        fast = fast->next->next ;
    }

    // If number of nodes are odd then update slow
    // by slow->next;
    if(fast)
        slow = slow->next ;

    return slow ;
}

// function to modify the contents of the linked list.
void modifyTheList(struct Node *head, struct Node *slow)
{
    // Create Stack.
    stack <int> s;
    Node *temp = head ;

    while(slow)
    {
        s.push( slow->data ) ;
        slow = slow->next ;
    }

    // Traverse the list by using temp until stack is empty.
    while( !s.empty() )
    {
        temp->data = temp->data - s.top() ;
        temp = temp->next ;
        s.pop() ;
    }

}

// Driver program to test above
int main()
{
```

```
struct Node *head = NULL, *mid ;  
  
// creating the linked list  
push(&head, 10);  
push(&head, 7);  
push(&head, 12);  
push(&head, 8);  
push(&head, 9);  
push(&head, 2);  
  
// Call Function to Find the starting point of second half of list.  
mid = find_mid(head) ;  
  
// Call function to modify the contents of the linked list.  
modifyTheList( head, mid);  
  
// print the modified linked list  
cout << "Modified List:" << endl;  
printList(head);  
return 0;  
}  
  
// This is contributed by Mr. Gera
```

Output:

```
Modified List:  
-8 -> 2 -> -4 -> 12 -> 7 -> 10
```

Time Complexity : O(n)  
Space Complexity : O(n/2)

**References:** <https://www.careercup.com/question?id=5657550909341696>

## Source

<https://www.geeksforgeeks.org/modify-contents-linked-list/>

## Chapter 158

# Move all occurrences of an element to end in a linked list

Move all occurrences of an element to end in a linked list - GeeksforGeeks

Given a linked list and a key in it, the task is to move all occurrences of given key to end of linked list, keeping order of all other elements same.

Examples:

```
Input : 1 -> 2 -> 2 -> 4 -> 3  
       key = 2  
Output : 1 -> 4 -> 3 -> 2 -> 2
```

```
Input : 6 -> 6 -> 7 -> 6 -> 3 -> 10  
       key = 6  
Output : 7 -> 3 -> 10 -> 6 -> 6 -> 6
```

A **simple solution** is to one by one find all occurrences of given key in linked list. For every found occurrence, insert it at the end. We do it till all occurrences of given key are moved to end.

Time Complexity :  $O(n^2)$

**Efficient Solution 1 :** is to keep two pointers:

**pCrawl** => Pointer to traverse the whole list one by one.

**pKey** => Pointer to an occurrence of key if a key is found. Else same as pCrawl.

We start both of the above pointers from head of linked list. We move **pKey** only when **pKey** is not pointing to a key. We always move **pCrawl**. So when **pCrawl** and **pKey** are not same, we must have found a key which lies before **pCrawl**, so we swap data of **pCrawl** and **pKey**, and move **pKey** to next location. The loop invariant is, after swapping of data, all elements from **pKey** to **pCrawl** are keys.

Below is the C++ implementation of this approach.

```
// C++ program to move all occurrences of a
// given key to end.
#include<bits/stdc++.h>

// A Linked list Node
struct Node
{
    int data;
    struct Node* next;
};

// A utility function to create a new node.
struct Node *newNode(int x)
{
    Node *temp = new Node;
    temp->data = x;
    temp->next = NULL;
}

// Utility function to print the elements
// in Linked list
void printList(Node *head)
{
    struct Node* temp = head;
    while (temp != NULL)
    {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}

// Moves all occurrences of given key to
// end of linked list.
void moveToEnd(Node *head, int key)
{
    // Keeps track of locations where key
    // is present.
    struct Node *pKey = head;

    // Traverse list
    struct Node *pCrawl = head;
    while (pCrawl != NULL)
    {
        // If current pointer is not same as pointer
        // to a key location, then we must have found
        // a key in linked list. We swap data of pCrawl
        // and pKey and move pKey to next position.
    }
}
```

```
if (pCrawl != pKey && pCrawl->data != key)
{
    pKey->data = pCrawl->data;
    pCrawl->data = key;
    pKey = pKey->next;
}

// Find next position where key is present
if (pKey->data != key)
    pKey = pKey->next;

// Moving to next Node
pCrawl = pCrawl->next;
}

// Driver code
int main()
{
    Node *head = newNode(10);
    head->next = newNode(20);
    head->next->next = newNode(10);
    head->next->next->next = newNode(30);
    head->next->next->next->next = newNode(40);
    head->next->next->next->next->next = newNode(10);
    head->next->next->next->next->next->next = newNode(60);

    printf("Before moveToEnd(), the Linked list is\n");
    printList(head);

    int key = 10;
    moveToEnd(head, key);

    printf("\nAfter moveToEnd(), the Linked list is\n");
    printList(head);

    return 0;
}
```

Output:

```
Before moveToEnd(), the Linked list is
10 20 10 30 40 10 60
```

```
After moveToEnd(), the Linked list is
20 30 40 60 10 10 10
```

Time Complexity : O(n) requires only one traversal of list.

**Efficient Solution 2 :**

1. Traverse the linked list and take a pointer at tail.
2. Now, check for the key and node->data, if they are equal, move the node to last-next, else move ahead.

**Java**

```
// Java code to remove key element to end of linked list
import java.util.*;

// Node class
class Node{
    int data;
    Node next;

    public Node(int data){
        this.data=data;
        this.next=null;
    }
}

class gfg{

    static Node root;

    // Function to remove key to end
    public static Node keyToEnd(Node head, int key){

        // Node to keep pointing to tail
        Node tail = head;

        if(head==null){
            return null;
        }

        while(tail.next !=null){
            tail=tail.next;
        }

        // Node to point to last of linked list
        Node last = tail;

        Node current = head;
        Node prev = null;

        // Node prev2 to point to previous when head.data!=key
        Node prev2 = null;
```

```
// loop to perform operations to remove key to end
while(current!=tail){
    if(current.data==key && prev2 == null){
        prev = current;
        current = current.next;
        head = current;
        last.next = prev;
        last = last.next;
        last.next = null;
        prev = null;
    }
    else{
        if(current.data==key && prev2!=null){
            prev = current;
            current = current.next;
            prev2.next = current;
            last.next = prev;
            last = last.next;
            last.next = null;
        }
        else
            if(current != tail){
                prev2 = current;
                current = current.next;
            }
    }
}
return head;
}

// Function to display linked list
public static void display(Node root){
    while(root!=null){
        System.out.print(root.data+" ");
        root = root.next;
    }
}

// Driver Code
public static void main(String args[]){
    root = new Node(5);
    root.next = new Node(2);
    root.next.next = new Node(2);
    root.next.next.next = new Node(7);
    root.next.next.next.next = new Node(2);
    root.next.next.next.next.next = new Node(2);
    root.next.next.next.next.next.next = new Node(2);
```

```
int key = 2;
System.out.println("Linked List before operations :");
display(root);
System.out.println("\nLinked List after operations :");
root = keyToEnd(root, key);
display(root);
}
}
```

Thanks to **Ravinder Kumar** for suggesting this method.

**Efficient Solution 3 :** is to maintain a separate list of keys. We initialize this list of keys as empty. We traverse given list. For every key found, we remove it from the original list and insert into the separate list of keys. We finally link list of keys at the end of remaining given list. Time complexity of this solution is also  $O(n)$  and it also requires only one traversal of list.

## Source

<https://www.geeksforgeeks.org/move-occurrences-element-end-linked-list/>

## Chapter 159

# Move all zeros to the front of the linked list

Move all zeros to the front of the linked list - GeeksforGeeks

Given a linked list. the task is to move all 0's to the front of the linked list. The order of all other element except 0 should be same after rearrangement.

Examples:

Input : 0 1 0 1 2 0 5 0 4 0  
Output :0 0 0 0 0 1 1 2 5 4

Input :1 1 2 3 0 0 0  
Output :0 0 0 1 1 2 3

A **simple solution** is to store all linked list element in an array. Then move all elements of array to beginning. Finally copy array elements back to linked list.

An **efficient solution** is to traverse the linked list from second node. For every node with 0 value, we disconnect it from its current position and move the node to front.

```
// CPP program to move all zeros
// to the end of the linked list.
#include <bits/stdc++.h>
using namespace std;

/* Link list node */
struct Node {
    int data;
    struct Node *next;
};


```

```
/* Given a reference (pointer to pointer) to
the head of a list and an int, push a new
node on the front of the list. */
void push(struct Node **head_ref, int new_data) {
    struct Node *new_node = new Node;
    new_node->data = new_data;
    new_node->next = (*head_ref);
    (*head_ref) = new_node;
}

/* moving zeroes to the begining in linked list */
void moveZeroes(struct Node **head) {
    if (*head == NULL)
        return;

    // Traverse the list from second node.
    struct Node *temp = (*head)->next, *prev = *head;
    while (temp != NULL) {

        // If current node is 0, move to
        // beginning of linked list
        if (temp->data == 0) {

            // Disconnect node from its
            // current position
            Node *curr = temp;
            temp = temp->next;
            prev->next = temp;

            // Move to beginning
            curr->next = (*head);
            *head = curr;
        }

        // For non-zero values
        else {
            prev = temp;
            temp = temp->next;
        }
    }
}

// function to displaying nodes
void display(struct Node *head) {
    while (head != NULL) {
        cout << head->data << "->";
        head = head->next;
    }
}
```

```
}

cout << "NULL";
}

/* Drier program to test above function*/
int main() {

    /* Start with the empty list */
    struct Node *head = NULL;

    /* Use push() to construct below list
    0->0->1->0->1->0->2->0->3->0 */
    push(&head, 0);
    push(&head, 3);
    push(&head, 0);
    push(&head, 2);
    push(&head, 0);
    push(&head, 1);
    push(&head, 0);
    push(&head, 1);
    push(&head, 0);
    push(&head, 0);

    // displaying list before rearrangement
    cout << "Linked list before rearrangement\n";
    display(head);

    /* Check the move_zeroes function */
    moveZeroes(&head);

    // displaying list after rearrangement
    cout << "\n Linked list after rearrangement \n";
    display(head);

    return 0;
}
```

Output:

```
Linked list before rearrangement
0->0->1->0->1->0->2->0->3->0->NULL
```

```
Linked list after rearrangement
0->0->0->0->0->1->1->2->3->NULL
```

**Source**

<https://www.geeksforgeeks.org/move-zeroes-front-linked-list/>

## Chapter 160

# Move first element to end of a given Linked List

Move first element to end of a given Linked List - GeeksforGeeks

Write a C function that moves first element to end in a given Singly Linked List. For example, if the given Linked List is 1->2->3->4->5, then the function should change the list to 2->3->4->5->1.

Algorithm:

Traverse the list till last node. Use two pointers: one to store the address of last node(last) and other for address of first node(first). After the end of loop do following operations.

- i) Make head as second node (\*head\_ref = first->next).
- ii) Set next of first as NULL (first->next = NULL).
- iii) Set next of last as first ( last->next = first)

```
/* C++ Program to move first element to end
   in a given linked list */
#include <stdio.h>
#include <stdlib.h>

/* A linked list node */
struct Node {
    int data;
    struct Node* next;
};

/* We are using a double pointer head_ref
   here because we change head of the linked
   list inside this function.*/
void moveToEnd(struct Node** head_ref)
{
    /* If linked list is empty, or it contains
       only one node then head_ref will still
       point to the head and hence we are
       not changing anything */
    if (*head_ref == NULL || (*head_ref)->next == NULL)
        return;

    /* Start traversing from head until
       we find the last node */
    struct Node* last = *head_ref;
    while (last->next != NULL)
        last = last->next;

    /* last now points to the last node.
       Make the next of last node as
       address of head */
    last->next = *head_ref;

    /* Change head to point to second
       node */
    *head_ref = (*head_ref)->next;
}
```

```
only one node, then nothing needs to be
done, simply return */
if (*head_ref == NULL || (*head_ref)->next == NULL)
    return;

/* Initialize first and last pointers */
struct Node* first = *head_ref;
struct Node* last = *head_ref;

/*After this loop last contains address
of last node in Linked List */
while (last->next != NULL) {
    last = last->next;
}

/* Change the head pointer to point
   to second node now */
*head_ref = first->next;

/* Set the next of first as NULL */
first->next = NULL;

/* Set the next of last as first */
last->next = first;
}

/* UTILITY FUNCTIONS */
/* Function to add a node at the beginning
   of Linked List */
void push(struct Node** head_ref, int new_data)
{
    struct Node* new_node = new Node;
    new_node->data = new_data;
    new_node->next = (*head_ref);
    (*head_ref) = new_node;
}

/* Function to print nodes in a given linked list */
void printList(struct Node* node)
{
    while (node != NULL) {
        printf("%d ", node->data);
        node = node->next;
    }
}

/* Driver program to test above function */
int main()
```

```
{  
    struct Node* start = NULL;  
  
    /* The constructed linked list is:  
       1->2->3->4->5 */  
    push(&start, 5);  
    push(&start, 4);  
    push(&start, 3);  
    push(&start, 2);  
    push(&start, 1);  
  
    printf("\n Linked list before moving first to end\n");  
    printList(start);  
  
    moveToEnd(&start);  
  
    printf("\n Linked list after moving first to end\n");  
    printList(start);  
  
    return 0;  
}
```

**Output:**

```
Linked list before moving first to end  
1 2 3 4 5  
Linked list after moving first to end  
2 3 4 5 1
```

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

**Source**

<https://www.geeksforgeeks.org/move-first-element-to-end-of-a-given-linked-list/>

## Chapter 161

# Move last element to front of a given Linked List

Move last element to front of a given Linked List - GeeksforGeeks

Write a C function that moves last element to front in a given Singly Linked List. For example, if the given Linked List is 1->2->3->4->5, then the function should change the list to 5->1->2->3->4.

Algorithm:

Traverse the list till last node. Use two pointers: one to store the address of last node and other for address of second last node. After the end of loop do following operations.

- i) Make second last as last (secLast->next = NULL).
- ii) Set next of last as head (last->next = \*head\_ref).
- iii) Make last as head (\*head\_ref = last)

C/C++

```
/* C Program to move last element to front in a given linked list */
#include<stdio.h>
#include<stdlib.h>

/* A linked list node */
struct Node
{
    int data;
    struct Node *next;
};

/* We are using a double pointer head_ref here because we change
   head of the linked list inside this function.*/
void moveToFront(struct Node **head_ref)
{
    /* If linked list is empty, or it contains only one node,
```

```
    then nothing needs to be done, simply return */
if (*head_ref == NULL || (*head_ref)->next == NULL)
    return;

/* Initialize second last and last pointers */
struct Node *secLast = NULL;
struct Node *last = *head_ref;

/*After this loop secLast contains address of second last
node and last contains address of last node in Linked List */
while (last->next != NULL)
{
    secLast = last;
    last = last->next;
}

/* Set the next of second last as NULL */
secLast->next = NULL;

/* Set next of last as head node */
last->next = *head_ref;

/* Change the head pointer to point to last node now */
*head_ref = last;
}

/* UTILITY FUNCTIONS */
/* Function to add a node at the begining of Linked List */
void push(struct Node** head_ref, int new_data)
{
    /* allocate node */
    struct Node* new_node =
        (struct Node*) malloc(sizeof(struct Node));

    /* put in the data */
    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;
}

/* Function to print nodes in a given linked list */
void printList(struct Node *node)
{
```

```
while(node != NULL)
{
    printf("%d ", node->data);
    node = node->next;
}
}

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

    /* The constructed linked list is:
     1->2->3->4->5 */
    push(&start, 5);
    push(&start, 4);
    push(&start, 3);
    push(&start, 2);
    push(&start, 1);

    printf("\n Linked list before moving last to front\n");
    printList(start);

    moveToFront(&start);

    printf("\n Linked list after removing last to front\n");
    printList(start);

    return 0;
}
```

### Java

```
/* Java Program to move last element to front in a given linked list */
class LinkedList
{
    Node head; // head of list

    /* Linked list Node*/
    class Node
    {
        int data;
        Node next;
        Node(int d) {data = d; next = null; }
    }

    void moveToFront()
    {
```

```
/* If linked list is empty or it contains only
   one node then simply return. */
if(head == null || head.next == null)
    return;

/* Initialize second last and last pointers */
Node secLast = null;
Node last = head;

/* After this loop secLast contains address of
   second last node and last contains address of
   last node in Linked List */
while (last.next != null)
{
    secLast = last;
    last = last.next;
}

/* Set the next of second last as null */
secLast.next = null;

/* Set the next of last as head */
last.next = head;

/* Change head to point to last node. */
head = last;
}

/* Utility functions */

/* Inserts a new Node at front of the list. */
public void push(int new_data)
{
    /* 1 & 2: Allocate the Node &
       Put in the data*/
    Node new_node = new Node(new_data);

    /* 3. Make next of new Node as head */
    new_node.next = head;

    /* 4. Move the head to point to new Node */
    head = new_node;
}

/* Function to print linked list */
void printList()
{
```

```
Node temp = head;
while(temp != null)
{
    System.out.print(temp.data+" ");
    temp = temp.next;
}
System.out.println();
}

/* Drier program to test above functions */
public static void main(String args[])
{
    LinkedList llist = new LinkedList();
    /* Constructed Linked List is 1->2->3->4->5->null */
    llist.push(5);
    llist.push(4);
    llist.push(3);
    llist.push(2);
    llist.push(1);

    System.out.println("Linked List before moving last to front ");
    llist.printList();

    llist.moveToFront();

    System.out.println("Linked List after moving last to front ");
    llist.printList();
}
}

/* This code is contributed by Rajat Mishra */
```

Output:

```
Linked list before moving last to front
1 2 3 4 5
Linked list after removing last to front
5 1 2 3 4
```

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

**Improved By :** [Soumith Bsv](#)

## Source

<https://www.geeksforgeeks.org/move-last-element-to-front-of-a-given-linked-list/>

## Chapter 162

# Multiply two numbers represented as linked lists into a third list

Multiply two numbers represented as linked lists into a third list - GeeksforGeeks

Given two numbers represented by linked lists, write a function that returns the head of the new linked list that represents the number that is the product of those numbers.

Examples:

```
Input : 9->4->6  
        8->4  
Output : 7->9->4->6->4
```

```
Input : 9->9->9->4->6->9  
        9->9->8->4->9  
Output : 9->9->7->9->5->9->8->0->1->8->1
```

We have already discussed a solution in below post.

[Multiply two numbers represented by Linked Lists](#)

The solution discussed above store result in an integer. Here we store result in a third list so that large numbers can be handled.

Remember old school multiplication? we imitate that process. On paper, we take the last digit of a number and multiply with the second number and write the product. Now leave the last column and same way each digit of one number is multiplied with every digit of other number and every time result is written by leaving one last column. then add these columns that forms the number. Now assume these columns as nodes of the resultant linked list. We make resultant linked list in reversed fashion.

**Algorithm**

Reverse both linked lists  
Make a linked list of maximum result size ( $m + n + 1$ )  
For each node of one list  
    For each node of second list  
        a) Multiply nodes  
        b) Add digit in result LL at corresponding position  
        c) Now resultant node itself can be higher than one digit  
        d) Make carry for next node  
    Leave one last column means next time start  
From next node in result list  
Reverse the resulted linked list

```
// C program to Multiply two numbers
// represented as linked lists
#include <stdio.h>
#include <stdlib.h>

// Linked list Node
struct Node {
    int data;
    struct Node* next;
};

// Function to create a new Node
// with given data
struct Node* newNode(int data)
{
    struct Node* new_node =
        (struct Node*)malloc(sizeof(struct Node));
    new_node->data = data;
    new_node->next = NULL;
    return new_node;
}

// Function to insert a Node at the
// beginning of the Linked List
void push(struct Node** head_ref, int new_data)
{
    // allocate Node
    struct Node* new_node = newNode(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;
}
```

```
(*head_ref) = new_node;
}

// Function to reverse the linked list and return
// its length
int reverse(struct Node** head_ref)
{
    struct Node* prev = NULL;
    struct Node* current = *head_ref;
    struct Node* next;
    int len = 0;
    while (current != NULL) {
        len++;
        next = current->next;
        current->next = prev;
        prev = current;
        current = next;
    }
    *head_ref = prev;
    return len;
}

// Function to make an empty linked list of
// given size
struct Node* make_empty_list(int size)
{
    struct Node* head = NULL;
    while (size--)
        push(&head, 0);
    return head;
}

// Multiply contents of two linked lists => store
// in another list and return its head
struct Node* multiplyTwoLists(struct Node* first,
                             struct Node* second)
{
    // reverse the lists to multiply from end
    // m and n lengths of linked lists to make
    // an empty list
    int m = reverse(&first), n = reverse(&second);

    // make a list that will contain the result
    // of multiplication.
    // m+n+1 can be max size of the list
    struct Node* result = make_empty_list(m + n + 1);

    // pointers for traverse linked lists and also
```

```
// to reverse them after
struct Node *second_ptr = second,
    *result_ptr1 = result, *result_ptr2, *first_ptr;

// multiply each Node of second list with first
while (second_ptr) {

    int carry = 0;

    // each time we start from the next of Node
    // from which we started last time
    result_ptr2 = result_ptr1;

    first_ptr = first;

    while (first_ptr) {

        // multiply a first list's digit with a
        // current second list's digit
        int mul = first_ptr->data * second_ptr->data
                  + carry;

        // Assign the product to corresponding Node
        // of result
        result_ptr2->data += mul % 10;

        // now resultant Node itself can have more
        // than 1 digit
        carry = mul / 10 + result_ptr2->data / 10;
        result_ptr2->data = result_ptr2->data % 10;

        first_ptr = first_ptr->next;
        result_ptr2 = result_ptr2->next;
    }

    // if carry is remaining from last multiplication
    if (carry > 0) {
        result_ptr2->data += carry;
    }

    result_ptr1 = result_ptr1->next;
    second_ptr = second_ptr->next;
}

// reverse the result_list as it was populated
// from last Node
reverse(&result);
reverse(&first);
```

```
reverse(&second);

// remove if there are zeros at starting
while (result->data == 0) {
    struct Node* temp = result;
    result = result->next;
    free(temp);
}

// Return head of multiplication list
return result;
}

// A utility function to print a linked list
void printList(struct Node* Node)
{
    while (Node != NULL) {
        printf("%d", Node->data);
        if (Node->next)
            printf("->");
        Node = Node->next;
    }
    printf("\n");
}

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

    // create first list 9->9->9->4->6->9
    push(&first, 9);
    push(&first, 6);
    push(&first, 4);
    push(&first, 9);
    push(&first, 9);
    printf("First List is: ");
    printList(first);

    // create second list 9->9->8->4->9
    push(&second, 9);
    push(&second, 4);
    push(&second, 8);
    push(&second, 9);
    push(&second, 9);
    printf("Second List is: ");
```

```
printList(second);

// Multiply the two lists and see result
struct Node* result = multiplyTwoLists(first, second);
printf("Resultant list is: ");
printList(result);

return 0;
}
```

Output:

```
First List is: 9->9->9->4->6->9
Second List is: 9->9->8->4->9
Resultant list is: 9->9->7->9->5->9->8->0->1->8->1
```

Note: we can take care of resultant node that can have more than 1 digit outside the loop just traverse the result list and add carry to next digit before reversing.

## Source

<https://www.geeksforgeeks.org/multiply-two-numbers-represented-linked-lists-third-list/>

## Chapter 163

# Multiply two numbers represented by Linked Lists

Multiply two numbers represented by Linked Lists - GeeksforGeeks

Given two numbers represented by linked lists, write a function that returns the multiplication of these two linked lists.

**Examples:**

```
Input : 9->4->6
        8->4
Output : 79464
```

```
Input : 3->2->1
        1->2
Output : 3852
```

**Solution:**

Traverse both lists and generate the required numbers to be multiplied and then return the multiplied values of the two numbers.

Algorithm to generate the number from linked list representation:

- 1) Initialize a variable to zero
- 2) Start traversing the linked list
- 3) Add the value of first node to this variable
- 4) From the second node, multiply the variable by 10 first and then add the value of the node to this variable.
- 5) Repeat step 4 until we reach the last node of the list.

Use the above algorithm with both of linked lists to generate the numbers.  
Below is the program for multiplying two numbers represented as linked lists:

**C++**

```
// C program to Multiply two numbers
// represented as linked lists
#include<stdio.h>
#include<stdlib.h>

// Linked list node
struct node
{
    int data;
    struct node* next;
};

// Function to create a new node
// with given data
struct node *newNode(int data)
{
    struct node *new_node = (struct node *) malloc(sizeof(struct node));
    new_node->data = data;
    new_node->next = NULL;
    return new_node;
}

// Function to insert a node at the
// beginning of the Linked List
void push(struct node** head_ref, int new_data)
{
    // allocate node
    struct node* new_node = newNode(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;
}

// Multiply contents of two linked lists
long multiplyTwoLists (struct node* first, struct node* second)
{
    int num1 = 0, num2 = 0;

    // Generate numbers from linked lists
    while (first || second)
```

```
{  
    if (first)  
    {  
        num1 = num1*10 + first->data;  
        first = first->next;  
    }  
    if (second)  
    {  
        num2 = num2*10 + second->data;  
        second = second->next;  
    }  
}  
  
// Return multiplication of  
// two numbers  
return num1*num2;  
}  
  
// A utility function to print a linked list  
void printList(struct node *node)  
{  
    while(node != NULL)  
    {  
        printf("%d", node->data);  
        if(node->next)  
            printf("->");  
        node = node->next;  
    }  
    printf("\n");  
}  
  
// Driver program to test above function  
int main(void)  
{  
    struct node* first = NULL;  
    struct node* second = NULL;  
  
    // create first list 9->4->6  
    push(&first, 6);  
    push(&first, 4);  
    push(&first, 9);  
    printf("First List is: ");  
    printList(first);  
  
    // create second list 8->4  
    push(&second, 4);  
    push(&second, 8);  
    printf("Second List is: ");
```

```
printList(second);

// Multiply the two lists and see result
printf("Result is: ");
printf("%ld", multiplyTwoLists(first, second));

return 0;
}
```

**Python3**

```
# Python program to Multiply
# two linkedlists

# Linked List Node
class Node():
    def __init__(self, data):
        self.data = data
        self.next = None

# Linked List class
class singly_linked_list():
    def __init__(self):
        self.head = None

    def add(self, node):

        '''Add a node to the linked list'''
        if not self.head:

            # If no nodes exist, assign the
            # new node to the head node.
            self.head = node
        else:

            # Link the old stuff
            # to the new node
            node.next = self.head

            # Move the head node to
            # point to the new node
            self.head = node

    def __repr__(self):

        '''Returns a string object containing
           a representation of the linked list.'''
        next = self.head
```

```
str_repr = ""
while(next):
    str_repr += str(next.data)
    if next.next: str_repr += "->"
    next = next.next
return str_repr

def multiply_linked_lists(list1, list2):

    '''Generates the numeric representation
       of 2 linked lists and returns their product.'''
    first_number = 0
    second_number = 0

    # Generate the first number
    next = list1.head
    while next:
        first_number = (first_number *
                        10 + next.data)
        next = next.next

    # Generate the second number
    next = list2.head
    while next:
        second_number = (second_number *
                          10 + next.data)
        next = next.next

    # Return the product
    return first_number * second_number

# Driver Code

# Instantiate 2 linked lists
l_list1 = singly_linked_list()
l_list2 = singly_linked_list()

# Create first list
l_list1.add(Node(6))
l_list1.add(Node(4))
l_list1.add(Node(9))

# Create second list
l_list2.add(Node(4))
l_list2.add(Node(8))

print("First list is: ", l_list1)
print("Second list is: ", l_list2)
```

```
print("Results is: ",  
      multiply_linked_lists(l_list1, l_list2))  
  
# This code is contributed  
# by harishkumar88
```

**Output:**

```
First List is: 9->4->6  
Second List is: 8->4  
Result is: 79464
```

Improved By : [harishkumar88](#)

**Source**

<https://www.geeksforgeeks.org/multiply-two-numbers-represented-linked-lists/>

## Chapter 164

# Multiply two polynomials

Multiply two polynomials - GeeksforGeeks

Given two polynomials represented by two arrays, write a function that multiplies given two polynomials.

Example:

```
Input: A[] = {5, 0, 10, 6}
       B[] = {1, 2, 4}
Output: prod[] = {5, 10, 30, 26, 52, 24}
```

```
The first input array represents "5 + 0x^1 + 10x^2 + 6x^3"
The second array represents "1 + 2x^1 + 4x^2"
And Output is "5 + 10x^1 + 30x^2 + 26x^3 + 52x^4 + 24x^5"
```

A simple solution is to one by one consider every term of first polynomial and multiply it with every term of second polynomial. Following is algorithm of this simple method.

```
multiply(A[0..m-1], B[0..n-1])
1) Create a product array prod[] of size m+n-1.
2) Initialize all entries in prod[] as 0.
3) Travers array A[] and do following for every element A[i]
... (3.a) Traverse array B[] and do following for every element B[j]
        prod[i+j] = prod[i+j] + A[i] * B[j]
4) Return prod[].
```

The following is C++ implementation of above algorithm.

```
// Simple C++ program to multiply two polynomials
```

```
#include <iostream>
using namespace std;

// A[] represents coefficients of first polynomial
// B[] represents coefficients of second polynomial
// m and n are sizes of A[] and B[] respectively
int *multiply(int A[], int B[], int m, int n)
{
    int *prod = new int[m+n-1];

    // Initialize the product polynomial
    for (int i = 0; i<m+n-1; i++)
        prod[i] = 0;

    // Multiply two polynomials term by term

    // Take every term of first polynomial
    for (int i=0; i<m; i++)
    {
        // Multiply the current term of first polynomial
        // with every term of second polynomial.
        for (int j=0; j<n; j++)
            prod[i+j] += A[i]*B[j];
    }

    return prod;
}

// A utility function to print a polynomial
void printPoly(int poly[], int n)
{
    for (int i=0; i<n; i++)
    {
        cout << poly[i];
        if (i != 0)
            cout << "x^" << i ;
        if (i != n-1)
            cout << " + ";
    }
}

// Driver program to test above functions
int main()
{
    // The following array represents polynomial 5 + 10x^2 + 6x^3
    int A[] = {5, 0, 10, 6};

    // The following array represents polynomial 1 + 2x + 4x^2
```

```

int B[] = {1, 2, 4};
int m = sizeof(A)/sizeof(A[0]);
int n = sizeof(B)/sizeof(B[0]);

cout << "First polynomial is n";
printPoly(A, m);
cout << "nSecond polynomial is n";
printPoly(B, n);

int *prod = multiply(A, B, m, n);

cout << "nProduct polynomial is n";
printPoly(prod, m+n-1);

return 0;
}

```

Output

```

First polynomial is
5 + 0x^1 + 10x^2 + 6x^3
Second polynomial is
1 + 2x^1 + 4x^2
Product polynomial is
5 + 10x^1 + 30x^2 + 26x^3 + 52x^4 + 24x^5

```

Time complexity of the above solution is  $O(mn)$ . If size of two polynomials same, then time complexity is  $O(n^2)$ .

#### Can we do better?

There are methods to do multiplication faster than  $O(n^2)$  time. These methods are mainly based on [divide and conquer](#). Following is one simple method that divides the given polynomial (of degree  $n$ ) into two polynomials one containing lower degree terms(lower than  $n/2$ ) and other containing higher degree terms (higher than or equal to  $n/2$ )

Let the two given polynomials be A and B.

For simplicity, Let us assume that the given two polynomials are of same degree and have degree in powers of 2, i.e.,  $n = 2i$

The polynomial 'A' can be written as  $A_0 + A_1 \cdot x^n/2$   
 The polynomial 'B' can be written as  $B_0 + B_1 \cdot x^n/2$

For example  $1 + 10x + 6x^2 - 4x^3 + 5x^4$  can be written as  $(1 + 10x) + (6 - 4x + 5x^2) \cdot x^2$

$A * B = (A_0 + A_1 \cdot x^n/2) * (B_0 + B_1 \cdot x^n/2)$

$$\begin{aligned}
 &= A_0 * B_0 + A_0 * B_1 * x^n / 2 + A_1 * B_0 * x^n / 2 + A_1 * B_1 * x^n \\
 &= A_0 * B_0 + (A_0 * B_1 + A_1 * B_0) x^n / 2 + A_1 * B_1 * x^n
 \end{aligned}$$

So the above divide and conquer approach requires 4 multiplications and  $O(n)$  time to add all 4 results. Therefore the time complexity is  $T(n) = 4T(n/2) + O(n)$ . The solution of the recurrence is  $O(n^2)$  which is same as the above simple solution.

The idea is to reduce number of multiplications to 3 and make the recurrence as  $T(n) = 3T(n/2) + O(n)$

***How to reduce number of multiplications?***

This requires a little trick similar to [Strassen's Matrix Multiplication](#). We do following 3 multiplications.

```

X = (A0 + A1)*(B0 + B1) // First Multiplication
Y = A0B0 // Second
Z = A1B1 // Third

```

The missing middle term in above multiplication equation  $A_0 * B_0 + (A_0 * B_1 + A_1 * B_0) x^n / 2 + A_1 * B_1 * x^n$  can obtained using below.

$$A_0B_1 + A_1B_0 = X - Y - Z$$

***In Depth Explanation***

Conventional polynomial multiplication uses 4 coefficient multiplications:

$$(ax + b)(cx + d) = acx^2 + (ad + bc)x + bd$$

However, notice the following relation:

$$(a + b)(c + d) = ad + bc + ac + bd$$

The rest of the two components are exactly the middle coefficient for product of two polynomials. Therefore, the product can be computed as:

$$\begin{aligned}
 (ax + b)(cx + d) &= acx^2 + \\
 ((a + b)(c + d), ac, bd)x + bd
 \end{aligned}$$

Hence, the latter expression has only three multiplications.

So the time taken by this algorithm is  $T(n) = 3T(n/2) + O(n)$

The solution of above recurrence is  $O(n^{\lg 3})$  which is better than  $O(n^2)$ .

We will soon be discussing implementation of above approach.

There is a  $O(n\log n)$  algorithm also that uses Fast Fourier Transform to multiply two polynomials (Refer [this](#) and [this](#) for details)

**Sources:**

<http://www.cse.ust.hk/~dekai/271/notes/L03/L03.pdf>

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

**Improved By :** [Mr.L](#)

**Source**

<https://www.geeksforgeeks.org/multiply-two-polynomials-2/>

## Chapter 165

# Pairwise swap elements of a given linked list

Pairwise swap elements of a given linked list - GeeksforGeeks

Given a singly linked list, write a function to swap elements pairwise. For example, if the linked list is 1->2->3->4->5 then the function should change it to 2->1->4->3->5, and if the linked list is 1->2->3->4->5->6 then the function should change it to 2->1->4->3->6->5.

### METHOD 1 (Iterative)

Start from the head node and traverse the list. While traversing swap data of each node with its next node's data.

C/C++

```
/* C program to pairwise swap elements in a given linked list */
#include<stdio.h>
#include<stdlib.h>

/* A linked list node */
struct Node
{
    int data;
    struct Node *next;
};

/*Function to swap two integers at addresses a and b */
void swap(int *a, int *b);

/* Function to pairwise swap elements of a linked list */
void pairWiseSwap(struct Node *head)
{
    struct Node *temp = head;
```

```
/* Traverse further only if there are at-least two nodes left */
while (temp != NULL && temp->next != NULL)
{
    /* Swap data of node with its next node's data */
    swap(&temp->data, &temp->next->data);

    /* Move temp by 2 for the next pair */
    temp = temp->next->next;
}
}

/* UTILITY FUNCTIONS */
/* Function to swap two integers */
void swap(int *a, int *b)
{
    int temp;
    temp = *a;
    *a = *b;
    *b = temp;
}

/* Function to add a node at the begining of Linked List */
void push(struct Node** head_ref, int new_data)
{
    /* allocate node */
    struct Node* new_node =
        (struct Node*) malloc(sizeof(struct Node));

    /* put in the data */
    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;
}

/* Function to print nodes in a given linked list */
void printList(struct Node *node)
{
    while (node != NULL)
    {
        printf("%d ", node->data);
        node = node->next;
    }
}
```

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

    /* The constructed linked list is:
     1->2->3->4->5 */
    push(&start, 5);
    push(&start, 4);
    push(&start, 3);
    push(&start, 2);
    push(&start, 1);

    printf("Linked list before calling pairWiseSwap()\n");
    printList(start);

    pairWiseSwap(start);

    printf("\nLinked list after calling pairWiseSwap()\n");
    printList(start);

    return 0;
}
```

### Java

```
// Java program to pairwise swap elements of a linked list
class LinkedList
{
    Node head; // head of list

    /* Linked list Node*/
    class Node
    {
        int data;
        Node next;
        Node(int d) {data = d; next = null; }
    }

    void pairWiseSwap()
    {
        Node temp = head;

        /* Traverse only till there are atleast 2 nodes left */
        while (temp != null && temp.next != null) {

            /* Swap the data */

```

```
int k = temp.data;
temp.data = temp.next.data;
temp.next.data = k;
temp = temp.next.next;
}
}

/* Utility functions */

/* Inserts a new Node at front of the list. */
public void push(int new_data)
{
    /* 1 & 2: Allocate the Node &
       Put in the data*/
    Node new_node = new Node(new_data);

    /* 3. Make next of new Node as head */
    new_node.next = head;

    /* 4. Move the head to point to new Node */
    head = new_node;
}

/* Function to print linked list */
void printList()
{
    Node temp = head;
    while (temp != null)
    {
        System.out.print(temp.data+" ");
        temp = temp.next;
    }
    System.out.println();
}

/* Driver program to test above functions */
public static void main(String args[])
{
    LinkedList llist = new LinkedList();

    /* Created Linked List 1->2->3->4->5 */
    llist.push(5);
    llist.push(4);
    llist.push(3);
    llist.push(2);
    llist.push(1);
```

```
System.out.println("Linked List before calling pairWiseSwap() ");
llist.printList();

llist.pairWiseSwap();

System.out.println("Linked List after calling pairWiseSwap() ");
llist.printList();
}

}

/* This code is contributed by Rajat Mishra */
```

### Python

```
# Python program to swap the elements of linked list pairwise

# Node class
class Node:

    # Constructor to initialize the node object
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:

    # Function to initialize head
    def __init__(self):
        self.head = None

    # Function to pairwise swap elements of a linked list
    def pairwiseSwap(self):
        temp = self.head

        # There are no nodes in linked list
        if temp is None:
            return

        # Traverse furthur only if there are at least two
        # left
        while(temp is not None and temp.next is not None):

            # Swap data of node with its next node's data
            temp.data, temp.next.data = temp.next.data, temp.data

            # Move tempo by 2 fro the next pair
            temp = temp.next.next

    # Function to insert a new node at the beginning
```

```
def push(self, new_data):
    new_node = Node(new_data)
    new_node.next = self.head
    self.head = new_node

# Utility function to print the linked LinkedList
def printList(self):
    temp = self.head
    while(temp):
        print temp.data,
        temp = temp.next

# Driver program
llist = LinkedList()
llist.push(5)
llist.push(4)
llist.push(3)
llist.push(2)
llist.push(1)

print "Linked list before calling pairWiseSwap() "
llist.printList()

llist.pairwiseSwap()

print "\nLinked list after calling pairWiseSwap()"
llist.printList()

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

Output:

```
Linked List before calling pairWiseSwap()
1 2 3 4 5
Linked List after calling pairWiseSwap()
2 1 4 3 5
```

Time complexity: O(n)

#### METHOD 2 (Recursive)

If there are 2 or more than 2 nodes in Linked List then swap the first two nodes and recursively call for rest of the list.

```
/* Recursive function to pairwise swap elements of a linked list */
```

```
void pairWiseSwap(struct node *head)
{
    /* There must be at-least two nodes in the list */
    if (head != NULL && head->next != NULL)
    {
        /* Swap the node's data with data of next node */
        swap(&head->data, &head->next->data);

        /* Call pairWiseSwap() for rest of the list */
        pairWiseSwap(head->next->next);
    }
}
```

Time complexity:  $O(n)$

The solution provided there swaps data of nodes. If data contains many fields, there will be many swap operations. See [this](#)for an implementation that changes links rather than swapping data.

## Source

<https://www.geeksforgeeks.org/pairwise-swap-elements-of-a-given-linked-list/>

## Chapter 166

# Pairwise swap elements of a given linked list by changing links

Pairwise swap elements of a given linked list by changing links - GeeksforGeeks

Given a singly linked list, write a function to swap elements pairwise. For example, if the linked list is 1->2->3->4->5->6->7 then the function should change it to 2->1->4->3->6->5->7, and if the linked list is 1->2->3->4->5->6 then the function should change it to 2->1->4->3->6->5

This problem has been discussed [here](#). The solution provided there swaps data of nodes. If data contains many fields, there will be many swap operations. So changing links is a better idea in general. Following is a C implementation that changes links instead of swapping data.

C

```
/* This program swaps the nodes of linked list rather than swapping the
field from the nodes.
Imagine a case where a node contains many fields, there will be plenty
of unnecessary swap calls. */

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

/* A linked list node */
struct Node
{
    int data;
    struct Node *next;
}
```

```
};

/* Function to pairwise swap elements of a linked list */
void pairWiseSwap(struct Node **head)
{
    // If linked list is empty or there is only one node in list
    if (*head == NULL || (*head)->next == NULL)
        return;

    // Initialize previous and current pointers
    struct Node *prev = *head;
    struct Node *curr = (*head)->next;

    *head = curr; // Change head before proceeding

    // Traverse the list
    while (true)
    {
        struct Node *next = curr->next;
        curr->next = prev; // Change next of current as previous node

        // If next NULL or next is the last node
        if (next == NULL || next->next == NULL)
        {
            prev->next = next;
            break;
        }

        // Change next of previous to next next
        prev->next = next->next;

        // Update previous and curr
        prev = next;
        curr = prev->next;
    }
}

/* Function to add a node at the begining of Linked List */
void push(struct Node** head_ref, int new_data)
{
    /* allocate node */
    struct Node* new_node =
        (struct Node*) malloc(sizeof(struct Node));

    /* put in the data */
    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;
}

/* Function to print nodes in a given linked list */
void printList(struct Node *node)
{
    while(node != NULL)
    {
        printf("%d ", node->data);
        node = node->next;
    }
}

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

    /* The constructed linked list is:
       1->2->3->4->5->6->7 */
    push(&start, 7);
    push(&start, 6);
    push(&start, 5);
    push(&start, 4);
    push(&start, 3);
    push(&start, 2);
    push(&start, 1);

    printf("\n Linked list before calling pairWiseSwap() ");
    printList(start);

    pairWiseSwap(&start);

    printf("\n Linked list after calling pairWiseSwap() ");
    printList(start);

    getchar();
    return 0;
}
```

**Java**

```
// Java program to swap elements of linked list by changing links

class LinkedList {
```

```
static Node head;

static class Node {

    int data;
    Node next;

    Node(int d) {
        data = d;
        next = null;
    }
}

/* Function to pairwise swap elements of a linked list */
Node pairWiseSwap(Node node) {

    // If linked list is empty or there is only one node in list
    if (node == null || node.next == null) {
        return node;
    }

    // Initialize previous and current pointers
    Node prev = node;
    Node curr = node.next;

    node = curr; // Change head before proceeding

    // Traverse the list
    while (true) {
        Node next = curr.next;
        curr.next = prev; // Change next of current as previous node

        // If next NULL or next is the last node
        if (next == null || next.next == null) {
            prev.next = next;
            break;
        }

        // Change next of previous to next next
        prev.next = next.next;

        // Update previous and curr
        prev = next;
        curr = prev.next;
    }
    return node;
}
```

```
/* Function to print nodes in a given linked list */
void printList(Node node) {
    while (node != null) {
        System.out.print(node.data + " ");
        node = node.next;
    }
}

// Driver program to test above functions
public static void main(String[] args) {

    /* The constructed linked list is:
     * 1->2->3->4->5->6->7 */
    LinkedList list = new LinkedList();
    list.head = new Node(1);
    list.head.next = new Node(2);
    list.head.next.next = new Node(3);
    list.head.next.next.next = new Node(4);
    list.head.next.next.next.next = new Node(5);
    list.head.next.next.next.next.next = new Node(6);
    list.head.next.next.next.next.next.next = new Node(7);

    System.out.println("Linked list before calling pairwiseSwap() ");
    list.printList(head);
    Node st = list.pairWiseSwap(head);
    System.out.println("");
    System.out.println("Linked list after calling pairwiseSwap() ");
    list.printList(st);
    System.out.println("");

}

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

Output:

```
Linked list before calling pairWiseSwap() 1 2 3 4 5 6 7
Linked list after calling pairWiseSwap() 2 1 4 3 6 5 7
```

Time Complexity: Time complexity of the above program is  $O(n)$  where  $n$  is the number of nodes in a given linked list. The while loop does a traversal of the given linked list.

Following is **recursive implementation** of the same approach. We change first two nodes and recur for the remaining list. Thanks to geek and omer salem for suggesting this method.

C

```
/* This program swaps the nodes of linked list rather than swapping the
field from the nodes.
Imagine a case where a node contains many fields, there will be plenty
of unnecessary swap calls. */

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

/* A linked list node */
struct node
{
    int data;
    struct node *next;
};

/* Function to pairwise swap elements of a linked list.
   It returns head of the modified list, so return value
   of this node must be assigned */
struct node *pairWiseSwap(struct node* head)
{
    // Base Case: The list is empty or has only one node
    if (head == NULL || head->next == NULL)
        return head;

    // Store head of list after two nodes
    struct node* remaing = head->next->next;

    // Change head
    struct node* newhead = head->next;

    // Change next of second node
    head->next->next = head;

    // Recur for remaining list and change next of head
    head->next = pairWiseSwap(remaing);

    // Return new head of modified list
    return newhead;
}

/* Function to add a node at the begining of Linked List */
void push(struct node** head_ref, int new_data)
{
    /* allocate node */
    struct node* new_node =
        (struct node*) malloc(sizeof(struct node));
    new_node->data = new_data;
    new_node->next = (*head_ref);
    (*head_ref) = new_node;
}
```

```
/* put in the data */
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;
}

/* Function to print nodes in a given linked list */
void printList(struct node *node)
{
    while(node != NULL)
    {
        printf("%d ", node->data);
        node = node->next;
    }
}

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

    /* The constructed linked list is:
       1->2->3->4->5->6->7 */
    push(&start, 7);
    push(&start, 6);
    push(&start, 5);
    push(&start, 4);
    push(&start, 3);
    push(&start, 2);
    push(&start, 1);

    printf("\n Linked list before calling pairWiseSwap() ");
    printList(start);

    start = pairWiseSwap(start); // NOTE THIS CHANGE

    printf("\n Linked list after calling pairWiseSwap() ");
    printList(start);

    return 0;
}
```

Java

```
// Java program to swap elements of linked list by changing links

class LinkedList {

    static Node head;

    static class Node {

        int data;
        Node next;

        Node(int d) {
            data = d;
            next = null;
        }
    }

    /* Function to pairwise swap elements of a linked list.
     * It returns head of the modified list, so return value
     * of this node must be assigned */
    Node pairWiseSwap(Node node) {

        // Base Case: The list is empty or has only one node
        if (node == null || node.next == null) {
            return node;
        }

        // Store head of list after two nodes
        Node remaing = node.next.next;

        // Change head
        Node newhead = node.next;

        // Change next of second node
        node.next.next = node;

        // Recur for remaining list and change next of head
        node.next = pairWiseSwap(remaing);

        // Return new head of modified list
        return newhead;
    }

    /* Function to print nodes in a given linked list */
    void printList(Node node) {
        while (node != null) {
            System.out.print(node.data + " ");
            node = node.next;
        }
    }
}
```

```
        }
    }

// Driver program to test above functions
public static void main(String[] args) {

    /* The constructed linked list is:
     * 1->2->3->4->5->6->7 */
    LinkedList list = new LinkedList();
    list.head = new Node(1);
    list.head.next = new Node(2);
    list.head.next.next = new Node(3);
    list.head.next.next.next = new Node(4);
    list.head.next.next.next.next = new Node(5);
    list.head.next.next.next.next.next = new Node(6);
    list.head.next.next.next.next.next.next = new Node(7);

    System.out.println("Linked list before calling pairwiseSwap() ");
    list.printList(head);
    head = list.pairWiseSwap(head);
    System.out.println("");
    System.out.println("Linked list after calling pairwiseSwap() ");
    list.printList(head);
    System.out.println("");

}
}
```

```
Linked list before calling pairWiseSwap() 1 2 3 4 5 6 7
Linked list after calling pairWiseSwap() 2 1 4 3 6 5 7
```

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

Improved By : [Rishabh Jindal 2](#)

## Source

<https://www.geeksforgeeks.org/pairwise-swap-elements-of-a-given-linked-list-by-changing-links/>

## Chapter 167

# Partitioning a linked list around a given value and If we don't care about making the elements of the list “stable”

Partitioning a linked list around a given value and If we don't care about making the elements of the list “stable” - GeeksforGeeks

Given a linked list and a value x, partition a linked list around a value x, such that all nodes less than x come before all nodes greater than or equal to x. If x is contained within the list the values of x only need to be after the elements less than x (see below). The partition element x can appear anywhere in the “right partition”; it does not need to appear between the left and right partitions.

Similar problem : [Partitioning a linked list around a given value and keeping the original order](#)

Examples:

```
Input : 3 -> 5 -> 10 -> 2 -> 8 -> 2 -> 1  
       x = 5  
Output : 1-> 2-> 2-> 3-> 5-> 10-> 8
```

If we don't care about making the elements of the list “stable” then we can instead rearrange the elements by growing the list at the head and tail.

In this approach, we start a “new” list (using the existing nodes). Elements bigger than the pivot element are put at the tail and elements smaller are put at the head. Each time we insert an element, we update either the head or tail.

Below is C++ implementation of above idea.

```
// C++ program to partition a linked list around a
// given value.
#include<bits/stdc++.h>
using namespace std;

/* Link list Node */
struct Node
{
    int data;
    struct Node* next;
};

// A utility function to create a new node
Node *newNode(int data)
{
    struct Node* new_node = new Node;
    new_node->data = data;
    new_node->next = NULL;
    return new_node;
}

// Function to make a new list(using the existing
// nodes) and return head of new list.
struct Node *partition(struct Node *head, int x)
{
    /* Let us initialize start and tail nodes of
    new list */
    struct Node *tail = head;

    // Now iterate original list and connect nodes
    Node *curr = head;
    while (curr != NULL)
    {
        struct Node *next = curr->next;
        if (curr->data < x)
        {
            /* Insert node at head. */
            curr->next = head;
            head = curr;
        }
        else // Append to the list of greater values
        {
            /* Insert node at tail. */
            tail->next = curr;
            tail = curr;
        }
        curr = next;
    }
}
```

```
}

tail->next = NULL;

// The head has changed, so we need
// to return it to the user.
return head;
}

/* Function to print linked list */
void printList(struct Node *head)
{
    struct Node *temp = head;
    while (temp != NULL)
    {
        printf("%d ", temp->data);
        temp = temp->next;
    }
}

// Driver program to run the case
int main()
{
    /* Start with the empty list */
    struct Node* head = newNode(3);
    head->next = newNode(5);
    head->next->next = newNode(8);
    head->next->next->next = newNode(2);
    head->next->next->next->next = newNode(10);
    head->next->next->next->next->next = newNode(2);
    head->next->next->next->next->next = newNode(1);

    int x = 5;
    head = partition(head, x);
    printList(head);
    return 0;
}
```

Output:

```
1 2 2 3 5 8 10
```

## Source

<https://www.geeksforgeeks.org/partitioning-linked-list-around-given-value-dont-care-making-elements-list-stable/>

## Chapter 168

# Partitioning a linked list around a given value and keeping the original order

Partitioning a linked list around a given value and keeping the original order - GeeksforGeeks

Given a linked list and a value x, partition it such that all nodes less than x come first, then all nodes with value equal to x and finally nodes with value greater than or equal to x. The original relative order of the nodes in each of the three partitions should be preserved. The partition must work in-place.

Examples:

```
Input : 1->4->3->2->5->2->3,  
       x = 3  
Output: 1->2->2->3->3->4->5
```

```
Input : 1->4->2->10  
       x = 3  
Output: 1->2->4->10
```

```
Input : 10->4->20->10->3  
       x = 3  
Output: 3->10->4->20->10
```

To solve this problem we can use **partition method** of [Quick Sort](#) but this would not preserve the original relative order of the nodes in each of the two partitions.

Below is the algorithm to solve this problem :

- Initialize first and last nodes of below three linked lists as NULL.

1. Linked list of values smaller than x.
  2. Linked list of values equal to x.
  3. Linked list of values greater than x.
- Now iterate through the original linked list. If a node's value is less than x then append it at the end of smaller list. If the value is equal to x, then at the end of equal list. And if value is greater, then at the end of greater list.
  - Now concatenate three lists.

Below is C++ implementation of above idea.

```
// C++ program to partition a linked list around a
// given value.
#include<bits/stdc++.h>
using namespace std;

/* Link list Node */
struct Node
{
    int data;
    struct Node* next;
};

// A utility function to create a new node
Node *newNode(int data)
{
    struct Node* new_node = new Node;
    new_node->data = data;
    new_node->next = NULL;
    return new_node;
}

// Function to make two separate lists and return
// head after concatenating
struct Node *partition(struct Node *head, int x)
{
    /* Let us initialize first and last nodes of
       three linked lists
       1) Linked list of values smaller than x.
       2) Linked list of values equal to x.
       3) Linked list of values greater than x.*/
    struct Node *smallerHead = NULL, *smallerLast = NULL;
    struct Node *greaterLast = NULL, *greaterHead = NULL;
    struct Node *equalHead = NULL, *equalLast = NULL;

    // Now iterate original list and connect nodes
    // of appropriate linked lists.
    while (head != NULL)
```

```
{  
    // If current node is equal to x, append it  
    // to the list of x values  
    if (head->data == x)  
    {  
        if (equalHead == NULL)  
            equalHead = equalLast = head;  
        else  
        {  
            equalLast->next = head;  
            equalLast = equalLast->next;  
        }  
    }  
  
    // If current node is less than X, append  
    // it to the list of smaller values  
    else if (head->data < x)  
    {  
        if (smallerHead == NULL)  
            smallerLast = smallerHead = head;  
        else  
        {  
            smallerLast->next = head;  
            smallerLast = head;  
        }  
    }  
    else // Append to the list of greater values  
    {  
        if (greaterHead == NULL)  
            greaterLast = greaterHead = head;  
        else  
        {  
            greaterLast->next = head;  
            greaterLast = head;  
        }  
    }  
  
    head = head->next;  
}  
  
// Fix end of greater linked list to NULL if this  
// list has some nodes  
if (greaterLast != NULL)  
    greaterLast->next = NULL;  
  
// Connect three lists  
  
// If smaller list is empty
```

```
if (smallerHead == NULL)
{
    if (equalHead == NULL)
        return greaterHead;
    equalLast->next = greaterHead;
    return equalHead;
}

// If smaller list is not empty
// and equal list is empty
if (equalHead == NULL)
{
    smallerLast->next = greaterHead;
    return smallerHead;
}

// If both smaller and equal list
// are non-empty
smallerLast->next = equalHead;
equalLast->next = greaterHead;
return smallerHead;
}

/* Function to print linked list */
void printList(struct Node *head)
{
    struct Node *temp = head;
    while (temp != NULL)
    {
        printf("%d ", temp->data);
        temp = temp->next;
    }
}

// Driver program to run the case
int main()
{
    /* Start with the empty list */
    struct Node* head = newNode(10);
    head->next = newNode(4);
    head->next->next = newNode(5);
    head->next->next->next = newNode(30);
    head->next->next->next->next = newNode(2);
    head->next->next->next->next->next = newNode(50);

    int x = 3;
    head = partition(head, x);
    printList(head);
```

*Chapter 168. Partitioning a linked list around a given value and keeping the original order*

---

```
    return 0;  
}
```

Output:

```
2 10 4 5 30 50
```

**Source**

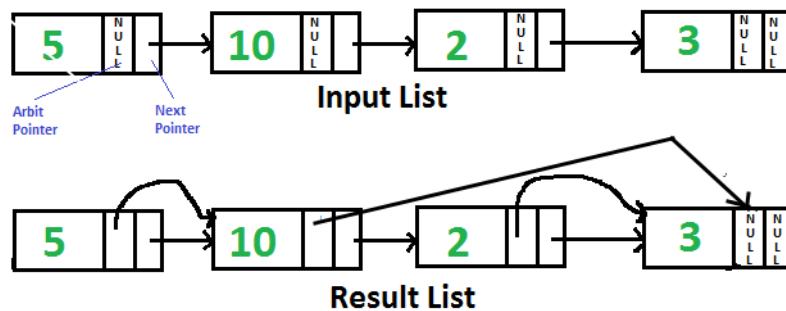
<https://www.geeksforgeeks.org/partitioning-a-linked-list-around-a-given-value-and-keeping-the-original-order/>

## Chapter 169

# Point arbit pointer to greatest value right side node in a linked list

Point arbit pointer to greatest value right side node in a linked list - GeeksforGeeks

Given singly linked list with every node having an additional “arbitrary” pointer that currently points to NULL. We need to make the “arbitrary” pointer to greatest value node in a linked list on its right side.



A **Simple Solution** is to traverse all nodes one by one. For every node, find the node which has greatest value on right side and change the next pointer. Time Complexity of this solution is  $O(n^2)$ .

An **Efficient Solution** can work in  $O(n)$  time. Below are steps.

1. Reverse given linked list.
2. Start traversing linked list and store maximum value node encountered so far. Make arbit of every node to point to max. If the data in current node is more than max node so far, update max.
3. Reverse modified linked list and return head.

Following is C++ implementation of above steps.

```
// C++ program to point arbit pointers to highest
// value on its right
#include<bits/stdc++.h>
using namespace std;

/* Link list node */
struct Node
{
    int data;
    Node* next, *arbit;
};

/* Function to reverse the linked list */
Node* reverse(Node *head)
{
    Node *prev = NULL, *current = head, *next;
    while (current != NULL)
    {
        next = current->next;
        current->next = prev;
        prev = current;
        current = next;
    }
    return prev;
}

// This function populates arbit pointer in every
// node to the greatest value to its right.
Node* populateArbit(Node *head)
{
    // Reverse given linked list
    head = reverse(head);

    // Initialize pointer to maximum value node
    Node *max = head;

    // Traverse the reversed list
    Node *temp = head->next;
    while (temp != NULL)
    {
        // Connect max through arbit pointer
        temp->arbit = max;

        // Update max if required
        if (max->data < temp->data)
            max = temp;
    }
}
```

```
// Move ahead in reversed list
temp = temp->next;
}

// Reverse modified linked list and return
// head.
return reverse(head);
}

// Utility function to print result linked list
void printNextArbitPointers(Node *node)
{
    printf("Node\tNext Pointer\tArbit Pointer\n");
    while (node!=NULL)
    {
        cout << node->data << "\t\t";

        if (node->next)
            cout << node->next->data << "\t\t";
        else cout << "NULL" << "\t\t";

        if (node->arbit)
            cout << node->arbit->data;
        else cout << "NULL";

        cout << endl;
        node = node->next;
    }
}

/* Function to create a new node with given data */
Node *newNode(int data)
{
    Node *new_node = new Node;
    new_node->data = data;
    new_node->next = NULL;
    return new_node;
}

/* Driver program to test above functions*/
int main()
{
    Node *head = newNode(5);
    head->next = newNode(10);
    head->next->next = newNode(2);
    head->next->next->next = newNode(3);
```

```
head = populateArbit(head);

printf("Resultant Linked List is: \n");
printNextArbitPointers(head);

return 0;
}
```

Output:

```
Resultant Linked List is:
Node    Next Pointer    Arbit Pointer
5        10             10
10       2              3
2        3              3
3        NULL            NULL
```

### Recursive Solution:

We can recursively reach the last node and traverse the linked list from end. Recursive solution doesn't require reversing of linked list. We can also use a stack in place of recursion to temporarily hold nodes. Thanks to Santosh Kumar Mishra for providing this solution.

```
// C++ program to point arbit pointers to highest
// value on its right
#include<bits/stdc++.h>
using namespace std;

/* Link list node */
struct Node
{
    int data;
    Node* next, *arbit;
};

// This function populates arbit pointer in every
// node to the greatest value to its right.
void populateArbit(Node *head)
{
    // using static maxNode to keep track of maximum
    // orbit node address on right side
    static Node *maxNode;

    // if head is null simply return the list
    if (head == NULL)
        return;
```

```
/* if head->next is null it means we reached at
   the last node just update the max and maxNode */
if (head->next == NULL)
{
    maxNode = head;
    return;
}

/* Calling the populateArbit to the next node */
populateArbit(head->next);

/* updating the arbit node of the current
   node with the maximum value on the right side */
head->arbit = maxNode;

/* if current Node value id greater then
   the previous right node then update it */
if (head->data > maxNode->data)
    maxNode = head;

return;
}

// Utility function to print result linked list
void printNextArbitPointers(Node *node)
{
    printf("Node\tNext Pointer\tArbit Pointer\n");
    while (node!=NULL)
    {
        cout << node->data << "\t\t";

        if(node->next)
            cout << node->next->data << "\t\t";
        else cout << "NULL" << "\t\t";

        if(node->arbit)
            cout << node->arbit->data;
        else cout << "NULL";

        cout << endl;
        node = node->next;
    }
}

/* Function to create a new node with given data */
Node *newNode(int data)
{
```

```
Node *new_node = new Node;
new_node->data = data;
new_node->next = NULL;
return new_node;
}

/* Driver program to test above functions*/
int main()
{
    Node *head = newNode(5);
    head->next = newNode(10);
    head->next->next = newNode(2);
    head->next->next->next = newNode(3);

    populateArbit(head);

    printf("Resultant Linked List is: \n");
    printNextArbitPointers(head);

    return 0;
}
```

Output:

```
Resultant Linked List is:
Node      Next Pointer      Arbit Pointer
5          10                10
10         2                  3
2          3                  3
3          NULL               NULL
```

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

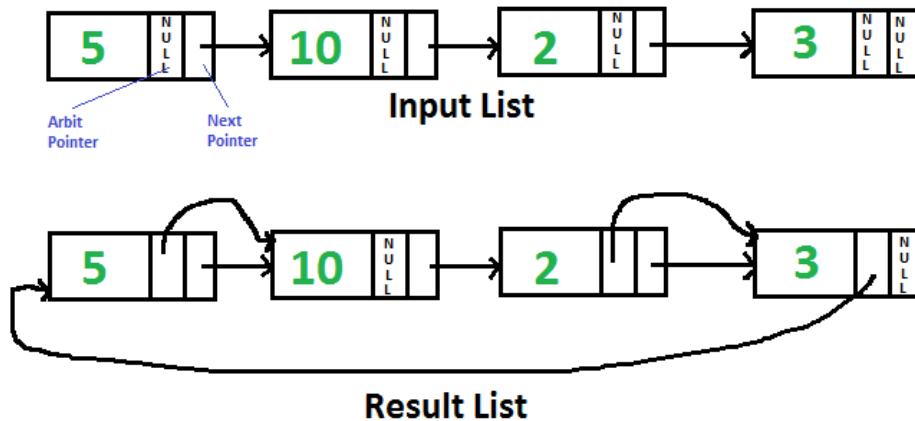
<https://www.geeksforgeeks.org/point-arbit-pointer-greatest-value-right-side-node-linked-list/>

## Chapter 170

# Point to next higher value node in a linked list with an arbitrary pointer

Point to next higher value node in a linked list with an arbitrary pointer - GeeksforGeeks

Given singly linked list with every node having an additional “arbitrary” pointer that currently points to NULL. Need to make the “arbitrary” pointer point to the next higher value node.



We strongly recommend to minimize your browser and try this yourself first

A **Simple Solution** is to traverse all nodes one by one, for every node, find the node which has next greater value of the current node and changes the next pointer. Time Complexity of this solution is  $O(n^2)$ .

An **Efficient Solution** works in  $O(n \log n)$  time. The idea is to use [Merge Sort for linked list](#).

- 1) Traverse input list and copy next pointer to arbit pointer for every node.
- 2) Do Merge Sort for the linked list formed by arbit pointers.

Below is C implementation of the above idea. All of the merger sort functions are taken from [here](#). The taken functions are modified here so that they work on arbit pointers instead of next pointers.

```
// C program to populate arbit pointers to next higher value
// using merge sort
#include<stdio.h>
#include<stdlib.h>

/* Link list node */
struct Node
{
    int data;
    struct Node* next, *arbit;
};

/* function prototypes */
struct Node* SortedMerge(struct Node* a, struct Node* b);
void FrontBackSplit(struct Node* source,
                    struct Node** frontRef, struct Node** backRef);

/* sorts the linked list formed by arbit pointers
(does not change next pointer or data) */
void MergeSort(struct Node** headRef)
{
    struct Node* head = *headRef;
    struct Node* a, *b;

    /* Base case -- length 0 or 1 */
    if ((head == NULL) || (head->arbit == NULL))
        return;

    /* Split head into 'a' and 'b' sublists */
    FrontBackSplit(head, &a, &b);

    /* Recursively sort the sublists */
    MergeSort(&a);
    MergeSort(&b);

    /* answer = merge the two sorted lists together */
    *headRef = SortedMerge(a, b);
}

/* See https://www.geeksforgeeks.org/?p=3622 for details of this
function */
struct Node* SortedMerge(struct Node* a, struct Node* b)
```

```
{  
    struct Node* result = NULL;  
  
    /* Base cases */  
    if (a == NULL)  
        return (b);  
    else if (b==NULL)  
        return (a);  
  
    /* Pick either a or b, and recur */  
    if (a->data <= b->data)  
    {  
        result = a;  
        result->arbit = SortedMerge(a->arbit, b);  
    }  
    else  
    {  
        result = b;  
        result->arbit = SortedMerge(a, b->arbit);  
    }  
  
    return (result);  
}  
  
/* Split the nodes of the given list into front and back halves,  
   and return the two lists using the reference parameters.  
   If the length is odd, the extra node should go in the front list.  
   Uses the fast/slow pointer strategy. */  
void FrontBackSplit(struct Node* source,  
                    struct Node** frontRef, struct Node** backRef)  
{  
    struct Node* fast, *slow;  
  
    if (source==NULL || source->arbit==NULL)  
    {  
        /* length < 2 cases */  
        *frontRef = source;  
        *backRef = NULL;  
        return;  
    }  
  
    slow = source, fast = source->arbit;  
  
    /* Advance 'fast' two nodes, and advance 'slow' one node */  
    while (fast != NULL)  
    {  
        fast = fast->arbit;  
        if (fast != NULL)
```

```
{  
    slow = slow->arbit;  
    fast = fast->arbit;  
}  
}  
  
/* 'slow' is before the midpoint in the list, so split it in two  
   at that point. */  
*frontRef = source;  
*backRef = slow->arbit;  
slow->arbit = NULL;  
}  
  
/* Function to insert a node at the beginning of the linked list */  
void push(struct Node** head_ref, int new_data)  
{  
    /* allocate node */  
    struct Node* new_node =  
        (struct Node*) malloc(sizeof(struct Node));  
  
    /* put in the data */  
    new_node->data = new_data;  
  
    /* link the old list off the new node */  
    new_node->next = (*head_ref);  
  
    new_node->arbit = NULL;  
  
    /* move the head to point to the new node */  
    (*head_ref) = new_node;  
}  
  
// Utility function to print result linked list  
void printListAfter(struct Node *node, struct Node *anode)  
{  
    printf("Traversal using Next Pointer\n");  
    while (node!=NULL)  
    {  
        printf("%d, ", node->data);  
        node = node->next;  
    }  
  
    printf("\nTraversal using Arbit Pointer\n");  
    while (anode!=NULL)  
    {  
        printf("%d, ", anode->data);  
        anode = anode->arbit;  
    }  
}
```

```
}

// This function populates arbit pointer in every node to the
// next higher value. And returns pointer to the node with
// minimum value
struct Node* populateArbit(struct Node *head)
{
    // Copy next pointers to arbit pointers
    struct Node *temp = head;
    while (temp != NULL)
    {
        temp->arbit = temp->next;
        temp = temp->next;
    }

    // Do merge sort for arbitrary pointers
    MergeSort(&head);

    // Return head of arbitrary pointer linked list
    return head;
}

/* Drier program to test above functions*/
int main()
{
    /* Start with the empty list */
    struct Node* head = NULL;

    /* Let us create the list shown above */
    push(&head, 3);
    push(&head, 2);
    push(&head, 10);
    push(&head, 5);

    /* Sort the above created Linked List */
    struct Node *ahead = populateArbit(head);

    printf("\nResult Linked List is: \n");
    printListAfter(head, ahead);

    getchar();
    return 0;
}
```

Output:

Result Linked List is:

Traversal using Next Pointer

5, 10, 2, 3,

Traversal using Arbit Pointer

2, 3, 5, 10,

This article is contributed by **Saurabh Bansal**. 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/point-to-next-higher-value-node-in-a-linked-list-with-an-arbitrary-pointer/>

## Chapter 171

# Practice questions for Linked List and Recursion

Practice questions for Linked List and Recursion - GeeksforGeeks

Assume the structure of a Linked List node is as follows.

```
struct Node
{
    int data;
    struct Node *next;
};
```

Explain the functionality of following C functions.

**1. What does the following function do for a given Linked List?**

```
void fun1(struct Node* head)
{
    if(head == NULL)
        return;

    fun1(head->next);
    printf("%d ", head->data);
}
```

fun1() prints the given Linked List in reverse manner. For Linked List 1->2->3->4->5, fun1() prints 5->4->3->2->1.

**2. What does the following function do for a given Linked List ?**

```
void fun2(struct Node* head)
```

```
{  
    if(head== NULL)  
        return;  
    printf("%d  ", head->data);  
  
    if(head->next != NULL )  
        fun2(head->next->next);  
    printf("%d  ", head->data);  
}
```

fun2() prints alternate nodes of the given Linked List, first from head to end, and then from end to head. If Linked List has even number of nodes, then fun2() skips the last node. For Linked List 1->2->3->4->5, fun2() prints 1 3 5 5 3 1. For Linked List 1->2->3->4->5->6, fun2() prints 1 3 5 5 3 1.

Below is a complete running program to test above functions.

```
#include<stdio.h>  
#include<stdlib.h>  
  
/* A linked list node */  
struct Node  
{  
    int data;  
    struct Node *next;  
};  
  
/* Prints a linked list in reverse manner */  
void fun1(struct Node* head)  
{  
    if(head == NULL)  
        return;  
  
    fun1(head->next);  
    printf("%d  ", head->data);  
}  
  
/* prints alternate nodes of a Linked List, first  
   from head to end, and then from end to head. */  
void fun2(struct Node* start)  
{  
    if(start == NULL)  
        return;  
    printf("%d  ", start->data);  
  
    if(start->next != NULL )  
        fun2(start->next->next);
```

```
    printf("%d  ", start->data);
}

/* UTILITY FUNCTIONS TO TEST fun1() and fun2() */
/* Given a reference (pointer to pointer) to the head
   of a list and an int, push a new node on the front
   of the list. */
void push(struct Node** head_ref, int new_data)
{
    /* allocate node */
    struct Node* new_node =
        (struct Node*) malloc(sizeof(struct Node));

    /* put in the data */
    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;
}

/* Driver program to test above functions */
int main()
{
    /* Start with the empty list */
    struct Node* head = NULL;

    /* Using push() to construct below list
       1->2->3->4->5 */
    push(&head, 5);
    push(&head, 4);
    push(&head, 3);
    push(&head, 2);
    push(&head, 1);

    printf("\n Output of fun1() for list 1->2->3->4->5 \n");
    fun1(head);

    printf("\n Output of fun2() for list 1->2->3->4->5 \n");
    fun2(head);

    getchar();
    return 0;
}
```

**Source**

<https://www.geeksforgeeks.org/practice-questions-for-linked-list-and-recursion/>

## Chapter 172

# Print Reverse a linked list using Stack

Print Reverse a linked list using Stack - GeeksforGeeks

Given a linked list, print reverse of it without modifying the list.

Examples:

Input : 1 2 3 4 5 6

Output : 6 5 4 3 2 1

Input : 12 23 34 45 56 67 78

Output : 78 67 56 45 34 23 12

Given a [Linked List](#), display the linked list in reverse without using recursion, stack or modifications to given list.

Examples:

Input : 1->2->3->4->5->NULL

Output : 5->4->3->2->1->NULL

Input : 10->5->15->20->24->NULL

Output : 24->20->15->5->10->NULL

Below are different solutions that are now allowed here as we cannot use extra space and modify list.

- 1) [Recursive solution to print reverse a linked list](#). Requires extra space.

- 2) [Reverse linked list](#) and then print. This requires modifications to original list.
- 3) [A O\(n<sup>2</sup>\) solution to print reverse of linked list](#) that first count nodes and then prints k-th node from end.

In this post, an efficient stack based solution is discussed.

1. First insert all the element in stack
2. Print stack till stack is not empty

Note: Instead of inserting data from each node into the stack, insert the node's address onto the stack. This is because the size of the node's data will be generally more than the size of the node's address. Thus the stack would end up requiring more memory if it directly stored the data elements. Also, we cannot insert the node's data onto the stack if each node contained more than one data member. Hence the simpler and efficient solution would be to simply insert the node's address.

```
// C/C++ program to print reverse of linked list
// using stack.
#include<bits/stdc++.h>
using namespace std;

/* Link list node */
struct Node
{
    int data;
    struct Node* next;
};

/* Given a reference (pointer to pointer) to the head
   of a list and an int, push a new node on the front
   of the list. */
void push(struct Node** head_ref, int new_data)
{
    struct Node* new_node =
        (struct Node*) malloc(sizeof(struct Node));
    new_node->data = new_data;
    new_node->next = (*head_ref);
    (*head_ref) = new_node;
}

/* Counts no. of nodes in linked list */
int getCount(struct Node* head)
{
    int count = 0; // Initialize count
    struct Node* current = head; // Initialize current
    while (current != NULL)
    {
        count++;
        current = current->next;
    }
}
```

```
        }
        return count;
    }

/* Takes head pointer of the linked list and index
   as arguments and return data at index*/
int getNth(struct Node* head, int n)
{
    struct Node* curr = head;
    for (int i=0; i<n-1 && curr != NULL; i++)
        curr = curr->next;
    return curr->data;
}

void printReverse(Node *head)
{
    // store Node addresses in stack
    stack<Node *> stk;
    Node* ptr = head;
    while (ptr != NULL)
    {
        stk.push(ptr);
        ptr = ptr->next;
    }

    // print data from stack
    while (!stk.empty())
    {
        cout << stk.top()->data << " ";
        stk.pop(); // pop after print
    }
    cout << "\n";
}

/* Drier program to test count function*/
int main()
{
    /* Start with the empty list */
    struct Node* head = NULL;

    /* Use push() to construct below list
       1->2->3->4->5 */
    push(&head, 5);
    push(&head, 4);
    push(&head, 3);
    push(&head, 2);
    push(&head, 1);
```

```
    printReverse(head);  
  
    return 0;  
}
```

Output:

5 4 3 2 1

**Improved By :** [Sayan Mahapatra](#)

### Source

<https://www.geeksforgeeks.org/print-reverse-linked-list-using-stack/>

## Chapter 173

# Print alternate nodes of a linked list using recursion

Print alternate nodes of a linked list using recursion - GeeksforGeeks

Given a linked list, print alternate nodes of this linked list.

Examples :

Input : 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9 → 10  
Output : 1 → 3 → 5 → 7 → 9

Input : 10 → 9  
Output : 10

### Recursive Approach :

1. Initialize a static variable(say flag)
2. If flag is odd print the node
3. increase head and flag by 1, and recurse for next nodes.

C++

```
// CPP code to print alternate nodes
// of a linked list using recursion
#include <bits/stdc++.h>
using namespace std;

// A linked list node
struct Node {
    int data;
    struct Node* next;
};


```

```
// Inserting node at the beginning
void push(struct Node** head_ref, int new_data)
{
    struct Node* new_node =
        (struct Node*)malloc(sizeof(struct Node));
    new_node->data = new_data;
    new_node->next = (*head_ref);
    (*head_ref) = new_node;
}

// Function to print alternate nodes of linked list.
// The boolean flag isOdd is used to find if the current
// node is even or odd.
void printAlternate(struct Node* node, bool isOdd=true)
{
    if (node == NULL)
        return;
    if (isOdd == true)
        cout << node->data << " ";
    printAlternate(node->next, !isOdd);
}

// Driver code
int main()
{
    // Start with the empty list
    struct Node* head = NULL;

    // construct below list
    // 1->2->3->4->5->6->7->8->9->10

    push(&head, 10);
    push(&head, 9);
    push(&head, 8);
    push(&head, 7);
    push(&head, 6);
    push(&head, 5);
    push(&head, 4);
    push(&head, 3);
    push(&head, 2);
    push(&head, 1);

    printAlternate(head);

    return 0;
}
```

**Output:**

1 3 5 7 9

**Source**

<https://www.geeksforgeeks.org/print-alternate-nodes-linked-list-using-recursion/>

## Chapter 174

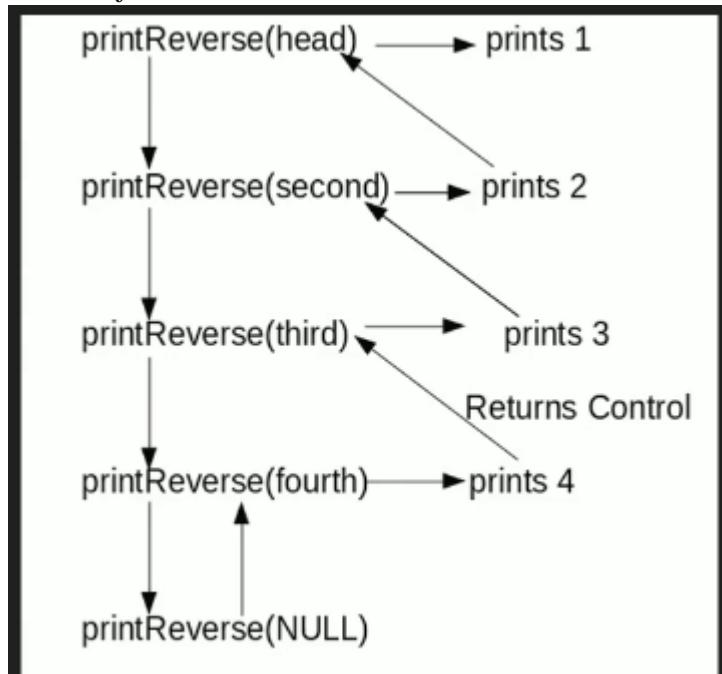
# Print reverse of a Linked List without actually reversing

Print reverse of a Linked List without actually reversing - GeeksforGeeks

Given a linked list, print reverse of it using a recursive function. For example, if the given linked list is 1->2->3->4, then output should be 4->3->2->1.

Note that the question is only about printing the reverse. To reverse the list itself see [this](#)

**Difficulty Level:** Rookie



Algorithm

```
printReverse(head)
1. call print reverse for head->next
2. print head->data
```

**Implementation:**

C

```
// C program to print reverse of a linked list
#include<stdio.h>
#include<stdlib.h>

/* Link list node */
struct Node
{
    int data;
    struct Node* next;
};

/* Function to reverse the linked list */
void printReverse(struct Node* head)
{
    // Base case
    if (head == NULL)
        return;

    // print the list after head node
    printReverse(head->next);

    // After everything else is printed, print head
    printf("%d ", head->data);
}

/*UTILITY FUNCTIONS*/
/* Push a node to linked list. Note that this function
   changes the head */
void push(struct Node** head_ref, char new_data)
{
    /* allocate node */
    struct Node* new_node =
        (struct Node*) malloc(sizeof(struct Node));

    /* put in the data */
    new_node->data = new_data;

    /* link the old list off the new node */
    new_node->next = *head_ref;
    *head_ref = new_node;
}
```

```
new_node->next = (*head_ref);

/* move the head to pochar to the new node */
(*head_ref)    = new_node;
}

/* Drier program to test above function*/
int main()
{
    // Let us create linked list 1->2->3->4
    struct Node* head = NULL;
    push(&head, 4);
    push(&head, 3);
    push(&head, 2);
    push(&head, 1);

    printReverse(head);
    return 0;
}
```

### Java

```
// Java program to print reverse of a linked list
class LinkedList
{
    Node head; // head of list

    /* Linked list Node*/
    class Node
    {
        int data;
        Node next;
        Node(int d) {data = d; next = null; }
    }

    /* Function to print reverse of linked list */
    void printReverse(Node head)
    {
        if (head == null) return;

        // print list of head node
        printReverse(head.next);

        // After everything else is printed
        System.out.print(head.data+" ");
    }

    /* Utility Functions */
```

```
/* Inserts a new Node at front of the list. */
public void push(int new_data)
{
    /* 1 & 2: Allocate the Node &
       Put in the data*/
    Node new_node = new Node(new_data);

    /* 3. Make next of new Node as head */
    new_node.next = head;

    /* 4. Move the head to point to new Node */
    head = new_node;
}

/*Drier function to test the above methods*/
public static void main(String args[])
{
    // Let us create linked list 1->2->3->4
    LinkedList llist = new LinkedList();
    llist.push(4);
    llist.push(3);
    llist.push(2);
    llist.push(1);

    llist.printReverse(llist.head);
}
}
/* This code is contributed by Rajat Mishra */
```

Output:

4 3 2 1

**Time Complexity:** O(n)

**Improved By :** Aashish Kumar 7

## Source

<https://www.geeksforgeeks.org/print-reverse-of-a-linked-list-without-actually-reversing/>

## Chapter 175

# Print reverse of a Linked List without extra space and modifications

Print reverse of a Linked List without extra space and modifications - GeeksforGeeks

Given a [Linked List](#), display the linked list in reverse without using recursion, stack or modifications to given list.

Examples:

Input : 1->2->3->4->5->NULL  
Output :5->4->3->2->1->NULL

Input :10->5->15->20->24->NULL  
Output :24->20->15->5->10->NULL

Below are different solutions that are now allowed here as we cannot use extra space and modify list.

- 1) [Recursive solution to print reverse a linked list](#). Requires extra space.
- 2) [Reverse linked list](#) and then print. This requires modifications to original list.
- 3) [Stack based solution to print linked list reverse](#). Push all nodes one by one to a stack. Then one by one pop elements from stack and print. This also requires extra space.

Algorithms:

- 1) Find n = count nodes in linked list.

2) For  $i = n$  to 1, do following.  
Print  $i$ -th node using get  $n$ -th node function

```
// C/C++ program to print reverse of linked list
// without extra space and without modifications.
#include<stdio.h>
#include<stdlib.h>

/* Link list node */
struct Node
{
    int data;
    struct Node* next;
};

/* Given a reference (pointer to pointer) to the head
   of a list and an int, push a new node on the front
   of the list. */
void push(struct Node** head_ref, int new_data)
{
    struct Node* new_node =
        (struct Node*) malloc(sizeof(struct Node));
    new_node->data = new_data;
    new_node->next = (*head_ref);
    (*head_ref) = new_node;
}

/* Counts no. of nodes in linked list */
int getCount(struct Node* head)
{
    int count = 0; // Initialize count
    struct Node* current = head; // Initialize current
    while (current != NULL)
    {
        count++;
        current = current->next;
    }
    return count;
}

/* Takes head pointer of the linked list and index
   as arguments and return data at index*/
int getNth(struct Node* head, int n)
{
    struct Node* curr = head;
    for (int i=0; i<n-1 && curr != NULL; i++)
        curr = curr->next;
    return curr->data;
```

```
}

void printReverse(Node *head)
{
    // Count nodes
    int n = getCount(head);

    for (int i=n; i>=1; i--)
        printf("%d ", getNth(head, i));
}

/* Drier program to test count function*/
int main()
{
    /* Start with the empty list */
    struct Node* head = NULL;

    /* Use push() to construct below list
     * 1->2->3->4->5 */
    push(&head, 5);
    push(&head, 4);
    push(&head, 3);
    push(&head, 2);
    push(&head, 1);

    printReverse(head);

    return 0;
}
```

Output:

5 4 3 2 1

## Source

<https://www.geeksforgeeks.org/print-reverse-linked-list-without-extra-space-modifications/>

## Chapter 176

# Print the alternate nodes of linked list (Iterative Method)

Print the alternate nodes of linked list (Iterative Method) - GeeksforGeeks

Given a linked list, print the alternate nodes of linked list.

Examples:

Input : 1 → 8 → 3 → 10 → 17 → 22 → 29 → 42

Output : 1 → 3 → 17 → 29

Alternate nodes : 1 → 3 → 17 → 29

Input : 10 → 17 → 33 → 38 → 73

Output : 10 → 33 → 73

Alternate nodes : 10 → 33 → 73

### Approach :

1. Traverse the whole linked list.
2. Set count = 0.
3. Print node when count is even.
4. Visit the next node.

C

```
// CPP code to print Alternate Nodes
#include <stdio.h>
#include <stdlib.h>

/* Link list node */
struct Node {
    int data;
```

```
    struct Node* next;
};

/* Function to get the alternate
   nodes of the linked list */
void printAlternateNode(struct Node* head)
{
    int count = 0;

    while (head != NULL) {

        // when count is even print the nodes
        if (count % 2 == 0)
            printf(" %d ", head->data);

        // count the nodes
        count++;

        // move on the next node.
        head = head->next;
    }
}

// Function to push node at head
void push(struct Node** head_ref, int new_data)
{
    struct Node* new_node =
        (struct Node*)malloc(sizeof(struct Node));
    new_node->data = new_data;
    new_node->next = (*head_ref);
    (*head_ref) = new_node;
}

// Driver code
int main()
{
    /* Start with the empty list */
    struct Node* head = NULL;

    /* Use push() function to construct
       the below list 8 -> 23 -> 11 -> 29 -> 12 */
    push(&head, 12);
    push(&head, 29);
    push(&head, 11);
    push(&head, 23);
    push(&head, 8);

    printAlternateNode(head);
```

```
    return 0;  
}
```

**Python3**

```
# Python3 code to print Alternate Nodes  
  
# Link list node  
class Node :  
  
    def __init__(self, data = None) :  
        self.data = data  
        self.next = None  
  
    # Function to push node at head  
    def push(self, data) :  
  
        new = Node(data)  
        new.next = self  
        return new  
  
    # Function to get the alternate  
    # nodes of the linked list  
    def printAlternateNode(self) :  
        head = self  
  
        while head and head.next != None :  
  
            print(head.data, end = " ")  
            head = head.next.next  
  
# Driver Code  
node = Node()  
  
# Use push() function to construct  
# the below list 8 -> 23 -> 11 -> 29 -> 12  
node = node.push(12)  
node = node.push(29)  
node = node.push(11)  
node = node.push(23)  
node = node.push(8)  
  
node.printAlternateNode()
```

**Output :**

8 11 12

**Time Complexity :**  $O(n)$

**Auxiliary Space :**  $O(1)$

**Asked in :** Govivace

# This code is contributed by 'dc\_flash'

### Source

<https://www.geeksforgeeks.org/print-alternate-nodes-linked-list-iterative-method/>

## Chapter 177

# Priority Queue using Linked List

Priority Queue using Linked List - GeeksforGeeks

Implement Priority Queue using Linked Lists.

- `push()`: This function is used to insert a new data into the queue.
- `pop()`: This function removes the element with the highest priority form the queue.
- `peek() / top()`: This function is used to get the highest priority element in the queue without removing it from the queue.

Priority Queues can be implemented using common data structures like arrays, linked-lists, heaps and binary trees.

**Prerequisites :**

[Linked Lists](#), [Priority Queues](#)

The list is so created so that the highest priority element is always at the head of the list. The list is arranged in descending order of elements based on their priority. This allow us to remove the highest priority element in  $O(1)$  time. To insert an element we must traverse the list and find the proper position to insert the node so that the overall order of the priority queue is maintained. This makes the `push()` operation takes  $O(N)$  time. The `pop()` and `peek()` operations are performed in constant time.

**Algorithm :**

`PUSH(HEAD, DATA, PRIORITY)`

Step 1: Create new node with DATA and PRIORITY

Step 2: Check if HEAD has lower priority. If true follow Steps 3-4 and end. Else goto Step 5.

Step 3: `NEW -> NEXT = HEAD`

Step 4: `HEAD = NEW`

Step 5: Set TEMP to head of the list

Step 6: While TEMP -> NEXT != NULL and TEMP -> NEXT -> PRIORITY > PRIORITY

Step 7: TEMP = TEMP -> NEXT

[END OF LOOP]

Step 8: NEW -> NEXT = TEMP -> NEXT

Step 9: TEMP -> NEXT = NEW

Step 10: End

POP(HEAD)

Step 2: Set the head of the list to the next node in the list. HEAD = HEAD -> NEXT.

Step 3: Free the node at the head of the list

Step 4: End

PEEK(HEAD):

Step 1: Return HEAD -> DATA

Step 2: End

Below is the implementation of the algorithm :

C

```
// C code to implement Priority Queue
// using Linked List
#include <stdio.h>
#include <stdlib.h>

// Node
typedef struct node {
    int data;

    // Lower values indicate higher priority
    int priority;

    struct node* next;
} Node;

// Function to Create A New Node
Node* newNode(int d, int p)
{
    Node* temp = (Node*)malloc(sizeof(Node));
    temp->data = d;
    temp->priority = p;
    temp->next = NULL;

    return temp;
}

// Return the value at head
int peek(Node** head)
```

```
{  
    return (*head)->data;  
}  
  
// Removes the element with the  
// highest priority form the list  
void pop(Node** head)  
{  
    Node* temp = *head;  
    (*head) = (*head)->next;  
    free(temp);  
}  
  
// Function to push according to priority  
void push(Node** head, int d, int p)  
{  
    Node* start = (*head);  
  
    // Create new Node  
    Node* temp = newNode(d, p);  
  
    // Special Case: The head of list has lesser  
    // priority than new node. So insert new  
    // node before head node and change head node.  
    if ((*head)->priority > p) {  
  
        // Insert New Node before head  
        temp->next = *head;  
        (*head) = temp;  
    }  
    else {  
  
        // Traverse the list and find a  
        // position to insert new node  
        while (start->next != NULL &&  
              start->next->priority < p) {  
            start = start->next;  
        }  
  
        // Either at the ends of the list  
        // or at required position  
        temp->next = start->next;  
        start->next = temp;  
    }  
}  
  
// Function to check is list is empty  
int isEmpty(Node** head)
```

```
{  
    return (*head) == NULL;  
}  
  
// Driver code  
int main()  
{  
    // Create a Priority Queue  
    // 7->4->5->6  
    Node* pq = newNode(4, 1);  
    push(&pq, 5, 2);  
    push(&pq, 6, 3);  
    push(&pq, 7, 0);  
  
    while (!isEmpty(&pq)) {  
        printf("%d ", peek(&pq));  
        pop(&pq);  
    }  
  
    return 0;  
}
```

**Output:**

7 4 5 6

**Time Complexities and Comparison with Binary Heap:**

	peek()	push()	pop()
Linked List	O(1)	O(n)	O(1)
Binary Heap	O(1)	O(Log n)	O(Log n)

**Source**

<https://www.geeksforgeeks.org/priority-queue-using-linked-list/>

## Chapter 178

# Priority Queue using doubly linked list

Priority Queue using doubly linked list - GeeksforGeeks

Given Nodes with their priority, implement a priority queue using doubly linked list.

**Prerequisite :** Priority Queue

- **push():** This function is used to insert a new data into the queue.
- **pop():** This function removes the element with the lowest priority value from the queue.
- **peek() / top():** This function is used to get the lowest priority element in the queue without removing it from the queue.

**Approach :**

1. Create a doubly linked list having fields info(hold the information of the Node), priority(hold the priority of the Node), prev(point to previous Node), next(point to next Node).
2. Insert the element and priority in the Node.
3. Arrange the Nodes in the increasing order of the priority.

Below is the implementation of above steps :

C++

```
// CPP code to implement priority
// queue using doubly linked list
#include <bits/stdc++.h>
using namespace std;

// Linked List Node
struct Node {
    int info;
```

```
int priority;
struct Node *prev, *next;
};

// Function to insert a new Node
void push(Node** fr, Node** rr, int n, int p)
{
    Node* news = (Node*)malloc(sizeof(Node));
    news->info = n;
    news->priority = p;

    // If linked list is empty
    if (*fr == NULL) {
        *fr = news;
        *rr = news;
        news->next = NULL;
    }
    else {
        // If p is less than or equal front
        // node's priority, then insert at
        // the front.
        if (p <= (*fr)->priority) {
            news->next = *fr;
            (*fr)->prev = news->next;
            *fr = news;
        }

        // If p is more rear node's priority,
        // then insert after the rear.
        else if (p > (*rr)->priority) {
            news->next = NULL;
            (*rr)->next = news;
            news->prev = (*rr)->next;
            *rr = news;
        }
    }

    // Handle other cases
    else {

        // Find position where we need to
        // insert.
        Node* start = (*fr)->next;
        while (start->priority > p)
            start = start->next;
        (start->prev)->next = news;
        news->next = start->prev;
        news->prev = (start->prev)->next;
        start->prev = news->next;
    }
}
```

```
        }
    }
}

// Return the value at rear
int peek(Node *fr)
{
    return fr->info;
}

bool isEmpty(Node *fr)
{
    return (fr == NULL);
}

// Removes the element with the
// least priority value form the list
int pop(Node** fr, Node** rr)
{
    Node* temp = *fr;
    int res = temp->info;
    (*fr) = (*fr)->next;
    free(temp);
    if (*fr == NULL)
        *rr = NULL;
    return res;
}

// Diver code
int main()
{
    Node *front = NULL, *rear = NULL;
    push(&front, &rear, 2, 3);
    push(&front, &rear, 3, 4);
    push(&front, &rear, 4, 5);
    push(&front, &rear, 5, 6);
    push(&front, &rear, 6, 7);
    push(&front, &rear, 1, 2);

    cout << pop(&front, &rear) << endl;
    cout << peek(front);

    return 0;
}
```

**Output:**

1  
2

**Related Article :**

[Priority Queue using Singly Linked List](#)

**Time Complexities and Comparison with [Binary Heap](#):**

	peek()	push()	pop()
Linked List	O(1)	O(n)	O(1)
Binary Heap	O(1)	O(Log n)	O(Log n)

**Source**

<https://www.geeksforgeeks.org/priority-queue-using-doubly-linked-list/>

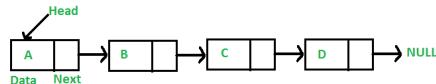
## Chapter 179

# Program for n'th node from the end of a Linked List

Program for n'th node from the end of a Linked List - GeeksforGeeks

Given a Linked List and a number n, write a function that returns the value at the n'th node from end of the Linked List.

For example, if input is below list and n = 3, then output is “B”



### Method 1 (Use length of linked list)

- 1) Calculate the length of Linked List. Let the length be len.
- 2) Print the (len - n + 1)th node from the begining of the Linked List.

C

```
// Simple C program to find n'th node from end
#include<stdio.h>
#include<stdlib.h>

/* Link list node */
struct Node
{
    int data;
    struct Node* next;
};

/* Function to get the nth node from the last of a linked list*/
void printNthFromLast(struct Node* head, int n)
{
```

```
int len = 0, i;
struct Node *temp = head;

// 1) count the number of nodes in Linked List
while (temp != NULL)
{
    temp = temp->next;
    len++;
}

// check if value of n is not more than length of the linked list
if (len < n)
    return;

temp = head;

// 2) get the (len-n+1)th node from the begining
for (i = 1; i < len-n+1; i++)
    temp = temp->next;

printf ("%d", temp->data);

return;
}

void push(struct Node** head_ref, int new_data)
{
/* allocate node */
struct Node* new_node =
    (struct Node*) malloc(sizeof(struct Node));

/* put in the data */
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;
}

/* Drier program to test above function*/
int main()
{
/* Start with the empty list */
struct Node* head = NULL;

// create linked 35->15->4->20
```

```
push(&head, 20);
push(&head, 4);
push(&head, 15);
push(&head, 35);

printNthFromLast(head, 4);
return 0;
}
```

**Java**

```
// Simple Java program to find n'th node from end of linked list
class LinkedList
{
    Node head; // head of the list

    /* Linked List node */
    class Node
    {
        int data;
        Node next;
        Node(int d)
        {
            data = d;
            next = null;
        }
    }

    /* Function to get the nth node from the last of a
       linked list */
    void printNthFromLast(int n)
    {
        int len = 0;
        Node temp = head;

        // 1) count the number of nodes in Linked List
        while (temp != null)
        {
            temp = temp.next;
            len++;
        }

        // check if value of n is not more than length of
        // the linked list
        if (len < n)
            return;

        temp = head;
```

```
// 2) get the (len-n+1)th node from the begining
for (int i = 1; i < len-n+1; i++)
    temp = temp.next;

System.out.println(temp.data);
}

/* Inserts a new Node at front of the list. */
public void push(int new_data)
{
    /* 1 & 2: Allocate the Node &
       Put in the data*/
    Node new_node = new Node(new_data);

    /* 3. Make next of new Node as head */
    new_node.next = head;

    /* 4. Move the head to point to new Node */
    head = new_node;
}

/*Drier program to test above methods */
public static void main(String [] args)
{
    LinkedList llist = new LinkedList();
    llist.push(20);
    llist.push(4);
    llist.push(15);
    llist.push(35);

    llist.printNthFromLast(4);
}
}// This code is contributed by Rajat Mishra
```

Output:

35

Following is a recursive C code for the same method. Thanks to Anuj Bansal for providing following code.

```
void printNthFromLast(struct Node* head, int n)
{
    static int i = 0;
    if (head == NULL)
        return;
```

```
    printNthFromLast(head->next, n);
    if (++i == n)
        printf("%d", head->data);
}
```

**Time Complexity:** O(n) where n is the length of linked list.

**Method 2 (Use two pointers)**

Maintain two pointers – reference pointer and main pointer. Initialize both reference and main pointers to head. First move reference pointer to n nodes from head. Now move both pointers one by one until reference pointer reaches end. Now main pointer will point to nth node from the end. Return main pointer.

**Implementation:**

C

```
// C program to find n'th node from end using slow and
// fast pointers
#include<stdio.h>
#include<stdlib.h>

/* Link list node */
struct Node
{
    int data;
    struct Node* next;
};

/* Function to get the nth node from the last of a linked list*/
void printNthFromLast(struct Node *head, int n)
{
    struct Node *main_ptr = head;
    struct Node *ref_ptr = head;

    int count = 0;
    if(head != NULL)
    {
        while( count < n )
        {
            if(ref_ptr == NULL)
            {
                printf("%d is greater than the no. of "
                       "nodes in list", n);
                return;
            }
            ref_ptr = ref_ptr->next;
            count++;
        } /* End of while*/
```

```
while(ref_ptr != NULL)
{
    main_ptr = main_ptr->next;
    ref_ptr = ref_ptr->next;
}
printf("Node no. %d from last is %d ",
       n, main_ptr->data);
}

void push(struct Node** head_ref, int new_data)
{
    /* allocate node */
    struct Node* new_node =
        (struct Node*) malloc(sizeof(struct Node));

    /* put in the data */
    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;
}

/* Drier program to test above function*/
int main()
{
    /* Start with the empty list */
    struct Node* head = NULL;
    push(&head, 20);
    push(&head, 4);
    push(&head, 15);
    push(&head, 35);

    printNthFromLast(head, 4);
}
```

### Java

```
// Java program to find  $n$ 'th node from end using slow and
// fast pointers
class LinkedList
{
    Node head; // head of the list
```

```
/* Linked List node */
class Node
{
    int data;
    Node next;
    Node(int d)
    {
        data = d;
        next = null;
    }
}

/* Function to get the nth node from end of list */
void printNthFromLast(int n)
{
    Node main_ptr = head;
    Node ref_ptr = head;

    int count = 0;
    if (head != null)
    {
        while (count < n)
        {
            if (ref_ptr == null)
            {
                System.out.println(n+" is greater than the no "+
                    " of nodes in the list");
                return;
            }
            ref_ptr = ref_ptr.next;
            count++;
        }
        while (ref_ptr != null)
        {
            main_ptr = main_ptr.next;
            ref_ptr = ref_ptr.next;
        }
        System.out.println("Node no. "+n+" from last is "+
            main_ptr.data);
    }
}

/* Inserts a new Node at front of the list. */
public void push(int new_data)
{
    /* 1 & 2: Allocate the Node &
       Put in the data*/
    Node new_node = new Node(new_data);
```

```
/* 3. Make next of new Node as head */
new_node.next = head;

/* 4. Move the head to point to new Node */
head = new_node;
}

/*Drier program to test above methods */
public static void main(String [] args)
{
    LinkedList llist = new LinkedList();
    llist.push(20);
    llist.push(4);
    llist.push(15);
    llist.push(35);

    llist.printNthFromLast(4);
}
} // This code is contributed by Rajat Mishra
```

### Python

```
# Python program to find n'th node from end using slow
# and fast pointer

# Node class
class Node:

    # Constructor to initialize the node object
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:

    # Function to initialize head
    def __init__(self):
        self.head = None

    # Function to insert a new node at the beginning
    def push(self, new_data):
        new_node = Node(new_data)
        new_node.next = self.head
        self.head = new_node

    def printNthFromLast(self, n):
        main_ptr = self.head
```

```
ref_ptr = self.head

count = 0
if(self.head is not None):
    while(count < n):
        if(ref_ptr is None):
            print "%d is greater than the no. pf nodes in list" %(n)
            return

        ref_ptr = ref_ptr.next
        count += 1

while(ref_ptr is not None):
    main_ptr = main_ptr.next
    ref_ptr = ref_ptr.next

print "Node no. %d from last is %d " %(n, main_ptr.data)

# Driver program to test above function
llist = LinkedList()
llist.push(20)
llist.push(4)
llist.push(15)
llist.push(35)

llist.printNthFromLast(4)

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

Output:

```
Node no. 4 from last is 35
```

**Time Complexity:**  $O(n)$  where  $n$  is the length of linked list.

**Improved By :** SudhirPal, anurag singh 21

## Source

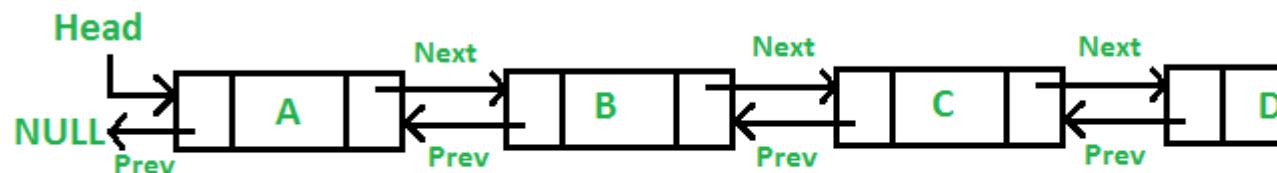
<https://www.geeksforgeeks.org/nth-node-from-the-end-of-a-linked-list/>

## Chapter 180

# Program to find size of Doubly Linked List

Program to find size of Doubly Linked List - GeeksforGeeks

Given a [doubly linked list](#), the task is to find the size of that doubly linked list. For example, size of below linked list is 4.



A doubly linked list is a linked data structure that consists of a set of sequentially linked records called nodes. Each node contains two fields, called links, that are references to the previous and to the next node in the sequence of nodes.

Traversal of a doubly linked list can be in either direction. In fact, the direction of traversal can change many times, if desired.

For example the function should return 3 for the above doubly linked list.

- 1) Initialize size to 0.
- 2) Initialize a node pointer, temp = head.
- 3) Do following while temp is not NULL
  - .....a) temp = temp -> next
  - .....b) size++;
- 4) Return size.

```
// A complete working C++ program to
// find size of doubly linked list.
#include <bits/stdc++.h>
using namespace std;

// A linked list node
struct Node
{
    int data;
    struct Node *next;
    struct Node *prev;
};

/* Function to add a node to front of doubly
   linked list */
void push(struct Node** head_ref, int new_data)
{
    struct Node* new_node = new Node;
    new_node->data = new_data;
    new_node->next = (*head_ref);
    new_node->prev = NULL;
    if ((*head_ref) != NULL)
        (*head_ref)->prev = new_node ;
    (*head_ref) = new_node;
}

// This function returns size of linked list
int findSize(struct Node *node)
{
    int res = 0;
    while (node != NULL)
    {
        res++;
        node = node->next;
    }
    return res;
}

/* Drier program to test above functions*/
int main()
{
    struct Node* head = NULL;
    push(&head, 4);
    push(&head, 3);
    push(&head, 2);
    push(&head, 1);
    cout << findSize(head);
    return 0;
}
```

}

**Output:**

4

**Source**

<https://www.geeksforgeeks.org/program-find-size-doubly-linked-list/>

## Chapter 181

# Queue based approach for first non-repeating character in a stream

Queue based approach for first non-repeating character in a stream - GeeksforGeeks

Given a stream of characters and we have to find first non repeating character each time a character is inserted to the stream.

Examples:

Input : a a b c  
Output : a -1 b b

Input : a a c  
Output : a -1 c

We have already discussed a Doubly linked list based approach in the [previous post](#).

### Approach-

1. Create a count array of size 26(assuming only lower case characters are present) and initialize it with zero.
2. Create a queue of char datatype.
3. Store each character in queue and increase its frequency in the hash array.
4. For every character of stream, we check front of the queue.
5. If the frequency of character at the front of queue is one, then that will be the first non repeating character.
6. Else if frequency is more than 1, then we pop that element.
7. If queue became empty that means there are no non repeating character so we will print -1.

C++

```
// C++ program for a Queue based approach to find first non-repeating
// character
#include <bits/stdc++.h>
using namespace std;
const int CHAR_MAX = 26;

// function to find first non repeating
// character of sa stream
void firstnonrepeating(char str[])
{
    queue<char> q;
    int charCount[CHAR_MAX] = { 0 };

    // traverse whole stream
    for (int i = 0; str[i]; i++) {

        // push each character in queue
        q.push(str[i]);

        // increment the frequency count
        charCount[str[i]-'a']++;

        // check for the non pepeating character
        while (!q.empty())
        {
            if (charCount[q.front()-'a'] > 1)
                q.pop();
            else
            {
                cout << q.front() << " ";
                break;
            }
        }

        if (q.empty())
            cout << -1 << " ";
    }
    cout << endl;
}

// Driver function
int main()
{
    char str[] = "aabc";
    firstnonrepeating(str);
    return 0;
}
```

}

**Java**

```
//Java Program for a Queue based approach to find first non-repeating
//character

import java.util.LinkedList;
import java.util.Queue;

public class NonRepeatingCQueue
{
    final static int CHAR_MAX = 26;

    // function to find first non repeating
    // character of stream
    static void firstNonRepeating(String str)
    {
        //count array of size 26(assuming only lower case characters are present)
        int[] charCount = new int[CHAR_MAX];

        //Queue to store Characters
        Queue<Character> q = new LinkedList<Character>();

        // traverse whole stream
        for(int i=0; i<str.length(); i++)
        {
            char c = str.charAt(i);

            // push each character in queue
            q.add(c);

            // increment the frequency count
            charCount++;

            // check for the non repeating character
            while(!q.isEmpty())
            {
                if(charCount[q.peek() - 'a'] > 1)
                    q.remove();
                else
                {
                    System.out.print(q.peek() + " ");
                    break;
                }
            }
            if(q.isEmpty())
                System.out.print(-1 + " ");
        }
    }
}
```

```
        }
        System.out.println();
    }

// Driver function
public static void main(String[] args)
{
    String str = "aabc";
    firstNonRepeating(str);
}

}
//This code is Contributed by Sumit Ghosh
```

Output:

a -1 b b

## Source

<https://www.geeksforgeeks.org/queue-based-approach-for-first-non-repeating-character-in-a-stream/>

## Chapter 182

# Queue | Set 2 (Linked List Implementation)

Queue | Set 2 (Linked List Implementation) - GeeksforGeeks

In the [previous post](#), we introduced Queue and discussed array implementation. In this post, linked list implementation is discussed. The following two main operations must be implemented efficiently.

In a Queue data structure, we maintain two pointers, *front* and *rear*. The *front* points the first item of queue and *rear* points to last item.

**enQueue()** This operation adds a new node after *rear* and moves *rear* to the next node.

**deQueue()** This operation removes the front node and moves *front* to the next node.

C

```
// A C program to demonstrate linked list based implementation of queue
#include <stdlib.h>
#include <stdio.h>

// A linked list (LL) node to store a queue entry
struct QNode
{
    int key;
    struct QNode *next;
};

// The queue, front stores the front node of LL and rear stores the
// last node of LL
struct Queue
{
    struct QNode *front, *rear;
```

```
};

// A utility function to create a new linked list node.
struct QNode* newNode(int k)
{
    struct QNode *temp = (struct QNode*)malloc(sizeof(struct QNode));
    temp->key = k;
    temp->next = NULL;
    return temp;
}

// A utility function to create an empty queue
struct Queue *createQueue()
{
    struct Queue *q = (struct Queue*)malloc(sizeof(struct Queue));
    q->front = q->rear = NULL;
    return q;
}

// The function to add a key k to q
void enqueue(struct Queue *q, int k)
{
    // Create a new LL node
    struct QNode *temp = newNode(k);

    // If queue is empty, then new node is front and rear both
    if (q->rear == NULL)
    {
        q->front = q->rear = temp;
        return;
    }

    // Add the new node at the end of queue and change rear
    q->rear->next = temp;
    q->rear = temp;
}

// Function to remove a key from given queue q
struct QNode *deQueue(struct Queue *q)
{
    // If queue is empty, return NULL.
    if (q->front == NULL)
        return NULL;

    // Store previous front and move front one node ahead
    struct QNode *temp = q->front;
    q->front = q->front->next;
```

```
// If front becomes NULL, then change rear also as NULL
if (q->front == NULL)
    q->rear = NULL;
return temp;
}

// Driver Program to test above functions
int main()
{
    struct Queue *q = createQueue();
    enQueue(q, 10);
    enQueue(q, 20);
    deQueue(q);
    deQueue(q);
    enQueue(q, 30);
    enQueue(q, 40);
    enQueue(q, 50);
    struct QNode *n = deQueue(q);
    if (n != NULL)
        printf("Dequeued item is %d", n->key);
    return 0;
}
```

### Java

```
// Java program for linked-list implementation of queue

// A linked list (LL) node to store a queue entry
class QNode
{
    int key;
    QNode next;

    // constructor to create a new linked list node
    public QNode(int key) {
        this.key = key;
        this.next = null;
    }
}

// A class to represent a queue
//The queue, front stores the front node of LL and rear stores the
//last node of LL
class Queue
{
    QNode front, rear;

    public Queue() {
```

```
    this.front = this.rear = null;
}

// Method to add an key to the queue.
void enqueue(int key)
{
    // Create a new LL node
    QNode temp = new QNode(key);

    // If queue is empty, then new node is front and rear both
    if (this.rear == null)
    {
        this.front = this.rear = temp;
        return;
    }

    // Add the new node at the end of queue and change rear
    this.rear.next = temp;
    this.rear = temp;
}

// Method to remove an key from queue.
QNode dequeue()
{
    // If queue is empty, return NULL.
    if (this.front == null)
        return null;

    // Store previous front and move front one node ahead
    QNode temp = this.front;
    this.front = this.front.next;

    // If front becomes NULL, then change rear also as NULL
    if (this.front == null)
        this.rear = null;
    return temp;
}

// Driver class
public class Test
{
    public static void main(String[] args)
    {
        Queue q=new Queue();
        q.enqueue(10);
```

```
    q.enqueue(20);
    q.dequeue();
    q.dequeue();
    q.enqueue(30);
    q.enqueue(40);
    q.enqueue(50);

    System.out.println("Dequeued item is "+ q.dequeue().key);
}
}

// This code is contributed by Gaurav Miglani
```

### Python3

```
# Python3 program to demonstrate linked list
# based implementation of queue

# A linked list (LL) node
# to store a queue entry
class Node:

    def __init__(self, data):
        self.data = data
        self.next = None

# A class to represent a queue

# The queue, front stores the front node
# of LL and rear stores the last node of LL
class Queue:

    def __init__(self):
        self.front = self.rear = None

    def isEmpty(self):
        return self.front == None

    # Method to add an item to the queue
    def EnQueue(self, item):
        temp = Node(item)

        if self.rear == None:
            self.front = self.rear = temp
            return
        self.rear.next = temp
        self.rear = temp

    # Method to remove an item from queue
```

```
def DeQueue(self):  
  
    if self.isEmpty():  
        return  
    temp = self.front  
    self.front = temp.next  
  
    if(self.front == None):  
        self.rear = None  
    return str(temp.data)  
  
# Driver Code  
if __name__== '__main__':  
    q = Queue()  
    q.Enqueue(10)  
    q.Enqueue(20)  
    q.DeQueue()  
    q.DeQueue()  
    q.Enqueue(30)  
    q.Enqueue(40)  
    q.Enqueue(50)  
  
    print("Dequeued item is " + q.DeQueue())
```

Output:

Dequeued item is 30

**Time Complexity:** Time complexity of both operations enqueue() and dequeue() is O(1) as we only change few pointers in both operations. There is no loop in any of the operations.

## Source

<https://www.geeksforgeeks.org/queue-set-2-linked-list-implementation/>

## Chapter 183

# QuickSort on Doubly Linked List

QuickSort on Doubly Linked List - GeeksforGeeks

Following is a typical recursive implementation of [QuickSort](#) for arrays. The implementation uses last element as pivot.

```
/* A typical recursive implementation of Quicksort for array*/  
  
/* This function takes last element as pivot, places the pivot element at its  
correct position in sorted array, and places all smaller (smaller than  
pivot) to left of pivot and all greater elements to right of pivot */  
int partition (int arr[], int l, int h)  
{  
    int x = arr[h];  
    int i = (l - 1);  
  
    for (int j = l; j <= h- 1; j++)  
    {  
        if (arr[j] <= x)  
        {  
            i++;  
            swap (&arr[i], &arr[j]);  
        }  
    }  
    swap (&arr[i + 1], &arr[h]);  
    return (i + 1);  
}  
  
/* A[] --> Array to be sorted, l --> Starting index, h --> Ending index */  
void quickSort(int A[], int l, int h)  
{
```

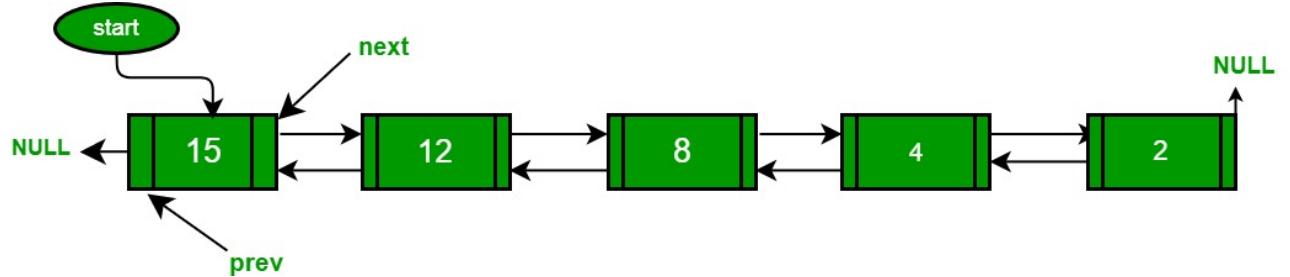
```

if (l < h)
{
    int p = partition(A, l, h); /* Partitioning index */
    quickSort(A, l, p - 1);
    quickSort(A, p + 1, h);
}
}

```

### Can we use same algorithm for Linked List?

Following is C++ implementation for doubly linked list. The idea is simple, we first find out pointer to last node. Once we have pointer to last node, we can recursively sort the linked list using pointers to first and last nodes of linked list, similar to the above recursive function where we pass indexes of first and last array elements. The partition function for linked list is also similar to partition for arrays. Instead of returning index of the pivot element, it returns pointer to the pivot element. In the following implementation, quickSort() is just a wrapper function, the main recursive function is `_quickSort()` which is similar to quickSort() for array implementation.



C++

```

// A C++ program to sort a linked list using Quicksort
#include <iostream>
#include <stdio.h>
using namespace std;

/* a node of the doubly linked list */
struct Node
{
    int data;
    struct Node *next;
    struct Node *prev;
};

/* A utility function to swap two elements */
void swap ( int* a, int* b )
{   int t = *a;      *a = *b;      *b = t; }

// A utility function to find last node of linked list

```

```

struct Node *lastNode(Node *root)
{
    while (root && root->next)
        root = root->next;
    return root;
}

/* Considers last element as pivot, places the pivot element at its
   correct position in sorted array, and places all smaller (smaller than
   pivot) to left of pivot and all greater elements to right of pivot */
Node* partition(Node *l, Node *h)
{
    // set pivot as h element
    int x = h->data;

    // similar to i = l-1 for array implementation
    Node *i = l->prev;

    // Similar to "for (int j = l; j <= h- 1; j++)"
    for (Node *j = l; j != h; j = j->next)
    {
        if (j->data <= x)
        {
            // Similar to i++ for array
            i = (i == NULL)? l : i->next;

            swap(&(i->data), &(j->data));
        }
    }
    i = (i == NULL)? l : i->next; // Similar to i++
    swap(&(i->data), &(h->data));
    return i;
}

/* A recursive implementation of quicksort for linked list */
void _quickSort(struct Node* l, struct Node *h)
{
    if (h != NULL && l != h && l != h->next)
    {
        struct Node *p = partition(l, h);
        _quickSort(l, p->prev);
        _quickSort(p->next, h);
    }
}

// The main function to sort a linked list. It mainly calls _quickSort()
void quickSort(struct Node *head)
{

```

```
// Find last node
struct Node *h = lastNode(head);

// Call the recursive QuickSort
_quickSort(head, h);
}

// A utility function to print contents of arr
void printList(struct Node *head)
{
    while (head)
    {
        cout << head->data << " ";
        head = head->next;
    }
    cout << endl;
}

/* Function to insert a node at the beginning of the Doubly Linked List */
void push(struct Node** head_ref, int new_data)
{
    struct Node* new_node = new Node; /* allocate node */
    new_node->data = new_data;

    /* since we are adding at the begining, prev is always NULL */
    new_node->prev = NULL;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* change prev of head node to new node */
    if ((*head_ref) != NULL) (*head_ref)->prev = new_node;

    /* move the head to point to the new node */
    (*head_ref) = new_node;
}

/* Driver program to test above function */
int main()
{
    struct Node *a = NULL;
    push(&a, 5);
    push(&a, 20);
    push(&a, 4);
    push(&a, 3);
    push(&a, 30);

    cout << "Linked List before sorting \n";
```

```
    printList(a);

    quickSort(a);

    cout << "Linked List after sorting \n";
    printList(a);

    return 0;
}
```

**Java**

```
// A Java program to sort a linked list using Quicksort
class QuickSort_using_Doubly_LinkedList{
    Node head;

    /* a node of the doubly linked list */
    static class Node{
        private int data;
        private Node next;
        private Node prev;

        Node(int d){
            data = d;
            next = null;
            prev = null;
        }
    }

    // A utility function to find last node of linked list
    Node lastNode(Node node){
        while(node.next!=null)
            node = node.next;
        return node;
    }

    /* Considers last element as pivot, places the pivot element at its
       correct position in sorted array, and places all smaller (smaller than
       pivot) to left of pivot and all greater elements to right of pivot */
    Node partition(Node l,Node h)
    {
        // set pivot as h element
        int x = h.data;

        // similar to i = l-1 for array implementation
        Node i = l.prev;
```

```
// Similar to "for (int j = l; j <= h- 1; j++)"
for(Node j=l; j!=h; j=j.next)
{
    if(j.data <= x)
    {
        // Similar to i++ for array
        i = (i==null) ? l : i.next;
        int temp = i.data;
        i.data = j.data;
        j.data = temp;
    }
    i = (i==null) ? l : i.next; // Similar to i++
    int temp = i.data;
    i.data = h.data;
    h.data = temp;
    return i;
}

/* A recursive implementation of quicksort for linked list */
void _quickSort(Node l,Node h)
{
    if(h!=null && l!=h && l!=h.next){
        Node temp = partition(l,h);
        _quickSort(l,temp.prev);
        _quickSort(temp.next,h);
    }
}

// The main function to sort a linked list. It mainly calls _quickSort()
public void quickSort(Node node)
{
    // Find last node
    Node head = lastNode(node);

    // Call the recursive QuickSort
    _quickSort(node,head);
}

// A utility function to print contents of arr
public void printList(Node head)
{
    while(head!=null){
        System.out.print(head.data+" ");
        head = head.next;
    }
}
```

```
/* Function to insert a node at the begining of the Doubly Linked List */
void push(int new_Data)
{
    Node new_Node = new Node(new_Data);      /* allocate node */

    // if head is null, head = new_Node
    if(head==null){
        head = new_Node;
        return;
    }

    /* link the old list off the new node */
    new_Node.next = head;

    /* change prev of head node to new node */
    head.prev = new_Node;

    /* since we are adding at the begining, prev is always NULL */
    new_Node.prev = null;

    /* move the head to point to the new node */
    head = new_Node;
}

/* Driver program to test above function */
public static void main(String[] args){
    QuickSort_using_Doubly_LinkedList list = new QuickSort_using_Doubly_LinkedList();

    list.push(5);
    list.push(20);
    list.push(4);
    list.push(3);
    list.push(30);

    System.out.println("Linked List before sorting ");
    list.printList(list.head);
    System.out.println("\nLinked List after sorting");
    list.quickSort(list.head);
    list.printList(list.head);

}

// This code has been contributed by Amit Khandelwal
```

Output :

Linked List before sorting

30 3 4 20 5

Linked List after sorting

3 4 5 20 30

**Time Complexity:** Time complexity of the above implementation is same as time complexity of QuickSort() for arrays. It takes  $O(n^2)$  time in worst case and  $O(n \log n)$  in average and best cases. The worst case occurs when the linked list is already sorted.

Can we implement random quick sort for linked list?

Quicksort can be implemented for Linked List only when we can pick a fixed point as pivot (like last element in above implementation). Random QuickSort cannot be efficiently implemented for Linked Lists by picking random pivot.

**Exercise:**

The above implementation is for doubly linked list. Modify it for singly linked list. Note that we don't have prev pointer in singly linked list.

Refer [QuickSort on Singly Linked List](#) for solution.

## Source

<https://www.geeksforgeeks.org/quicksort-for-linked-list/>

## Chapter 184

# QuickSort on Singly Linked List

QuickSort on Singly Linked List - GeeksforGeeks

QuickSort on Doubly Linked List is discussed [here](#). QuickSort on Singly linked list was given as an exercise. Following is C++ implementation for same. The important things about implementation are, it changes pointers rather swapping data and time complexity is same as the implementation for Doubly Linked List.

In **partition()**, we consider last element as pivot. We traverse through the current list and if a node has value greater than pivot, we move it after tail. If the node has smaller value, we keep it at its current position.

In **QuickSortRecur()**, we first call **partition()** which places pivot at correct position and returns pivot. After pivot is placed at correct position, we find tail node of left side (list before pivot) and recur for left list. Finally, we recur for right list.

```
// C++ program for Quick Sort on Singly Linled List
#include <iostream>
#include <cstdio>
using namespace std;

/* a node of the singly linked list */
struct Node
{
    int data;
    struct Node *next;
};

/* A utility function to insert a node at the beginning of linked list */
void push(struct Node** head_ref, int new_data)
{
    /* allocate node */
    struct Node* new_node = new Node;

    /* put in the data */
    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;
}
```

```

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;
}

/* A utility function to print linked list */
void printList(struct Node *node)
{
    while (node != NULL)
    {
        printf("%d ", node->data);
        node = node->next;
    }
    printf("\n");
}

// Returns the last node of the list
struct Node *getTail(struct Node *cur)
{
    while (cur != NULL && cur->next != NULL)
        cur = cur->next;
    return cur;
}

// Partitions the list taking the last element as the pivot
struct Node *partition(struct Node *head, struct Node *end,
                      struct Node **newHead, struct Node **newEnd)
{
    struct Node *pivot = end;
    struct Node *prev = NULL, *cur = head, *tail = pivot;

    // During partition, both the head and end of the list might change
    // which is updated in the newHead and newEnd variables
    while (cur != pivot)
    {
        if (cur->data < pivot->data)
        {
            // First node that has a value less than the pivot - becomes
            // the new head
            if ((*newHead) == NULL)
                (*newHead) = cur;

            prev = cur;
            cur = cur->next;
        }
    }
}

```

```

    }

    else // If cur node is greater than pivot
    {
        // Move cur node to next of tail, and change tail
        if (prev)
            prev->next = cur->next;
        struct Node *tmp = cur->next;
        cur->next = NULL;
        tail->next = cur;
        tail = cur;
        cur = tmp;
    }
}

// If the pivot data is the smallest element in the current list,
// pivot becomes the head
if ((*newHead) == NULL)
    (*newHead) = pivot;

// Update newEnd to the current last node
(*newEnd) = tail;

// Return the pivot node
return pivot;
}

//here the sorting happens exclusive of the end node
struct Node *quickSortRecur(struct Node *head, struct Node *end)
{
    // base condition
    if (!head || head == end)
        return head;

    Node *newHead = NULL, *newEnd = NULL;

    // Partition the list, newHead and newEnd will be updated
    // by the partition function
    struct Node *pivot = partition(head, end, &newHead, &newEnd);

    // If pivot is the smallest element - no need to recur for
    // the left part.
    if (newHead != pivot)
    {
        // Set the node before the pivot node as NULL
        struct Node *tmp = newHead;
        while (tmp->next != pivot)
            tmp = tmp->next;
    }
}

```

```
tmp->next = NULL;

// Recur for the list before pivot
newHead = quickSortRecur(newHead, tmp);

// Change next of last node of the left half to pivot
tmp = getTail(newHead);
tmp->next = pivot;
}

// Recur for the list after the pivot element
pivot->next = quickSortRecur(pivot->next, newEnd);

return newHead;
}

// The main function for quick sort. This is a wrapper over recursive
// function quickSortRecur()
void quickSort(struct Node **headRef)
{
    (*headRef) = quickSortRecur(*headRef, getTail(*headRef));
    return;
}

// Driver program to test above functions
int main()
{
    struct Node *a = NULL;
    push(&a, 5);
    push(&a, 20);
    push(&a, 4);
    push(&a, 3);
    push(&a, 30);

    cout << "Linked List before sorting \n";
    printList(a);

    quickSort(&a);

    cout << "Linked List after sorting \n";
    printList(a);

    return 0;
}
```

Output:

```
Linked List before sorting
```

```
30 3 4 20 5  
Linked List after sorting  
3 4 5 20 30
```

This article is contributed by **Balasubramanian.N**. 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/quicksort-on-singly-linked-list/>

## Chapter 185

# Rearrange a Linked List in Zig-Zag fashion

Rearrange a Linked List in Zig-Zag fashion - GeeksforGeeks

Given a linked list, rearrange it such that converted list should be of the form a < b > c < d > e < f .. where a, b, c.. are consecutive data node of linked list. Examples :

Input: 1->2->3->4  
Output: 1->3->2->4

Input: 11->15->20->5->10  
Output: 11->20->5->15->10

A **simple approach** to do this, is to [sort the linked list using merge sort](#) and then swap alternate, but that requires  $O(n \log n)$  time complexity. Here n is number of elements in linked list.

An **efficient approach** which requires  $O(n)$  time is, using a single scan similar to bubble sort and then maintain a flag for representing which order (< or >) currently we are. If the current two elements are not in that order then swap those elements otherwise not. Please refer [this](#) for detailed explanation of swapping order.

```
// C++ program to arrange linked list in zigzag fashion
#include <bits/stdc++.h>
using namespace std;

/* Link list Node */
struct Node
{
    int data;
```

```
    struct Node* next;
};

// This function distributes the Node in zigzag fashion
void zigZagList(Node *head)
{
    // If flag is true, then next node should be greater
    // in the desired output.
    bool flag = true;

    // Traverse linked list starting from head.
    Node* current = head;
    while (current->next != NULL)
    {
        if (flag) /* "<" relation expected */
        {
            /* If we have a situation like A > B > C
               where A, B and C are consecutive Nodes
               in list we get A > B < C by swapping B
               and C */
            if (current->data > current->next->data)
                swap(current->data, current->next->data);
        }
        else /* ">" relation expected */
        {
            /* If we have a situation like A < B < C where
               A, B and C are consecutive Nodes in list we
               get A < C > B by swapping B and C */
            if (current->data < current->next->data)
                swap(current->data, current->next->data);
        }

        current = current->next;
        flag = !flag; /* flip flag for reverse checking */
    }
}

/* UTILITY FUNCTIONS */
/* Function to push a Node */
void push(Node** head_ref, int new_data)
{
    /* allocate Node */
    struct Node* new_Node = new Node;

    /* put in the data */
    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;
}

/* Function to print linked list */
void printList(struct Node *Node)
{
    while (Node != NULL)
    {
        printf("%d->", Node->data);
        Node = Node->next;
    }
    printf("NULL");
}

/* Drier program to test above function*/
int main(void)
{
    /* Start with the empty list */
    struct Node* head = NULL;

    // create a list 4 -> 3 -> 7 -> 8 -> 6 -> 2 -> 1
    // answer should be -> 3  7  4  8  2  6  1
    push(&head, 1);
    push(&head, 2);
    push(&head, 6);
    push(&head, 8);
    push(&head, 7);
    push(&head, 3);
    push(&head, 4);

    printf("Given linked list \n");
    printList(head);

    zigZagList(head);

    printf("\nZig Zag Linked list \n");
    printList(head);

    return (0);
}
```

Output :

Given linked list

4->3->7->8->6->2->1->NULL

Zig Zag Linked list  
3->7->4->8->2->6->1->NULL

In above code, push function pushes the node at the front of the linked list, the code can be easily modified for pushing node at the end of list. Other thing to note is, swapping of data between two nodes is done by swap by value not swap by links for simplicity, for swap by links technique please see [this](#).

Time complexity : O(n)

Auxiliary Space : O(1)

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/linked-list-in-zig-zag-fashion/>

## Chapter 186

# Rearrange a Linked List in Zig-Zag fashion | Set-2

Rearrange a Linked List in Zig-Zag fashion | Set-2 - GeeksforGeeks

Given a linked list, rearrange it such that converted list should be of the form a < b > c < d > e < f .. where a, b, c.. are consecutive data node of linked list. Note that it is not allowed to swap data.

Examples:

Input: 1->2->3->4  
Output: 1->3->2->4

Input: 11->15->20->5->10  
Output: 11->20->5->15->10

### Approach:

A solution that converts given list into zigzag form is discussed in [previous post](#). The solution discussed performs conversion by swapping data of nodes. Swapping data of nodes may be expensive in many situations when the data contains many fields. In this post, a solution that performs conversion by swapping links is discussed.

The idea is to traverse the given linked list and check if current node maintains the zigzag order or not. To check if given node maintains zigzag order or not, a variable *ind* is used. If **ind = 0**, then the current node's data should be less than its adjacent node's data and if **ind = 1**, then current node's data should be greater than its adjacent node's data. If the current node violates the zigzag order, then swap the position of both nodes. For doing this step, maintain two pointers **prev** and **next**. *prev* stores previous node of current node and *next* stores new next node of current node. To swap both nodes, the following steps are performed:

- Make next node of current node, the next node of previous node.

- Make the current node next node of its adjacent node.
- Make current node next = next node.

Below is the implementation of above approach:

```
// CPP program to arrange linked list in
// zigzag fashion
#include <bits/stdc++.h>
using namespace std;

/* Link list Node */
struct Node {
    int data;
    struct Node* next;
};

// This function converts the Linked list in
// zigzag fashion
Node* zigZagList(Node* head)
{
    if (head == NULL || head->next == NULL) {
        return head;
    }

    // to store new head
    Node* res = NULL;

    // to traverse linked list
    Node* curr = head;

    // to store previous node of current node
    Node* prev = NULL;

    // to store new next node of current node
    Node* next;

    // to check if current element should
    // be less than or greater than.
    // ind = 0 --> less than
    // ind = 1 --> greater than
    int ind = 0;

    while (curr->next) {

        // If elements are not in zigzag fashion
        // swap them.
        if ((ind == 0 && curr->data > curr->next->data)
            || (ind == 1 && curr->data < curr->next->data)) {
```

```
if (res == NULL)
    res = curr->next;

// Store new next element of current
// node
next = curr->next->next;

// Previous node of current node will
// now point to next node of current node
if (prev)
    prev->next = curr->next;

// Change next pointers of both
// adjacent nodes
curr->next->next = curr;
curr->next = next;

// Change previous pointer.
if (prev)
    prev = prev->next;
else
    prev = res;
}

// If already in zig zag form, then move
// to next element.
else {
    if (res == NULL) {
        res = curr;
    }

    prev = curr;
    curr = curr->next;
}

// Update info whether next element should
// be less than or greater than.
ind = 1 - ind;
}

return res;
}

/* UTILITY FUNCTIONS */
/* Function to push a Node */
void push(Node** head_ref, int new_data)
{
```

```
/* allocate Node */
struct Node* new_Node = new Node;

/* put in the data */
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;
}

/* Function to print linked list */
void printList(struct Node* Node)
{
    while (Node != NULL) {
        printf("%d->", Node->data);
        Node = Node->next;
    }
}

/* Driver program to test above function*/
int main(void)
{
    /* Start with the empty list */
    struct Node* head = NULL;

    // create a list 4 -> 3 -> 7 -> 8 -> 6 -> 2 -> 1
    // answer should be -> 3 7 4 8 2 6 1
    push(&head, 1);
    push(&head, 2);
    push(&head, 6);
    push(&head, 8);
    push(&head, 7);
    push(&head, 3);
    push(&head, 4);

    printf("Given linked list \n");
    printList(head);

    head = zigZagList(head);

    printf("\nZig Zag Linked list \n");
    printList(head);

    return 0;
}
```

**Output:**

```
Given linked list  
4->3->7->8->6->2->1->  
Zig Zag Linked list  
3->7->4->8->2->6->1->
```

**Time Complexity:**  $O(N)$   
**Auxiliary Space:**  $O(1)$

**Source**

<https://www.geeksforgeeks.org/rearrange-a-linked-list-in-zig-zag-fashion-set-2/>

## Chapter 187

# Rearrange a given linked list in-place.

Rearrange a given linked list in-place. - GeeksforGeeks

Given a singly linked list  $L_0 \rightarrow L_1 \rightarrow \dots \rightarrow L_{n-1} \rightarrow L_n$ . Rearrange the nodes in the list so that the new formed list is :  $L_0 \rightarrow L_n \rightarrow L_1 \rightarrow L_{n-1} \rightarrow L_2 \rightarrow L_{n-2} \dots$

You are required to do this in-place without altering the nodes' values.

**Example:**

Input: 1 → 2 → 3 → 4  
Output: 1 → 4 → 2 → 3

Input: 1 → 2 → 3 → 4 → 5  
Output: 1 → 5 → 2 → 4 → 3

### Simple Solution

- 1) Initialize current node as head.
- 2) While next of current node is not null, do following
  - a) Find the last node, remove it from the end and insert it as next of the current node.
  - b) Move current to next to next of current

Time complexity of the above simple solution is  $O(n^2)$  where n is the number of nodes in the linked list.

### Better Solution

- 1) Copy contents of given linked list to a vector.

2) Rearrange given vector by swapping nodes from both ends.

3) Copy the modified vector back to the linked list.

Implementation of this approach : <https://ide.geeksforgeeks.org/1eGSEy>

Thanks to Arushi Dhamija for suggesting this approach.

**Efficient Solution:**

1) Find the middle point using tortoise and hare method.

2) Split the linked list into two halves using found middle point in step 1.

3) Reverse the second half.

4) Do alternate merge of first and second halves.

Time Complexity of this solution is O(n).

Below is the implementation of this method.

C++

```
// C++ program to rearrange a linked list in-place
#include<bits/stdc++.h>
using namespace std;

// Linkedlist Node structure
struct Node
{
    int data;
    struct Node *next;
};

// Function to create newNode in a linkedlist
Node* newNode(int key)
{
    Node *temp = new Node;
    temp->data = key;
    temp->next = NULL;
    return temp;
}

// Function to reverse the linked list
void reverselist(Node **head)
{
    // Initialize prev and current pointers
    Node *prev = NULL, *curr = *head, *next;

    while (curr)
    {
        next = curr->next;
        curr->next = prev;
        prev = curr;
        curr = next;
    }
    *head = prev;
}
```

```
        prev = curr;
        curr = next;
    }

    *head = prev;
}

// Function to print the linked list
void printlist(Node *head)
{
    while (head != NULL)
    {
        cout << head->data << " ";
        if(head->next) cout << "-> ";
        head = head->next;
    }
    cout << endl;
}

// Function to rearrange a linked list
void rearrange(Node **head)
{
    // 1) Find the middle point using tortoise and hare method
    Node *slow = *head, *fast = slow->next;
    while (fast && fast->next)
    {
        slow = slow->next;
        fast = fast->next->next;
    }

    // 2) Split the linked list in two halves
    // head1, head of first half    1 -> 2
    // head2, head of second half   3 -> 4
    Node *head1 = *head;
    Node *head2 = slow->next;
    slow->next = NULL;

    // 3) Reverse the second half, i.e.,  4 -> 3
    reverselist(&head2);

    // 4) Merge alternate nodes
    *head = newNode(0); // Assign dummy Node

    // curr is the pointer to this dummy Node, which will
    // be used to form the new list
    Node *curr = *head;
    while (head1 || head2)
    {
```

```
// First add the element from list
if (head1)
{
    curr->next = head1;
    curr = curr->next;
    head1 = head1->next;
}

// Then add the element from the second list
if (head2)
{
    curr->next = head2;
    curr = curr->next;
    head2 = head2->next;
}
}

// Assign the head of the new list to head pointer
*head = (*head)->next;
}

// Driver program
int main()
{
    Node *head = newNode(1);
    head->next = newNode(2);
    head->next->next = newNode(3);
    head->next->next->next = newNode(4);
    head->next->next->next->next = newNode(5);

    printlist(head);      // Print original list
    rearrange(&head);    // Modify the list
    printlist(head);      // Print modified list
    return 0;
}
```

### Java

```
// Java program to rearrange link list in place

// Linked List Class
class LinkedList {

    static Node head; // head of the list

    /* Node Class */
    static class Node {
```

```
int data;
Node next;

// Constructor to create a new node
Node(int d) {
    data = d;
    next = null;
}
}

void printlist(Node node) {
    if (node == null) {
        return;
    }
    while (node != null) {
        System.out.print(node.data + " -> ");
        node = node.next;
    }
}

Node reverselist(Node node) {
    Node prev = null, curr = node, next;
    while (curr != null) {
        next = curr.next;
        curr.next = prev;
        prev = curr;
        curr = next;
    }
    node = prev;
    return node;
}

void rearrange(Node node) {

    // 1) Find the middle point using tortoise and hare method
    Node slow = node, fast = slow.next;
    while (fast != null && fast.next != null) {
        slow = slow.next;
        fast = fast.next.next;
    }

    // 2) Split the linked list in two halves
    // node1, head of first half    1 -> 2 -> 3
    // node2, head of second half   4 -> 5
    Node node1 = node;
    Node node2 = slow.next;
    slow.next = null;
```

```
// 3) Reverse the second half, i.e., 5 -> 4
node2 = reverselist(node2);

// 4) Merge alternate nodes
node = new Node(0); // Assign dummy Node

// curr is the pointer to this dummy Node, which will
// be used to form the new list
Node curr = node;
while (node1 != null || node2 != null) {

    // First add the element from first list
    if (node1 != null) {
        curr.next = node1;
        curr = curr.next;
        node1 = node1.next;
    }

    // Then add the element from second list
    if (node2 != null) {
        curr.next = node2;
        curr = curr.next;
        node2 = node2.next;
    }
}

// Assign the head of the new list to head pointer
node = node.next;
}

public static void main(String[] args) {

    LinkedList list = new LinkedList();
    list.head = new Node(1);
    list.head.next = new Node(2);
    list.head.next.next = new Node(3);
    list.head.next.next.next = new Node(4);
    list.head.next.next.next.next = new Node(5);

    list.printlist(head); // print original list
    list.rearrange(head); // rearrange list as per ques
    System.out.println("");
    list.printlist(head); // print modified list

}
```

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

Output:

```
1 -> 2 -> 3 -> 4 -> 5  
1 -> 5 -> 2 -> 4 -> 3
```

Time Complexity: O(n)

Auxiliary Space: O(1)

Thanks to Gaurav Ahirwar for suggesting the above approach.

**Another approach :**

1. Take two pointers prev and curr, which hold the addresses of head and head-> next.
2. Compare their data and swap.

After that, a new linked list is formed.

Below is the implementation :

```
// CPP code to rearrange linked list in place  
#include<iostream>  
#include<bits/stdc++.h>  
using namespace std;  
  
struct node{  
    int data;  
    struct node *next;  
};  
typedef struct node Node;  
  
// function for rearranging a linked list with high and low value.  
void rearrange(Node *head)  
{  
    if(head == NULL) //Base case.  
        return;  
  
    // two pointer variable.  
    Node *prev = head, *curr = head -> next;  
  
    while(curr)  
    {  
        // swap function for swapping data.  
        if(prev -> data > curr -> data)  
            swap(prev -> data, curr -> data);  
  
        // swap function for swapping data.  
        if(curr -> next && curr -> next -> data > curr -> data)  
            swap(curr -> next -> data , curr -> data);  
    }  
}
```

```
prev = curr -> next;

if( !curr -> next)
    break;
curr = curr -> next -> next;
}

}

// function to insert a node in the linked list at the beginning.
void push(Node **head, int k)
{
    Node *tem = (Node*)malloc(sizeof(Node));
    tem -> data = k;
    tem -> next = *head;
    *head = tem;
}

// function to display node of linked list.
void display(Node *head)
{
    Node *curr = head;
    while(curr != NULL)
    {
        printf("%d ", curr -> data);
        curr = curr -> next;
    }
}

// driver code
int main()
{

Node *head = NULL;

//let create a linked list.
//9 -> 6 -> 8 -> 3 -> 7
push(&head, 7);
push(&head, 3);
push(&head, 8);
push(&head, 6);
push(&head, 9);

rearrange(head);

display(head);

return 0;
```

}

Time Complexity :  $O(n)$

Auxiliary Space :  $O(1)$

Thanks to [Aditya](#) for suggesting this approach.

**Improved By :** [Balraj Parmar](#)

## Source

<https://www.geeksforgeeks.org/rearrange-a-given-linked-list-in-place/>

## Chapter 188

# Rearrange a given list such that it consists of alternating minimum maximum elements

Rearrange a given list such that it consists of alternating minimum maximum elements - GeeksforGeeks

Given a list of integers, rearrange the list such that it consists of alternating minimum maximum elements **using only list operations**. The first element of the list should be minimum and second element should be maximum of all elements present in the list. Similarly, third element will be next minimum element and fourth element is next maximum element and so on. Use of extra space is not permitted.

Examples:

Input: [1 3 8 2 7 5 6 4]  
Output: [1 8 2 7 3 6 4 5]

Input: [1 2 3 4 5 6 7]  
Output: [1 7 2 6 3 5 4]

Input: [1 6 2 5 3 4]  
Output: [1 6 2 5 3 4]

The idea is to sort the list in ascending order first. Then we start popping elements from the end of the list and insert them into their correct position in the list.

Below is C++ implementation of above idea -

C/C++

```
// C++ program to rearrange a given list such that it
// consists of alternating minimum maximum elements
#include <bits/stdc++.h>
using namespace std;

// Function to rearrange a given list such that it
// consists of alternating minimum maximum elements
void alternateSort(list<int>& inp)
{
    // sort the list in ascending order
    inp.sort();

    // get iterator to first element of the list
    list<int>::iterator it = inp.begin();
    it++;

    for (int i=1; i<(inp.size() + 1)/2; i++)
    {
        // pop last element (next greatest)
        int val = inp.back();
        inp.pop_back();

        // insert it after next minimum element
        inp.insert(it, val);

        // increment the pointer for next pair
        ++it;
    }
}

// Driver code
int main()
{
    // input list
    list<int> inp({ 1, 3, 8, 2, 7, 5, 6, 4 });

    // rearrange the given list
    alternateSort(inp);

    // print the modified list
    for (int i : inp)
        cout << i << " ";

    return 0;
}
```

Java

```
// Java program to rearrange a given list such that it
// consists of alternating minimum maximum elements
import java.util.*;

class AlternateSort
{
    // Function to rearrange a given list such that it
    // consists of alternating minimum maximum elements
    // using LinkedList
    public static void alternateSort(LinkedList<Integer> ll)
    {
        Collections.sort(ll);

        for (int i = 1; i < (ll.size() + 1)/2; i++)
        {
            Integer x = ll.getLast();
            ll.removeLast();
            ll.add(2*i - 1, x);
        }

        System.out.println(ll);
    }

    public static void main (String[] args) throws java.lang.Exception
    {
        // input list
        Integer arr[] = {1, 3, 8, 2, 7, 5, 6, 4};

        // convert array to LinkedList
        LinkedList<Integer> ll = new LinkedList<Integer>(Arrays.asList(arr));

        // rearrange the given list
        alternateSort(ll);
    }
}
```

Output:

```
1 8 2 7 3 6 4 5
```

## Source

<https://www.geeksforgeeks.org/rearrange-given-list-consists-alternating-minimum-maximum-elements/>

## Chapter 189

# Rearrange a linked list in to alternate first and last element

Rearrange a linked list in to alternate first and last element - GeeksforGeeks

Given a linked list. arrange the linked list in manner of alternate first and last element.

Examples:

Input : 1->2->3->4->5->6->7->8  
Output :1->8->2->7->3->6->4->5

Input :10->11->15->13  
Output :10->13->11->15

We have discussed three different solution in [Rearrange a given linked list in-place](#).

In this post a different [Deque](#) based solution is discussed.

### Method

- 1) Create an empty deque
- 2) Insert all element from the linked list to the deque
- 3) Insert the element back to the linked list from deque in alternate fashion i.e first then last and so on

```
// CPP program to rearrange a linked list in given manner
#include <bits/stdc++.h>
using namespace std;

/* Link list node */
struct Node {
    int data;
    struct Node* next;
}
```

```
};

/* Function to reverse the linked list */
void arrange(struct Node* head)
{
    struct Node* temp = head;
    deque<int> d; // defining a deque

    // push all the elemnts of linked list in to deque
    while (temp != NULL) {
        d.push_back(temp->data);
        temp = temp->next;
    }

    // Alternatively push the first and last elements
    // from deque to back to the linked list and pop
    int i = 0;
    temp = head;
    while (!d.empty()) {
        if (i % 2 == 0) {
            temp->data = d.front();
            d.pop_front();
        }
        else {
            temp->data = d.back();
            d.pop_back();
        }
        i++;
        temp = temp->next; // increse temp
    }
}

/*UTILITY FUNCTIONS*/
/* Push a node to linked list. Note that this function
changes the head */
void push(struct Node** head_ref, char new_data)
{
    /* allocate node */
    struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));

    /* put in the data */
    new_node->data = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to pochar to the new node */
    (*head_ref) = new_node;
```

```
}

// printing the linked list
void printList(struct Node* head)
{
    struct Node* temp = head;
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
}

/* Drier program to test above function*/
int main()
{
    // Let us create linked list 1->2->3->4
    struct Node* head = NULL;

    push(&head, 5);
    push(&head, 4);
    push(&head, 3);
    push(&head, 2);
    push(&head, 1);
    cout << "Given linked list\t";
    printList(head);
    arrange(head);
    cout << "\nAfter rearrangement\t";
    printList(head);
    return 0;
}
```

Output:

```
Given linked list      1 2 3 4 5
After rearrangement   1 5 2 4 3
```

## Source

<https://www.geeksforgeeks.org/rearrange-linked-list-alternate-first-last-element/>

## Chapter 190

# Rearrange a linked list such that all even and odd positioned nodes are together

Rearrange a linked list such that all even and odd positioned nodes are together - Geeks-forGeeks

Rearrange a linked list in such a way that all odd position nodes are together and all even positions node are together,

Examples:

Input: 1->2->3->4  
Output: 1->3->2->4

Input: 10->22->30->43->56->70  
Output: 10->30->56->22->43->70

The important thing in this question is to make sure that all below cases are handled

- 1) Empty linked list
- 2) A linked list with only one node
- 3) A linked list with only two nodes
- 4) A linked list with odd number of nodes
- 5) A linked list with even number of nodes

The below program maintains two pointers ‘odd’ and ‘even’ for current nodes at odd and even positions respectively. We also store first node of even linked list so that we can attach the even list at the end of odd list after all odd and even nodes are connected together in two different lists.

```
// C++ program to rearrange a linked list in such a
```

```
// way that all odd positioned node are stored before
// all even positioned nodes
#include<bits/stdc++.h>
using namespace std;

// Linked List Node
struct Node
{
    int data;
    struct Node* next;
};

// A utility function to create a new node
Node* newNode(int key)
{
    Node *temp = new Node;
    temp->data = key;
    temp->next = NULL;
    return temp;
}

// Rearranges given linked list such that all even
// positioned nodes are before odd positioned.
// Returns new head of linked List.
Node *rearrangeEvenOdd(Node *head)
{
    // Corner case
    if (head == NULL)
        return NULL;

    // Initialize first nodes of even and
    // odd lists
    Node *odd = head;
    Node *even = head->next;

    // Remember the first node of even list so
    // that we can connect the even list at the
    // end of odd list.
    Node *evenFirst = even;

    while (1)
    {
        // If there are no more nodes, then connect
        // first node of even list to the last node
        // of odd list
        if (!odd || !even || !(even->next))
        {
            odd->next = evenFirst;
```

```
        break;
    }

    // Connecting odd nodes
    odd->next = even->next;
    odd = even->next;

    // If there are NO more even nodes after
    // current odd.
    if (odd->next == NULL)
    {
        even->next = NULL;
        odd->next = evenFirst;
        break;
    }

    // Connecting even nodes
    even->next = odd->next;
    even = odd->next;
}

return head;
}

// A utility function to print a linked list
void printlist(Node * node)
{
    while (node != NULL)
    {
        cout << node->data << "->";
        node = node->next;
    }
    cout << "NULL" << endl;
}

// Driver code
int main(void)
{
    Node *head = newNode(1);
    head->next = newNode(2);
    head->next->next = newNode(3);
    head->next->next->next = newNode(4);
    head->next->next->next->next = newNode(5);

    cout << "Given Linked List\n";
    printlist(head);

    head = rearrangeEvenOdd(head);
```

```
cout << "\nModified Linked List\n";
printlist(head);

return 0;
}
```

Output:

```
Given Linked List
1->2->3->4->5->NULL
Modified Linked List
1->3->5->2->4->NULL
```

## Source

<https://www.geeksforgeeks.org/rearrange-a-linked-list-such-that-all-even-and-odd-positioned-nodes-are-together/>

## Chapter 191

# Recursive Approach to find nth node from the end in the linked list

Recursive Approach to find nth node from the end in the linked list - GeeksforGeeks

Find the nth node from the end in the given linked list using a recursive approach.

**Examples:**

```
Input : list: 4->2->1->5->3
        n = 2
Output : 5
```

**Algorithm:**

```
findNthFromLast(head, n, count, nth_last)
    if head == NULL then
        return

    findNthFromLast(head->next, n, count, nth_last)
    count = count + 1
    if count == n then
        nth_last = head

findNthFromLastUtil(head, n)
    Initialize nth_last = NULL
    Initialize count = 0
```

```
findNthFromLast(head, n, &count, &nth_last)

if nth_last != NULL then
    print nth_last->data
else
    print "Node does not exists"
```

**Note:** Parameters **count** and **nth\_last** will be pointer variables in **findNthFromLast()**.

C++

```
// C++ implementation to recursively find the nth node from
// the last of the linked list
#include <bits/stdc++.h>

using namespace std;

// structure of a node of a linked list
struct Node {
    int data;
    Node* next;
};

// function to get a new node
Node* getNode(int data)
{
    // allocate space
    Node* newNode = new Node;

    // put in data
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}

// function to recursively find the nth node from
// the last of the linked list
void findNthFromLast(Node* head, int n, int* count,
                      Node** nth_last)
{
    // if list is empty
    if (!head)
        return;

    // recursive call
    findNthFromLast(head->next, n, count, nth_last);

    // increment count
    (*count)++;
}
```

```
*count = *count + 1;

// if true, then head is the nth node from the last
if (*count == n)
    *nth_last = head;
}

// utility function to find the nth node from
// the last of the linked list
void findNthFromLastUtil(Node* head, int n)
{
    // Initialize
    Node* nth_last = NULL;
    int count = 0;

    // find nth node from the last
    findNthFromLast(head, n, &count, &nth_last);

    // if node exists, then print it
    if (nth_last != NULL)
        cout << "Nth node from last is: "
            << nth_last->data;
    else
        cout << "Node does not exists";
}

// Driver program to test above
int main()
{
    // linked list: 4->2->1->5->3
    Node* head = getNode(4);
    head->next = getNode(2);
    head->next->next = getNode(1);
    head->next->next->next = getNode(5);
    head->next->next->next->next = getNode(3);

    int n = 2;

    findNthFromLastUtil(head, n);

    return 0;
}
```

### Java

```
// Java implementation to recursively
// find the nth node from the last
// of the linked list
import java.util.*;
```

```
class GFG
{
    static int count = 0, data = 0;

    // a node of a linked list
    static class Node
    {
        int data;
        Node next;
    }

    // function to get a new node
    static Node getNode(int data)
    {
        // allocate space
        Node newNode = new Node();

        // put in data
        newNode.data = data;
        newNode.next = null;
        return newNode;
    }

    // function to recursively
    // find the nth node from
    // the last of the linked list
    static void findNthFromLast(Node head, int n,
                                Node nth_last)
    {
        // if list is empty
        if (head == null)
            return;

        // recursive call
        findNthFromLast(head.next, n, nth_last);

        // increment count
        count = count + 1;

        // if true, then head is the
        // nth node from the last
        if (count == n)
        {
            data = head.data;
        }
    }

    // utility function to find
    // the nth node from the last
    // of the linked list
    static void findNthFromLastUtil(Node head, int n)
    {
```

```
// Initialize
Node nth_last = new Node();
count = 0;

// find nth node from the last
findNthFromLast(head, n, nth_last);

// if node exists, then print it
if (nth_last != null)
    System.out.println("Nth node from last is: " +
        data);
else
    System.out.println("Node does not exists");
}

// Driver Code
public static void main(String args[])
{
// linked list: 4.2.1.5.3
Node head = getNode(4);
head.next = getNode(2);
head.next.next = getNode(1);
head.next.next.next = getNode(5);
head.next.next.next.next = getNode(3);

int n = 2;
findNthFromLastUtil(head, n);
}
}

// This code is contributed
// by Arnab Kundu
```

**Output:**

```
Nth node from last is: 5
```

**Time Complexity:** O(n), where ‘n’ is the number of nodes in the linked list.

**Improved By :** [andrew1234](#)

**Source**

<https://www.geeksforgeeks.org/recursive-approach-to-find-nth-node-from-the-end-in-the-linked-list/>

## Chapter 192

# Recursive approach for alternating split of Linked List

Recursive approach for alternating split of Linked List - GeeksforGeeks

Given a linked list, split the linked list into two with alternate nodes.

Examples:

```
Input : 1 2 3 4 5 6 7
Output : 1 3 5 7
          2 4 6
```

```
Input : 1 4 5 6
Output : 1 5
          4 6
```

We have discussed [Iterative splitting of linked list](#).

The idea is to begin from two nodes first and second. Let us call these nodes as ‘a’ and ‘b’. We recurs

**CPP**

```
// CPP code to split linked list
#include <bits/stdc++.h>
using namespace std;

// Node structure
struct Node {
    int data;
    struct Node* next;
};

Node* splitList(Node* head) {
    if (head == NULL || head->next == NULL)
        return head;

    Node* a = head;
    Node* b = head->next;

    Node* prev_a = NULL;
    Node* curr = b;

    while (curr != NULL) {
        curr->next = prev_a;
        prev_a = curr;
        curr = curr->next;
    }

    head = a;
    b->next = NULL;
    return head;
}
```

```
// Function to push nodes
// into linked list
void push(Node** head, int new_data)
{
    Node* new_node = new Node;
    new_node->data = new_data;
    new_node->next = (*head);
    (*head) = new_node;
}

// We basically remove link between 'a'
// and its next. Similarly we remove link
// between 'b' and its next. Then we recur
// for remaining lists.
void moveNode(Node* a, Node* b)
{
    if (b == NULL || a == NULL)
        return;

    if (a->next != NULL)
        a->next = a->next->next;

    if (b->next != NULL)
        b->next = b->next->next;

    moveNode(a->next, b->next);
}

// function to split linked list
void alternateSplitLinkedList(Node* head, Node** aRef,
                             Node** bRef)
{
    Node* curr = head;
    *aRef = curr;
    *bRef = curr->next;
    moveNode(*aRef, *bRef);
}

void display(Node* head)
{
    Node* curr = head;
    while (curr != NULL) {
        printf("%d ", curr->data);
        curr = curr->next;
    }
}
```

```
// Driver code
int main()
{
    Node* head = NULL;
    Node *a = NULL, *b = NULL;

    push(&head, 7);
    push(&head, 6);
    push(&head, 5);
    push(&head, 4);
    push(&head, 3);
    push(&head, 2);
    push(&head, 1);

    alternateSplitLinkedList(head, &a, &b);

    printf("a : ");
    display(a);
    printf("\nb : ");
    display(b);

    return 0;
}
```

**Output:**

```
a : 1 3 5 7
b : 2 4 6
```

**Source**

<https://www.geeksforgeeks.org/recursive-approach-alternating-split-linked-list/>

## Chapter 193

# Recursive function to delete k-th node from linked list

Recursive function to delete k-th node from linked list - GeeksforGeeks

Given a singly linked list delete node at k-th position without using loop.

Examples:

```
Input : list = 9->8->3->5->2->1
        k = 4
Output : 9->8->3->2->1
```

```
Input : list = 0->0->1->6->2->3
        k = 3
Output : 0->0->6->2->3
```

We recursively reduce value of k. When k reaches 1, we delete current node and return next of current node as new node. When function returns, we link the returned node as next of previous node.

```
// Recursive CPP program to delete k-th node
// of a linked list
#include <bits/stdc++.h>
using namespace std;

struct Node {
    int data;
    struct Node* next;
};

// Deletes k-th node and returns new header.
```

```
Node* deleteNode(Node* start, int k)
{
    // If invalid k
    if (k < 1)
        return start;

    // If linked list is empty
    if (start == NULL)
        return NULL;

    // Base case (start needs to be deleted)
    if (k == 1)
    {
        Node *res = start->next;
        delete(start);
        return res;
    }

    start->next = deleteNode(start->next, k-1);
    return start;
}

/* Utility function to insert a node at the beginning */
void push(struct Node **head_ref, int new_data)
{
    struct Node *new_node = new Node;
    new_node->data = new_data;
    new_node->next = *head_ref;
    *head_ref = new_node;
}

/* Utility function to print a linked list */
void printList(struct Node *head)
{
    while (head!=NULL)
    {
        cout << head->data << " ";
        head = head->next;
    }
    printf("\n");
}

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

    /* Create following linked list
       head->data = 10
       head->next->data = 20
       head->next->next->data = 30
       head->next->next->next->data = 40
       head->next->next->next->next = NULL
```

```
12->15->10->11->5->6->2->3 */
push(&head,3);
push(&head,2);
push(&head,6);
push(&head,5);
push(&head,11);
push(&head,10);
push(&head,15);
push(&head,12);

int k = 3;
head = deleteNode(head, k);

printf("\nModified Linked List: ");
printList(head);

return 0;
}
```

Output:

```
Modified Linked List: 12 15 11 5 6 2 3
```

## Source

<https://www.geeksforgeeks.org/recursive-function-delete-k-th-node-linked-list/>

## Chapter 194

# Recursive insertion and traversal linked list

Recursive insertion and traversal linked list - GeeksforGeeks

We have discussed different methods of [linked list insertion](#). How to recursively create a linked list?

### Recursively inserting at the end:

To create a Linked list using recursion follow these steps. Below steps insert a new node recursively at the end of linked list.

```
// Function to insert a new node at the
// end of linked list using recursion.
Node* insertEnd(Node* head, int data)
{
    // If linked list is empty, create a
    // new node (Assuming newNode() allocates
    // a new node with given data)
    if (head == NULL)
        return newNode(data);

    // If we have not reached end, keep traversing
    // recursively.
    else
        head->next = insertEnd(head->next, data);
    return head;
}
```

### Recursively traversing the list:

The idea is simple, we print current node and recur for remaining list.

```
void traverse(Node* head)
{
    if (head == NULL)
        return;

    // If head is not NULL, print current node
    // and recur for remaining list
    cout << head->data << " ";

    traverse(head->next);
}
```

**Complete Program:**

Below is complete program to demonstrate working of insert and traverse a linked list.

```
// Recursive CPP program to recursively insert
// a node and recursively print the list.
#include <bits/stdc++.h>
using namespace std;
struct Node {
    int data;
    Node* next;
};

// Allocates a new node with given data
Node *newNode(int data)
{
    Node *new_node = new Node;
    new_node->data = data;
    new_node->next = NULL;
    return new_node;
}

// Function to insert a new node at the
// end of linked list using recursion.
Node* insertEnd(Node* head, int data)
{
    // If linked list is empty, create a
    // new node (Assuming newNode() allocates
    // a new node with given data)
    if (head == NULL)
        return newNode(data);

    // If we have not reached end, keep traversing
    // recursively.
    else
```

```
        head->next = insertEnd(head->next, data);
        return head;
    }

void traverse(Node* head)
{
    if (head == NULL)
        return;

    // If head is not NULL, print current node
    // and recur for remaining list
    cout << head->data << " ";

    traverse(head->next);
}

// Driver code
int main()
{
    Node* head = NULL;
    head = insertEnd(head, 6);
    head = insertEnd(head, 8);
    head = insertEnd(head, 10);
    head = insertEnd(head, 12);
    head = insertEnd(head, 14);
    traverse(head);
}
```

Output:

6 8 10 12 14

## Source

<https://www.geeksforgeeks.org/recursive-insertion-and-traversal-linked-list/>

## Chapter 195

# Recursive selection sort for singly linked list | Swapping node links

Recursive selection sort for singly linked list | Swapping node links - GeeksforGeeks

Given a singly linked list containing  $n$  nodes. The problem is to sort the list using recursive selection sort technique. The approach should be such that it involves swapping node links instead of swapping nodes data.

Examples:

```
Input : 10 -> 12 -> 8 -> 4 -> 6
Output : 4 -> 6 -> 8 -> 10 -> 12
```

In Selection Sort, we first find minimum element, swap it with the beginning node and recur for remaining list. Below is recursive implementation of these steps for linked list.

```
recurSelectionSort(head)
    if head->next == NULL
        return head
    Initialize min = head
    Initialize beforeMin = NULL
    Initialize ptr = head

    while ptr->next != NULL
        if min->data > ptr->next->data
            min = ptr->next
            beforeMin = ptr
```

```
ptr = ptr->next

if min != head
    swapNodes(&head, head, min, beforeMin)

head->next = recurSelectionSort(head->next)
return head

swapNodes(head_ref, currX, currY, prevY)
    head_ref = currY
    prevY->next = currX

    Initialize temp = currY->next
    currY->next = currX->next
    currX->next = temp
```

The `swapNodes(head_ref, currX, currY, prevY)` is based on the approach discussed [here](#) but it is modified accordingly for the implementation of this post.

```
// C++ implementation of recursive selection sort
// for singly linked list | Swapping node links
#include <bits/stdc++.h>
using namespace std;

// A Linked list node
struct Node {
    int data;
    struct Node* next;
};

// function to swap nodes 'currX' and 'currY' in a
// linked list without swapping data
void swapNodes(struct Node** head_ref, struct Node* currX,
               struct Node* currY, struct Node* prevY)
{
    // make 'currY' as new head
    *head_ref = currY;

    // adjust links
    prevY->next = currX;

    // Swap next pointers
    struct Node* temp = currY->next;
    currY->next = currX->next;
    currX->next = temp;
}

// function to sort the linked list using
```

```
// recursive selection sort technique
struct Node* recurSelectionSort(struct Node* head)
{
    // if there is only a single node
    if (head->next == NULL)
        return head;

    // 'min' - pointer to store the node having
    // minimum data value
    struct Node* min = head;

    // 'beforeMin' - pointer to store node previous
    // to 'min' node
    struct Node* beforeMin = NULL;
    struct Node* ptr;

    // traverse the list till the last node
    for (ptr = head; ptr->next != NULL; ptr = ptr->next) {

        // if true, then update 'min' and 'beforeMin'
        if (ptr->next->data < min->data) {
            min = ptr->next;
            beforeMin = ptr;
        }
    }

    // if 'min' and 'head' are not same,
    // swap the head node with the 'min' node
    if (min != head)
        swapNodes(&head, head, min, beforeMin);

    // recursively sort the remaining list
    head->next = recurSelectionSort(head->next);

    return head;
}

// function to sort the given linked list
void sort(struct Node** head_ref)
{
    // if list is empty
    if ((*head_ref) == NULL)
        return;

    // sort the list using recursive selection
    // sort technique
    *head_ref = recurSelectionSort(*head_ref);
}
```

```
// function to insert a node at the
// beginning of the linked list
void push(struct Node** head_ref, int new_data)
{
    // allocate node
    struct Node* new_node =
        (struct Node*)malloc(sizeof(struct Node));

    // put in the data
    new_node->data = new_data;

    // link the old list to the new node
    new_node->next = (*head_ref);

    // move the head to point to the new node
    (*head_ref) = new_node;
}

// function to print the linked list
void printList(struct Node* head)
{
    while (head != NULL) {
        cout << head->data << " ";
        head = head->next;
    }
}

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

    // create linked list 10->12->8->4->6
    push(&head, 6);
    push(&head, 4);
    push(&head, 8);
    push(&head, 12);
    push(&head, 10);

    cout << "Linked list before sorting:n";
    printList(head);

    // sort the linked list
    sort(&head);

    cout << "\nLinked list after sorting:n";
    printList(head);
}
```

```
    return 0;  
}
```

Output:

```
Linked list before sorting:  
10 12 8 4 6  
Linked list after sorting:  
4 6 8 10 12
```

Time Complexity:  $O(n^2)$

## Source

<https://www.geeksforgeeks.org/recursive-selection-sort-singly-linked-list-swapping-node-links/>

## Chapter 196

# Recursively Reversing a linked list (A simple implementation)

Recursively Reversing a linked list (A simple implementation) - GeeksforGeeks

Given pointer to the head node of a linked list, the task is to recursively reverse the linked list. We need to reverse the list by changing links between nodes.

Examples:

```
Input : Head of following linked list  
       1->2->3->4->NULL  
Output : Linked list should be changed to,  
        4->3->2->1->NULL
```

```
Input : Head of following linked list  
       1->2->3->4->5->NULL  
Output : Linked list should be changed to,  
        5->4->3->2->1->NULL
```

```
Input : NULL  
Output : NULL
```

```
Input : 1->NULL  
Output : 1->NULL
```

We have discussed an iterative and two recursive approaches in [previous post on reverse a linked list](#).

In this approach of reversing a linked list by passing a single pointer what we are trying to do is that we are making the previous node of the current node as his next node to reverse the linked list.

1. We return the pointer of next node to his previous(current) node and then make the previous node as the next node of returned node and then returning the current node.
2. We first traverse till the last node and making the last node as the head node of reversed linked list and then applying the above procedure in the recursive manner.

```
// Recursive C++ program to reverse
// a linked list
#include <iostream>
using namespace std;

/* Link list node */
struct Node {
    int data;
    struct Node* next;
    Node(int data)
    {
        this->data = data;
        next = NULL;
    }
};

struct LinkedList {
    Node* head;
    LinkedList()
    {
        head = NULL;
    }

    /* Function to reverse the linked list */
    Node* reverse(Node* node)
    {
        if (node == NULL)
            return NULL;
        if (node->next == NULL) {
            head = node;
            return node;
        }
        Node* node1 = reverse(node->next);
        node1->next = node;
        node->next = NULL;
        return node;
    }

    /* Function to print linked list */
    void print()
    {
        struct Node* temp = head;
        while (temp != NULL) {
```

```
        cout << temp->data << " ";
        temp = temp->next;
    }
}

void push(int data)
{
    Node* temp = new Node(data);
    temp->next = head;
    head = temp;
}
};

/* Driver program to test above function*/
int main()
{
    /* Start with the empty list */
    LinkedList ll;
    ll.push(20);
    ll.push(4);
    ll.push(15);
    ll.push(85);

    cout << "Given linked list\n";
    ll.print();

    ll.reverse(ll.head);

    cout << "\nReversed Linked list \n";
    ll.print();
    return 0;
}
```

**Output:**

```
Given linked list
85 15 4 20
Reversed Linked list
20 4 15 85
```

**Source**

<https://www.geeksforgeeks.org/recursively-reversing-a-linked-list-a-simple-implementation/>

## Chapter 197

# Remove all occurrences of duplicates from a sorted Linked List

Remove all occurrences of duplicates from a sorted Linked List - GeeksforGeeks

Given a sorted linked list, delete all nodes that have duplicate numbers (all occurrences), leaving only numbers that appear once in the original list.

**Examples:**

Input : 23->28->28->35->49->49->53->53  
Output : 23->35

Input : 11->11->11->11->75->75  
Output : empty List

Note that this is different from [Remove Duplicates From Linked List](#)

The idea is to maintain a pointer (*prev*) to the node which just previous to the block of nodes we are checking for duplicates. In the first example, the pointer *prev* would point to 23 while we check for duplicates for the node 28. Once we reach the last duplicate node with value 28 (name it *current* pointer), we can make the next field of *prev* node to be the next of *current* and update *current=next*. This would delete the block of nodes with value 28 which has duplicates.

C++

```
// C++ program to remove all
// occurrences of duplicates
// from a sorted linked list.
```

```
#include <bits/stdc++.h>
using namespace std;

// A linked list node
struct Node
{
    int data;
    struct Node *next;
};

// Utility function
// to create a new Node
struct Node *newNode(int data)
{
    Node *temp = new Node;
    temp -> data = data;
    temp -> next = NULL;
    return temp;
}

// Function to print nodes
// in a given linked list.
void printList(struct Node *node)
{
    while (node != NULL)
    {
        printf("%d ", node -> data);
        node = node -> next;
    }
}

// Function to remove all occurrences
// of duplicate elements
void removeAllDuplicates(struct Node* &start)
{
    // create a dummy node
    // that acts like a fake
    // head of list pointing
    // to the original head
    Node* dummy = new Node;

    // dummy node points
    // to the original head
    dummy -> next = start;

    // Node pointing to last
    // node which has no duplicate.
    Node* prev = dummy;
```

```
// Node used to traverse
// the linked list.
Node* current = start;

while(current != NULL)
{
    // Until the current and
    // previous values are
    // same, keep updating current
    while(current -> next != NULL &&
          prev -> next -> data == current -> next -> data)
        current = current -> next;

    // if current has unique value
    // i.e current is not updated,
    // Move the prev pointer to
    // next node
    if (prev -> next == current)
        prev = prev -> next;

    // when current is updated
    // to the last duplicate
    // value of that segment,
    // make prev the next of current
    else
        prev -> next = current -> next;

    current = current -> next;
}

// update original head to
// the next of dummy node
start = dummy -> next;
}

// Driver Code
int main()
{
    // 23->28->28->35->49->49->53->53
    struct Node* start = newNode(23);
    start -> next = newNode(28);
    start -> next -> next = newNode(28);
    start -> next ->
    next -> next = newNode(35);
    start -> next ->
    next -> next -> next = newNode(49);
    start -> next ->
```

```
next -> next ->
next -> next = newNode(49);
start -> next ->
next -> next ->
next -> next -> next = newNode(53);
start -> next ->
next -> next ->
next -> next ->
next -> next = newNode(53);
cout << "List before removal " <<
        "of duplicates\n";
printList(start);

removeAllDuplicates(start);

cout << "\nList after removal " <<
        "of duplicates\n";
printList(start);
return 0;
}

// This code is contributed
// by NIKHIL JINDAL
```

**Java**

```
/* Java program to remove all occurrences of
   duplicates from a sorted linked list */

/* class to create Linked lIst */
class LinkedList
{
    Node head=null; /* head of linked list */
    class Node
    {
        int val; /* value in the node */
        Node next;
        Node(int v)
        {
            /* default value of the next
               pointer field */
            val = v;
            next = null;
        }
    }

    /* Function to insert data nodes into
       the Linked List at the front */
```

```
public void insert(int data)
{
    Node new_node = new Node(data);
    new_node.next = head;
    head = new_node;
}

/* Function to remove all occurrences
   of duplicate elements */
public void removeAllDuplicates()
{
    /* create a dummy node that acts like a fake
       head of list pointing to the original head*/
    Node dummy = new Node(0);

    /* dummy node points to the original head*/
    dummy.next = head;
    Node prev = dummy;
    current = head;

    while (current != null)
    {
        /* Until the current and previous values
           are same, keep updating current */
        while (current.next != null &&
               prev.next.val == current.next.val)
            current = current.next;

        /* if current has unique value i.e current
           is not updated, Move the prev pointer
           to next node*/
        if (prev.next == current)
            prev = prev.next;

        /* when current is updated to the last
           duplicate value of that segment, make
           prev the next of current*/
        else
            prev.next = current.next;

        current = current.next;
    }

    /* update original head to the next of dummy
       node */
    head = dummy.next;
}
```

```
/* Function to print the list elements */
public void printList()
{
    Node trav=head;
    if (head==null)
        System.out.print(" List is empty" );
    while (trav != null)
    {
        System.out.print(trav.val + " ");
        trav = trav.next;
    }
}

/* Driver program to test above functions */
public static void main(String[] args)
{
    LinkedList ll = new LinkedList();
    ll.insert(53);
    ll.insert(53);
    ll.insert(49);
    ll.insert(49);
    ll.insert(35);
    ll.insert(28);
    ll.insert(28);
    ll.insert(23);
    System.out.println("Before removal of duplicates");
    ll.printList();

    ll.removeAllDuplicates();

    System.out.println("\nAfter removal of duplicates");
    ll.printList();
}
```

Output:

```
List before removal of duplicates
23 28 28 35 49 49 53 53
List after removal of duplicates
23 35
```

Time Complexity : O(n)

Improved By : [nik1996](#)

**Source**

<https://www.geeksforgeeks.org/remove-occurrences-duplicates-sorted-linked-list/>

## Chapter 198

# Remove duplicates from a sorted doubly linked list

Remove duplicates from a sorted doubly linked list - GeeksforGeeks

Given a sorted doubly linked list containing **n** nodes. The problem is to remove duplicate nodes from the given list.

Examples:

```
Input : DLL: 4<->4<->4<->4<->6<->8<->8<->10<->12<->12
Output : 4<->6<->8<->10<->12
```

**Algorithm:**

```
removeDuplicates(head_ref, x)
    if head_ref == NULL
        return
    Initialize current = head_ref
    while current->next != NULL
        if current->data == current->next->data
            deleteNode(head_ref, current->next)
        else
            current = current->next
```

The algorithm for **deleteNode(head\_ref, current)** (which deletes the node using the pointer to the node) is discussed in [this](#) post.

```
/* C++ implementation to remove duplicates from a
sorted doubly linked list */
#include <bits/stdc++.h>
```

```
using namespace std;

/* a node of the doubly linked list */
struct Node {
    int data;
    struct Node* next;
    struct Node* prev;
};

/* Function to delete a node in a Doubly Linked List.
   head_ref --> pointer to head node pointer.
   del --> pointer to node to be deleted. */
void deleteNode(struct Node** head_ref, struct Node* del)
{
    /* base case */
    if (*head_ref == NULL || del == NULL)
        return;

    /* If node to be deleted is head node */
    if (*head_ref == del)
        *head_ref = del->next;

    /* Change next only if node to be deleted
       is NOT the last node */
    if (del->next != NULL)
        del->next->prev = del->prev;

    /* Change prev only if node to be deleted
       is NOT the first node */
    if (del->prev != NULL)
        del->prev->next = del->next;

    /* Finally, free the memory occupied by del*/
    free(del);
}

/* function to remove duplicates from a
   sorted doubly linked list */
void removeDuplicates(struct Node** head_ref)
{
    /* if list is empty */
    if ((*head_ref) == NULL)
        return;

    struct Node* current = *head_ref;
    struct Node* next;
```

```
/* traverse the list till the last node */
while (current->next != NULL) {

    /* Compare current node with next node */
    if (current->data == current->next->data)

        /* delete the node pointed to by
           'current->next' */
        deleteNode(head_ref, current->next);

    /* else simply move to the next node */
    else
        current = current->next;
}

/* Function to insert a node at the beginning
   of the Doubly Linked List */
void push(struct Node** head_ref, int new_data)
{
    /* allocate node */
    struct Node* new_node =
        (struct Node*)malloc(sizeof(struct Node));

    /* put in the data */
    new_node->data = new_data;

    /* since we are adding at the begining,
       prev is always NULL */
    new_node->prev = NULL;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* change prev of head node to new node */
    if ((*head_ref) != NULL)
        (*head_ref)->prev = new_node;

    /* move the head to point to the new node */
    (*head_ref) = new_node;
}

/* Function to print nodes in a given doubly linked list */
void printList(struct Node* head)
{
    /* if list is empty */
    if (head == NULL)
        cout << "Doubly Linked list empty";
```

```
while (head != NULL) {
    cout << head->data << " ";
    head = head->next;
}
}

/* Driver program to test above functions*/
int main()
{
    /* Start with the empty list */
    struct Node* head = NULL;

    /* Create the doubly linked list:
     * 4<->4<->4<->4<->6<->8<->8<->10<->12<->12 */
    push(&head, 12);
    push(&head, 12);
    push(&head, 10);
    push(&head, 8);
    push(&head, 8);
    push(&head, 6);
    push(&head, 4);
    push(&head, 4);
    push(&head, 4);
    push(&head, 4);

    cout << "Original Doubly linked list:n";
    printList(head);

    /* remove duplicate nodes */
    removeDuplicates(&head);

    cout << "\nDoubly linked list after"
        " removing duplicates:n";
    printList(head);

    return 0;
}
```

Output:

```
Original Doubly linked list:
4 4 4 4 6 8 8 10 12 12
Doubly linked list after removing duplicates:
4 6 8 10 12
```

Time Complexity: O(n)

**Source**

<https://www.geeksforgeeks.org/remove-duplicates-sorted-doubly-linked-list/>

## Chapter 199

# Remove duplicates from a sorted linked list

Remove duplicates from a sorted linked list - GeeksforGeeks

Write a removeDuplicates() function which takes a list sorted in non-decreasing order and deletes any duplicate nodes from the list. The list should only be traversed once.

For example if the linked list is 11->11->11->21->43->43->60 then removeDuplicates() should convert the list to 11->21->43->60.

### Algorithm:

Traverse the list from the head (or start) node. While traversing, compare each node with its next node. If data of next node is same as current node then delete the next node. Before we delete a node, we need to store next pointer of the node

### Implementation:

Functions other than removeDuplicates() are just to create a linked linked list and test removeDuplicates().

## C

```
/* C Program to remove duplicates from a sorted linked list */
#include<stdio.h>
#include<stdlib.h>

/* Link list node */
struct Node
{
    int data;
    struct Node* next;
};

/* The function removes duplicates from a sorted list */
```

```
void removeDuplicates(struct Node* head)
{
    /* Pointer to traverse the linked list */
    struct Node* current = head;

    /* Pointer to store the next pointer of a node to be deleted*/
    struct Node* next_next;

    /* do nothing if the list is empty */
    if (current == NULL)
        return;

    /* Traverse the list till last node */
    while (current->next != NULL)
    {
        /* Compare current node with next node */
        if (current->data == current->next->data)
        {
            /* The sequence of steps is important*/
            next_next = current->next->next;
            free(current->next);
            current->next = next_next;
        }
        else /* This is tricky: only advance if no deletion */
        {
            current = current->next;
        }
    }
}

/* UTILITY FUNCTIONS */
/* Function to insert a node at the beginning of the linked list */
void push(struct Node** head_ref, int new_data)
{
    /* allocate node */
    struct Node* new_node =
        (struct Node*) malloc(sizeof(struct Node));

    /* put in the data */
    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;
}
```

```
/* Function to print nodes in a given linked list */
void printList(struct Node *node)
{
    while (node!=NULL)
    {
        printf("%d ", node->data);
        node = node->next;
    }
}

/* Drier program to test above functions*/
int main()
{
    /* Start with the empty list */
    struct Node* head = NULL;

    /* Let us create a sorted linked list to test the functions
     * Created linked list will be 11->11->11->13->13->20 */
    push(&head, 20);
    push(&head, 13);
    push(&head, 13);
    push(&head, 11);
    push(&head, 11);
    push(&head, 11);

    printf("\n Linked list before duplicate removal ");
    printList(head);

    /* Remove duplicates from linked list */
    removeDuplicates(head);

    printf("\n Linked list after duplicate removal ");
    printList(head);

    return 0;
}
```

### Java

```
// Java program to remove duplicates from a sorted linked list
class LinkedList
{
    Node head; // head of list

    /* Linked list Node*/
    class Node
    {
        int data;
```

```
Node next;
Node(int d) {data = d; next = null; }
}

void removeDuplicates()
{
    /*Another reference to head*/
    Node current = head;

    /* Pointer to store the next pointer of a node to be deleted*/
    Node next_next;

    /* do nothing if the list is empty */
    if (head == null)
        return;

    /* Traverse list till the last node */
    while (current.next != null) {

        /*Compare current node with the next node */
        if (current.data == current.next.data) {
            next_next = current.next.next;
            current.next = null;
            current.next = next_next;
        }
        else // advance if no deletion
            current = current.next;
    }
}

/* Utility functions */

/* Inserts a new Node at front of the list. */
public void push(int new_data)
{
    /* 1 & 2: Allocate the Node &
       Put in the data*/
    Node new_node = new Node(new_data);

    /* 3. Make next of new Node as head */
    new_node.next = head;

    /* 4. Move the head to point to new Node */
    head = new_node;
}

/* Function to print linked list */
void printList()
```

```
{  
    Node temp = head;  
    while (temp != null)  
    {  
        System.out.print(temp.data+" ");  
        temp = temp.next;  
    }  
    System.out.println();  
}  
  
/* Drier program to test above functions */  
public static void main(String args[])  
{  
    LinkedList llist = new LinkedList();  
    llist.push(20);  
    llist.push(13);  
    llist.push(13);  
    llist.push(11);  
    llist.push(11);  
    llist.push(11);  
  
    System.out.println("List before removal of duplicates");  
    llist.printList();  
  
    llist.removeDuplicates();  
  
    System.out.println("List after removal of elements");  
    llist.printList();  
}  
}  
/* This code is contributed by Rajat Mishra */
```

Output:

```
Linked list before duplicate removal 11 11 11 13 13 20  
Linked list after duplicate removal 11 13 20
```

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

**Recursive Approach :**

```
/* C recursive Program to remove duplicates from a sorted linked list */  
#include<stdio.h>  
#include<stdlib.h>  
  
/* Link list node */
```

```
struct Node
{
    int data;
    struct Node* next;
};

/* UTILITY FUNCTIONS */
/* Function to insert a node at the beginning of the linked list */
void push(struct Node** head_ref, int new_data)
{
    /* allocate node */
    struct Node* new_node =
        (struct Node*) malloc(sizeof(struct Node));

    /* put in the data */
    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;
}

/* Function to print nodes in a given linked list */
void printList(struct Node *node)
{
    while (node!=NULL)
    {
        printf("%d ", node->data);
        node = node->next;
    }
}

Node* deleteDuplicates(Node* head) {
    if (head == nullptr) return nullptr;
    if (head->next == nullptr) return head;
    if (head->data == head->next->data) {
        //if find next element duplicate, skip it and return deleted-duplicates with current head
        head->next = head->next->next;
        return deleteDuplicates(head);
    }
    else {
        // if doesn't find next element duplicate, leave head and check from next element
        head->next = deleteDuplicates(head->next);
        return head;
    }
}
```

```
int main()
{
    /* Start with the empty list */
    struct Node* head = NULL;

    /* Let us create a sorted linked list to test the functions
     * Created linked list will be 11->11->11->13->13->20 */
    push(&head, 20);
    push(&head, 13);
    push(&head, 13);
    push(&head, 11);
    push(&head, 11);
    push(&head, 11);

    printf("\n Linked list before duplicate removal ");
    printList(head);

    /* Remove duplicates from linked list */
    head = deleteDuplicates(head);

    printf("\n Linked list after duplicate removal ");
    printList(head);

    return 0;
}
/* This code is contributed by Yogesh shukla */
```

#### Output

```
Linked list before duplicate removal 11 11 11 13 13 20
Linked list after duplicate removal 11 13 20
```

#### Related Article :

[Remove all occurrences of duplicates from a sorted Linked List](#)

#### References:

[cslibrary.stanford.edu/105/LinkedListProblems.pdf](http://cslibrary.stanford.edu/105/LinkedListProblems.pdf)

Improved By : [Yogesh Shukla](#) 1

#### Source

<https://www.geeksforgeeks.org/remove-duplicates-from-a-sorted-linked-list/>

## Chapter 200

# Remove duplicates from a sorted linked list using recursion

Remove duplicates from a sorted linked list using recursion - GeeksforGeeks

Write a removeDuplicates() function which takes a list sorted in non-decreasing order and deletes any duplicate nodes from the list. The list should only be traversed once.

For example if the linked list is 11->11->11->21->43->43->60 then removeDuplicates() should convert the list to 11->21->43->60.

### Algorithm:

Traverse the list recursively from the head (or start) to end and after completion of recursion calls, compare the next node(returned node) and current node(head). If data of both nodes are equal then return the next (**head-> next**) node else return the current **node(head)**.

### Implementation:

Functions other than removeDuplicates() are just to create a linked linked list and test removeDuplicates().

```
/* C Program to remove duplicates  
from a sorted linked list */  
#include <bits/stdc++.h>  
#include <stdlib.h>  
  
/* Link list node */  
struct Node {  
    int data;  
    struct Node* next;  
};  
  
/* The function removes duplicates from a sorted list */  
struct Node* removeDuplicates(struct Node* head)  
{
```

```
/* if head is null then return*/
if (head == NULL)
    return NULL;

/* Remove duplicates from list after head */
head->next = removeDuplicates(head->next);

// Check if head itself is duplicate
if (head->next != NULL &&
    head->next->data == head->data) {

    Node* res = head->next;
    delete head;
    return res;
}

return head;
}

/* UTILITY FUNCTIONS */
/* Function to insert a node at
   the beginning of the linked list */
void push(struct Node** head_ref, int new_data)
{
    struct Node* new_node = new Node;
    new_node->data = new_data;
    new_node->next = (*head_ref);
    (*head_ref) = new_node;
}

/* Function to print nodes in a given linked list */
void printList(struct Node* node)
{
    while (node != NULL) {
        printf("%d ", node->data);
        node = node->next;
    }
}

/* Drier program to test above functions*/
int main()
{
    /* Start with the empty list */
    struct Node* head = NULL;

    /* Let us create a sorted linked list to test the functions
       Created linked list will be 11->11->11->13->13->20 */
    push(&head, 20);
```

```
push(&head, 13);
push(&head, 13);
push(&head, 11);
push(&head, 11);
push(&head, 11);

printf("\n Linked list before duplicate removal ");
printList(head);

/* Remove duplicates from linked list */
struct Node* h = removeDuplicates(head);

printf("\n Linked list after duplicate removal ");
printList(h);

return 0;
}
```

**Output:**

```
Linked list before duplicate removal 11 11 11 13 13 20
Linked list after duplicate removal 11 13 20
```

**Source**

<https://www.geeksforgeeks.org/remove-duplicates-sorted-linked-list-using-recursion/>

# Chapter 201

## Remove duplicates from an unsorted doubly linked list

Remove duplicates from an unsorted doubly linked list - GeeksforGeeks

Given an unsorted doubly linked list containing  $n$  nodes. The problem is to remove duplicate nodes from the given list.

Examples:

```
Input : DLL: 8<->4<->4<->6<->4<->8<->4<->10<->12<->12
Output : 8<->4<->6<->10<->12
```

### Method 1 (Naive Approach):

This is the simplest way where two loops are used. Outer loop is used to pick the elements one by one and inner loop compares the picked element with rest of the elements.

```
// C++ implementation to remove duplicates from an
// unsorted doubly linked list
#include <bits/stdc++.h>

using namespace std;

// a node of the doubly linked list
struct Node {
    int data;
    struct Node* next;
    struct Node* prev;
};

// Function to delete a node in a Doubly Linked List.
// head_ref --> pointer to head node pointer.
// del --> pointer to node to be deleted.
```

```
void deleteNode(struct Node** head_ref, struct Node* del)
{
    // base case
    if (*head_ref == NULL || del == NULL)
        return;

    // If node to be deleted is head node
    if (*head_ref == del)
        *head_ref = del->next;

    // Change next only if node to be deleted
    // is NOT the last node
    if (del->next != NULL)
        del->next->prev = del->prev;

    // Change prev only if node to be deleted
    // is NOT the first node
    if (del->prev != NULL)
        del->prev->next = del->next;

    // Finally, free the memory occupied by del
    free(del);
}

// function to remove duplicates from
// an unsorted doubly linked list
void removeDuplicates(struct Node** head_ref)
{
    // if DLL is empty or if it contains only
    // a single node
    if ((*head_ref) == NULL ||
        (*head_ref)->next == NULL)
        return;

    struct Node* ptr1, *ptr2;

    // pick elements one by one
    for (ptr1 = *head_ref; ptr1 != NULL; ptr1 = ptr1->next) {
        ptr2 = ptr1->next;

        // Compare the picked element with the
        // rest of the elements
        while (ptr2 != NULL) {

            // if duplicate, then delete it
            if (ptr1->data == ptr2->data) {

                // store pointer to the node next to 'ptr2'
                if (ptr2->next != NULL)
                    ptr1->next = ptr2->next;
                else
                    *head_ref = NULL;
                free(ptr2);
            }
            ptr2 = ptr2->next;
        }
    }
}
```

```
    struct Node* next = ptr2->next;

    // delete node pointed to by 'ptr2'
    deleteNode(head_ref, ptr2);

    // update 'ptr2'
    ptr2 = next;
}

// else simply move to the next node
else
    ptr2 = ptr2->next;
}
}

// Function to insert a node at the beginning
// of the Doubly Linked List
void push(struct Node** head_ref, int new_data)
{
    // allocate node
    struct Node* new_node =
        (struct Node*)malloc(sizeof(struct Node));

    // put in the data
    new_node->data = new_data;

    // since we are adding at the begining,
    // prev is always NULL
    new_node->prev = NULL;

    // link the old list off the new node
    new_node->next = (*head_ref);

    // change prev of head node to new node
    if ((*head_ref) != NULL)
        (*head_ref)->prev = new_node;

    // move the head to point to the new node
    (*head_ref) = new_node;
}

// Function to print nodes in a given doubly
// linked list
void printList(struct Node* head)
{
    // if list is empty
    if (head == NULL)
```

```
cout << "Doubly Linked list empty";

while (head != NULL) {
    cout << head->data << " ";
    head = head->next;
}
}

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

    // Create the doubly linked list:
    // 8<->4<->4<->6<->4<->8<->4<->10<->12<->12
    push(&head, 12);
    push(&head, 12);
    push(&head, 10);
    push(&head, 4);
    push(&head, 8);
    push(&head, 4);
    push(&head, 6);
    push(&head, 4);
    push(&head, 4);
    push(&head, 8);

    cout << "Original Doubly linked list:n";
    printList(head);

    /* remove duplicate nodes */
    removeDuplicates(&head);

    cout << "\nDoubly linked list after "
        "removing duplicates:n";
    printList(head);

    return 0;
}
```

Output:

```
Original Doubly linked list:
8 4 4 6 4 8 4 10 12 12
Doubly linked list after removing duplicates:
8 4 6 10 12
```

Time Complexity:  $O(n^2)$

Auxiliary Space: O(1)

**Method 2 (Sorting):** Following are the steps:

1. Sort the elements of the doubly linked list using Merge Sort. Refer [this post](#).
2. Remove duplicates in linear time using the [algorithm to remove duplicates from a sorted doubly linked list](#).

Time Complexity: O(nLogn)

Auxiliary Space: O(1)

Note that this method doesn't preserve the original order of elements.

**Method 3 Efficient Approach(Hashing):**

We traverse the doubly linked list from head to end. For every newly encountered element, we check whether it is in the hash table: if yes, we remove it; otherwise we put it in the hash table. Hash table is implemented using [unordered\\_set in C++](#).

```
// C++ implementation to remove duplicates from an
// unsorted doubly linked list
#include <bits/stdc++.h>

using namespace std;

// a node of the doubly linked list
struct Node {
    int data;
    struct Node* next;
    struct Node* prev;
};

// Function to delete a node in a Doubly Linked List.
// head_ref --> pointer to head node pointer.
// del --> pointer to node to be deleted.
void deleteNode(struct Node** head_ref, struct Node* del)
{
    // base case
    if (*head_ref == NULL || del == NULL)
        return;

    // If node to be deleted is head node
    if (*head_ref == del)
        *head_ref = del->next;

    // Change next only if node to be deleted
    // is NOT the last node
    if (del->next != NULL)
        del->next->prev = del->prev;
```

```
// Change prev only if node to be deleted
// is NOT the first node
if (del->prev != NULL)
    del->prev->next = del->next;

// Finally, free the memory occupied by del
free(del);
}

// function to remove duplicates from
// an unsorted doubly linked list
void removeDuplicates(struct Node** head_ref)
{
    // if doubly linked list is empty
    if ((*head_ref) == NULL)
        return;

    // unordered_set 'us' implemented as hash table
    unordered_set<int> us;

    struct Node* current = *head_ref, *next;

    // traverse up to the end of the list
    while (current != NULL) {

        // if current data is seen before
        if (us.find(current->data) != us.end()) {

            // store pointer to the node next to
            // 'current' node
            next = current->next;

            // delete the node pointed to by 'current'
            deleteNode(head_ref, current);

            // update 'current'
            current = next;
        }
        else {

            // insert the current data in 'us'
            us.insert(current->data);

            // move to the next node
            current = current->next;
        }
    }
}
```

```
}  
  
// Function to insert a node at the beginning  
// of the Doubly Linked List  
void push(struct Node** head_ref, int new_data)  
{  
    // allocate node  
    struct Node* new_node =  
        (struct Node*)malloc(sizeof(struct Node));  
  
    // put in the data  
    new_node->data = new_data;  
  
    // since we are adding at the beginning,  
    // prev is always NULL  
    new_node->prev = NULL;  
  
    // link the old list off the new node  
    new_node->next = (*head_ref);  
  
    // change prev of head node to new node  
    if ((*head_ref) != NULL)  
        (*head_ref)->prev = new_node;  
  
    // move the head to point to the new node  
    (*head_ref) = new_node;  
}  
  
// Function to print nodes in a given doubly  
// linked list  
void printList(struct Node* head)  
{  
    // if list is empty  
    if (head == NULL)  
        cout << "Doubly Linked list empty";  
  
    while (head != NULL) {  
        cout << head->data << " ";  
        head = head->next;  
    }  
}  
  
// Driver program to test above  
int main()  
{  
    struct Node* head = NULL;  
  
    // Create the doubly linked list:
```

```
// 8<->4<->4<->6<->4<->8<->4<->10<->12<->12
push(&head, 12);
push(&head, 12);
push(&head, 10);
push(&head, 4);
push(&head, 8);
push(&head, 4);
push(&head, 6);
push(&head, 4);
push(&head, 4);
push(&head, 8);

cout << "Original Doubly linked list:n";
printList(head);

/* remove duplicate nodes */
removeDuplicates(&head);

cout << "\nDoubly linked list after "
      "removing duplicates:n";
printList(head);

return 0;
}
```

Output:

```
Original Doubly linked list:
8 4 4 6 4 8 4 10 12 12
Doubly linked list after removing duplicates:
8 4 6 10 12
```

Time Complexity: O(n)  
Auxiliary Space: O(n)

## Source

<https://www.geeksforgeeks.org/remove-duplicates-unsorted-doubly-linked-list/>

## Chapter 202

# Remove duplicates from an unsorted linked list

Remove duplicates from an unsorted linked list - GeeksforGeeks

Write a removeDuplicates() function which takes a list and deletes any duplicate nodes from the list. The list is not sorted.

For example if the linked list is 12->11->12->21->41->43->21 then removeDuplicates() should convert the list to 12->11->21->41->43.

### METHOD 1 (Using two loops)

This is the simple way where two loops are used. Outer loop is used to pick the elements one by one and inner loop compares the picked element with rest of the elements.

Thanks to Gaurav Saxena for his help in writing this code.

C++

```
/* Program to remove duplicates in an unsorted
linked list */
#include<bits/stdc++.h>
using namespace std;

/* A linked list node */
struct Node
{
    int data;
    struct Node *next;
};

// Utility function to create a new Node
struct Node *newNode(int data)
{
    Node *temp = new Node;
```

```
temp->data = data;
temp->next = NULL;
return temp;
}

/* Function to remove duplicates from a
   unsorted linked list */
void removeDuplicates(struct Node *start)
{
    struct Node *ptr1, *ptr2, *dup;
    ptr1 = start;

    /* Pick elements one by one */
    while (ptr1 != NULL && ptr1->next != NULL)
    {
        ptr2 = ptr1;

        /* Compare the picked element with rest
           of the elements */
        while (ptr2->next != NULL)
        {
            /* If duplicate then delete it */
            if (ptr1->data == ptr2->next->data)
            {
                /* sequence of steps is important here */
                dup = ptr2->next;
                ptr2->next = ptr2->next->next;
                delete(dup);
            }
            else /* This is tricky */
                ptr2 = ptr2->next;
        }
        ptr1 = ptr1->next;
    }

    /* Function to print nodes in a given linked list */
    void printList(struct Node *node)
    {
        while (node != NULL)
        {
            printf("%d ", node->data);
            node = node->next;
        }
    }

    /* Druver program to test above function */
    int main()
```

```
{  
    /* The constructed linked list is:  
       10->12->11->11->12->11->10*/  
    struct Node *start = newNode(10);  
    start->next = newNode(12);  
    start->next->next = newNode(11);  
    start->next->next->next = newNode(11);  
    start->next->next->next->next = newNode(12);  
    start->next->next->next->next->next =  
        newNode(11);  
    start->next->next->next->next->next->next =  
        newNode(10);  
  
    printf("Linked list before removing duplicates ");  
    printList(start);  
  
    removeDuplicates(start);  
  
    printf("\nLinked list after removing duplicates ");  
    printList(start);  
  
    return 0;  
}
```

**Java**

```
// Java program to remove duplicates from unsorted  
// linked list  
  
class LinkedList {  
  
    static Node head;  
  
    static class Node {  
  
        int data;  
        Node next;  
  
        Node(int d) {  
            data = d;  
            next = null;  
        }  
    }  
  
    /* Function to remove duplicates from an  
       unsorted linked list */  
    void remove_duplicates() {  
        Node ptr1 = null, ptr2 = null, dup = null;
```

```
ptr1 = head;

/* Pick elements one by one */
while (ptr1 != null && ptr1.next != null) {
    ptr2 = ptr1;

    /* Compare the picked element with rest
       of the elements */
    while (ptr2.next != null) {

        /* If duplicate then delete it */
        if (ptr1.data == ptr2.next.data) {

            /* sequence of steps is important here */
            dup = ptr2.next;
            ptr2.next = ptr2.next.next;
            System.gc();
        } else /* This is tricky */ {
            ptr2 = ptr2.next;
        }
    }
    ptr1 = ptr1.next;
}

void printList(Node node) {
    while (node != null) {
        System.out.print(node.data + " ");
        node = node.next;
    }
}

public static void main(String[] args) {
    LinkedList list = new LinkedList();
    list.head = new Node(10);
    list.head.next = new Node(12);
    list.head.next.next = new Node(11);
    list.head.next.next.next = new Node(11);
    list.head.next.next.next.next = new Node(12);
    list.head.next.next.next.next.next = new Node(11);
    list.head.next.next.next.next.next.next = new Node(10);

    System.out.println("Linked List before removing duplicates : \n");
    list.printList(head);

    list.remove_duplicates();
    System.out.println("");
    System.out.println("Linked List after removing duplicates : \n");
}
```

```
    list.printList(head);
}
// This code has been contributed by Mayank Jaiswal
```

Output :

```
Linked list before removing duplicates:
10 12 11 11 12 11 10
Linked list after removing duplicates:
10 12 11
```

Time Complexity:  $O(n^2)$

### METHOD 2 (Use Sorting)

In general, Merge Sort is the best-suited sorting algorithm for sorting linked lists efficiently.

- 1) Sort the elements using Merge Sort. We will soon be writing a post about sorting a linked list.  $O(n\log n)$
- 2) Remove duplicates in linear time using the [algorithm for removing duplicates in sorted Linked List.  \$O\(n\)\$](#)

Please note that this method doesn't preserve the original order of elements.

Time Complexity:  $O(n\log n)$

### METHOD 3 (Use Hashing)

We traverse the link list from head to end. For every newly encountered element, we check whether it is in the hash table: if yes, we remove it; otherwise we put it in the hash table.

C++

```
/* Program to remove duplicates in an unsorted
   linked list */
#include<bits/stdc++.h>
using namespace std;

/* A linked list node */
struct Node
{
    int data;
    struct Node *next;
};

// Utility function to create a new Node
struct Node *newNode(int data)
{
    Node *temp = new Node;
    temp->data = data;
    temp->next = NULL;
```

```
    return temp;
}

/* Function to remove duplicates from a
   unsorted linked list */
void removeDuplicates(struct Node *start)
{
    // Hash to store seen values
    unordered_set<int> seen;

    /* Pick elements one by one */
    struct Node *curr = start;
    struct Node *prev = NULL;
    while (curr != NULL)
    {
        // If current value is seen before
        if (seen.find(curr->data) != seen.end())
        {
            prev->next = curr->next;
            delete (curr);
        }
        else
        {
            seen.insert(curr->data);
            prev = curr;
        }
        curr = prev->next;
    }
}

/* Function to print nodes in a given linked list */
void printList(struct Node *node)
{
    while (node != NULL)
    {
        printf("%d ", node->data);
        node = node->next;
    }
}

/* Driver program to test above function */
int main()
{
    /* The constructed linked list is:
       10->12->11->11->12->11->10*/
    struct Node *start = newNode(10);
    start->next = newNode(12);
    start->next->next = newNode(11);
```

```
start->next->next->next = newNode(11);
start->next->next->next->next = newNode(12);
start->next->next->next->next->next =
                                newNode(11);
start->next->next->next->next->next->next =
                                newNode(10);

printf("Linked list before removing duplicates : \n");
printList(start);

removeDuplicates(start);

printf("\nLinked list after removing duplicates : \n");
printList(start);

return 0;
}
```

**Java**

```
// Java program to remove duplicates
// from unsorted linkedlist

import java.util.HashSet;

public class removeDuplicates
{
    static class node
    {
        int val;
        node next;

        public node(int val)
        {
            this.val = val;
        }
    }

    /* Function to remove duplicates from a
       unsorted linked list */
    static void removeDuplicate(node head)
    {
        // Hash to store seen values
        HashSet<Integer> hs = new HashSet<>();

        /* Pick elements one by one */
        node current = head;
        node prev = null;
```

```
while (current != null)
{
    int curval = current.val;

    // If current value is seen before
    if (hs.contains(curval)) {
        prev.next = current.next;
    } else {
        hs.add(curval);
        prev = current;
    }
    current = current.next;
}

/*
 * Function to print nodes in a given linked list */
static void printList(node head)
{
    while (head != null)
    {
        System.out.print(head.val + " ");
        head = head.next;
    }
}

public static void main(String[] args)
{
    /* The constructed linked list is:
       10->12->11->11->12->11->10*/
    node start = new node(10);
    start.next = new node(12);
    start.next.next = new node(11);
    start.next.next.next = new node(11);
    start.next.next.next.next = new node(12);
    start.next.next.next.next.next = new node(11);
    start.next.next.next.next.next.next = new node(10);

    System.out.println("Linked list before removing duplicates :");
    printList(start);

    removeDuplicate(start);

    System.out.println("\nLinked list after removing duplicates :");
    printList(start);
}
}
```

```
// This code is contributed by Rishabh Mahrsee
```

Output :

```
Linked list before removing duplicates:
```

```
10 12 11 11 12 11 10
```

```
Linked list after removing duplicates:
```

```
10 12 11
```

Thanks to bearwang for suggesting this method.

Time Complexity:  $O(n)$  on average (assuming that hash table access time is  $O(1)$  on average).

Please write comments if you find any of the above explanations/algorithms incorrect, or a better ways to solve the same problem.

## Source

<https://www.geeksforgeeks.org/remove-duplicates-from-an-unsorted-linked-list/>

## Chapter 203

# Remove every k-th node of the linked list

Remove every k-th node of the linked list - GeeksforGeeks

Given a singly linked list, Your task is to remove every K-th node of the linked list. Assume that K is always less than or equal to length of Linked List.

**Examples :**

```
Input : 1->2->3->4->5->6->7->8
        k = 3
Output : 1->2->4->5->7->8
As 3 is the k-th node after its deletion list
would be 1->2->4->5->6->7->8
And now 4 is the starting node then from it, 6
would be the k-th node. So no other kth node
could be there. So, final list is:
1->2->4->5->7->8.
```

```
Input: 1->2->3->4->5->6
      k = 1
Output: Empty list
All nodes need to be deleted
```

The idea is traverse the list from beginning and keep track of nodes visited after last deletion. Whenever count becomes k, delete current node and reset count as 0.

- (1) Traverse list and do following
  - (a) Count node before deletion.
  - (b) If (count == k) that means current

```
node is to be deleted.  
(i) Delete current node i.e. do  
  
    // assign address of next node of  
    // current node to the previous node  
    // of the current node.  
    prev->next = ptr->next i.e.  
  
    (ii) Reset count as 0, i.e., do count = 0.  
(c) Update prev node if count != 0 and if  
    count is 0 that means that node is a  
    starting point.  
(d) Update ptr and continue until all  
    k-th node gets deleted.
```

Below is C++ implementation.

```
// C++ program to delete every k-th Node of  
// a singly linked list.  
#include<iostream>  
using namespace std;  
  
/* Linked list Node */  
struct Node  
{  
    int data;  
    struct Node* next;  
};  
  
// To remove complete list (Needed for  
// case when k is 1)  
void freeList(Node *node)  
{  
    while (node != NULL)  
    {  
        Node *next = node->next;  
        delete (node);  
        node = next;  
    }  
}  
  
// Deletes every k-th node and returns head  
// of modified list.  
Node *deleteKthNode(struct Node *head, int k)  
{  
    // If linked list is empty  
    if (head == NULL)  
        return NULL;
```

```
if (k == 1)
{
    freeList(head);
    return NULL;
}

// Initialize ptr and prev before starting
// traversal.
struct Node *ptr = head, *prev = NULL;

// Traverse list and delete every k-th node
int count = 0;
while (ptr != NULL)
{
    // increment Node count
    count++;

    // check if count is equal to k
    // if yes, then delete current Node
    if (k == count)
    {
        // put the next of current Node in
        // the next of previous Node
        delete(prev->next);
        prev->next = ptr->next;

        // set count = 0 to reach further
        // k-th Node
        count = 0;
    }

    // update prev if count is not 0
    if (count != 0)
        prev = ptr;

    ptr = prev->next;
}

return head;
}

/* Function to print linked list */
void displayList(struct Node *head)
{
    struct Node *temp = head;
    while (temp != NULL)
    {
```

```
        cout<<temp->data<<" ";
        temp = temp->next;
    }
}

// Utility function to create a new node.
struct Node *newNode(int x)
{
    Node *temp = new Node;
    temp->data = x;
    temp->next = NULL;
    return temp;
}

/* Driver program to test count function*/
int main()
{
    /* Start with the empty list */
    struct Node* head = newNode(1);
    head->next = newNode(2);
    head->next->next = newNode(3);
    head->next->next->next = newNode(4);
    head->next->next->next->next = newNode(5);
    head->next->next->next->next->next = newNode(6);
    head->next->next->next->next->next->next =
                                newNode(7);
    head->next->next->next->next->next->next->next =
                                newNode(8);

    int k = 3;
    head = deleteKthNode(head, k);

    displayList(head);

    return 0;
}
```

Output:

1 2 4 5 7 8

Time Complexity : O(n)

## Source

<https://www.geeksforgeeks.org/remove-every-k-th-node-linked-list/>

## Chapter 204

# Replace nodes with duplicates in linked list

Replace nodes with duplicates in linked list - GeeksforGeeks

Given a linked list that contains some random integers from 1 to n with many duplicates. Replace each duplicate element that is present in the linked list with the values n+1, n+2, n+3 and so on(starting from left to right in the given linked list).

Examples:

```
Input : 1 3 1 4 4 2 1
Output : 1 3 5 4 6 2 7
Replace 2nd occurrence of 1 with 5 i.e. (4+1)
        2nd occurrence of 4 with 6 i.e. (4+2)
        3rd occurrence of 1 with 7 i.e. (4+3)
```

```
Input : 1 1 1 4 3 2 2
Output : 1 5 6 4 3 2 7
```

### Approach :

1. Traverse the linked list, store the frequencies of every number present in linked list in a map and alongwith it find the maximum integer present in linked list i.e. **maxNum**.
2. Now, traverse the linked list again and if the frequency of any number is more than 1, set its value to -1 in map on its first occurrence.
- 3.The reason for this is that on next occurrence of this number we will find its value -1 which means this number has occurred before, change its data with maxNum + 1 and increment maxNum.

Below is the implementation of idea.

```
// C++ Program to replace duplicates
```

```
// in an unsorted linked list
#include <bits/stdc++.h>
using namespace std;

/* A linked list node */
struct Node {
    int data;
    struct Node* next;
};

// Utility function to create a new Node
struct Node* newNode(int data)
{
    Node* temp = new Node;
    temp->data = data;
    temp->next = NULL;
    return temp;
}

// Function to replace duplicates from a
// linked list
void replaceDuplicates(struct Node* head)
{
    // map to store the frequency of numbers
    unordered_map<int, int> mymap;

    Node* temp = head;

    // variable to store the maximum number
    // in linked list
    int maxNum = 0;

    // traverse the linked list to store
    // the frequency of every number and
    // find the maximum integer
    while (temp) {
        mymap[temp->data]++;
        if (maxNum < temp->data)
            maxNum = temp->data;
        temp = temp->next;
    }

    // Traverse again the linked list
    while (head) {

        // Mark the node with frequency more
        // than 1 so that we can change the
        // 2nd occurrence of that number
```

```
if (mymap[head->data] > 1)
    mymap[head->data] = -1;

    // -1 means number has occurred
    // before change its value
    else if (mymap[head->data] == -1)
        head->data = ++maxNum;

    head = head->next;
}
}

/* Function to print nodes in a given
linked list */
void printList(struct Node* node)
{
    while (node != NULL) {
        printf("%d ", node->data);
        node = node->next;
    }
    cout << endl;
}

/* Driver program to test above function */
int main()
{
    /* The constructed linked list is:
    1->3->1->4->4->2->1*/
    struct Node* head = newNode(1);
    head->next = newNode(3);
    head->next->next = newNode(1);
    head->next->next->next = newNode(4);
    head->next->next->next->
                                newNode(4);
    head->next->next->next->next->
                                newNode(2);
    head->next->next->next->next->
                                newNode(1);

    cout << "Linked list before replacing"
        << " duplicates\n";
    printList(head);

    replaceDuplicates(head);

    cout << "Linked list after replacing"
        << " duplicates\n";
    printList(head);
```

```
    return 0;  
}
```

Output:

```
Linked list before replacing duplicates  
1 3 1 4 4 2 1  
Linked list after replacing duplicates  
1 3 5 4 6 2 7
```

## Source

<https://www.geeksforgeeks.org/replace-nodes-duplicates-linked-list/>

## Chapter 205

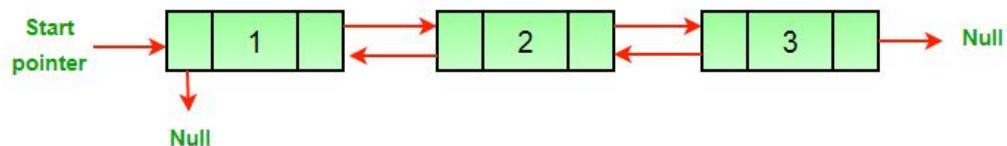
# Reverse a Doubly Linked List

Reverse a Doubly Linked List - GeeksforGeeks

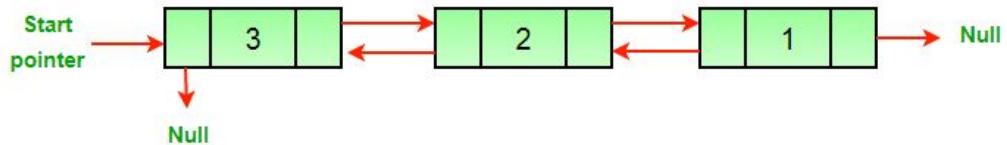
Write a C function to reverse a given Doubly Linked List

See below diagrams for example.

(a) Original Doubly Linked List



(b) Reversed Doubly Linked List



Here is a simple method for reversing a Doubly Linked List. All we need to do is swap prev and next pointers for all nodes, change prev of the head (or start) and change the head pointer in the end.

C

```
/* Program to reverse a doubly linked list */
#include <stdio.h>
#include <stdlib.h>

/* a node of the doubly linked list */
struct Node
{
    int data;
    struct Node *next;
    struct Node *prev;
};

/* Function to reverse a Doubly Linked List */
void reverse(struct Node **head_ref)
{
    struct Node *temp = NULL;
    struct Node *current = *head_ref;

    /* swap next and prev for all nodes of
       doubly linked list */
    while (current != NULL)
    {
        temp = current->prev;
        current->prev = current->next;
        current->next = temp;
        current = current->prev;
    }

    /* Before changing head, check for the cases like empty
       list and list with only one node */
    if(temp != NULL )
        *head_ref = temp->prev;
}

/* UTILITY FUNCTIONS */
/* Function to insert a node at the beginning of the Doubly Linked List */
void push(struct Node** head_ref, int new_data)
{
    /* allocate node */
    struct Node* new_node =
        (struct Node*) malloc(sizeof(struct Node));

    /* put in the data */
    new_node->data = new_data;

    /* since we are adding at the begining,
```

```
    prev is always NULL */
new_node->prev = NULL;

/* link the old list off the new node */
new_node->next = (*head_ref);

/* change prev of head node to new node */
if((*head_ref) != NULL)
    (*head_ref)->prev = new_node ;

/* move the head to point to the new node */
(*head_ref)      = new_node;
}

/* Function to print nodes in a given doubly linked list
   This function is same as printList() of singly linked lsit */
void printList(struct Node *node)
{
    while(node!=NULL)
    {
        printf("%d ", node->data);
        node = node->next;
    }
}

/* Drier program to test above functions*/
int main()
{
    /* Start with the empty list */
    struct Node* head = NULL;

    /* Let us create a sorted linked list to test the functions
       Created linked list will be 10->8->4->2 */
    push(&head, 2);
    push(&head, 4);
    push(&head, 8);
    push(&head, 10);

    printf("\n Original Linked list ");
    printList(head);

    /* Reverse doubly linked list */
    reverse(&head);

    printf("\n Reversed Linked list ");
    printList(head);

    getchar();
}
```

}

**Java**

```
// Java program to reverse a doubly linked list

class LinkedList {

    static Node head;

    static class Node {

        int data;
        Node next, prev;

        Node(int d) {
            data = d;
            next = prev = null;
        }
    }

    /* Function to reverse a Doubly Linked List */
    void reverse() {
        Node temp = null;
        Node current = head;

        /* swap next and prev for all nodes of
           doubly linked list */
        while (current != null) {
            temp = current.prev;
            current.prev = current.next;
            current.next = temp;
            current = current.prev;
        }

        /* Before changing head, check for the cases like empty
           list and list with only one node */
        if (temp != null)
            head = temp.prev;
    }
}

/* UTILITY FUNCTIONS */
/* Function to insert a node at the beginning of the Doubly Linked List */
void push(int new_data) {
    /* allocate node */
    Node new_node = new Node(new_data);
```

```
/* since we are adding at the begining,
prev is always NULL */
new_node.prev = null;

/* link the old list off the new node */
new_node.next = head;

/* change prev of head node to new node */
if (head != null) {
    head.prev = new_node;
}

/* move the head to point to the new node */
head = new_node;
}

/* Function to print nodes in a given doubly linked list
This function is same as printList() of singly linked lsit */
void printList(Node node) {
    while (node != null) {
        System.out.print(node.data + " ");
        node = node.next;
    }
}

public static void main(String[] args) {
    LinkedList list = new LinkedList();

    /* Let us create a sorted linked list to test the functions
    Created linked list will be 10->8->4->2 */
    list.push(2);
    list.push(4);
    list.push(8);
    list.push(10);

    System.out.println("Original linked list ");
    list.printList(head);

    list.reverse();
    System.out.println("");
    System.out.println("The reversed Linked List is ");
    list.printList(head);
}

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

Python

```
# Program to reverse a doubly linked list

# A node of the doubly linked list
class Node:

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

class DoublyLinkedList:
    # Constructor for empty Doubly Linked List
    def __init__(self):
        self.head = None

    # Function reverse a Doubly Linked List
    def reverse(self):
        temp = None
        current = self.head

        # Swap next and prev for all nodes of
        # doubly linked list
        while current is not None:
            temp = current.prev
            current.prev = current.next
            current.next = temp
            current = current.prev

        # Before changing head, check for the cases like
        # empty list and list with only one node
        if temp is not None:
            self.head = temp.prev

    # Given a reference to the head of a list and an
    # integer, inserts a new node on the front of list
    def push(self, new_data):

        # 1. Allocates node
        # 2. Put the data in it
        new_node = Node(new_data)

        # 3. Make next of new node as head and
        # previous as None (already None)
        new_node.next = self.head

        # 4. change prev of head node to new_node
        if self.head is not None:
            self.head.prev = new_node
```

```
        self.head.prev = new_node

    # 5. move the head to point to the new node
    self.head = new_node

def printList(self, node):
    while(node is not None):
        print node.data,
        node = node.next

# Driver program to test the above functions
dll = DoublyLinkedList()
dll.push(2);
dll.push(4);
dll.push(8);
dll.push(10);

print "\nOriginal Linked List"
dll.printList(dll.head)

# Reverse doubly linked list
dll.reverse()

print "\nReversed Linked List"
dll.printList(dll.head)

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

Time Complexity: O(n)

We can also swap data instead of pointers to reverse the Doubly Linked List. [Method used for reversing array](#) can be used to swap data. Swapping data can be costly compared to pointers if size of data item(s) is more.

**Improved By :** [gurunath10](#)

## Source

<https://www.geeksforgeeks.org/reverse-a-doubly-linked-list/>

## Chapter 206

# Reverse a Doubly Linked List | Set 4 (Swapping Data)

Reverse a Doubly Linked List | Set 4 (Swapping Data) - GeeksforGeeks

Given a Doubly Linked List, we are asked to reverse the list in-place without using any extra space.

**Examples:**

Input : 1 <--> 2 <--> 5 <--> 6 <--> 7  
Output : 7 <--> 6 <--> 5 <--> 2 <--> 1

Input : 11 <--> 22 <--> 33 <--> 22 <--> 1  
Output : 1 <--> 22 <--> 33 <--> 22 <--> 11

We have discussed at three methods to reverse a doubly linked list: [Reverse a doubly linked list](#), [Reverse a Doubly Linked List \(Set 2\)](#) and [Reverse a Doubly linked list using recursion](#).

The first two methods work in  $O(n)$  time and require no extra space. The first method works by swapping the next and previous pointers of each node. The second method takes each node from the list and adds it to the beginning of the list.

There is another approach that is a bit more intuitive, but also a bit more costly. This method is similar to reverse an array. To reverse an array, we put two pointers-one at the beginning and another at the end of the list. We then swap the data of the two pointers and advance both pointers toward each other. We stop either when the two pointers meet or when they cross each other. We perform exactly  $n/2$  swaps, and the time complexity is also  $O(N)$ .

A doubly linked list has both a previous and a next pointer, which means we can traverse in both forward and backward directions in the list. So if we put a pointer( say left pointer), at the beginning of the list and another right pointer at the end of the list, we can move these pointers toward each other by advancing left pointer and receding the right pointer.

### Algorithm

```
Step 1: Set LEFT to head of list
Step 2: Traverse the list and set RIGHT to end of the list
Step 3: Repeat following steps while LEFT != RIGHT and
        LEFT->PREV != RIGHT
Step 4: Swap LEFT->DATA and RIGHT->DATA
Step 5: Advance LEFT pointer by one, LEFT = LEFT->NEXT
Step 6: Recede RIGHT pointer by one, i.e RIGHT = RIGHT->PREV
        [END OF LOOP]
Step 7: End
```

### A Note on the comparative efficiency of the three methods

A few things must be mentioned. This method is simple to implement, but it is also more costly when compared to say the pointer-exchange method. This is because we swap data and not pointers. Swapping data can be more costly if the nodes are large complex data types with multiple data members. In contrast, the pointer to the node will always be simpler data type and either of 4 or 8 bytes.

Below is the CPP implementation of the algorithm.

```
// Cpp Program to Reverse a List using Data Swapping

#include <iostream>
using namespace std;

struct Node {
    int data;
    struct Node *prev, *next;
};

Node* newNode(int val)
{
    Node* temp = new Node;
    temp->data = val;
    temp->prev = temp->next = nullptr;
    return temp;
}

void printList(Node* head)
{
    while (head->next != nullptr) {
        cout << head->data << " <-> ";
        head = head->next;
    }
    cout << head->data << endl;
```

```
}

// Insert a new node at the head of the list
void insert(Node** head, int val)
{
    Node* temp = newNode(val);
    temp->next = *head;
    (*head)->prev = temp;
    (*head) = temp;
}

// Function to reverse the list
void reverseList(Node** head)
{
    Node* left = *head, * right = *head;

    // Traverse the list and set right pointer to
    // end of list
    while (right->next != nullptr)
        right = right->next;

    // Swap data of left and right pointer and move
    // them towards each other until they meet or
    // cross each other
    while (left != right && left->prev != right) {

        // Swap data of left and right pointer
        swap(left->data, right->data);

        // Advance left pointer
        left = left->next;

        // Advance right pointer
        right = right->prev;
    }
}

// Driver code
int main()
{
    Node* head = newNode(5);
    insert(&head, 4);
    insert(&head, 3);
    insert(&head, 2);
    insert(&head, 1);

    printList(head);
    cout << "List After Reversing" << endl;
```

```
reverseList(&head);
printList(head);

return 0;
}
```

**Output:**

```
1 <--> 2 <--> 3 <--> 4 <--> 5
List After Reversing
5 <--> 4 <--> 3 <--> 2 <--> 1
```

**Source**

<https://www.geeksforgeeks.org/reverse-doubly-linked-list-set-4-swapping-data/>

## Chapter 207

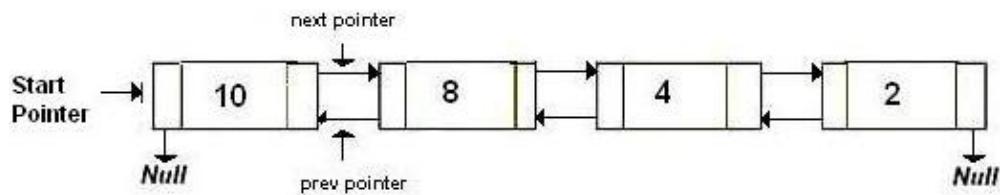
# Reverse a Doubly Linked List | Set-2

Reverse a Doubly Linked List | Set-2 - GeeksforGeeks

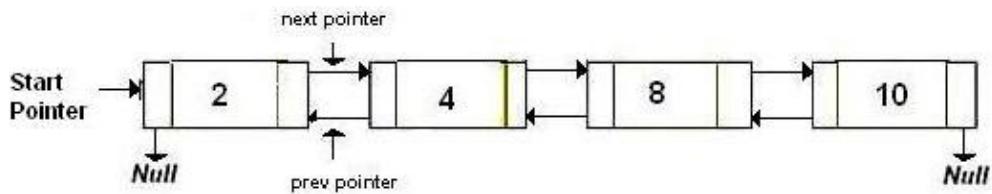
Write a program to reverse the given Doubly Linked List.

See below diagrams for example.

(a) Original Doubly Linked List



(b) Reversed Doubly Linked List



**Approach:** In the [previous post](#), doubly linked list is being reversed by swapping prev and next pointers for all nodes, changing prev of the head (or start) and then changing the head

pointer in the end. In this post, we create a push function that adds the given node at the beginning of the given list. We traverse the original list and one by one pass the current node pointer to the push function. This process will reverse the list. Finally return the new head of this reversed list.

```
// C++ implementation to reverse
// a doubly linked list
#include <bits/stdc++.h>

using namespace std;

// a node of the doubly linked list
struct Node {
    int data;
    Node *next, *prev;
};

// function to get a new node
Node* getNode(int data)
{
    // allocate space
    Node* new_node = (Node*)malloc(sizeof(Node));

    // put in the data
    new_node->data = data;
    new_node->next = new_node->prev = NULL;
    return new_node;
}

// function to insert a node at the begining
// of the Doubly Linked List
void push(Node** head_ref, Node* new_node)
{
    // since we are adding at the begining,
    // prev is always NULL
    new_node->prev = NULL;

    // link the old list off the new node
    new_node->next = (*head_ref);

    // change prev of head node to new node
    if ((*head_ref) != NULL)
        (*head_ref)->prev = new_node;

    // move the head to point to the new node
    (*head_ref) = new_node;
}
```

```
// function to reverse a doubly linked list
void reverseList(Node** head_ref)
{
    // if list is empty or it contains
    // a single node only
    if (!(*head_ref) || !((*head_ref)->next))
        return;

    Node* new_head = NULL;
    Node *curr = *head_ref, *next;

    while (curr != NULL) {

        // get pointer to next node
        next = curr->next;

        // push 'curr' node at the beginning of the
        // list with starting with 'new_head'
        push(&new_head, curr);

        // update 'curr'
        curr = next;
    }

    // update 'head_ref'
    *head_ref = new_head;
}

// Function to print nodes in a
// given doubly linked list
void printList(Node* head)
{
    while (head != NULL) {
        cout << head->data << " ";
        head = head->next;
    }
}

// Driver program to test above
int main()
{
    // Start with the empty list
    Node* head = NULL;

    // Create doubly linked: 10<->8<->4<->2 */
    push(&head, getNode(2));
    push(&head, getNode(4));
    push(&head, getNode(8));
```

```
push(&head, getNode(10));  
  
cout << "Original list: ";  
printList(head);  
  
// Reverse doubly linked list  
reverseList(&head);  
  
cout << "\nReversed list: ";  
printList(head);  
  
return 0;  
}
```

Output:

```
Original list: 10 8 4 2  
Reversed list: 2 4 8 10
```

Time Complexity: O(n).

## Source

<https://www.geeksforgeeks.org/reverse-doubly-linked-list-set-2/>

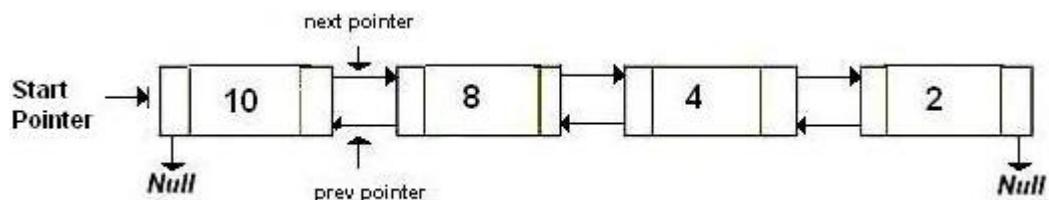
## Chapter 208

# Reverse a Doubly linked list using recursion

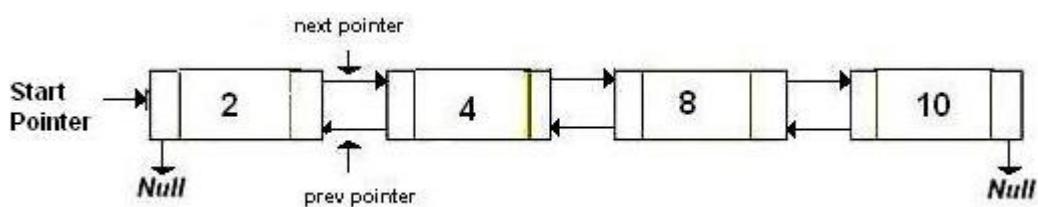
Reverse a Doubly linked list using recursion - GeeksforGeeks

Given a doubly linked list. Reverse it using recursion.

Original Doubly linked list



Reversed Doubly linked list



We have discussed

[Iterative solution to reverse a Doubly Linked List](#)

### Algorithm

- 1) If list is empty, return
- 2) Reverse head by swapping head->prev and head->next
- 3) If prev = NULL it means that list is fully reversed. Else reverse(head->prev)

```
// C++ implementation to reverse a doubly
// linked list using recursion
#include <bits/stdc++.h>
using namespace std;

// a node of the doubly linked list
struct Node {
    int data;
    Node *next, *prev;
};

// function to get a new node
Node* getNode(int data)
{
    // allocate space
    Node* new_node = new Node;
    new_node->data = data;
    new_node->next = new_node->prev = NULL;
    return new_node;
}

// function to insert a node at the beginning
// of the Doubly Linked List
void push(Node** head_ref, Node* new_node)
{
    // since we are adding at the beginning,
    // prev is always NULL
    new_node->prev = NULL;

    // link the old list off the new node
    new_node->next = (*head_ref);

    // change prev of head node to new node
    if ((*head_ref) != NULL)
        (*head_ref)->prev = new_node;

    // move the head to point to the new node
    (*head_ref) = new_node;
}

// function to reverse a doubly linked list
Node* Reverse(Node* node)
{
    // If empty list, return
    if (!node)
        return NULL;

    // Otherwise, swap the next and prev
```

```
Node* temp = node->next;
node->next = node->prev;
node->prev = temp;

// If the prev is now NULL, the list
// has been fully reversed
if (!node->prev)
    return node;

// Otherwise, keep going
return Reverse(node->prev);
}

// Function to print nodes in a given doubly
// linked list
void printList(Node* head)
{
    while (head != NULL) {
        cout << head->data << " ";
        head = head->next;
    }
}

// Driver program to test above
int main()
{
    // Start with the empty list
    Node* head = NULL;

    // Create doubly linked: 10<->8<->4<->2 */
    push(&head, getNode(2));
    push(&head, getNode(4));
    push(&head, getNode(8));
    push(&head, getNode(10));
    cout << "Original list: ";
    printList(head);

    // Reverse doubly linked list
    head = Reverse(head);
    cout << "\nReversed list: ";
    printList(head);
    return 0;
}
```

Output:

```
Original list: 10 8 4 2
Reversed list: 2 4 8 10
```

**Source**

<https://www.geeksforgeeks.org/reverse-doubly-linked-list-using-recursion/>

## Chapter 209

# Reverse a Linked List according to its Size

Reverse a Linked List according to its Size - GeeksforGeeks

Given a linked list with n nodes, reverse it in the following way :

1. If n is even, reverse it in group of  $n/2$  nodes.
2. If n is odd, keep the middle node as it is, reverse first  $n/2$  elements and reverse last  $n/2$  elements.

Examples:

Input : 1 2 3 4 5 6 (n is even)

Output : 3 2 1 6 5 4

Input : 1 2 3 4 5 6 7 (n is odd)

Output : 3 2 1 4 7 6 5

**Approach:** The idea is similar to [Reversing a linked list in groups of size k](#) where k is  $n/2$ . Just need to check for mid node.

- If n is even, divide the linked list into two parts i.e. first  $n/2$  elements and last  $n/2$  elements and reverse both the parts.
- If n is odd, divide the linked list into three parts i.e. first  $n/2$  elements,  $(n/2 + 1)$  th element and last  $n/2$  elements and reverse both the parts except  $(n/2 + 1)$  th element

```
// C++ program to reverse given
// linked list according to its size
#include <bits/stdc++.h>
using namespace std;
```

```
struct Node {  
    int data;  
    Node* next;  
};  
  
// Function to create a new Node  
Node* newNode(int data)  
{  
    Node *temp = new Node;  
    temp->data = data;  
    temp->next = NULL;  
    return temp;  
}  
  
// Prints a list.  
void printList(Node* head)  
{  
    Node *temp = head;  
    while (temp) {  
        cout << temp->data << " ";  
        temp = temp->next;  
    }  
    cout << endl;  
}  
  
/* Function to push a Node */  
void push(Node** head_ref, int new_data)  
{  
    Node* new_Node = new Node;  
    new_Node->data = new_data;  
    new_Node->next = (*head_ref);  
    (*head_ref) = new_Node;  
}  
  
// Returns size of list.  
int getSize(Node* head)  
{  
    Node* curr = head;  
    int count = 0;  
    while (curr) {  
        curr = curr->next;  
        count++;  
    }  
    return count;  
}  
  
// Function to reverse the linked
```

```
// list according to its size
Node* reverseSizeBy2Util(Node* head, int k,
                         bool skipMiddle)
{
    if (!head)
        return NULL;

    int count = 0;
    Node* curr = head;
    Node* prev = NULL;
    Node* next;

    // Reverse current block of list.
    while (curr && count < k) {
        next = curr->next;
        curr->next = prev;
        prev = curr;
        curr = next;
        count++;
    }

    // If size is even, reverse next block too.
    if (!skipMiddle)
        head->next = reverseSizeBy2Util(next, k, false);

    else {

        // if size is odd, skip next element
        // and reverse the block after that.
        head->next = next;
        if (next)
            next->next = reverseSizeBy2Util(next->next,
                                              k, true);
    }
    return prev;
}

Node* reverseBySizeBy2(Node* head)
{
    // Get the size of list.
    int n = getSize(head);

    // If the size is even, no need
    // to skip middle Node.
    if (n % 2 == 0)
        return reverseSizeBy2Util(head, n/2, false);

    // If size is odd, middle Node has
```

```
// to be skipped.  
else  
    return reverseSizeBy2Util(head, n/2, true);  
}  
  
// Drivers code  
int main()  
{  
    /* Start with the empty list */  
    Node* head = NULL;  
  
    /* Created Linked list is 1->2->3->4->5->6->7->8->9 */  
    push(&head, 9);  
    push(&head, 8);  
    push(&head, 7);  
    push(&head, 6);  
    push(&head, 5);  
    push(&head, 4);  
    push(&head, 3);  
    push(&head, 2);  
    push(&head, 1);  
  
    cout << "Original List : ";  
    printList(head);  
  
    cout << "Reversed List : ";  
    Node* reversedHead = reverseBySizeBy2(head);  
    printList(reversedHead);  
  
    return 0;  
}
```

**Output:**

```
Original List : 1 2 3 4 5 6 7 8 9  
Reversed List : 9 8 7 6 5 4 3 2 1
```

**Source**

<https://www.geeksforgeeks.org/reverse-a-linked-list-according-to-its-size/>

## Chapter 210

# Reverse a Linked List in groups of given size | Set 1

Reverse a Linked List in groups of given size | Set 1 - GeeksforGeeks

Given a linked list, write a function to reverse every k nodes (where k is an input to the function).

**Example:**

Inputs: 1->2->3->4->5->6->7->8->NULL and k = 3  
Output: 3->2->1->6->5->4->8->7->NULL.

Inputs: 1->2->3->4->5->6->7->8->NULL and k = 5  
Output: 5->4->3->2->1->8->7->6->NULL.

Algorithm: *reverse(head, k)*

- 1) Reverse the first sub-list of size k. While reversing keep track of the next node and previous node. Let the pointer to the next node be *next* and pointer to the previous node be *prev*. See [this post](#) for reversing a linked list.
- 2) *head->next = reverse(next, k)* /\* Recursively call for rest of the list and link the two sub-lists \*/
- 3) return *prev* /\* *prev* becomes the new head of the list (see the diagrams of iterative method of [this post](#)) \*/

**C/C++**

```
// C program to reverse a linked list in groups of given size
#include<stdio.h>
#include<stdlib.h>

/* Link list node */

```

```
struct Node
{
    int data;
    struct Node* next;
};

/* Reverses the linked list in groups of size k and returns the
   pointer to the new head node. */
struct Node *reverse (struct Node *head, int k)
{
    struct Node* current = head;
    struct Node* next = NULL;
    struct Node* prev = NULL;
    int count = 0;

    /*reverse first k nodes of the linked list */
    while (current != NULL && count < k)
    {
        next = current->next;
        current->next = prev;
        prev = current;
        current = next;
        count++;
    }

    /* next is now a pointer to (k+1)th node
       Recursively call for the list starting from current.
       And make rest of the list as next of first node */
    if (next != NULL)
        head->next = reverse(next, k);

    /* prev is new head of the input list */
    return prev;
}

/* UTILITY FUNCTIONS */
/* Function to push a node */
void push(struct Node** head_ref, int new_data)
{
    /* allocate node */
    struct Node* new_node =
        (struct Node*) malloc(sizeof(struct Node));

    /* put in the data */
    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;
}

/* Function to print linked list */
void printList(struct Node *node)
{
    while (node != NULL)
    {
        printf("%d  ", node->data);
        node = node->next;
    }
}

/* Driver program to test above function*/
int main(void)
{
    /* Start with the empty list */
    struct Node* head = NULL;

    /* Created Linked list is 1->2->3->4->5->6->7->8->9 */
    push(&head, 9);
    push(&head, 8);
    push(&head, 7);
    push(&head, 6);
    push(&head, 5);
    push(&head, 4);
    push(&head, 3);
    push(&head, 2);
    push(&head, 1);

    printf("\nGiven linked list \n");
    printList(head);
    head = reverse(head, 3);

    printf("\nReversed Linked list \n");
    printList(head);

    return(0);
}
```

**Java**

```
// Java program to reverse a linked list in groups of
// given size
class LinkedList
{
```

```
Node head; // head of list

/* Linked list Node*/
class Node
{
    int data;
    Node next;
    Node(int d) {data = d; next = null; }
}

Node reverse(Node head, int k)
{
    Node current = head;
    Node next = null;
    Node prev = null;

    int count = 0;

    /* Reverse first k nodes of linked list */
    while (count < k && current != null)
    {
        next = current.next;
        current.next = prev;
        prev = current;
        current = next;
        count++;
    }

    /* next is now a pointer to (k+1)th node
       Recursively call for the list starting from current.
       And make rest of the list as next of first node */
    if (next != null)
        head.next = reverse(next, k);

    // prev is now head of input list
    return prev;
}

/* Utility functions */

/* Inserts a new Node at front of the list. */
public void push(int new_data)
{
    /* 1 & 2: Allocate the Node &
       Put in the data*/
    Node new_node = new Node(new_data);
```

```
/* 3. Make next of new Node as head */
new_node.next = head;

/* 4. Move the head to point to new Node */
head = new_node;
}

/* Function to print linked list */
void printList()
{
    Node temp = head;
    while (temp != null)
    {
        System.out.print(temp.data+" ");
        temp = temp.next;
    }
    System.out.println();
}

/* Drier program to test above functions */
public static void main(String args[])
{
    LinkedList llist = new LinkedList();

    /* Constructed Linked List is 1->2->3->4->5->6->
       7->8->8->9->null */
    llist.push(9);
    llist.push(8);
    llist.push(7);
    llist.push(6);
    llist.push(5);
    llist.push(4);
    llist.push(3);
    llist.push(2);
    llist.push(1);

    System.out.println("Given Linked List");
    llist.printList();

    llist.head = llist.reverse(llist.head, 3);

    System.out.println("Reversed list");
    llist.printList();
}
}

/* This code is contributed by Rajat Mishra */
```

Python

```
# Python program to reverse a linked list in group of given size

# Node class
class Node:

    # Constructor to initialize the node object
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:

    # Function to initialize head
    def __init__(self):
        self.head = None

    def reverse(self, head, k):
        current = head
        next = None
        prev = None
        count = 0

        # Reverse first k nodes of the linked list
        while(current is not None and count < k):
            next = current.next
            current.next = prev
            prev = current
            current = next
            count += 1

        # next is now a pointer to (k+1)th node
        # recursively call for the list starting
        # from current . And make rest of the list as
        # next of first node
        if next is not None:
            head.next = self.reverse(next, k)

        # prev is new head of the input list
        return prev

    # Function to insert a new node at the beginning
    def push(self, new_data):
        new_node = Node(new_data)
        new_node.next = self.head
        self.head = new_node

    # Utility function to print the linked LinkedList
    def printList(self):
```

```
temp = self.head
while(temp):
    print temp.data,
    temp = temp.next

# Driver program
llist = LinkedList()
llist.push(9)
llist.push(8)
llist.push(7)
llist.push(6)
llist.push(5)
llist.push(4)
llist.push(3)
llist.push(2)
llist.push(1)

print "Given linked list"
llist.printList()
llist.head = llist.reverse(llist.head, 3)

print "\nReversed Linked list"
llist.printList()

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

Output:

```
Given Linked List
1 2 3 4 5 6 7 8 9
Reversed list
3 2 1 6 5 4 9 8 7
```

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

## Source

<https://www.geeksforgeeks.org/reverse-a-list-in-groups-of-given-size/>

## Chapter 211

# Reverse a Linked List in groups of given size | Set 2

Reverse a Linked List in groups of given size | Set 2 - GeeksforGeeks

Given a linked list, write a function to reverse every k nodes (where k is an input to the function).

Examples:

Inputs: 1->2->3->4->5->6->7->8->NULL and k = 3  
Output: 3->2->1->6->5->4->8->7->NULL.

Inputs: 1->2->3->4->5->6->7->8->NULL and k = 5  
Output: 5->4->3->2->1->8->7->6->NULL.

We have already discussed its solution in below post

[Reverse a Linked List in groups of given size | Set 1](#)

In this post, we have used a stack which will store the nodes of the given linked list. Firstly, push the k elements of the linked list in the stack. Now pop elements one by one and keep track of the previously popped node. Point the next pointer of prev node to top element of stack. Repeat this process, until NULL is reached.

This algorithm uses O(k) extra space.

```
// C++ program to reverse a linked list in groups
// of given size
#include <bits/stdc++.h>
using namespace std;

/* Link list node */
```

```
struct Node {  
    int data;  
    struct Node* next;  
};  
  
/* Reverses the linked list in groups of size k  
   and returns the pointer to the new head node. */  
struct Node* Reverse(struct Node* head, int k)  
{  
    // Create a stack of Node*  
    stack<Node*> mystack;  
    struct Node* current = head;  
    struct Node* prev = NULL;  
  
    while (current != NULL) {  
  
        // Terminate the loop whichever comes first  
        // either current == NULL or count >= k  
        int count = 0;  
        while (current != NULL && count < k) {  
            mystack.push(current);  
            current = current->next;  
            count++;  
        }  
  
        // Now pop the elements of stack one by one  
        while (mystack.size() > 0) {  
  
            // If final list has not been started yet.  
            if (prev == NULL) {  
                prev = mystack.top();  
                head = prev;  
                mystack.pop();  
            } else {  
                prev->next = mystack.top();  
                prev = prev->next;  
                mystack.pop();  
            }  
        }  
    }  
  
    // Next of last element will point to NULL.  
    prev->next = NULL;  
  
    return head;  
}  
  
/* UTILITY FUNCTIONS */
```

```
/* Function to push a node */
void push(struct Node** head_ref, int new_data)
{
    /* allocate node */
    struct Node* new_node =
        (struct Node*)malloc(sizeof(struct Node));

    /* put in the data */
    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;
}

/* Function to print linked list */
void printList(struct Node* node)
{
    while (node != NULL) {
        printf("%d ", node->data);
        node = node->next;
    }
}

/* Drier program to test above function*/
int main(void)
{
    /* Start with the empty list */
    struct Node* head = NULL;

    /* Created Linked list is 1->2->3->4->5->6->7->8->9 */
    push(&head, 9);
    push(&head, 8);
    push(&head, 7);
    push(&head, 6);
    push(&head, 5);
    push(&head, 4);
    push(&head, 3);
    push(&head, 2);
    push(&head, 1);

    printf("\nGiven linked list \n");
    printList(head);
    head = Reverse(head, 3);

    printf("\nReversed Linked list \n");
}
```

```
    printList(head);  
  
    return 0;  
}
```

Output:

```
Given Linked List  
1 2 3 4 5 6 7 8 9  
Reversed list  
3 2 1 6 5 4 9 8 7
```

### Source

<https://www.geeksforgeeks.org/reverse-linked-list-groups-given-size-set-2/>

## Chapter 212

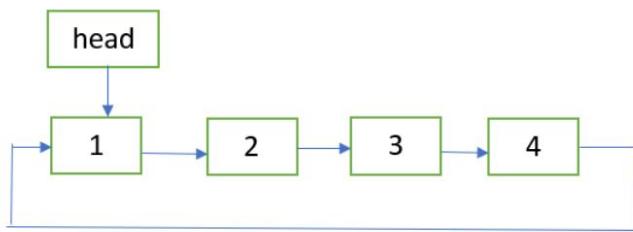
# Reverse a circular linked list

Reverse a circular linked list - GeeksforGeeks

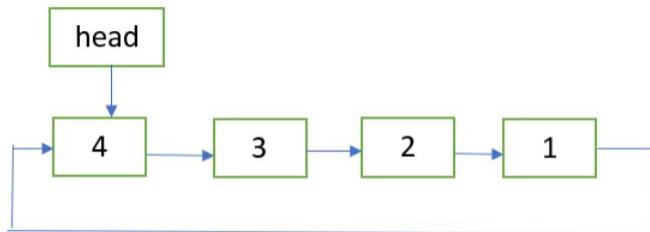
Given a circular linked list of size **n**. The problem is to reverse the given circular linked list by changing links between the nodes.

Examples:

**INPUT:**



**OUTPUT:**



**Approach:** The approach is same as followed in [reversing a singly linked list](#). Only here we have to make one more adjustment by linking the last node of the reversed list to the first node.

```
// C++ implementation to reverse  
// a circular linked list
```

```
#include <bits/stdc++.h>

using namespace std;

// Linked list node
struct Node {
    int data;
    Node* next;
};

// function to get a new node
Node* getNode(int data)
{
    // allocate memory for node
    Node* newNode = new Node;

    // put in the data
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}

// Function to reverse the circular linked list
void reverse(Node** head_ref)
{
    // if list is empty
    if (*head_ref == NULL)
        return;

    // reverse procedure same as reversing a
    // singly linked list
    Node* prev = NULL;
    Node* current = *head_ref;
    Node* next;
    do {
        next = current->next;
        current->next = prev;
        prev = current;
        current = next;
    } while (current != (*head_ref));

    // adjusting the links so as to make the
    // last node point to the first node
    (*head_ref)->next = prev;
    *head_ref = prev;
}

// Function to print circular linked list
```

```
void printList(Node* head)
{
    if (head == NULL)
        return;

    Node* temp = head;
    do {
        cout << temp->data << " ";
        temp = temp->next;
    } while (temp != head);
}

// Driver program to test above
int main()
{
    // Create a circular linked list
    // 1->2->3->4->1
    Node* head = getNode(1);
    head->next = getNode(2);
    head->next->next = getNode(3);
    head->next->next->next = getNode(4);
    head->next->next->next->next = head;

    cout << "Given circular linked list: ";
    printList(head);

    reverse(&head);

    cout << "\nReversed circular linked list: ";
    printList(head);

    return 0;
}
```

Output:

```
Given circular linked list: 1 2 3 4
Reversed circular linked list: 4 3 2 1
```

Time Complexity: O(n)

## Source

<https://www.geeksforgeeks.org/reverse-circular-linked-list/>

## Chapter 213

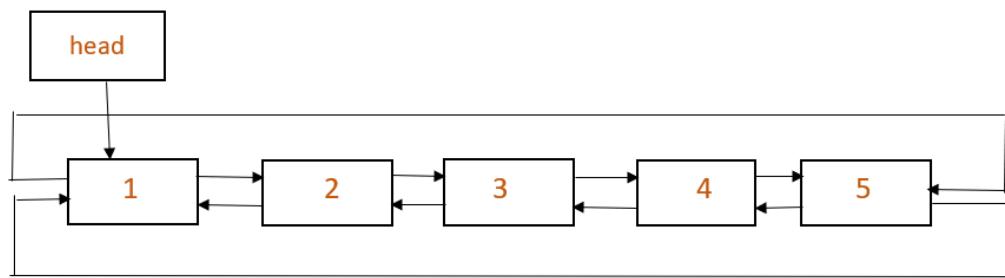
# Reverse a doubly circular linked list

Reverse a doubly circular linked list - GeeksforGeeks

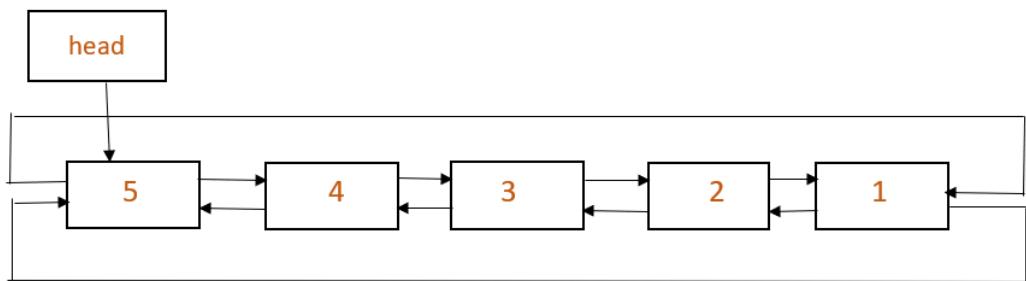
The problem is to reverse the given doubly circular linked list.

Examples:

Input:



Output:



Algorithm:

```
insertEnd(head, new_node)
    Declare last

    if head == NULL then
        new_node->next = new_node->prev = new_node
        head = new_node
        return

    last = head->prev
    new_node->next = head
    head->prev = new_node
    new_node->prev = last
    last->next = new_node

reverse(head)
    Initialize new_head = NULL
    Declare last

    last = head->prev
    Initialize curr = last, prev

    while curr->prev != last
        prev = curr->prev
        insertEnd(&new_head, curr)
        curr = prev
        insertEnd(&new_head, curr)

    return new_head
```

**Explanation:** The variable **head** in the parameter list of **insertEnd()** is pointer to a pointer variable. **reverse()** traverses the doubly circular linked list starting with **head** pointer in backward direction and one by one gets the node in the traversal. It inserts those nodes at the end of the list that starts with the **new\_head** pointer with the help of the function **insertEnd()** and finally returns **new\_head**.

```
// C++ implementation to reverse a
// doubly circular linked list
#include <bits/stdc++.h>

using namespace std;

// structure of a node of linked list
struct Node {
    int data;
    Node *next, *prev;
};
```

```
// function to create and return a new node
Node* getNode(int data)
{
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->data = data;
    return newNode;
}

// Function to insert at the end
void insertEnd(Node** head, Node* new_node)
{
    // If the list is empty, create a single node
    // circular and doubly list
    if (*head == NULL) {
        new_node->next = new_node->prev = new_node;
        *head = new_node;
        return;
    }

    // If list is not empty

    /* Find last node */
    Node* last = (*head)->prev;

    // Start is going to be next of new_node
    new_node->next = *head;

    // Make new node previous of start
    (*head)->prev = new_node;

    // Make last previous of new node
    new_node->prev = last;

    // Make new node next of old last
    last->next = new_node;
}

// Utility function to reverse a
// doubly circular linked list
Node* reverse(Node* head)
{
    if (!head)
        return NULL;

    // Initialize a new head pointer
    Node* new_head = NULL;

    // get pointer to the the last node
```

```
Node* last = head->prev;

// set 'curr' to last node
Node *curr = last, *prev;

// traverse list in backward direction
while (curr->prev != last) {
    prev = curr->prev;

    // insert 'curr' at the end of the list
    // starting with the 'new_head' pointer
    insertEnd(&new_head, curr);
    curr = prev;
}
insertEnd(&new_head, curr);

// head pointer of the reversed list
return new_head;
}

// function to display a doubly circular list in
// forward and backward direction
void display(Node* head)
{
    if (!head)
        return;

    Node* temp = head;

    cout << "Forward direction: ";
    while (temp->next != head) {
        cout << temp->data << " ";
        temp = temp->next;
    }
    cout << temp->data;

    Node* last = head->prev;
    temp = last;

    cout << "\nBackward direction: ";
    while (temp->prev != last) {
        cout << temp->data << " ";
        temp = temp->prev;
    }
    cout << temp->data;
}

// Driver program to test above
```

```
int main()
{
    Node* head = NULL;

    insertEnd(&head, getNode(1));
    insertEnd(&head, getNode(2));
    insertEnd(&head, getNode(3));
    insertEnd(&head, getNode(4));
    insertEnd(&head, getNode(5));

    cout << "Current list:\n";
    display(head);

    head = reverse(head);

    cout << "\n\nReversed list:\n";
    display(head);

    return 0;
}
```

Output:

```
Current list:
Forward direction: 1 2 3 4 5
Backward direction: 5 4 3 2 1

Reversed list:
Forward direction: 5 4 3 2 1
Backward direction: 1 2 3 4 5
```

Time Complexity: O(n).

## Source

<https://www.geeksforgeeks.org/reverse-a-doubly-circular-linked-list/>

## Chapter 214

# Reverse a doubly linked list in groups of given size

Reverse a doubly linked list in groups of given size - GeeksforGeeks

Given a doubly linked list containing **n** nodes. The problem is to reverse every group of **k** nodes in the list.

Examples:

**Input :** DLL: 10<->8<->4<->2

**k = 2**

**Output :** 8<->10<->2<->4

**Input :** 1<->2<->3<->4<->5<->6<->7<->8

**k = 3**

**Output :** 3<->2<->1<->6<->5<->4<->8<->7

**Prerequisite:** [Reverse a doubly linked list | Set-2.](#)

**Approach:** Create a recursive function say **reverse(head, k)**. This function receives the head or the first node of each group of **k** nodes. It reverses those group of **k** nodes by applying the approach discussed in [Reverse a doubly linked list | Set-2.](#) After reversing the group of **k** nodes the function checks whether next group of nodes exists in the list or not. If group exists then it makes a recursive call to itself with the first node of the next group and makes the necessary adjustments with the next and previous links of that group. Finally it returns the new head node of the reversed group.

```
// C++ implementation to reverse a doubly linked list
// in groups of given size
#include <bits/stdc++.h>
```

```
using namespace std;

// a node of the doubly linked list
struct Node {
    int data;
    Node *next, *prev;
};

// function to get a new node
Node* getNode(int data)
{
    // allocate space
    Node* new_node = (Node*)malloc(sizeof(Node));

    // put in the data
    new_node->data = data;
    new_node->next = new_node->prev = NULL;
    return new_node;
}

// function to insert a node at the beginning
// of the Doubly Linked List
void push(Node** head_ref, Node* new_node)
{
    // since we are adding at the beginning,
    // prev is always NULL
    new_node->prev = NULL;

    // link the old list off the new node
    new_node->next = (*head_ref);

    // change prev of head node to new node
    if ((*head_ref) != NULL)
        (*head_ref)->prev = new_node;

    // move the head to point to the new node
    (*head_ref) = new_node;
}

// function to reverse a doubly linked list
// in groups of given size
Node* revListInGroupOfGivenSize(Node* head, int k)
{
    Node *current = head;
    Node* next = NULL;
    Node* newHead = NULL;
    int count = 0;
```

```
// reversing the current group of k
// or less than k nodes by adding
// them at the beginning of list
// 'newHead'
while (current != NULL && count < k)
{
    next = current->next;
    push(&newHead, current);
    current = next;
    count++;
}

// if next group exists then making the desired
// adjustments in the link
if (next != NULL)
{
    head->next = revListInGroupOfGivenSize(next, k);
    head->next->prev = head;
}

// pointer to the new head of the
// reversed group
return newHead;
}

// Function to print nodes in a
// given doubly linked list
void printList(Node* head)
{
    while (head != NULL) {
        cout << head->data << " ";
        head = head->next;
    }
}

// Driver program to test above
int main()
{
    // Start with the empty list
    Node* head = NULL;

    // Create doubly linked: 10<->8<->4<->2
    push(&head, getNode(2));
    push(&head, getNode(4));
    push(&head, getNode(8));
    push(&head, getNode(10));

    int k = 2;
```

```
cout << "Original list: ";
printList(head);

// Reverse doubly linked list in groups of
// size 'k'
head = revListInGroupOfGivenSize(head, k);

cout << "\nModified list: ";
printList(head);

return 0;
}
```

Output:

```
Original list: 10 8 4 2
Modified list: 8 10 2 4
```

Time Complexity: O(n).

## Source

<https://www.geeksforgeeks.org/reverse-doubly-linked-list-groups-given-size/>

# Chapter 215

## Reverse a linked list

Reverse a linked list - GeeksforGeeks

Given pointer to the head node of a linked list, the task is to reverse the linked list. We need to reverse the list by changing links between nodes.

Examples:

```
Input : Head of following linked list  
       1->2->3->4->NULL  
Output : Linked list should be changed to,  
        4->3->2->1->NULL
```

```
Input : Head of following linked list  
       1->2->3->4->5->NULL  
Output : Linked list should be changed to,  
        5->4->3->2->1->NULL
```

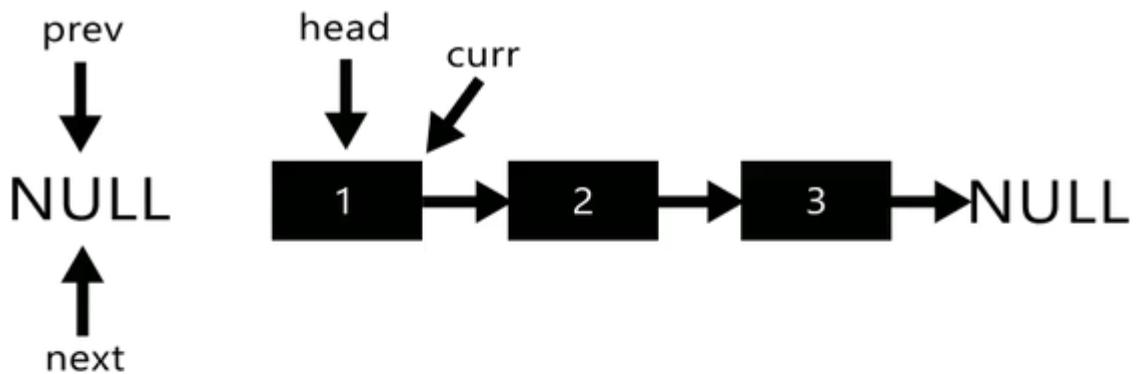
```
Input : NULL  
Output : NULL
```

```
Input : 1->NULL  
Output : 1->NULL
```

### Iterative Method

1. Initialize three pointers prev as NULL, curr as head and next as NULL.
2. Iterate through the linked list. In loop, do following.  
 // Before changing next of current,  
 // store next node  
 next = curr->next

```
// Now change next of current  
// This is where actual reversing happens  
curr->next = prev  
// Move prev and curr one step forward  
prev = curr  
curr = next
```



```
while (current != NULL)  
{  
    next = current->next;  
    current->next = prev;  
    prev = current;  
    current = next;  
}  
*head_ref = prev;
```

C++

```
// Iterative C++ program to reverse
```

```
// a linked list
#include<iostream>
using namespace std;

/* Link list node */
struct Node
{
    int data;
    struct Node* next;
    Node (int data)
    {
        this->data = data;
        next = NULL;
    }
};

struct LinkedList
{
    Node *head;
    LinkedList()
    {
        head = NULL;
    }

    /* Function to reverse the linked list */
    void reverse()
    {
        // Initialize current, previous and
        // next pointers
        Node *current = head;
        Node *prev = NULL, *next = NULL;

        while (current != NULL)
        {
            // Store next
            next = current->next;

            // Reverse current node's pointer
            current->next = prev;

            // Move pointers one position ahead.
            prev = current;
            current = next;
        }
        head = prev;
    }
}
```

```
/* Function to print linked list */
void print()
{
    struct Node *temp = head;
    while (temp != NULL)
    {
        cout << temp->data << " ";
        temp = temp->next;
    }
}

void push(int data)
{
    Node *temp = new Node(data);
    temp->next = head;
    head = temp;
}
};

/* Driver program to test above function*/
int main()
{
    /* Start with the empty list */
    LinkedList ll;
    ll.push(20);
    ll.push(4);
    ll.push(15);
    ll.push(85);

    cout << "Given linked list\n";
    ll.print();

    ll.reverse();

    cout << "\nReversed Linked list \n";
    ll.print();
    return 0;
}
```

C

```
// Iterative C program to reverse a linked list
#include<stdio.h>
#include<stdlib.h>

/* Link list node */
struct Node
```

```
{  
    int data;  
    struct Node* next;  
};  
  
/* Function to reverse the linked list */  
static void reverse(struct Node** head_ref)  
{  
    struct Node* prev = NULL;  
    struct Node* current = *head_ref;  
    struct Node* next = NULL;  
    while (current != NULL)  
    {  
        // Store next  
        next = current->next;  
  
        // Reverse current node's pointer  
        current->next = prev;  
  
        // Move pointers one position ahead.  
        prev = current;  
        current = next;  
    }  
    *head_ref = prev;  
}  
  
/* Function to push a node */  
void push(struct Node** head_ref, int new_data)  
{  
    struct Node* new_node =  
        (struct Node*) malloc(sizeof(struct Node));  
    new_node->data = new_data;  
    new_node->next = (*head_ref);  
    (*head_ref) = new_node;  
}  
  
/* Function to print linked list */  
void printList(struct Node *head)  
{  
    struct Node *temp = head;  
    while(temp != NULL)  
    {  
        printf("%d ", temp->data);  
        temp = temp->next;  
    }  
}  
  
/* Driver program to test above function*/
```

```
int main()
{
    /* Start with the empty list */
    struct Node* head = NULL;

    push(&head, 20);
    push(&head, 4);
    push(&head, 15);
    push(&head, 85);

    printf("Given linked list\n");
    printList(head);
    reverse(&head);
    printf("\nReversed Linked list \n");
    printList(head);
    getchar();
}
```

**Java**

```
// Java program for reversing the linked list

class LinkedList {

    static Node head;

    static class Node {

        int data;
        Node next;

        Node(int d) {
            data = d;
            next = null;
        }
    }

    /* Function to reverse the linked list */
    Node reverse(Node node) {
        Node prev = null;
        Node current = node;
        Node next = null;
        while (current != null) {
            next = current.next;
            current.next = prev;
            prev = current;
            current = next;
        }
    }
}
```

```
        node = prev;
        return node;
    }

    // prints content of double linked list
    void printList(Node node) {
        while (node != null) {
            System.out.print(node.data + " ");
            node = node.next;
        }
    }

    public static void main(String[] args) {
        LinkedList list = new LinkedList();
        list.head = new Node(85);
        list.head.next = new Node(15);
        list.head.next.next = new Node(4);
        list.head.next.next.next = new Node(20);

        System.out.println("Given Linked list");
        list.printList(head);
        head = list.reverse(head);
        System.out.println("");
        System.out.println("Reversed linked list ");
        list.printList(head);
    }
}
```

// This code has been contributed by Mayank Jaiswal

### Python

```
# Python program to reverse a linked list
# Time Complexity : O(n)
# Space Complexity : O(1)

# Node class
class Node:

    # Constructor to initialize the node object
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:

    # Function to initialize head
```

```
def __init__(self):
    self.head = None

# Function to reverse the linked list
def reverse(self):
    prev = None
    current = self.head
    while(current is not None):
        next = current.next
        current.next = prev
        prev = current
        current = next
    self.head = prev

# Function to insert a new node at the beginning
def push(self, new_data):
    new_node = Node(new_data)
    new_node.next = self.head
    self.head = new_node

# Utility function to print the linked LinkedList
def printList(self):
    temp = self.head
    while(temp):
        print temp.data,
        temp = temp.next

# Driver program to test above functions
llist = LinkedList()
llist.push(20)
llist.push(4)
llist.push(15)
llist.push(85)

print "Given Linked List"
llist.printList()
llist.reverse()
print "\nReversed Linked List"
llist.printList()

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

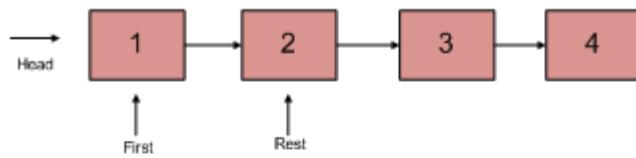
Given linked list  
85 15 4 20  
Reversed Linked list  
20 4 15 85

**Time Complexity:** O(n)  
**Space Complexity:** O(1)

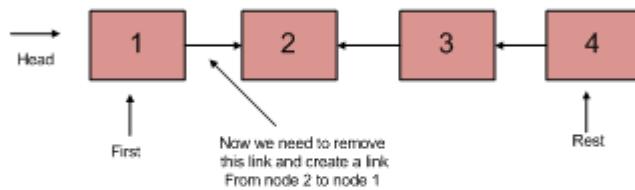
**Recursive Method:**

- 1) Divide the list in two parts - first node and rest of the linked list.
- 2) Call reverse for the rest of the linked list.
- 3) Link rest to first.
- 4) Fix head pointer

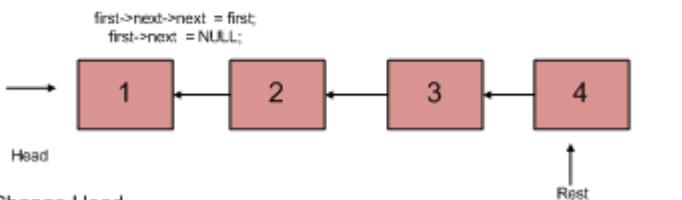
Divide the List in two parts



Reverse Rest



Link Rest to First



Change Head



```

void recursiveReverse(struct Node** head_ref)
{
    struct Node* first;
    struct Node* rest;

    /* empty list */
    if (*head_ref == NULL)
        return;
  
```

```
/* suppose first = {1, 2, 3}, rest = {2, 3} */
first = *head_ref;
rest  = first->next;

/* List has only one node */
if (rest == NULL)
    return;

/* reverse the rest list and put the first element at the end */
recursiveReverse(&rest);
first->next->next  = first;

/* tricky step -- see the diagram */
first->next  = NULL;

/* fix the head pointer */
*head_ref = rest;
}
```

**Time Complexity:**  $O(n)$

**Space Complexity:**  $O(1)$

#### A Simpler and Tail Recursive Method

Below is C++ implementation of this method.

C++

```
// A simple and tail recursive C++ program to reverse
// a linked list
#include<bits/stdc++.h>
using namespace std;

struct Node
{
    int data;
    struct Node *next;
};

void reverseUtil(Node *curr, Node *prev, Node **head);

// This function mainly calls reverseUtil()
// with prev as NULL
void reverse(Node **head)
{
    if (!head)
        return;
    reverseUtil(*head, NULL, head);
}
```

```
// A simple and tail recursive function to reverse
// a linked list. prev is passed as NULL initially.
void reverseUtil(Node *curr, Node *prev, Node **head)
{
    /* If last node mark it head*/
    if (!curr->next)
    {
        *head = curr;

        /* Update next to prev node */
        curr->next = prev;
        return;
    }

    /* Save curr->next node for recursive call */
    Node *next = curr->next;

    /* and update next ...*/
    curr->next = prev;

    reverseUtil(next, curr, head);
}

// A utility function to create a new node
Node *newNode(int key)
{
    Node *temp = new Node;
    temp->data = key;
    temp->next = NULL;
    return temp;
}

// A utility function to print a linked list
void printlist(Node *head)
{
    while(head != NULL)
    {
        cout << head->data << " ";
        head = head->next;
    }
    cout << endl;
}

// Driver program to test above functions
int main()
{
    Node *head1 = newNode(1);
```

```
head1->next = newNode(2);
head1->next->next = newNode(3);
head1->next->next->next = newNode(4);
head1->next->next->next->next = newNode(5);
head1->next->next->next->next->next = newNode(6);
head1->next->next->next->next->next->next = newNode(7);
head1->next->next->next->next->next->next->next = newNode(8);
cout << "Given linked list\n";
printlist(head1);
reverse(&head1);
cout << "\nReversed linked list\n";
printlist(head1);
return 0;
}
```

**Java**

```
// Java program for reversing the Linked list

class LinkedList {

    static Node head;

    static class Node {

        int data;
        Node next;

        Node(int d) {
            data = d;
            next = null;
        }
    }

    // A simple and tail recursive function to reverse
    // a linked list. prev is passed as NULL initially.
    Node reverseUtil(Node curr, Node prev) {

        /* If last node mark it head*/
        if (curr.next == null) {
            head = curr;

            /* Update next to prev node */
            curr.next = prev;

            return head;
        }
    }
}
```

```
/* Save curr->next node for recursive call */
Node next1 = curr.next;

/* and update next ...*/
curr.next = prev;

reverseUtil(next1, curr);
return head;
}

// prints content of double linked list
void printList(Node node) {
    while (node != null) {
        System.out.print(node.data + " ");
        node = node.next;
    }
}

public static void main(String[] args) {
    LinkedList list = new LinkedList();
    list.head = new Node(1);
    list.head.next = new Node(2);
    list.head.next.next = new Node(3);
    list.head.next.next.next = new Node(4);
    list.head.next.next.next.next = new Node(5);
    list.head.next.next.next.next.next = new Node(6);
    list.head.next.next.next.next.next.next = new Node(7);
    list.head.next.next.next.next.next.next.next = new Node(8);

    System.out.println("Original Linked list ");
    list.printList(head);
    Node res = list.reverseUtil(head, null);
    System.out.println("");
    System.out.println("");
    System.out.println("Reversed linked list ");
    list.printList(res);
}
}

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

### Python

```
# Simple and tail recursive Python program to
# reverse a linked list

# Node class
```

```
class Node:

    # Constructor to initialize the node object
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:

    # Function to initialize head
    def __init__(self):
        self.head = None

    def reverseUtil(self, curr, prev):

        # If last node mark it head
        if curr.next is None :
            self.head = curr

        # Update next to prev node
        curr.next = prev
        return

        # Save curr.next node for recursive call
        next = curr.next

        # And update next
        curr.next = prev

        self.reverseUtil(next, curr)

    # This function mainly calls reverseUtil()
    # with previous as None
    def reverse(self):
        if self.head is None:
            return
        self.reverseUtil(self.head, None)

    # Function to insert a new node at the beginning
    def push(self, new_data):
        new_node = Node(new_data)
        new_node.next = self.head
        self.head = new_node

    # Utility function to print the linked LinkedList
```

```
def printList(self):
    temp = self.head
    while(temp):
        print temp.data,
        temp = temp.next

# Driver program
llist = LinkedList()
llist.push(8)
llist.push(7)
llist.push(6)
llist.push(5)
llist.push(4)
llist.push(3)
llist.push(2)
llist.push(1)

print "Given linked list"
llist.printList()

llist.reverse()

print "\nReverse linked list"
llist.printList()

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

Output:

```
Given linked list
1 2 3 4 5 6 7 8
```

```
Reversed linked list
8 7 6 5 4 3 2 1
```

Thanks to Gaurav Ahirwar for suggesting this solution.

[Iteratively Reverse a linked list using only 2 pointers \(An Interesting Method\)](#)

**References:**

<http://cslibrary.stanford.edu/105/LinkedListProblems.pdf>

**Improved By :** [AshishKumar22](#)

**Source**

<https://www.geeksforgeeks.org/reverse-a-linked-list/>

## Chapter 216

# Reverse a stack without using extra space in O(n)

Reverse a stack without using extra space in O(n) - GeeksforGeeks

Reverse a [Stack](#) without using recursion and extra space .Even the functional Stack is not allowed.

Examples:

Input : 1->2->3->4  
Output : 4->3->2->1

Input : 6->5->4  
Output : 4->5->6

We have discussed a way of reversing a string in below post.

[Reverse a Stack using Recursion](#)

The above solution requires O(n) extra space. We can reverse a string in O(1) time if we internally represent the stack as linked list. Reverse a stack would require a reversing a linked list which can be done with O(n) time and O(1) extra space.

Note that push() and pop() operations still take O(1) time.

C++

```
// CPP program to implement Stack
// using linked list so that reverse
// can be done with O(1) extra space.
#include<bits/stdc++.h>
```

```
using namespace std;

class StackNode {
public:
    int data;
    StackNode *next;

    StackNode(int data)
    {
        this->data = data;
        this->next = NULL;
    }
};

class Stack {
    StackNode *top;

public:
    // Push and pop operations
    void push(int data)
    {
        if (top == NULL) {
            top = new StackNode(data);
            return;
        }
        StackNode *s = new StackNode(data);
        s->next = top;
        top = s;
    }

    StackNode* pop()
    {
        StackNode *s = top;
        top = top->next;
        return s;
    }

    // prints contents of stack
    void display()
    {
        StackNode *s = top;
        while (s != NULL) {
            cout << s->data << " ";
            s = s->next;
        }
        cout << endl;
    }
};
```

```
}

// Reverses the stack using simple
// linked list reversal logic.
void reverse()
{
    StackNode *prev, *cur, *succ;
    cur = prev = top;
    cur = cur->next;
    prev->next = NULL;
    while (cur != NULL) {

        succ = cur->next;
        cur->next = prev;
        prev = cur;
        cur = succ;
    }
    top = prev;
}
};

// driver code
int main()
{
    Stack *s = new Stack();
    s->push(1);
    s->push(2);
    s->push(3);
    s->push(4);
    cout << "Original Stack" << endl;;
    s->display();
    cout << endl;

    // reverse
    s->reverse();

    cout << "Reversed Stack" << endl;
    s->display();

    return 0;
}
// This code is contribute by Chhavi.
```

### Java

```
// Java program to implement Stack using linked
// list so that reverse can be done with O(1)
// extra space.
```

```
class StackNode {  
    int data;  
    StackNode next;  
    public StackNode(int data)  
    {  
        this.data = data;  
        this.next = null;  
    }  
}  
  
class Stack {  
    StackNode top;  
  
    // Push and pop operations  
    public void push(int data)  
    {  
        if (this.top == null) {  
            top = new StackNode(data);  
            return;  
        }  
        StackNode s = new StackNode(data);  
        s.next = this.top;  
        this.top = s;  
    }  
    public StackNode pop()  
    {  
        StackNode s = this.top;  
        this.top = this.top.next;  
        return s;  
    }  
  
    // prints contents of stack  
    public void display()  
    {  
        StackNode s = this.top;  
        while (s != null) {  
            System.out.print(" " + s.data);  
            s = s.next;  
        }  
        System.out.println();  
    }  
  
    // Reverses the stack using simple  
    // linked list reversal logic.  
    public void reverse()  
    {  
        StackNode prev, cur, succ;  
        cur = prev = this.top;
```

```
        cur = cur.next;
        prev.next = null;
        while (cur != null) {

            succ = cur.next;
            cur.next = prev;
            prev = cur;
            cur = succ;
        }
        this.top = prev;
    }

public class reverseStackWithoutSpace {
    public static void main(String[] args)
    {
        Stack s = new Stack();
        s.push(1);
        s.push(2);
        s.push(3);
        s.push(4);
        System.out.println("Original Stack");
        s.display();

        // reverse
        s.reverse();

        System.out.println("Reversed Stack");
        s.display();
    }
}
```

Output:

```
Original Stack
4 3 2 1
Reversed Stack
1 2 3 4
```

## Source

<https://www.geeksforgeeks.org/reverse-stack-without-using-extra-space/>

## Chapter 217

# Reverse a sublist of linked list

Reverse a sublist of linked list - GeeksforGeeks

We are given a linked list and positions m and n. We need to reverse the linked list from position m to n.

Examples:

```
Input : 10->20->30->40->50->60->70->NULL
        m = 3, n = 6
Output : 10->20->60->50->40->30->70->NULL
```

```
Input : 1->2->3->4->5->6->NULL
        m = 2, n = 4
Output : 1->4->3->2->5->6->NULL
```

To reverse the linked list from position m to n, we find addresses of start and end position of the linked list by running a loop, and then we unlink this part from the rest of the list and then use the normal [linked list reverse function](#) which we have earlier used for reversing the complete linked list, and use it to reverse the portion of the linked list which need to be reversed. After reversal, we again attach the portion reversed to the main list.

```
// C program to reverse a linked list
// from position m to position n
#include <stdio.h>
#include <stdlib.h>

// Linked list node
struct Node {
    int data;
    struct Node* next;
};


```

```
// the standard reverse function used
// to reverse a linked list
struct Node* reverse(struct Node* head)
{
    struct Node* prev = NULL;
    struct Node* curr = head;

    while (curr) {
        struct Node* next = curr->next;
        curr->next = prev;
        prev = curr;
        curr = next;
    }
    return prev;
}

// function used to reverse a linked list
// from position m to n which uses reverse
// function
Node* reverseBetween(Node* head, int m, int n)
{
    if (m == n)
        return head;

    // revs and revend is start and end respectively
    // of the portion of the linked list which
    // need to be reversed. revs_prev is previous
    // of starting position and revend_next is next
    // of end of list to be reversed.
    Node* revs = NULL, *revs_prev = NULL;
    Node* revend = NULL, *revend_next = NULL;

    // Find values of above pointers.
    int i = 1;
    Node* curr = head;
    while (curr && i <= n) {
        if (i < m)
            revs_prev = curr;

        if (i == m)
            revs = curr;

        if (i == n) {
            revend = curr;
            revend_next = curr->next;
        }
    }
}
```

```
        curr = curr->next;
        i++;
    }
    revend->next = NULL;

    // Reverse linked list starting with
    // revs.
    revend = reverse(revs);

    // If starting position was not head
    if (revs_prev)
        revs_prev->next = revend;

    // If starting position was head
    else
        head = revend;

    revs->next = revend_next;
    return head;
}

void print(struct Node* head)
{
    while (head != NULL) {
        printf("%d ", head->data);
        head = head->next;
    }
    printf("\n");
}

// function to add a new node at the
// begining of the list
void push(struct Node** head_ref, int new_data)
{
    struct Node* new_node = new Node;
    new_node->data = new_data;
    new_node->next = (*head_ref);
    (*head_ref) = new_node;
}

// Driver code
int main()
{
    struct Node* head = NULL;
    push(&head, 70);
    push(&head, 60);
    push(&head, 50);
    push(&head, 40);
```

```
push(&head, 30);
push(&head, 20);
push(&head, 10);
reverseBetween(head, 3, 6);
print(head);
return 0;
}
```

Output:

```
10 20 60 50 40 30 70
```

## Source

<https://www.geeksforgeeks.org/reverse-sublist-linked-list/>

## Chapter 218

# Reverse alternate K nodes in a Singly Linked List

Reverse alternate K nodes in a Singly Linked List - GeeksforGeeks

Given a linked list, write a function to reverse every alternate k nodes (where k is an input to the function) in an efficient way. Give the complexity of your algorithm.

Example:

Inputs: 1->2->3->4->5->6->7->8->9->NULL and k = 3  
Output: 3->2->1->4->5->6->9->8->7->NULL.

### Method 1 (Process 2k nodes and recursively call for rest of the list)

This method is basically an extension of the method discussed in [this post](#).

```
kAltReverse(struct node *head, int k)
1) Reverse first k nodes.
2) In the modified list head points to the kth node. So change next
   of head to (k+1)th node
3) Move the current pointer to skip next k nodes.
4) Call the kAltReverse() recursively for rest of the n - 2k nodes.
5) Return new head of the list.
```

C

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

/* Link list node */
```

```

struct Node
{
    int data;
    struct Node* next;
};

/* Reverses alternate k nodes and
   returns the pointer to the new head node */
struct Node *kAltReverse(struct Node *head, int k)
{
    struct Node* current = head;
    struct Node* next;
    struct Node* prev = NULL;
    int count = 0;

    /*1) reverse first k nodes of the linked list */
    while (current != NULL && count < k)
    {
        next = current->next;
        current->next = prev;
        prev = current;
        current = next;
        count++;
    }

    /* 2) Now head points to the kth node. So change next
       of head to (k+1)th node*/
    if(head != NULL)
        head->next = current;

    /* 3) We do not want to reverse next k nodes. So move the current
       pointer to skip next k nodes */
    count = 0;
    while(count < k-1 && current != NULL )
    {
        current = current->next;
        count++;
    }

    /* 4) Recursively call for the list starting from current->next.
       And make rest of the list as next of first node */
    if(current != NULL)
        current->next = kAltReverse(current->next, k);

    /* 5) prev is new head of the input list */
    return prev;
}

```

```
/* UTILITY FUNCTIONS */
/* Function to push a node */
void push(struct Node** head_ref, int new_data)
{
    /* allocate node */
    struct Node* new_node =
        (struct Node*) malloc(sizeof(struct Node));

    /* put in the data */
    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;
}

/* Function to print linked list */
void printList(struct Node *node)
{
    int count = 0;
    while(node != NULL)
    {
        printf("%d ", node->data);
        node = node->next;
        count++;
    }
}

/* Driver program to test above function*/
int main(void)
{
    /* Start with the empty list */
    struct Node* head = NULL;
    int i;
    // create a list 1->2->3->4->5..... ->20
    for(i = 20; i > 0; i--)
        push(&head, i);

    printf("\n Given linked list \n");
    printList(head);
    head = kAltReverse(head, 3);

    printf("\n Modified Linked list \n");
    printList(head);

    getchar();
}
```

```
        return(0);
}
```

**Java**

```
// Java program to reverse alternate k nodes in a linked list

class LinkedList {

    static Node head;

    class Node {

        int data;
        Node next;

        Node(int d) {
            data = d;
            next = null;
        }
    }

    /* Reverses alternate k nodes and
     * returns the pointer to the new head node */
    Node kAltReverse(Node node, int k) {
        Node current = node;
        Node next = null, prev = null;
        int count = 0;

        /*1) reverse first k nodes of the linked list */
        while (current != null && count < k) {
            next = current.next;
            current.next = prev;
            prev = current;
            current = next;
            count++;
        }

        /* 2) Now head points to the kth node. So change next
         * of head to (k+1)th node*/
        if (node != null) {
            node.next = current;
        }

        /* 3) We do not want to reverse next k nodes. So move the current
         * pointer to skip next k nodes */
        count = 0;
        while (count < k - 1 && current != null) {
```

```
        current = current.next;
        count++;
    }

    /* 4) Recursively call for the list starting from current->next.
       And make rest of the list as next of first node */
    if (current != null) {
        current.next = kAltReverse(current.next, k);
    }

    /* 5) prev is new head of the input list */
    return prev;
}

void printList(Node node) {
    while (node != null) {
        System.out.print(node.data + " ");
        node = node.next;
    }
}

void push(int newdata) {
    Node mynode = new Node(newdata);
    mynode.next = head;
    head = mynode;
}

public static void main(String[] args) {
    LinkedList list = new LinkedList();

    // Creating the linkedlist
    for (int i = 20; i > 0; i--) {
        list.push(i);
    }
    System.out.println("Given Linked List :");
    list.printList(head);
    head = list.kAltReverse(head, 3);
    System.out.println("");
    System.out.println("Modified Linked List :");
    list.printList(head);

}
}

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

Output:  
*Given linked list*

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

*Modified Linked list*

3 2 1 4 5 6 9 8 7 10 11 12 15 14 13 16 17 18 20 19

Time Complexity: O(n)

## Source

<https://www.geeksforgeeks.org/reverse-alternate-k-nodes-in-a-singly-linked-list/>

## Chapter 219

# Reverse each word in a linked list node

Reverse each word in a linked list node - GeeksforGeeks

Given a linked list of strings, we need to reverse each word of the string in the given linked list.

Examples:

Input: geeksforgeeks a computer science portal for geeks  
Output: skeegrofskeeg a retupmoc ecneics latrop rof skeeg

Input: Publish your own articles on geeksforgeeks  
Output: hsilbuP ruoy nwo selcitra no skeegrofskeeg

Using a loop iterate the list till null and take string from each node and reverse the string.

```
// C++ program to reverse each word
// in a linked list
#include <bits/stdc++.h>
using namespace std;

// Linked list Node structure
struct Node {
    string c;
    struct Node* next;
};

// Function to create newNode
// in a linked list
struct Node* newNode(string c)
```

```
{  
    Node* temp = new Node;  
    temp->c = c;  
    temp->next = NULL;  
    return temp;  
}  
  
// reverse each node data  
void reverse_word(string& str)  
{  
    reverse(str.begin(), str.end());  
}  
  
void reverse(struct Node* head)  
{  
    struct Node* ptr = head;  
  
    // iterate each node and call reverse_word  
    // for each node data  
    while (ptr != NULL) {  
        reverse_word(ptr->c);  
        ptr = ptr->next;  
    }  
}  
  
// printing linked list  
void printList(struct Node* head)  
{  
    while (head != NULL) {  
        cout << head->c << " ";  
        head = head->next;  
    }  
}  
  
// Driver program  
int main()  
{  
    Node* head = newNode("Geeksforgeeks");  
    head->next = newNode("a");  
    head->next->next = newNode("computer");  
    head->next->next->next = newNode("science");  
    head->next->next->next->next = newNode("portal");  
    head->next->next->next->next->next = newNode("for");  
    head->next->next->next->next->next->next = newNode("geeks");  
  
    cout << "List before reverse: \n";  
    printList(head);  
}
```

```
reverse(head);

cout << "\n\nList after reverse: \n";
printList(head);

return 0;
}
```

**Output:**

```
List before reverse:
Geeksforgeeks a computer science portal for geeks
```

```
List after reverse:
skeegrofskeeG a retupmoc ecneics latrop rof skeeg
```

**Source**

<https://www.geeksforgeeks.org/reverse-word-linked-list-node/>

## Chapter 220

# Reverse first K elements of given linked list

Reverse first K elements of given linked list - GeeksforGeeks

Given pointer to the head node of a linked list and a number K, the task is to reverse the first K nodes of the linked list. We need to reverse the list by changing links between nodes. check also [Reversal of a linked list](#)

Examples:

Input : 1->2->3->4->5->6->7->8->9->10->NULL

k = 3

Output :3->2->1->4->5->6->7->8->9->10->NULL

Input :10->18->20->25->35->NULL

k = 2

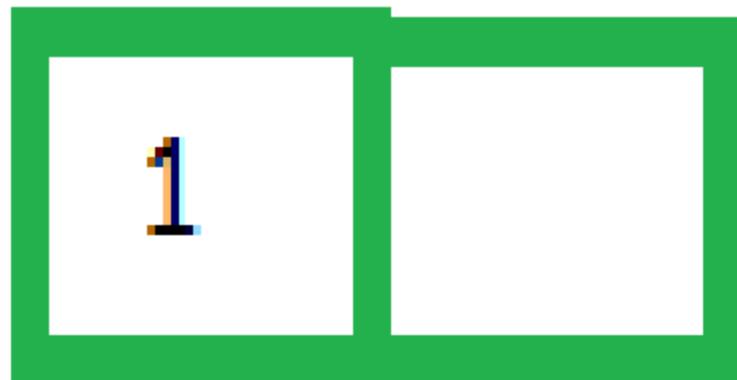
Output :18->10->20->25->35->NULL

Explanation of the method:

suppose linked list is 1->2->3->4->5->NULL nad k=3

- 1) Traverse the linked list till K-th point.
- 2) Break the linked list in to two parts from k-th point. After partition linked list will look like 1->2->3->NULL & 4->5->NULL
- 3) [Reverse first part of the linked list](#) leave second part as it is 3->2->1->NULL and 4->5->NULL
- 4) Join both the parts of the linked list, we get 3->2->1->4->5->NULL

A pictorial representation of how the algorithm works



for



```
// C++ program for reversal of first k elements
// of given linked list
#include <bits/stdc++.h>
using namespace std;

/* Link list node */
struct Node {
    int data;
    struct Node* next;
};

/* Function to reverse first k elements of linked list */
static void reverseKNodes(struct Node** head_ref, int k)
{
    // traverse the linked list until break
    // point not meet
    struct Node* temp = *head_ref;
    int count = 1;
    while (count < k) {
        temp = temp->next;
        count++;
    }

    // backup the joint point
    struct Node* joint_point = temp->next;
    temp->next = NULL; // break the list

    // reverse the list till break point
    struct Node* prev = NULL;
    struct Node* current = *head_ref;
    struct Node* next;
    while (current != NULL) {
        next = current->next;
        current->next = prev;
        prev = current;
        current = next;
    }

    // join both parts of the linked list
    // traverse the list until NULL is not
    // found
    *head_ref = prev;
    current = *head_ref;
    while (current->next != NULL)
        current = current->next;

    // joint both part of the list
    current->next = joint_point;
```

```
}

/* Function to push a node */
void push(struct Node** head_ref, int new_data)
{
    struct Node* new_node =
        (struct Node*)malloc(sizeof(struct Node));
    new_node->data = new_data;
    new_node->next = (*head_ref);
    (*head_ref) = new_node;
}

/* Function to print linked list */
void printList(struct Node* head)
{
    struct Node* temp = head;
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
}

/* Driver program to test above function*/
int main()
{
    // Create a linked list 1->2->3->4->5
    struct Node* head = NULL;
    push(&head, 5);
    push(&head, 4);
    push(&head, 3);
    push(&head, 2);
    push(&head, 1);

    // k should be less than the
    // numbers of nodes
    int k = 3;

    cout << "\nGiven list\n";
    printList(head);

    reverseKNodes(&head, k);

    cout << "\nModified list\n";
    printList(head);

    return 0;
}
```

Output:

```
Given list  
1 2 3 4 5  
Modified list  
3 2 1 4 5
```

### Source

<https://www.geeksforgeeks.org/reverse-first-k-elements-given-linked-list/>

## Chapter 221

# Reverse nodes of a linked list without affecting the special characters

Reverse nodes of a linked list without affecting the special characters - GeeksforGeeks

Given a linked list of alphabets and special characters. Reverse the given linked list without affecting the position of the special characters.

**Examples:**

**Input:** g -> @ -> e -> # -> e -> \$ -> k -> s -> NULL

**Output:** s -> @ -> k -> # -> e -> \$ -> e -> g -> NULL

**Explanation:** Here we can see that in the output the position of special character in not change and also linked list is reverse.

The idea is to traverse the linked list and store the characters excluding the special characters in a temporary array. Again traverse the linked list and copy elements from the array to the nodes of the linked list in a reverse manner.

**Below is the step by step algorithm:**

1. Take a temporary array, TEMP\_ARR.
2. Traverse the linked list and do the following
  - if the current element is an alphabet, store that element of the linked list to TEMP\_ARR.
  - else, increase node pointer by one
3.     • if the current element is an alphabet, copy the last element of TEMP\_ARR to the current linked list node and decrease the current index of TEMP\_ARR for the next iteration.

- else, increase node by one

Below is the implementation of above approach:

C++

```
// CPP program to reverse a linked list
// without affecting special characters

#include <iostream>

using namespace std;

// Link list node
struct Node {
    char data;
    struct Node* next;
};

// Function to reverse the linked list
// without affecting special characters
void reverse(struct Node** head_ref, int size)
{
    struct Node* current = *head_ref;

    char TEMP_ARR[size];

    int i = 0;

    // Traverse the linked list and insert
    // linked list elements to TEMP_ARR
    while (current != NULL) {
        // if the cuurent data is any alphabet than
        // store it in to TEMP_ARR
        if ((current->data >= 97 && current->data <= 122) ||
            (current->data >= 65 && current->data <= 90)) {
            TEMP_ARR[i++] = current->data;
            current = current->next;
        }
        // else increase the node position
        else
            current = current->next;
    }

    current = *head_ref;
    // Traverse the linked list again
```

```
while (current != NULL)
{
    // if current character is an alphabet than
    // replace the current element in the linked list
    // with the last element of the TEMP_ARR
    if ((current->data >= 97 && current->data <= 122) ||
        (current->data >= 65 && current->data <= 90)) {
        current->data = TEMP_ARR[--i];
        current = current->next;
    }
    // else increase the node
    else
        current = current->next;
}

// Function to push a node
void push(struct Node** head_ref, char new_data)
{
    /* allocate node */
    struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));

    /* put in the data */
    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;
}

/* Function to print linked list */
void printList(struct Node* head)
{
    struct Node* temp = head;
    while (temp != NULL) {
        cout << temp->data;
        temp = temp->next;
    }
}

// Driver program to test above function
int main()
{
    /* Start with the empty list */
    struct Node* head = NULL;
```

```
push(&head, 's');
push(&head, '$');
push(&head, 'k');
push(&head, 'e');
push(&head, 'e');
push(&head, '@');
push(&head, '#');
push(&head, 'g');
push(&head, 'r');
push(&head, 'o');
push(&head, 'f');
push(&head, 's');
push(&head, '$');
push(&head, 'k');
push(&head, 'e');
push(&head, 'e');
push(&head, 'g');

cout << "Given linked list: ";
printList(head);

reverse(&head, 13);

cout << "\nReversed Linked list: ";
printList(head);

return 0;
}
```

### Java

```
// Java program to reverse a
// linked list without affecting
// special characters
class GFG
{

    // Link list node
    public static class Node
    {
        char data;
        Node next;
    }

    // Function to reverse the linked
    // list without affecting special
    // characters
    static void reverse(Node head_ref,
```

```
        int size)
{
Node current = head_ref;

char TEMP_ARR[] = new char[size];

int i = 0;

// Traverse the linked list
// and insert linked list
// elements to TEMP_ARR
while (current != null)
{
    // if the cuurent data
    // is any alphabet than
    // store it in to TEMP_ARR
    if ((current.data >= 97 &&
        current.data <= 122) ||
        (current.data >= 65 &&
        current.data <= 90))
    {
        TEMP_ARR[i++] = current.data;
        current = current.next;
    }

    // else increase the node position
    else
        current = current.next;
}

current = head_ref;

// Traverse the linked list again
while (current != null)
{
    // if current character is an
    // alphabet than replace the
    // current element in the linked
    // list with the last element
    // of the TEMP_ARR
    if ((current.data >= 97 &&
        current.data <= 122) ||
        (current.data >= 65 &&
        current.data <= 90))
    {
        current.data = TEMP_ARR[--i];
        current = current.next;
    }
}
```

```
}

// else increase the node
else
    current = current.next;
}
}

// Function to push a node
static Node push(Node head_ref,
                  char new_data)
{
    /* allocate node */
    Node new_node = new Node();

    /* put in the data */
    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;

    return head_ref;
}

/* Function to print linked list */
static void printList(Node head)
{
    Node temp = head;
    while (temp != null)
    {
        System.out.print(temp.data);
        temp = temp.next;
    }
}

// Driver Code
public static void main(String rags[])
{
    /* Start with the empty list */
    Node head = null;

    head = push(head, 's');
    head = push(head, '$');
```

```
head = push(head, 'k');
head = push(head, 'e');
head = push(head, 'e');
head = push(head, '@');
head = push(head, '#');
head = push(head, 'g');
head = push(head, 'r');
head = push(head, 'o');
head = push(head, 'f');
head = push(head, 's');
head = push(head, '$');
head = push(head, 'k');
head = push(head, 'e');
head = push(head, 'e');
head = push(head, 'g');

System.out.print( "Given linked list: ");
printList(head);

reverse(head, 13);

System.out.print("\nReversed Linked list: ");
printList(head);
}

}

// This code is contributed by Arnab Kundu
```

**Output:**

```
Given linked list: geek$sforg#@eek$s
Reversed Linked list: skee$grofs#@kee$g
```

**Improved By :** [andrew1234](#)

**Source**

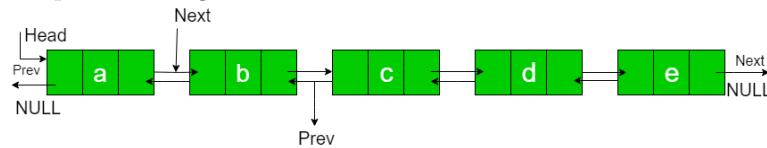
<https://www.geeksforgeeks.org/reverse-nodes-of-a-linked-list-without-affecting-the-special-characters/>

## Chapter 222

# Rotate Doubly linked list by N nodes

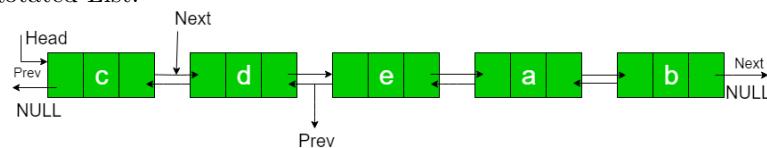
Rotate Doubly linked list by N nodes - GeeksforGeeks

Given a doubly linked list, rotate the linked list counter-clockwise by N nodes. Here N is a given positive integer and is smaller than the count of nodes in linked list.



$N = 2$

Rotated List:



Examples:

Input : a b c d e     $N = 2$   
Output : c d e a b

Input : a b c d e f g h     $N = 4$   
Output : e f g h a b c d

Asked in [Amazon](#)

To rotate the Doubly linked list, we need to change next of Nth node to NULL, next of last node to previous head node, and prev of head node to last node and finally change head to (N+1)th node and prev of new head node to NULL (Prev of Head node in doubly linked list is NULL)

So we need to get hold of three nodes: Nth node, (N+1)th node and last node. Traverse the list from beginning and stop at Nth node. Store pointer to Nth node. We can get (N+1)th node using NthNode->next. Keep traversing till end and store pointer to last node also. Finally, change pointers as stated above and at Last Print Rotated List using PrintList Function.

```
// C++ program to rotate a Doubly linked
// list counter clock wise by N times
#include <bits/stdc++.h>
using namespace std;

/* Link list node */
struct Node {
    char data;
    struct Node* prev;
    struct Node* next;
};

// This function rotates a doubly linked
// list counter-clockwise and updates the
// head. The function assumes that N is
// smallerthan size of linked list. It
// doesn't modify the list if N is greater
// than or equal to size
void rotate(struct Node** head_ref, int N)
{
    if (N == 0)
        return;

    // Let us understand the below code
    // for example N = 2 and
    // list = a <-> b <-> c <-> d <-> e.
    struct Node* current = *head_ref;

    // current will either point to Nth
    // or NULL after this loop. Current
    // will point to node 'b' in the
    // above example
    int count = 1;
    while (count < N && current != NULL) {
        current = current->next;
        count++;
    }
}
```

```
// If current is NULL, N is greater
// than or equal to count of nodes
// in linked list. Don't change the
// list in this case
if (current == NULL)
    return;

// current points to Nth node. Store
// it in a variable. NthNode points to
// node 'b' in the above example
struct Node* NthNode = current;

// current will point to last node
// after this loop current will point
// to node 'e' in the above example
while (current->next != NULL)
    current = current->next;

// Change next of last node to previous
// head. Next of 'e' is now changed to
// node 'a'
current->next = *head_ref;

// Change prev of Head node to current
// Prev of 'a' is now changed to node 'e'
(*head_ref)->prev = current;

// Change head to (N+1)th node
// head is now changed to node 'c'
*head_ref = NthNode->next;

// Change prev of New Head node to NULL
// Because Prev of Head Node in Doubly
// linked list is NULL
(*head_ref)->prev = NULL;

// change next of Nth node to NULL
// next of 'b' is now NULL
NthNode->next = NULL;
}

// Function to insert a node at the
// beginning of the Doubly Linked List
void push(struct Node** head_ref, int new_data)
{
    struct Node* new_node = new Node;
    new_node->data = new_data;
    new_node->prev = NULL;
```

```
new_node->next = (*head_ref);
if ((*head_ref) != NULL)
    (*head_ref)->prev = new_node;
*head_ref = new_node;
}

/* Function to print linked list */
void printList(struct Node* node)
{
    while (node->next != NULL) {
        cout << node->data << " "
            << "<=>"
            << " ";
        node = node->next;
    }
    cout << node->data;
}

// Driver's Code
int main(void)
{
    /* Start with the empty list */
    struct Node* head = NULL;

    /* Let us create the doubly
       linked list a<->b<->c<->d<->e */
    push(&head, 'e');
    push(&head, 'd');
    push(&head, 'c');
    push(&head, 'b');
    push(&head, 'a');

    int N = 2;

    cout << "Given linked list \n";
    printList(head);
    rotate(&head, N);

    cout << "\nRotated Linked list \n";
    printList(head);

    return 0;
}
```

**Output:**

```
Given linked list
```

```
a b c d e  
Rotated Linked list  
c d e a b
```

## Source

<https://www.geeksforgeeks.org/rotate-doubly-linked-list-n-nodes/>

## Chapter 223

# Rotate Linked List block wise

Rotate Linked List block wise - GeeksforGeeks

Given a Linked List of length  $n$  and block length  $k$  rotate in circular manner towards right/left each block by a number  $d$ . If  $d$  is positive rotate towards right else rotate towards left.

**Examples:**

Input: 1->2->3->4->5->6->7->8->9->NULL,  
k = 3  
d = 1

Output: 3->1->2->6->4->5->9->7->8->NULL

Explanation: Here blocks of size 3 are rotated towards right(as  $d$  is positive) by 1.

Input: 1->2->3->4->5->6->7->8->9->10->  
11->12->13->14->15->NULL,  
k = 4  
d = -1

Output: 2->3->4->1->6->7->8->5->10->11  
->12->9->14->15->13->NULL

Explanation: Here, at the end of linked list, remaining nodes are less than  $k$ , i.e. only three nodes are left while  $k$  is 4. Rotate those 3 nodes also by  $d$ .

Prerequisite: [Rotate a linked list](#)

The idea is if the absolute value of  $d$  is greater than the value of  $k$ , then rotate the link list by  $d \% k$  times. If  $d$  is 0, no need to rotate the linked list at all.

```
// C/C++ program to rotate a linked list block wise
#include <stdio.h>
#include <stdlib.h>

/* Link list node */
struct Node {
    int data;
    struct Node* next;
};

// Recursive function to rotate one block
struct Node* rotateHelper(struct Node* blockHead,
                          struct Node* blockTail,
                          int d, struct Node** tail,
                          int k)
{
    if (d == 0)
        return blockHead;

    // Rotate Clockwise
    if (d > 0) {
        struct Node* temp = blockHead;
        for (int i = 1; temp->next->next &&
             i < k - 1; i++)
            temp = temp->next;
        blockTail->next = blockHead;
        *tail = temp;
        return rotateHelper(blockTail, temp,
                            d - 1, tail, k);
    }

    // Rotate anti-Clockwise
    if (d < 0) {
        blockTail->next = blockHead;
        *tail = blockHead;
        return rotateHelper(blockHead->next,
                            blockHead, d + 1, tail, k);
    }
}

// Function to rotate the linked list block wise
struct Node* rotateByBlocks(struct Node* head,
                           int k, int d)
{
    // If length is 0 or 1 return head
    if (!head || !head->next)
        return head;
```

```
// if degree of rotation is 0, return head
if (d == 0)
    return head;

struct Node* temp = head, *tail = NULL;

// Traverse upto last element of this block
int i;
for (i = 1; temp->next && i < k; i++)
    temp = temp->next;

// storing the first node of next block
struct Node* nextBlock = temp->next;

// If nodes of this block are less than k.
// Rotate this block also
if (i < k)
    head = rotateHelper(head, temp, d % k,
                         &tail, i);
else
    head = rotateHelper(head, temp, d % k,
                         &tail, k);

// Append the new head of next block to
// the tail of this block
tail->next = rotateByBlocks(nextBlock, k,
                             d % k);

// return head of updated Linked List
return head;
}

/* UTILITY FUNCTIONS */
/* Function to push a node */
void push(struct Node** head_ref, int new_data)
{
    struct Node* new_node = new Node;
    new_node->data = new_data;
    new_node->next = (*head_ref);
    (*head_ref) = new_node;
}

/* Function to print linked list */
void printList(struct Node* node)
{
    while (node != NULL) {
        printf("%d ", node->data);
        node = node->next;
    }
}
```

```
    }

}

/* Driver program to test above function*/
int main()
{
    /* Start with the empty list */
    struct Node* head = NULL;

    // create a list 1->2->3->4->5->
    // 6->7->8->9->NULL
    for (int i = 9; i > 0; i -= 1)
        push(&head, i);

    printf("Given linked list \n");
    printList(head);

    // k is block size and d is number of
    // rotations in every block.
    int k = 3, d = 2;
    head = rotateByBlocks(head, k, d);

    printf("\nRotated by blocks Linked list \n");
    printList(head);

    return (0);
}
```

**Output:**

```
Given linked list
1 2 3 4 5 6 7 8 9
Rotated by blocks Linked list
2 3 1 5 6 4 8 9 7
```

Improved By : [codeShaurya](#)

**Source**

<https://www.geeksforgeeks.org/rotate-linked-list-block-wise/>

## Chapter 224

# Rotate a Linked List

Rotate a Linked List - GeeksforGeeks

Given a singly linked list, rotate the linked list counter-clockwise by k nodes. Where k is a given positive integer. For example, if the given linked list is 10->20->30->40->50->60 and k is 4, the list should be modified to 50->60->10->20->30->40. Assume that k is smaller than the count of nodes in linked list.

To rotate the linked list, we need to change next of kth node to NULL, next of last node to previous head node, and finally change head to (k+1)th node. So we need to get hold of three nodes: kth node, (k+1)th node and last node.

Traverse the list from beginning and stop at kth node. Store pointer to kth node. We can get (k+1)th node using kthNode->next. Keep traversing till end and store pointer to last node also. Finally, change pointers as stated above.

C/C++

```
// C/C++ program to rotate a linked list counter clock wise

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

/* Link list node */
struct Node
{
    int data;
    struct Node* next;
};

// This function rotates a linked list counter-clockwise and
// updates the head. The function assumes that k is smaller
// than size of linked list. It doesn't modify the list if
// k is greater than or equal to size
void rotate(struct Node **head_ref, int k)
```

```

{
    if (k == 0)
        return;

    // Let us understand the below code for example k = 4 and
    // list = 10->20->30->40->50->60.
    struct Node* current = *head_ref;

    // current will either point to kth or NULL after this loop.
    // current will point to node 40 in the above example
    int count = 1;
    while (count < k && current != NULL)
    {
        current = current->next;
        count++;
    }

    // If current is NULL, k is greater than or equal to count
    // of nodes in linked list. Don't change the list in this case
    if (current == NULL)
        return;

    // current points to kth node. Store it in a variable.
    // kthNode points to node 40 in the above example
    struct Node *kthNode = current;

    // current will point to last node after this loop
    // current will point to node 60 in the above example
    while (current->next != NULL)
        current = current->next;

    // Change next of last node to previous head
    // Next of 60 is now changed to node 10
    current->next = *head_ref;

    // Change head to (k+1)th node
    // head is now changed to node 50
    *head_ref = kthNode->next;

    // change next of kth node to NULL
    // next of 40 is now NULL
    kthNode->next = NULL;
}

/* UTILITY FUNCTIONS */
/* Function to push a node */
void push (struct Node** head_ref, int new_data)
{

```

```
/* allocate node */
struct Node* new_node =
    (struct Node*) malloc(sizeof(struct Node));

/* put in the data */
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;
}

/* Function to print linked list */
void printList(struct Node *node)
{
    while (node != NULL)
    {
        printf("%d ", node->data);
        node = node->next;
    }
}

/* Driver program to test above function*/
int main(void)
{
    /* Start with the empty list */
    struct Node* head = NULL;

    // create a list 10->20->30->40->50->60
    for (int i = 60; i > 0; i -= 10)
        push(&head, i);

    printf("Given linked list \n");
    printList(head);
    rotate(&head, 4);

    printf("\nRotated Linked list \n");
    printList(head);

    return (0);
}
```

### Java

```
// Java program to rotate a linked list
```

```
class LinkedList
{
    Node head; // head of list

    /* Linked list Node*/
    class Node
    {
        int data;
        Node next;
        Node(int d)
        {
            data = d;
            next = null;
        }
    }

    // This function rotates a linked list counter-clockwise
    // and updates the head. The function assumes that k is
    // smaller than size of linked list. It doesn't modify
    // the list if k is greater than or equal to size
    void rotate(int k)
    {
        if (k == 0) return;

        // Let us understand the below code for example k = 4
        // and list = 10->20->30->40->50->60.
        Node current = head;

        // current will either point to kth or NULL after this
        // loop. current will point to node 40 in the above example
        int count = 1;
        while (count < k && current != null)
        {
            current = current.next;
            count++;
        }

        // If current is NULL, k is greater than or equal to count
        // of nodes in linked list. Don't change the list in this case
        if (current == null)
            return;

        // current points to kth node. Store it in a variable.
        // kthNode points to node 40 in the above example
        Node kthNode = current;

        // current will point to last node after this loop
        // current will point to node 60 in the above example
```

```
while (current.next != null)
    current = current.next;

// Change next of last node to previous head
// Next of 60 is now changed to node 10

current.next = head;

// Change head to (k+1)th node
// head is now changed to node 50
head = kthNode.next;

// change next of kth node to null
kthNode.next = null;

}

/* Given a reference (pointer to pointer) to the head
   of a list and an int, push a new node on the front
   of the list. */
void push(int new_data)
{
    /* 1 & 2: Allocate the Node &
       Put in the data*/
    Node new_node = new Node(new_data);

    /* 3. Make next of new Node as head */
    new_node.next = head;

    /* 4. Move the head to point to new Node */
    head = new_node;
}

void printList()
{
    Node temp = head;
    while(temp != null)
    {
        System.out.print(temp.data+" ");
        temp = temp.next;
    }
    System.out.println();
}

/* Drier program to test above functions */
public static void main(String args[])
{
    LinkedList llist = new LinkedList();
```

```
// create a list 10->20->30->40->50->60
for (int i = 60; i >= 10; i -= 10)
    llist.push(i);

System.out.println("Given list");
llist.printList();

llist.rotate(4);

System.out.println("Rotated Linked List");
llist.printList();
}
} /* This code is contributed by Rajat Mishra */
```

### Python

```
# Python program to rotate a linked list

# Node class
class Node:

    # Constructor to initialize the node object
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:

    # Function to initialize head
    def __init__(self):
        self.head = None

    # Function to insert a new node at the beginning
    def push(self, new_data):
        # allocate node and put the data
        new_node = Node(new_data)

        # Make next of new node as head
        new_node.next = self.head

        # move the head to point to the new Node
        self.head = new_node

    # Utility function to print the linked LinkedList
    def printList(self):
        temp = self.head
        while(temp):
```

```
print temp.data,
temp = temp.next

# This function rotates a linked list counter-clockwise and
# updates the head. The function assumes that k is smaller
# than size of linked list. It doesn't modify the list if
# k is greater than or equal to size
def rotate(self, k):
    if k == 0:
        return

    # Let us understand the below code for example k = 4
    # and list = 10->20->30->40->50->60
    current = self.head

    # current will either point to kth or NULL after
    # this loop
    # current will point to node 40 in the above example
    count = 1
    while(count <k and current is not None):
        current = current.next
        count += 1

    # If current is None, k is greater than or equal
    # to count of nodes in linked list. Don't change
    # the list in this case
    if current is None:
        return

    # current points to kth node. Store it in a variable
    # kth node points to node 40 in the above example
    kthNode = current

    # current will point to lsat node after this loop
    # current will point to node 60 in above example
    while(current.next is not None):
        current = current.next

    # Change next of last node to previous head
    # Next of 60 is now changed to node 10
    current.next = self.head

    # Change head to (k+1)th node
    # head is not changed to node 50
    self.head = kthNode.next

    # change next of kth node to NULL
    # next of 40 is not NULL
```

```
kthNode.next = None

# Driver program to test above function
llist = LinkedList()

# Create a list 10->20->30->40->50->60
for i in range(60, 0, -10):
    llist.push(i)

print "Given linked list"
llist.printList()
llist.rotate(4)

print "\nRotated Linked list"
llist.printList()

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

Output:

```
Given linked list
10 20 30 40 50 60
Rotated Linked list
50 60 10 20 30 40
```

Time Complexity: O(n) where n is the number of nodes in Linked List. The code traverses the linked list only once.

## Source

<https://www.geeksforgeeks.org/rotate-a-linked-list/>

## Chapter 225

# Search an element in a Linked List (Iterative and Recursive)

Search an element in a Linked List (Iterative and Recursive) - GeeksforGeeks

Write a function that searches a given key ‘x’ in a given singly linked list. The function should return true if x is present in linked list and false otherwise.

```
bool search(Node *head, int x)
```

For example, if the key to be searched is 15 and linked list is 14->21->11->30->10, then function should return false. If key to be searched is 14, then the function should return true.

### Iterative Solution

- 2) Initialize a node pointer, current = head.
- 3) Do following while current is not NULL
  - a) current->key is equal to the key being searched return true.
  - b) current = current->next
- 4) Return false

Following is iterative implementation of above algorithm to search a given key.

C

```
// Iterative C program to search an element in linked list
#include<stdio.h>
#include<stdlib.h>
#include<stdbool.h>
```

```
/* Link list node */
struct Node
{
    int key;
    struct Node* next;
};

/* Given a reference (pointer to pointer) to the head
   of a list and an int, push a new node on the front
   of the list. */
void push(struct Node** head_ref, int new_key)
{
    /* allocate node */
    struct Node* new_node =
        (struct Node*) malloc(sizeof(struct Node));

    /* put in the key */
    new_node->key = new_key;

    /* 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;
}

/* Checks whether the value x is present in linked list */
bool search(struct Node* head, int x)
{
    struct Node* current = head; // Initialize current
    while (current != NULL)
    {
        if (current->key == x)
            return true;
        current = current->next;
    }
    return false;
}

/* Driver program to test count function*/
int main()
{
    /* Start with the empty list */
    struct Node* head = NULL;
    int x = 21;

    /* Use push() to construct below list
       10
       |
       20
       |
       30
       |
       40
       |
       50
       |
       60
       |
       70
       |
       80
       |
       90
       |
       100
```

```
14->21->11->30->10  */
push(&head, 10);
push(&head, 30);
push(&head, 11);
push(&head, 21);
push(&head, 14);

search(head, 21)? printf("Yes") : printf("No");
return 0;
}
```

**Java**

```
// Iterative Java program to search an element
// in linked list

//Node class
class Node
{
    int data;
    Node next;
    Node(int d)
    {
        data = d;
        next = null;
    }
}

//Linked list class
class LinkedList
{
    Node head;      //Head of list

    //Inserts a new node at the front of the list
    public void push(int new_data)
    {
        //Allocate new node and putting data
        Node new_node = new Node(new_data);

        //Make next of new node as head
        new_node.next = head;

        //Move the head to point to new Node
        head = new_node;
    }

    //Checks whether the value x is present in linked list
    public boolean search(Node head, int x)
```

```
{  
    Node current = head;    //Initialize current  
    while (current != null)  
    {  
        if (current.data == x)  
            return true;    //data found  
        current = current.next;  
    }  
    return false;    //data not found  
}  
  
//Driver function to test the above functions  
public static void main(String args[])  
{  
  
    //Start with the empty list  
    LinkedList llist = new LinkedList();  
  
    /*Use push() to construct below list  
    14->21->11->30->10 */  
    llist.push(10);  
    llist.push(30);  
    llist.push(11);  
    llist.push(21);  
    llist.push(14);  
  
    if (llist.search(llist.head, 21))  
        System.out.println("Yes");  
    else  
        System.out.println("No");  
}  
}  
// This code is contributed by Pratik Agarwal
```

### Python

```
# Iterative Python program to search an element  
# in linked list  
  
# Node class  
class Node:  
  
    # Function to initialise the node object  
    def __init__(self, data):  
        self.data = data # Assign data  
        self.next = None # Initialize next as null  
  
# Linked List class
```

```
class LinkedList:
    def __init__(self):
        self.head = None # Initialize head as None

    # This function insert a new node at the
    # beginning of the linked list
    def push(self, new_data):

        # Create a new Node
        new_node = Node(new_data)

        # 3. Make next of new Node as head
        new_node.next = self.head

        # 4. Move the head to point to new Node
        self.head = new_node

    # This Function checks whether the value
    # x present in the linked list
    def search(self, x):

        # Initialize current to head
        current = self.head

        # loop till current not equal to None
        while current != None:
            if current.data == x:
                return True # data found

            current = current.next

        return False # Data Not found

# Code execution starts here
if __name__ == '__main__':

    # Start with the empty list
    llist = LinkedList()

    ''' Use push() to construct below list
        14->21->11->30->10 '''
    llist.push(10);
    llist.push(30);
    llist.push(11);
    llist.push(21);
    llist.push(14);
```

```
if llist.search(21):
    print("Yes")
else:
    print("No")

# This code is contributed by Ravi Shankar
```

Output:

Yes

### Recursive Solution

```
bool search(head, x)
1) If head is NULL, return false.
2) If head's key is same as x, return true;
2) Else return search(head->next, x)
```

Following is recursive implementation of above algorithm to search a given key.

### C

```
// Recursive C program to search an element in linked list
#include<stdio.h>
#include<stdlib.h>
#include<stdbool.h>
/* Link list node */
struct Node
{
    int key;
    struct Node* next;
};

/* Given a reference (pointer to pointer) to the head
   of a list and an int, push a new node on the front
   of the list. */
void push(struct Node** head_ref, int new_key)
{
    /* allocate node */
    struct Node* new_node =
        (struct Node*) malloc(sizeof(struct Node));

    /* put in the key */
    new_node->key = new_key;
```

```
/* 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;
}

/* Checks whether the value x is present in linked list */
bool search(struct Node* head, int x)
{
    // Base case
    if (head == NULL)
        return false;

    // If key is present in current node, return true
    if (head->key == x)
        return true;

    // Recur for remaining list
    return search(head->next, x);
}

/* Driver program to test count function*/
int main()
{
    /* Start with the empty list */
    struct Node* head = NULL;
    int x = 21;

    /* Use push() to construct below list
     * 14->21->11->30->10  */
    push(&head, 10);
    push(&head, 30);
    push(&head, 11);
    push(&head, 21);
    push(&head, 14);

    search(head, 21)? printf("Yes") : printf("No");
    return 0;
}
```

### Java

```
// Recursive Java program to search an element
// in linked list

// Node class
```

```
class Node
{
    int data;
    Node next;
    Node(int d)
    {
        data = d;
        next = null;
    }
}

// Linked list class
class LinkedList
{
    Node head;      //Head of list

    //Inserts a new node at the front of the list
    public void push(int new_data)
    {
        //Allocate new node and putting data
        Node new_node = new Node(new_data);

        //Make next of new node as head
        new_node.next = head;

        //Move the head to point to new Node
        head = new_node;
    }

    // Checks whether the value x is present
    // in linked list
    public boolean search(Node head, int x)
    {
        // Base case
        if (head == null)
            return false;

        // If key is present in current node,
        // return true
        if (head.data == x)
            return true;

        // Recur for remaining list
        return search(head.next, x);
    }

    // Driver function to test the above functions
    public static void main(String args[])

```

```
{  
    // Start with the empty list  
    LinkedList llist = new LinkedList();  
  
    /* Use push() to construct below list  
       14->21->11->30->10 */  
    llist.push(10);  
    llist.push(30);  
    llist.push(11);  
    llist.push(21);  
    llist.push(14);  
  
    if (llist.search(llist.head, 21))  
        System.out.println("Yes");  
    else  
        System.out.println("No");  
}  
}  
// This code is contributed by Pratik Agarwal
```

Output:

Yes

### Python

```
# Recursive Python program to  
# search an element in linked list  
  
# Node class  
class Node:  
  
    # Function to initialise  
    # the node object  
    def __init__(self, data):  
        self.data = data # Assign data  
        self.next = None # Initialize next as null  
  
class LinkedList:  
  
    def __init__(self):  
        self.head = None # Initialize head as None  
  
    # This function insert a new node at  
    # the beginning of the linked list  
    def push(self, new_data):
```

```
# Create a new Node
new_node = Node(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

# Checks whether the value key
# is present in linked list
def search(self, li, key):

    # Base case
    if(not li):
        return False

    # If key is present in
    # current node, return true
    if(li.data == key):
        return True

    # Recur for remaining list
    return self.search(li.next, key)

# Driver Code
if __name__=='__main__':
    li = LinkedList()

    li.push(1)
    li.push(2)
    li.push(3)
    li.push(4)

    key = 4

    if li.search(li.head,key):
        print("Yes")
    else:
        print("No")

# This code is contributed
# by Manoj Sharma
```

**Output:**

**Yes**

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

**Improved By :** [CodeDaemon](#)

## **Source**

<https://www.geeksforgeeks.org/search-an-element-in-a-linked-list-iterative-and-recursive/>

## Chapter 226

# Segregate even and odd nodes in a Linked List

Segregate even and odd nodes in a Linked List - GeeksforGeeks

Given a Linked List of integers, write a function to modify the linked list such that all even numbers appear before all the odd numbers in the modified linked list. Also, keep the order of even and odd numbers same.

Examples:

Input: 17->15->8->12->10->5->4->1->7->6->NULL  
Output: 8->12->10->4->6->17->15->5->1->7->NULL

Input: 8->12->10->5->4->1->6->NULL  
Output: 8->12->10->4->6->5->1->NULL

// If all numbers are even then do not change the list  
Input: 8->12->10->NULL  
Output: 8->12->10->NULL

// If all numbers are odd then do not change the list  
Input: 1->3->5->7->NULL  
Output: 1->3->5->7->NULL

### Method 1

The idea is to get pointer to the last node of list. And then traverse the list starting from the head node and move the odd valued nodes from their current position to end of the list.

Thanks to blunderboy for suggesting this method.

Algorithm:

- ...1) Get pointer to the last node.

- ...2) Move all the odd nodes to the end.  
.....a) Consider all odd nodes before the first even node and move them to end.  
.....b) Change the head pointer to point to the first even node.  
.....b) Consider all odd nodes after the first even node and move them to the end.

**C/C++**

```
// C/C++ program to segregate even and odd nodes in a
// Linked List
#include <stdio.h>
#include <stdlib.h>

/* a node of the singly linked list */
struct Node
{
    int data;
    struct Node *next;
};

void segregateEvenOdd(struct Node **head_ref)
{
    struct Node *end = *head_ref;
    struct Node *prev = NULL;
    struct Node *curr = *head_ref;

    /* Get pointer to the last node */
    while (end->next != NULL)
        end = end->next;

    struct Node *new_end = end;

    /* Consider all odd nodes before the first even node
       and move them after end */
    while (curr->data %2 != 0 && curr != end)
    {
        new_end->next = curr;
        curr = curr->next;
        new_end->next->next = NULL;
        new_end = new_end->next;
    }

    // 10->8->17->17->15
    /* Do following steps only if there is any even node */
    if (curr->data%2 == 0)
    {
        /* Change the head pointer to point to first even node */
        *head_ref = curr;

        /* now current points to the first even node */
    }
}
```

```
while (curr != end)
{
    if ( (curr->data)%2 == 0 )
    {
        prev = curr;
        curr = curr->next;
    }
    else
    {
        /* break the link between prev and current */
        prev->next = curr->next;

        /* Make next of curr as NULL */
        curr->next = NULL;

        /* Move curr to end */
        new_end->next = curr;

        /* make curr as new end of list */
        new_end = curr;

        /* Update current pointer to next of the moved node */
        curr = prev->next;
    }
}
}

/* We must have prev set before executing lines following this
   statement */
else prev = curr;

/* If there are more than 1 odd nodes and end of original list is
   odd then move this node to end to maintain same order of odd
   numbers in modified list */
if (new_end!=end && (end->data)%2 != 0)
{
    prev->next = end->next;
    end->next = NULL;
    new_end->next = end;
}
return;
}

/* UTILITY FUNCTIONS */
/* Function to insert a node at the beginning */
void push(struct Node** head_ref, int new_data)
{
    /* allocate node */

```

```
struct Node* new_node =
    (struct Node*) malloc(sizeof(struct Node));

/* put in the data */
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;
}

/* Function to print nodes in a given linked list */
void printList(struct Node *node)
{
    while (node!=NULL)
    {
        printf("%d ", node->data);
        node = node->next;
    }
}

/* Drier program to test above functions*/
int main()
{
    /* Start with the empty list */
    struct Node* head = NULL;

    /* Let us create a sample linked list as following
     * 0->2->4->6->8->10->11 */

    push(&head, 11);
    push(&head, 10);
    push(&head, 8);
    push(&head, 6);
    push(&head, 4);
    push(&head, 2);
    push(&head, 0);

    printf("\nOriginal Linked list \n");
    printList(head);

    segregateEvenOdd(&head);

    printf("\nModified Linked list \n");
    printList(head);
```

```
    return 0;
}
```

**Java**

```
// Java program to segregate even and odd nodes in a
// Linked List
class LinkedList
{
    Node head; // head of list

    /* Linked list Node*/
    class Node
    {
        int data;
        Node next;
        Node(int d)
        {
            data = d;
            next = null;
        }
    }

    void segregateEvenOdd()
    {
        Node end = head;
        Node prev = null;
        Node curr = head;

        /* Get pointer to last Node */
        while (end.next != null)
            end = end.next;

        Node new_end = end;

        // Consider all odd nodes before getting first eve node
        while (curr.data %2 !=0 && curr != end)
        {
            new_end.next = curr;
            curr = curr.next;
            new_end.next.next = null;
            new_end = new_end.next;
        }

        // do following steps only if there is an even node
        if (curr.data %2 ==0)
        {
            head = curr;
```

```
// now curr points to first even node
while (curr != end)
{
    if (curr.data % 2 == 0)
    {
        prev = curr;
        curr = curr.next;
    }
    else
    {
        /* Break the link between prev and curr*/
        prev.next = curr.next;

        /* Make next of curr as null */
        curr.next = null;

        /*Move curr to end */
        new_end.next = curr;

        /*Make curr as new end of list */
        new_end = curr;

        /*Update curr pointer */
        curr = prev.next;
    }
}
}

/* We have to set prev before executing rest of this code */
else prev = curr;

if (new_end != end && end.data %2 != 0)
{
    prev.next = end.next;
    end.next = null;
    new_end.next = end;
}
}

/* Given a reference (pointer to pointer) to the head
   of a list and an int, push a new node on the front
   of the list. */
void push(int new_data)
{
    /* 1 & 2: Allocate the Node &
       Put in the data*/
    Node new_node = new Node(new_data);
```

```
/* 3. Make next of new Node as head */
new_node.next = head;

/* 4. Move the head to point to new Node */
head = new_node;
}

// Utility function to print a linked list
void printList()
{
    Node temp = head;
    while(temp != null)
    {
        System.out.print(temp.data+" ");
        temp = temp.next;
    }
    System.out.println();
}

/* Drier program to test above functions */
public static void main(String args[])
{
    LinkedList llist = new LinkedList();
    llist.push(11);
    llist.push(10);
    llist.push(8);
    llist.push(6);
    llist.push(4);
    llist.push(2);
    llist.push(0);
    System.out.println("Original Linked List");
    llist.printList();

    llist.segregateEvenOdd();

    System.out.println("Modified Linked List");
    llist.printList();
}
} /* This code is contributed by Rajat Mishra */
```

Output:

```
Original Linked list 0 2 4 6 8 10 11
Modified Linked list 0 2 4 6 8 10 11
```

Time complexity: O(n)

### Method 2

The idea is to split the linked list into two: one containing all even nodes and other containing all odd nodes. And finally attach the odd node linked list after the even node linked list. To split the Linked List, traverse the original Linked List and move all odd nodes to a separate Linked List of all odd nodes. At the end of loop, the original list will have all the even nodes and the odd node list will have all the odd nodes. To keep the ordering of all nodes same, we must insert all the odd nodes at the end of the odd node list. And to do that in constant time, we must keep track of last pointer in the odd node list.

C++

```
// CPP program to segregate even and odd nodes in a
// Linked List
#include <stdio.h>
#include <stdlib.h>

/* a node of the singly linked list */
struct Node
{
    int data;
    struct Node *next;
};

// Function to segregate even and odd nodes.
void segregateEvenOdd(struct Node **head_ref)
{
    // Starting node of list having
    // even values.
    Node *evenStart = NULL;

    // Ending node of even values list.
    Node *evenEnd = NULL;

    // Starting node of odd values list.
    Node *oddStart = NULL;

    // Ending node of odd values list.
    Node *oddEnd = NULL;

    // Node to traverse the list.
    Node *currNode = *head_ref;

    while(currNode != NULL){
        int val = currNode -> data;

        // If current value is even, add
        // it to even values list.

```

```
if(val % 2 == 0) {
    if(evenStart == NULL){
        evenStart = currNode;
        evenEnd = evenStart;
    }

    else{
        evenEnd -> next = currNode;
        evenEnd = evenEnd -> next;
    }
}

// If current value is odd, add
// it to odd values list.
else{
    if(oddStart == NULL){
        oddStart = currNode;
        oddEnd = oddStart;
    }
    else{
        oddEnd -> next = currNode;
        oddEnd = oddEnd -> next;
    }
}

// Move head pointer one step in
// forward direction
currNode = currNode -> next;
}

// If either odd list or even list is empty,
// no change is required as all elements
// are either even or odd.
if(oddStart == NULL || evenStart == NULL){
    return;
}

// Add odd list after even list.
evenEnd -> next = oddStart;
oddEnd -> next = NULL;

// Modify head pointer to
// starting of even list.
*head_ref = evenStart;
}

/* UTILITY FUNCTIONS */
/* Function to insert a node at the beginning */
```

```
void push(struct Node** head_ref, int new_data)
{
    /* allocate node */
    struct Node* new_node =
        (struct Node*) malloc(sizeof(struct Node));

    /* put in the data */
    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;
}

/* Function to print nodes in a given linked list */
void printList(struct Node *node)
{
    while (node!=NULL)
    {
        printf("%d ", node->data);
        node = node->next;
    }
}

/* Drier program to test above functions*/
int main()
{
    /* Start with the empty list */
    struct Node* head = NULL;

    /* Let us create a sample linked list as following
     * 0->1->4->6->9->10->11 */

    push(&head, 11);
    push(&head, 10);
    push(&head, 9);
    push(&head, 6);
    push(&head, 4);
    push(&head, 1);
    push(&head, 0);

    printf("\nOriginal Linked list \n");
    printList(head);

    segregateEvenOdd(&head);
```

```
    printf("\nModified Linked list \n");
    printList(head);

    return 0;
}

// This code is contributed by NIKHIL JINDAL.
```

**Java**

```
// Java program to segregate even and odd nodes in a
// Linked List
import java.io.*;

class LinkedList {

    Node head; // head of list

    /* Linked list Node*/
    class Node
    {
        int data;
        Node next;
        Node(int d)
        {
            data = d;
            next = null;
        }
    }

    public void segregateEvenOdd() {

        Node evenStart = null;
        Node evenEnd = null;
        Node oddStart = null;
        Node oddEnd = null;
        Node currentNode = head;

        while(currentNode != null) {
            int element = currentNode.data;

            if(element % 2 == 0) {

                if(evenStart == null) {
                    evenStart = currentNode;
                    evenEnd = evenStart;
                } else {
                    evenEnd.next = currentNode;
                    evenEnd = evenStart;
                }
            } else {
                if(oddStart == null) {
                    oddStart = currentNode;
                    oddEnd = oddStart;
                } else {
                    oddEnd.next = currentNode;
                    oddEnd = oddStart;
                }
            }
            currentNode = currentNode.next;
        }
    }
}
```

```
        evenEnd = evenEnd.next;
    }

} else {

    if(oddStart == null) {
        oddStart = currentNode;
        oddEnd = oddStart;
    } else {
        oddEnd.next = currentNode;
        oddEnd = oddEnd.next;
    }
}

// Move head pointer one step in forward direction
currentNode = currentNode.next;
}

if(oddStart == null || evenStart == null) {
    return;
}

evenEnd.next = oddStart;
oddEnd.next = null;
head=evenStart;
}

/* Given a reference (pointer to pointer) to the head
   of a list and an int, push a new node on the front
   of the list. */
void push(int new_data)
{
    /* 1 & 2: Allocate the Node &
       Put in the data*/
    Node new_node = new Node(new_data);

    /* 3. Make next of new Node as head */
    new_node.next = head;

    /* 4. Move the head to point to new Node */
    head = new_node;
}

// Utility function to print a linked list
void printList()
{
    Node temp = head;
    while(temp != null)
```

```
{  
    System.out.print(temp.data+" ");  
    temp = temp.next;  
}  
System.out.println();  
}  
  
/* Drier program to test above functions */  
public static void main(String args[])  
{  
    LinkedList llist = new LinkedList();  
    llist.push(11);  
    llist.push(10);  
    llist.push(9);  
    llist.push(6);  
    llist.push(4);  
    llist.push(1);  
    llist.push(0);  
    System.out.println("Origional Linked List");  
    llist.printList();  
  
    llist.segregateEvenOdd();  
  
    System.out.println("Modified Linked List");  
    llist.printList();  
}  
}
```

Output:

```
Origional Linked List  
0 1 4 6 9 10 11  
Modified Linked List  
0 4 6 10 1 9 11
```

Time complexity: O(n)

Improved By : [nik1996](#)

## Source

<https://www.geeksforgeeks.org/segregate-even-and-odd-elements-in-a-linked-list/>

## Chapter 227

# Segregate even and odd nodes in a Linked List using Deque

Segregate even and odd nodes in a Linked List using Deque - GeeksforGeeks

Given a Linked List of integers. The task is to write a program to modify the linked list such that all even numbers appear before all the odd numbers in the modified linked list. It is not needed to keep the order of even and odd nodes same as that of the original list, the task is just to rearrange the nodes such that all even valued nodes appear before the odd valued nodes.

**See Also:** [Segregate even and odd nodes in a Linked List](#)

**Examples:**

**Input:** 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> 8 -> 9 -> 10 -> NULL

**Output:** 10 -> 8 -> 6 -> 4 -> 2 -> 1 -> 3 -> 5 -> 7 -> 9 -> NULL

**Input:** 4 -> 3 -> 2 -> 1 -> NULL

**Output:** 2 -> 4 -> 3 -> 1 -> NULL

The idea is to iteratively push all the elements of the linked list to deque as per the below conditions:

- Start traversing the linked list and if an element is even then push it to the front of the Deque and,
- If the element is odd then push it to the back of the Deque.

Finally, replace all elements of the linked list with the elements of Deque starting from the first element.

Below is the implementation of the above approach:

```
// CPP program to segregate even and
// odd noeds in a linked list using deque
#include <bits/stdc++.h>
using namespace std;

/* Link list node */
struct Node {
    int data;
    struct Node* next;
};

/*UTILITY FUNCTIONS*/
/* Push a node to linked list. Note that this function
changes the head */
void push(struct Node** head_ref, char new_data)
{
    /* allocate node */
    struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));

    /* put in the data */
    new_node->data = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to pochar to the new node */
    (*head_ref) = new_node;
}

// printing the linked list
void printList(struct Node* head)
{
    struct Node* temp = head;
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
}

// Function to rearrange even and odd
// elements in a linked list using deque
void evenOdd(struct Node* head)
{
    struct Node* temp = head;

    // Declaring a Deque
    deque<int> d;
```

```
// Push all the elements of
// linked list in to deque
while (temp != NULL) {

    // if element is even push it
    // to front of the deque
    if (temp->data % 2 == 0)
        d.push_front(temp->data);

    else // else push at the back of the deque
        d.push_back(temp->data);
    temp = temp->next; // increase temp
}

temp = head;

// Replace all elements of the linked list
// with the elements of Deque starting from
// the first element
while (!d.empty()) {
    temp->data = d.front();
    d.pop_front();
    temp = temp->next;
}
}

// Driver code
int main()
{
    struct Node* head = NULL;
    push(&head, 10);
    push(&head, 9);
    push(&head, 8);
    push(&head, 7);
    push(&head, 6);
    push(&head, 5);
    push(&head, 4);
    push(&head, 3);
    push(&head, 2);
    push(&head, 1);

    cout << "Given linked list: ";
    printList(head);

    evenOdd(head);

    cout << "\nAfter rearrangement: ";
    printList(head);
```

```
    return 0;  
}
```

**Output:**

```
Given linked list: 1 2 3 4 5 6 7 8 9 10  
After rearrangement: 10 8 6 4 2 1 3 5 7 9
```

**Time complexity:** O(N)

**Auxiliary Space:** O(N), where N is the total number of nodes in the linked list.

**Source**

<https://www.geeksforgeeks.org/segregate-even-and-odd-nodes-in-a-linked-list-using-deque/>

## Chapter 228

# Select a Random Node from a Singly Linked List

Select a Random Node from a Singly Linked List - GeeksforGeeks

Given a singly linked list, select a random node from linked list (the probability of picking a node should be  $1/N$  if there are  $N$  nodes in list). You are given a random number generator.

Below is a Simple Solution

- 1) Count number of nodes by traversing the list.
- 2) Traverse the list again and select every node with probability  $1/N$ . The selection can be done by generating a random number from 0 to  $N-1$  for  $i$ 'th node, and selecting the  $i$ 'th node only if generated number is equal to 0 (or any other fixed number from 0 to  $N-1$ ).

We get uniform probabilities with above schemes.

```
i = 1, probability of selecting first node = 1/N
i = 2, probability of selecting second node =
    [probability that first node is not selected] *
    [probability that second node is selected]
    = ((N-1)/N)* 1/(N-1)
    = 1/N
```

Similarly, probabilities of other selecting other nodes is  $1/N$

The above solution requires two traversals of linked list.

**How to select a random node with only one traversal allowed?**

The idea is to use [Reservoir Sampling](#). Following are the steps. This is a simpler version of [Reservoir Sampling](#) as we need to select only one key instead of  $k$  keys.

- (1) Initialize result as first node

```
result = head->key
(2) Initialize n = 2
(3) Now one by one consider all nodes from 2nd node onward.
    (3.a) Generate a random number from 0 to n-1.
          Let the generated random number is j.
    (3.b) If j is equal to 0 (we could choose other fixed number
          between 0 to n-1), then replace result with current node.
    (3.c) n = n+1
    (3.d) current = current->next
```

Below is the implementation of above algorithm.

C

```
/* C program to randomly select a node from a singly
   linked list */
#include<stdio.h>
#include<stdlib.h>
#include <time.h>

/* Link list node */
struct Node
{
    int key;
    struct Node* next;
};

// A reservoir sampling based function to print a
// random node from a linked list
void printRandom(struct Node *head)
{
    // IF list is empty
    if (head == NULL)
        return;

    // Use a different seed value so that we don't get
    // same result each time we run this program
    srand(time(NULL));

    // Initialize result as first node
    int result = head->key;

    // Iterate from the (k+1)th element to nth element
    struct Node *current = head;
    int n;
    for (n=2; current!=NULL; n++)
    {
```

```
// change result with probability 1/n
if (rand() % n == 0)
    result = current->key;

// Move to next node
current = current->next;
}

printf("Randomly selected key is %d\n", result);
}

/* BELOW FUNCTIONS ARE JUST UTILITY TO TEST */

/* A utility function to create a new node */
struct Node *newNode(int new_key)
{
    /* allocate node */
    struct Node* new_node =
        (struct Node*) malloc(sizeof(struct Node));

    /* put in the key */
    new_node->key = new_key;
    new_node->next = NULL;

    return new_node;
}

/* A utility function to insert a node at the beginning
   of linked list */
void push(struct Node** head_ref, int new_key)
{
    /* allocate node */
    struct Node* new_node = new Node;

    /* put in the key */
    new_node->key = new_key;

    /* 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;
}

// Driver program to test above functions
int main()
```

```
{  
    struct Node *head = NULL;  
    push(&head, 5);  
    push(&head, 20);  
    push(&head, 4);  
    push(&head, 3);  
    push(&head, 30);  
  
    printRandom(head);  
  
    return 0;  
}
```

**Java**

```
// Java program to select a random node from singly linked list  
  
import java.util.*;  
  
// Linked List Class  
class LinkedList {  
  
    static Node head; // head of list  
  
    /* Node Class */  
    static class Node {  
  
        int data;  
        Node next;  
  
        // Constructor to create a new node  
        Node(int d) {  
            data = d;  
            next = null;  
        }  
    }  
  
    // A reservoir sampling based function to print a  
    // random node from a linked list  
    void printrandom(Node node) {  
  
        // If list is empty  
        if (node == null) {  
            return;  
        }  
  
        // Use a different seed value so that we don't get  
        // same result each time we run this program
```

```
Math.abs(UUID.randomUUID().getMostSignificantBits());  
  
// Initialize result as first node  
int result = node.data;  
  
// Iterate from the (k+1)th element to nth element  
Node current = node;  
int n;  
for (n = 2; current != null; n++) {  
  
    // change result with probability 1/n  
    if (Math.random() % n == 0) {  
        result = current.data;  
    }  
  
    // Move to next node  
    current = current.next;  
}  
  
System.out.println("Randomly selected key is " + result);  
}  
  
// Driver program to test above functions  
public static void main(String[] args) {  
  
    LinkedList list = new LinkedList();  
    list.head = new Node(5);  
    list.head.next = new Node(20);  
    list.head.next.next = new Node(4);  
    list.head.next.next.next = new Node(3);  
    list.head.next.next.next.next = new Node(30);  
  
    list.printrandom(head);  
}  
}  
  
// This code has been contributed by Mayank Jaiswal
```

### Python

```
# Python program to randomly select a node from singly  
# linked list  
  
import random  
  
# Node class  
class Node:
```

```
# Constructor to initialize the node object
def __init__(self, data):
    self.data= data
    self.next = None

class LinkedList:

    # Function to initialize head
    def __init__(self):
        self.head = None

    # A reservoir sampling based function to print a
    # random node from a linked list
    def printRandom(self):

        # If list is empty
        if self.head is None:
            return

        # Use a different seed value so that we don't get
        # same result each time we run this program
        random.seed()

        # Initialize result as first node
        result = self.head.data

        # Iterate from the (k+1)th element nth element
        current = self.head
        n = 2
        while(current is not None):

            # change result with probability 1/n
            if (random.randrange(n) == 0 ):
                result = current.data

            # Move to next node
            current = current.next
            n += 1

        print "Randomly selected key is %d" %(result)

    # Function to insert a new node at the beginning
    def push(self, new_data):
        new_node = Node(new_data)
        new_node.next = self.head
        self.head = new_node
```

```
# Utility function to print the linked LinkedList
def printList(self):
    temp = self.head
    while(temp):
        print temp.data,
        temp = temp.next

# Driver program to test above function
llist = LinkedList()
llist.push(5)
llist.push(20)
llist.push(4)
llist.push(3)
llist.push(30)
llist.printRandom()

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

Note that the above program is based on outcome of a random function and may produce different output.

#### How does this work?

Let there be total N nodes in list. It is easier to understand from last node.

The probability that last node is result simply  $1/N$  [For last or  $N^{\text{th}}$  node, we generate a random number between 0 to  $N-1$  and make last node as result if the generated number is 0 (or any other fixed number)]

The probability that second last node is result should also be  $1/N$ .

The probability that the second last node is result  
= [Probability that the second last node replaces result] X  
[Probability that the last node doesn't replace the result]  
=  $[1 / (N-1)] * [(N-1)/N]$   
=  $1/N$

Similarly we can show probability for 3<sup>rd</sup> last node and other nodes.

This article is contributed by **Rajeev**. 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/select-a-random-node-from-a-singly-linked-list/>

## Chapter 229

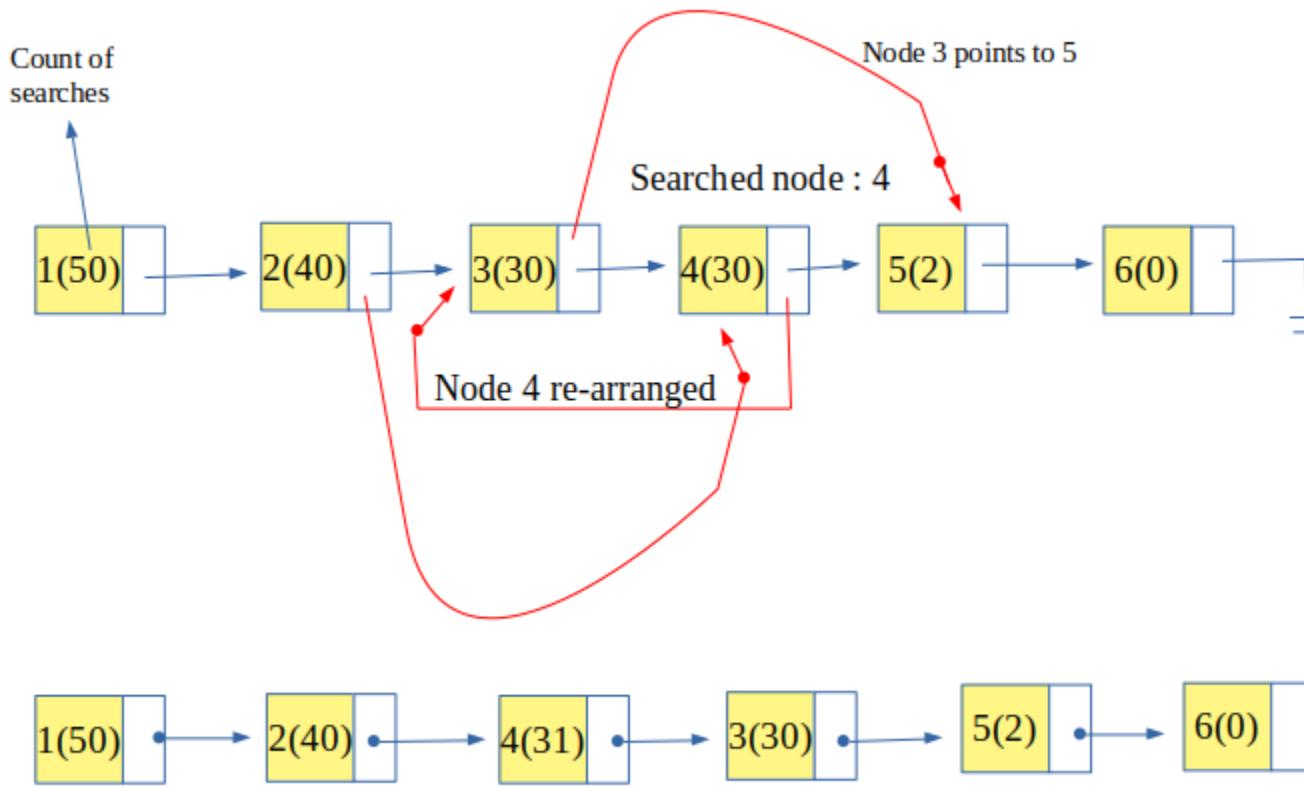
# Self Organizing List : Count Method

Self Organizing List : Count Method - GeeksforGeeks

[Self Organizing list](#) is a list that re-organizes or re-arranges itself for better performance. In a simple list, an item to be searched is looked for in a sequential manner which gives the time complexity of  $O(n)$ . But in real scenario not all the items are searched frequently and most of the time only few items are searched multiple times.

So, a self organizing list uses this property (also known as locality of reference) that brings the most frequent used items at the head of the list. This increases the probability of finding the item at the start of the list and those elements which are rarely used are pushed to the back of the list.

In **Count Method**, the number of time each node is searched for is counted (i.e. the frequency of search is maintained). So an extra storage is associated with each node that is incremented every time a node is searched. And then the nodes are arranged in non-increasing order of count or frequency of its searches. So this ensures that the most frequently accessed node is kept at the head of the list.



Examples:

```
Input : list : 1, 2, 3, 4, 5
       searched : 4
Output : list : 4, 1, 2, 3, 5
```

```
Input : list : 4, 1, 2, 3, 5
       searched : 5
       searched : 5
       searched : 2
Output : list : 5, 2, 4, 1, 3
Explanation : 5 is searched 2 times (i.e. the
most searched) 2 is searched 1 time and 4 is
also searched 1 time (but since 2 is searched
recently, it is kept ahead of 4) rest are not
searched, so they maintained order in which
they were inserted.
```

```
// CPP Program to implement self-organizing list
// using count method
#include <iostream>
using namespace std;

// structure for self organizing list
struct self_list {
    int value;
    int count;
    struct self_list* next;
};

// head and rear pointing to start and end of list resp.
self_list *head = NULL, *rear = NULL;

// function to insert an element
void insert_self_list(int number)
{
    // creating a node
    self_list* temp = (self_list*)malloc(sizeof(self_list));

    // assigning value to the created node;
    temp->value = number;
    temp->count = 0;
    temp->next = NULL;

    // first element of list
    if (head == NULL)
        head = rear = temp;

    // rest elements of list
    else {
        rear->next = temp;
        rear = temp;
    }
}

// function to search the key in list
// and re-arrange self-organizing list
bool search_self_list(int key)
{
    // pointer to current node
    self_list* current = head;

    // pointer to previous node
    self_list* prev = NULL;

    // searching for the key
```

```
while (current != NULL) {

    // if key is found
    if (current->value == key) {

        // increment the count of node
        current->count = current->count + 1;

        // if it is not the first element
        if (current != head) {
            self_list* temp = head;
            self_list* temp_prev = NULL;

            // finding the place to arrange the searched node
            while (current->count < temp->count) {
                temp_prev = temp;
                temp = temp->next;
            }

            // if the place is other than its own place
            if (current != temp) {
                prev->next = current->next;
                current->next = temp;

                // if it is to be placed at beginning
                if (temp == head)
                    head = current;
                else
                    temp_prev->next = current;
            }
        }
        return true;
    }
    prev = current;
    current = current->next;
}
return false;
}

// function to display the list
void display()
{
    if (head == NULL) {
        cout << "List is empty" << endl;
        return;
    }

    // temporary pointer pointing to head
```

```
self_list* temp = head;
cout << "List: ";

// sequentially displaying nodes
while (temp != NULL) {
    cout << temp->value << "(" << temp->count << ")";
    if (temp->next != NULL)
        cout << " --> ";
    // incrementing node pointer.
    temp = temp->next;
}
cout << endl
    << endl;
}

// Driver Code
int main()
{
    /* inserting five values */
    insert_self_list(1);
    insert_self_list(2);
    insert_self_list(3);
    insert_self_list(4);
    insert_self_list(5);

    // Display the list
    display();

    search_self_list(4);
    search_self_list(2);
    display();

    search_self_list(4);
    search_self_list(4);
    search_self_list(5);
    display();

    search_self_list(5);
    search_self_list(2);
    search_self_list(2);
    search_self_list(2);
    display();
    return 0;
}
```

Output:

List: 1(0) --> 2(0) --> 3(0) --> 4(0) --> 5(0)

List: 2(1) --> 4(1) --> 1(0) --> 3(0) --> 5(0)

List: 4(3) --> 5(1) --> 2(1) --> 1(0) --> 3(0)

List: 2(4) --> 4(3) --> 5(2) --> 1(0) --> 3(0)

## Source

<https://www.geeksforgeeks.org/self-organizing-list-count-method/>

## Chapter 230

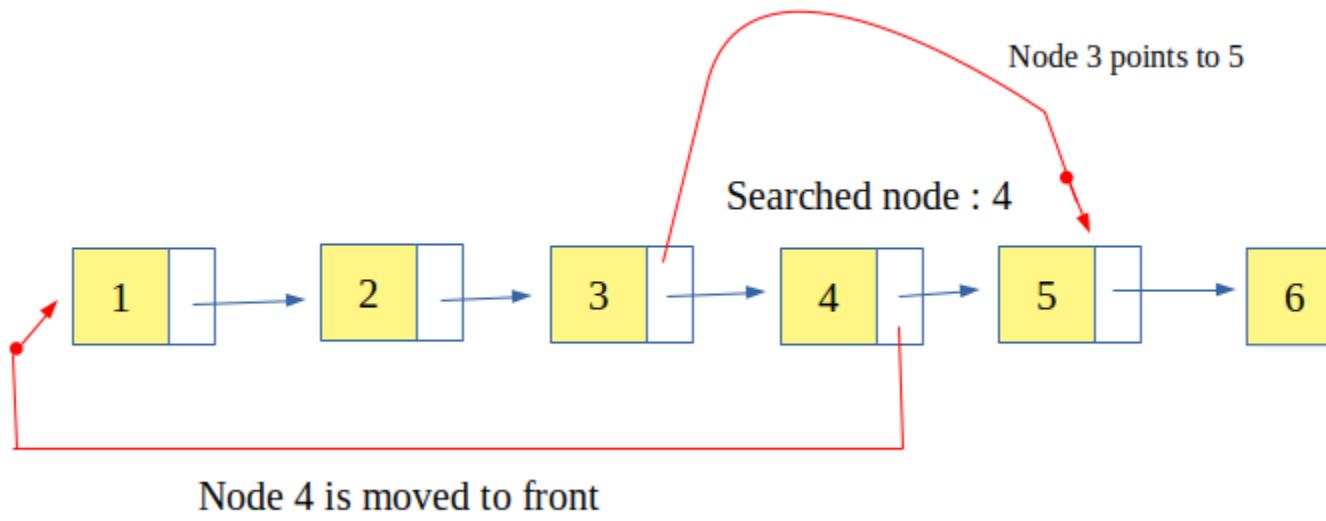
# Self Organizing List : Move to Front Method

Self Organizing List : Move to Front Method - GeeksforGeeks

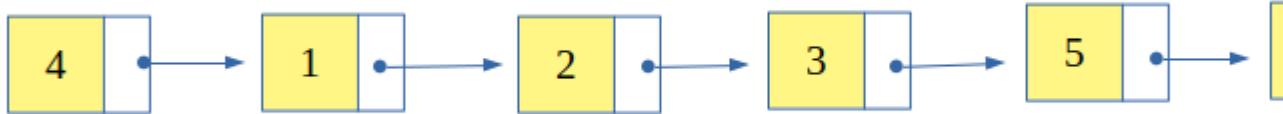
[Self Organizing list](#) is a list that re-organizes or re-arranges itself for better performance. In a simple list, an item to be searched is looked for in a sequential manner which gives the time complexity of  $O(n)$ . But in real scenario not all the items are searched frequently and most of the time only few items are searched multiple times.

So, a self organizing list uses this property (also known as **locality of reference**) that brings the most frequent used items at the head of the list. This increases the probability of finding the item at the start of the list and those elements which are rarely used are pushed to the back of the list.

In **Move to Front Method**, the recently searched item is moved to the front of the list. So, this method is quite easy to implement but it also moves in-frequent searched items to front. This moving of in-frequent searched items to the front is a big disadvantage of this method because it affects the access time.



Node 4 is moved to front



Examples:

Input : list : 1, 2, 3, 4, 5, 6  
searched: 4  
Output : list : 4, 1, 2, 3, 5, 6

Input : list : 4, 1, 2, 3, 5, 6  
searched : 2  
Output : list : 2, 4, 1, 3, 5, 6

```
// CPP Program to implement self-organizing list
// using move to front method
#include <iostream>
using namespace std;

// structure for self organizing list
```

```
struct self_list {
    int value;
    struct self_list* next;
};

// head and rear pointing to start and end of list resp.
self_list *head = NULL, *rear = NULL;

// function to insert an element
void insert_self_list(int number)
{
    // creating a node
    self_list* temp = (self_list*)malloc(sizeof(self_list));

    // assigning value to the created node;
    temp->value = number;
    temp->next = NULL;

    // first element of list
    if (head == NULL)
        head = rear = temp;

    // rest elements of list
    else {
        rear->next = temp;
        rear = temp;
    }
}

// function to search the key in list
// and re-arrange self-organizing list
bool search_self_list(int key)
{
    // pointer to current node
    self_list* current = head;

    // pointer to previous node
    self_list* prev = NULL;

    // searching for the key
    while (current != NULL) {

        // if key found
        if (current->value == key) {

            // if key is not the first element
            if (prev != NULL) {

                // move current node to front
                current->next = prev->next;
                prev->next = current;
            }
        }
    }
}
```

```
/* re-arranging the elements */
prev->next = current->next;
current->next = head;
head = current;
}
return true;
}
prev = current;
current = current->next;
}

// key not found
return false;
}

// function to display the list
void display()
{
    if (head == NULL) {
        cout << "List is empty" << endl;
        return;
    }

    // temporary pointer pointing to head
    self_list* temp = head;
    cout << "List: ";

    // sequentially displaying nodes
    while (temp != NULL) {
        cout << temp->value;
        if (temp->next != NULL)
            cout << " --> ";

        // incrementing node pointer.
        temp = temp->next;
    }
    cout << endl << endl;
}

// Driver Code
int main()
{
    /* inserting five values */
    insert_self_list(1);
    insert_self_list(2);
    insert_self_list(3);
    insert_self_list(4);
    insert_self_list(5);
```

```
// Display the list
display();

// search 4 and if found then re-arrange
if (search_self_list(4))
    cout << "Searched: 4" << endl;
else
    cout << "Not Found: 4" << endl;

// Display the list
display();

// search 2 and if found then re-arrange
if (search_self_list(2))
    cout << "Searched: 2" << endl;
else
    cout << "Not Found: 2" << endl;
display();

return 0;
}
```

Output:

List: 1 --> 2 --> 3 --> 4 --> 5

Searched: 4

List: 4 --> 1 --> 2 --> 3 --> 5

Searched: 2

List: 2 --> 4 --> 1 --> 3 --> 5

## Source

<https://www.geeksforgeeks.org/self-organizing-list-move-front-method/>

# Chapter 231

## Skip List | Set 2 (Insertion)

Skip List | Set 2 (Insertion) - GeeksforGeeks

We have already discussed the idea of Skip list and how they work in [Skip List | Set 1 \(Introduction\)](#). In this article, we will be discussing how to insert an element in Skip list.

### Deciding nodes level

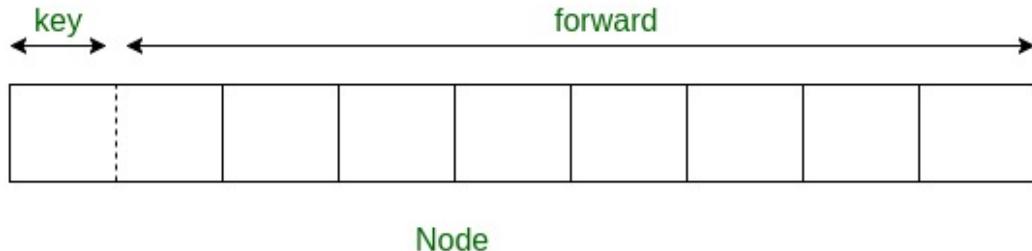
Each element in the list is represented by a node, the level of the node is chosen randomly while insertion in the list. **Level does not depend on the number of elements in the node.** The level for node is decided by the following algorithm –

```
randomLevel()
lvl := 1
//random() that returns a random value in [0...1)
while random() < p and lvl < MaxLevel do
    lvl := lvl + 1
return lvl
```

**MaxLevel** is the upper bound on number of levels in the skip list. It can be determined as  $\sum_{i=1}^{MaxLevel} p^i = \frac{1-p^{MaxLevel}}{1-p}$ . Above algorithm assure that random level will never be greater than MaxLevel. Here **p** is the fraction of the nodes with level **i** pointers also having level **i+1** pointers and **N** is the number of nodes in the list.

### Node Structure

Each node carries a key and a **forward** array carrying pointers to nodes of a different level. A level **i** node carries **i** forward pointers indexed through 0 to **i**.



### Insertion in Skip List

We will start from highest level in the list and compare key of next node of the current node with the key to be inserted. Basic idea is If –

1. Key of next node is less than key to be inserted then we keep on moving forward on the same level
2. Key of next node is greater than the key to be inserted then we store the pointer to current node **i** at **update[i]** and move one level down and continue our search.

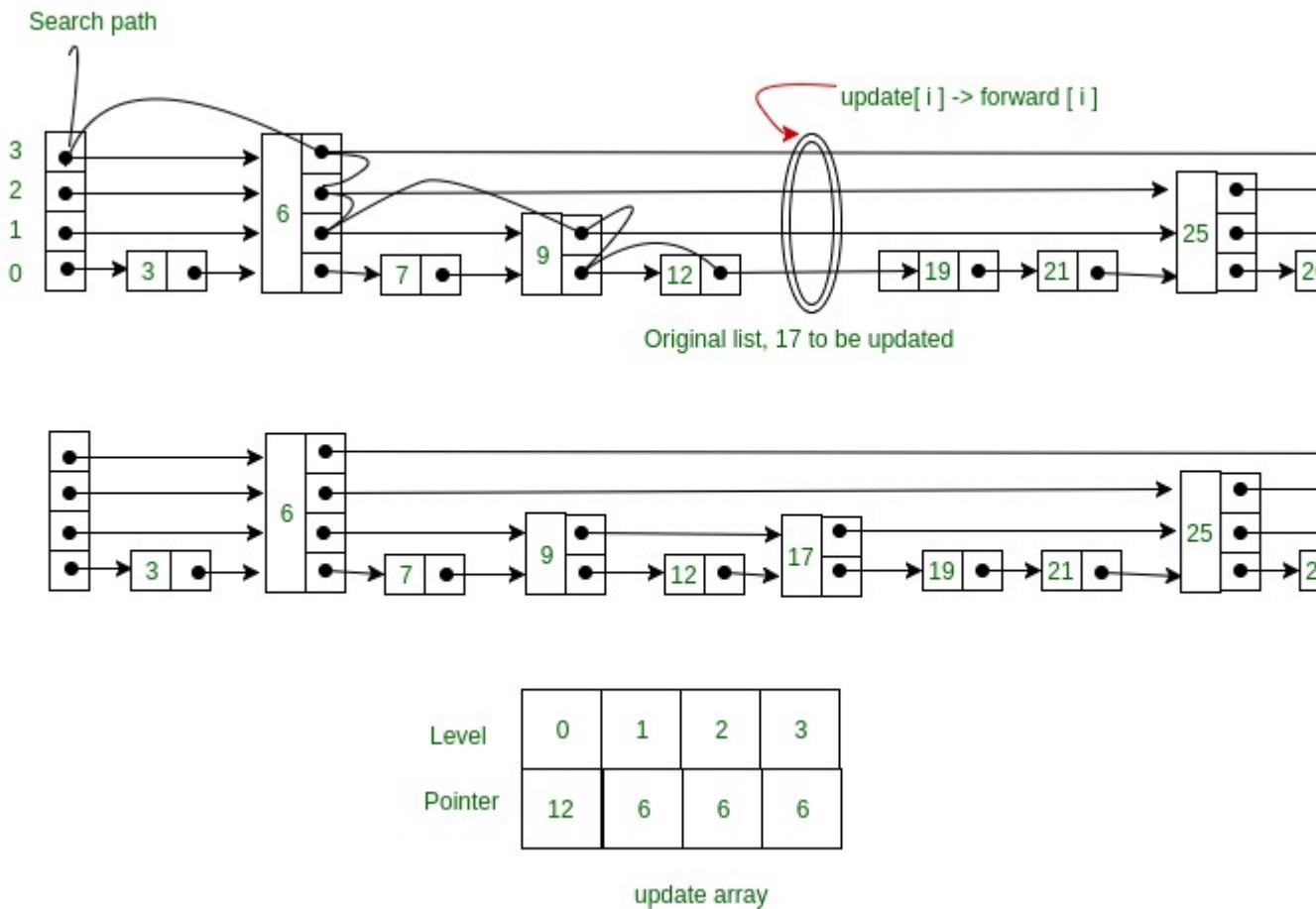
At the level 0, we will definitely find a position to insert given key. Following is the psuedo code for the insertion algorithm –

```

Insert(list, searchKey)
local update[0...MaxLevel+1]
x := list -> header
for i := list -> level downto 0 do
    while x -> forward[i] -> key   forward[i]
update[i] := x
x := x -> forward[0]
lvl := randomLevel()
if lvl > list -> level then
for i := list -> level + 1 to lvl do
    update[i] := list -> header
    list -> level := lvl
x := makeNode(lvl, searchKey, value)
for i := 0 to level do
    x -> forward[i] := update[i] -> forward[i]
    update[i] -> forward[i] := x

```

Here **update[i]** holds the pointer to node at level **i** from which we moved down to level **i-1** and pointer of node left to insertion position at level 0. Consider this example where we want to insert key 17 –



Following is the code for insertion of key in Skip list –

C++

```
// C++ code for inserting element in skip list

#include <bits/stdc++.h>
using namespace std;

// Class to implement node
class Node
{
public:
    int key;

    // Array to hold pointers to node of different level
    Node **forward;
    Node(int, int);
```

```
};

Node::Node(int key, int level)
{
    this->key = key;

    // Allocate memory to forward
    forward = new Node*[level+1];

    // Fill forward array with 0(NULL)
    memset(forward, 0, sizeof(Node*)*(level+1));
}

// Class for Skip list
class SkipList
{
    // Maximum level for this skip list
    int MAXLVL;

    // P is the fraction of the nodes with level
    // i pointers also having level i+1 pointers
    float P;

    // current level of skip list
    int level;

    // pointer to header node
    Node *header;
public:
    SkipList(int, float);
    int randomLevel();
    Node* createNode(int, int);
    void insertElement(int);
    void displayList();
};

SkipList::SkipList(int MAXLVL, float P)
{
    this->MAXLVL = MAXLVL;
    this->P = P;
    level = 0;

    // create header node and initialize key to -1
    header = new Node(-1, MAXLVL);
}

// create random level for node
int SkipList::randomLevel()
```

```

{
    float r = (float)rand()/RAND_MAX;
    int lvl = 0;
    while (r < P && lvl < MAXLVL)
    {
        lvl++;
        r = (float)rand()/RAND_MAX;
    }
    return lvl;
};

// create new node
Node* SkipList::createNode(int key, int level)
{
    Node *n = new Node(key, level);
    return n;
};

// Insert given key in skip list
void SkipList::insertElement(int key)
{
    Node *current = header;

    // create update array and initialize it
    Node *update[MAXLVL+1];
    memset(update, 0, sizeof(Node*)*(MAXLVL+1));

    /*      start from highest level of skip list
           move the current pointer forward while key
           is greater than key of node next to current
           Otherwise inserted current in update and
           move one level down and continue search
    */
    for (int i = level; i >= 0; i--)
    {
        while (current->forward[i] != NULL &&
               current->forward[i]->key < key)
            current = current->forward[i];
        update[i] = current;
    }

    /* reached level 0 and forward pointer to
       right, which is desired position to
       insert key.
    */
    current = current->forward[0];

    /* if current is NULL that means we have reached

```

```

to end of the level or current's key is not equal
to key to insert that means we have to insert
node between update[0] and current node */
if (current == NULL || current->key != key)
{
    // Generate a random level for node
    int rlevel = randomLevel();

    // If random level is greater than list's current
    // level (node with highest level inserted in
    // list so far), initialize update value with pointer
    // to header for further use
    if (rlevel > level)
    {
        for (int i=level+1;i<rlevel+1;i++)
            update[i] = header;

        // Update the list current level
        level = rlevel;
    }

    // create new node with random level generated
    Node* n = createNode(key, rlevel);

    // insert node by rearranging pointers
    for (int i=0;i<=rlevel;i++)
    {
        n->forward[i] = update[i]->forward[i];
        update[i]->forward[i] = n;
    }
    cout << "Successfully Inserted key " << key << "\n";
}
};

// Display skip list level wise
void SkipList::displayList()
{
    cout<<"\n*****Skip List*****"\<<"\n";
    for (int i=0;i<=level;i++)
    {
        Node *node = header->forward[i];
        cout << "Level " << i << ": ";
        while (node != NULL)
        {
            cout << node->key<< " ";
            node = node->forward[i];
        }
        cout << "\n";
    }
}

```

```
    }
};

// Driver to test above code
int main()
{
    // Seed random number generator
    srand((unsigned)time(0));

    // create SkipList object with MAXLVL and P
    SkipList lst(3, 0.5);

    lst.insertElement(3);
    lst.insertElement(6);
    lst.insertElement(7);
    lst.insertElement(9);
    lst.insertElement(12);
    lst.insertElement(19);
    lst.insertElement(17);
    lst.insertElement(26);
    lst.insertElement(21);
    lst.insertElement(25);
    lst.displayList();
}
```

### Python

```
# Python3 code for inserting element in skip list

import random

class Node(object):
    """
    Class to implement node
    """
    def __init__(self, key, level):
        self.key = key

        # list to hold references to node of different level
        self.forward = [None]*(level+1)

class SkipList(object):
    """
    Class for Skip list
    """
    def __init__(self, max_lvl, P):
        # Maximum level for this skip list
        self.MAXLVL = max_lvl
```

```

# P is the fraction of the nodes with level
# i references also having level i+1 references
self.P = P

# create header node and initialize key to -1
self.header = self.createNode(self.MAXLVL, -1)

# current level of skip list
self.level = 0

# create new node
def createNode(self, lvl, key):
    n = Node(key, lvl)
    return n

# create random level for node
def randomLevel(self):
    lvl = 0
    while random.random()<self.P and \
          lvl<self.MAXLVL:lvl += 1
    return lvl

# insert given key in skip list
def insertElement(self, key):
    # create update array and initialize it
    update = [None]*(self.MAXLVL+1)
    current = self.header

    """
    start from highest level of skip list
    move the current reference forward while key
    is greater than key of node next to current
    Otherwise inserted current in update and
    move one level down and continue search
    """
    for i in range(self.level, -1, -1):
        while current.forward[i] and \
              current.forward[i].key < key:
            current = current.forward[i]
        update[i] = current

    """
    reached level 0 and forward reference to
    right, which is desired position to
    insert key.
    """
    current = current.forward[0]

```

```

    """
    if current is NULL that means we have reached
        to end of the level or current's key is not equal
        to key to insert that means we have to insert
        node between update[0] and current node
    """

    if current == None or current.key != key:
        # Generate a random level for node
        rlevel = self.randomLevel()

    """

    If random level is greater than list's current
    level (node with highest level inserted in
    list so far), initialize update value with reference
    to header for further use
    """

    if rlevel > self.level:
        for i in range(self.level+1, rlevel+1):
            update[i] = self.header
        self.level = rlevel

    # create new node with random level generated
    n = self.createNode(rlevel, key)

    # insert node by rearranging references
    for i in range(rlevel+1):
        n.forward[i] = update[i].forward[i]
        update[i].forward[i] = n

    print("Successfully inserted key {}".format(key))

# Display skip list level wise
def displayList(self):
    print("\n*****Skip List*****")
    head = self.header
    for lvl in range(self.level+1):
        print("Level {}: ".format(lvl), end=" ")
        node = head.forward[lvl]
        while(node != None):
            print(node.key, end=" ")
            node = node.forward[lvl]
        print("")

# Driver to test above code
def main():
    lst = SkipList(3, 0.5)
    lst.insertElement(3)

```

```
lst.insertElement(6)
lst.insertElement(7)
lst.insertElement(9)
lst.insertElement(12)
lst.insertElement(19)
lst.insertElement(17)
lst.insertElement(26)
lst.insertElement(21)
lst.insertElement(25)
lst.displayList()

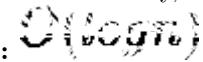
main()
```

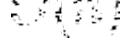
Output:

```
Successfully Inserted key 3
Successfully Inserted key 6
Successfully Inserted key 7
Successfully Inserted key 9
Successfully Inserted key 12
Successfully Inserted key 19
Successfully Inserted key 17
Successfully Inserted key 26
Successfully Inserted key 21
Successfully Inserted key 25
```

```
*****Skip List*****
Level 0: 3 6 7 9 12 17 19 21 25 26
Level 1: 3 6 12 17 25
Level 2: 6 12 17 25
Level 3: 12 17 25
```

**Note:** The level of nodes is decided randomly, so output may differ.

**Time complexity (Average):** 

**Time complexity (Worst):** 

In next article we will discuss searching and deletion in Skip List.

#### References

- <ftp://ftp.cs.umd.edu/pub/skipLists/skiplists.pdf>

#### Source

<https://www.geeksforgeeks.org/skip-list-set-2-insertion/>

## Chapter 232

# Skip List | Set 3 (Searching and Deletion)

Skip List | Set 3 (Searching and Deletion) - GeeksforGeeks

In previous article [Skip List | Set 2 \(Insertion\)](#) we discussed the structure of skip nodes and how to insert an element in the skip list. In this article we will discuss how to search and delete an element from skip list.

### Searching an element in Skip list

Searching an element is very similar to approach for searching a spot for inserting an element in Skip list. The basic idea is if –

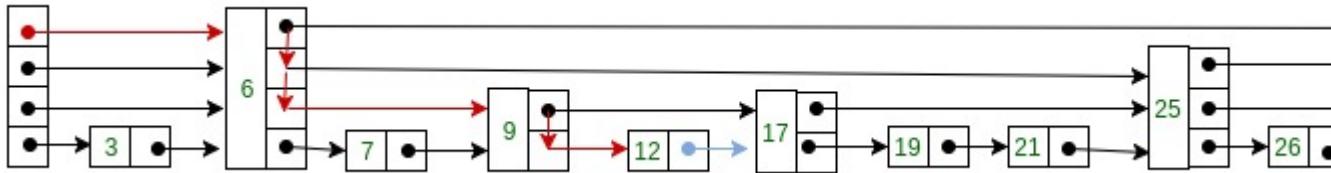
1. Key of next node is less than search key then we keep on moving forward on the same level.
2. Key of next node is greater than the key to be inserted then we store the pointer to current node **i** at **update[i]** and move one level down and continue our search.

At the lowest level (0), if the element next to the rightmost element (**update[0]**) has key equal to the search key, then we have found key otherwise failure.

Following is the pseudo code for searching element –

```
Search(list, searchKey)
x := list -> header
-- loop invariant: x -> key  level downto 0 do
    while x -> forward[i] -> key  forward[i]
x := x -> forward[0]
if x -> key = searchKey then return x -> value
else return failure
```

Consider this example where we want to search for key 17-



### Deleting an element from the Skip list

Deletion of an element  $k$  is preceded by locating element in the Skip list using above mentioned search algorithm. Once the element is located, rearrangement of pointers is done to remove element from list just like we do in singly linked list. We start from lowest level and do rearrangement until element next to  $\text{update}[i]$  is not  $k$ .

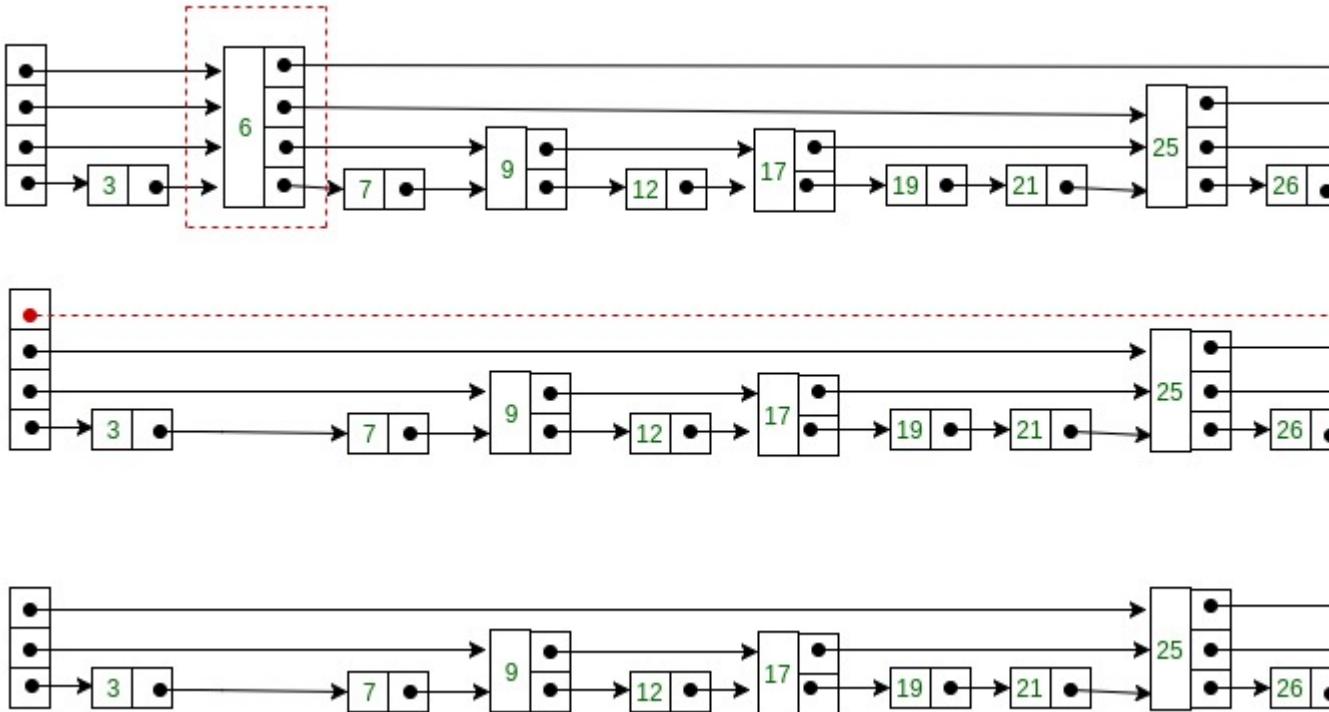
After deletion of element there could be levels with no elements, so we will remove these levels as well by decrementing the level of Skip list. Following is the pseudo code for deletion

```

Delete(list, searchKey)
local update[0..MaxLevel+1]
x := list -> header
for i := list -> level downto 0 do
    while x -> forward[i] -> key   forward[i]
        update[i] := x
    x := x -> forward[0]
if x -> key = searchKey then
    for i := 0 to list -> level do
        if update[i] -> forward[i] = x then break
        update[i] -> forward[i] := x -> forward[i]
    free(x)
while list -> level > 0 and list -> header -> forward[list -> level] = NIL do
    list -> level := list -> level - 1

```

Consider this example where we want to delete element 6 –



Here at level 3, there is no element (arrow in red) after deleting element 6. So we will decrement level of skip list by 1.

Following is the code for searching and deleting element from Skip List –

C++

```
// C++ code for searching and deleting element in skip list

#include <bits/stdc++.h>
using namespace std;

// Class to implement node
class Node
{
public:
    int key;

    // Array to hold pointers to node of different level
    Node **forward;
    Node(int, int);
};

Node::Node(int key, int level)
{
```

```
this->key = key;

// Allocate memory to forward
forward = new Node*[level+1];

// Fill forward array with 0(NULL)
memset(forward, 0, sizeof(Node*)*(level+1));
};

// Class for Skip list
class SkipList
{
    // Maximum level for this skip list
    int MAXLVL;

    // P is the fraction of the nodes with level
    // i pointers also having level i+1 pointers
    float P;

    // current level of skip list
    int level;

    // pointer to header node
    Node *header;
public:
    SkipList(int, float);
    int randomLevel();
    Node* createNode(int, int);
    void insertElement(int);
    void deleteElement(int);
    void searchElement(int);
    void displayList();
};

SkipList::SkipList(int MAXLVL, float P)
{
    this->MAXLVL = MAXLVL;
    this->P = P;
    level = 0;

    // create header node and initialize key to -1
    header = new Node(-1, MAXLVL);
};

// create random level for node
int SkipList::randomLevel()
{
    float r = (float)rand()/RAND_MAX;
```

```
int lvl = 0;
while(r < P && lvl < MAXLVL)
{
    lvl++;
    r = (float)rand()/RAND_MAX;
}
return lvl;
};

// create new node
Node* SkipList::createNode(int key, int level)
{
    Node *n = new Node(key, level);
    return n;
};

// Insert given key in skip list
void SkipList::insertElement(int key)
{
    Node *current = header;

    // create update array and initialize it
    Node *update[MAXLVL+1];
    memset(update, 0, sizeof(Node)*(MAXLVL+1));

    /*      start from highest level of skip list
        move the current pointer forward while key
        is greater than key of node next to current
        Otherwise inserted current in update and
        move one level down and continue search
    */
    for(int i = level; i >= 0; i--)
    {
        while(current->forward[i] != NULL &&
               current->forward[i]->key < key)
            current = current->forward[i];
        update[i] = current;
    }

    /* reached level 0 and forward pointer to
       right, which is desired position to
       insert key.
    */
    current = current->forward[0];

    /* if current is NULL that means we have reached
       to end of the level or current's key is not equal
       to key to insert that means we have to insert
    */
}
```

```
    node between update[0] and current node */
if (current == NULL || current->key != key)
{
    // Generate a random level for node
    int rlevel = randomLevel();

    /* If random level is greater than list's current
       level (node with highest level inserted in
       list so far), initialize update value with pointer
       to header for further use */
    if(rlevel > level)
    {
        for(int i=level+1;i<rlevel+1;i++)
            update[i] = header;

        // Update the list current level
        level = rlevel;
    }

    // create new node with random level generated
    Node* n = createNode(key, rlevel);

    // insert node by rearranging pointers
    for(int i=0;i<=rlevel;i++)
    {
        n->forward[i] = update[i]->forward[i];
        update[i]->forward[i] = n;
    }
    cout<<"Successfully Inserted key "<<key<<"\n";
}

// Delete element from skip list
void SkipList::deleteElement(int key)
{
    Node *current = header;

    // create update array and initialize it
    Node *update[MAXLVL+1];
    memset(update, 0, sizeof(Node)*(MAXLVL+1));

    /*      start from highest level of skip list
           move the current pointer forward while key
           is greater than key of node next to current
           Otherwise inserted current in update and
           move one level down and continue search
    */
    for(int i = level; i >= 0; i--)

```

```

{
    while(current->forward[i] != NULL &&
          current->forward[i]->key < key)
        current = current->forward[i];
    update[i] = current;
}

/* reached level 0 and forward pointer to
   right, which is possibly our desired node.*/
current = current->forward[0];

// If current node is target node
if(current != NULL and current->key == key)
{
    /* start from lowest level and rearrange
       pointers just like we do in singly linked list
       to remove target node */
    for(int i=0;i<=level;i++)
    {
        /* If at level i, next node is not target
           node, break the loop, no need to move
           further level */
        if(update[i]->forward[i] != current)
            break;

        update[i]->forward[i] = current->forward[i];
    }

    // Remove levels having no elements
    while(level>0 &&
          header->forward[level] == 0)
        level--;
    cout<<"Successfully deleted key "<<key<<"\n";
}
};

// Search for element in skip list
void SkipList::searchElement(int key)
{
    Node *current = header;

    /*      start from highest level of skip list
           move the current pointer forward while key
           is greater than key of node next to current
           Otherwise inserted current in update and
           move one level down and continue search
    */
    for(int i = level; i >= 0; i--)

```

```
{  
    while(current->forward[i] &&  
          current->forward[i]->key < key)  
        current = current->forward[i];  
  
}  
  
/* reached level 0 and advance pointer to  
   right, which is possibly our desired node*/  
current = current->forward[0];  
  
// If current node have key equal to  
// search key, we have found our target node  
if(current and current->key == key)  
    cout<<"Found key: "<<key<<"\n";  
};  
  
// Display skip list level wise  
void SkipList::displayList()  
{  
    cout<<"\n*****Skip List*****"<<"\n";  
    for(int i=0;i<=level;i++)  
    {  
        Node *node = header->forward[i];  
        cout<<"Level "<<i<<": ";  
        while(node != NULL)  
        {  
            cout<<node->key<<" ";  
            node = node->forward[i];  
        }  
        cout<<"\n";  
    }  
};  
  
// Driver to test above code  
int main()  
{  
    // Seed random number generator  
    srand((unsigned)time(0));  
  
    // create SkipList object with MAXLVL and P  
    SkipList lst(3, 0.5);  
  
    lst.insertElement(3);  
    lst.insertElement(6);  
    lst.insertElement(7);  
    lst.insertElement(9);  
    lst.insertElement(12);
```

```
lst.insertElement(19);
lst.insertElement(17);
lst.insertElement(26);
lst.insertElement(21);
lst.insertElement(25);
lst.displayList();

//Search for node 19
lst.searchElement(19);

//Delete node 19
lst.deleteElement(19);
lst.displayList();
}
```

### Python

```
# Python3 code for searching and deleting element in skip list

import random

class Node(object):
    """
    Class to implement node
    """
    def __init__(self, key, level):
        self.key = key

        # list to hold references to node of different level
        self.forward = [None]*(level+1)

class SkipList(object):
    """
    Class for Skip list
    """
    def __init__(self, max_lvl, P):
        # Maximum level for this skip list
        self.MAXLVL = max_lvl

        # P is the fraction of the nodes with level
        # i references also having level i+1 references
        self.P = P

        # create header node and initialize key to -1
        self.header = self.createNode(self.MAXLVL, -1)

        # current level of skip list
        self.level = 0
```

```
# create new node
def createNode(self, lvl, key):
    n = Node(key, lvl)
    return n

# create random level for node
def randomLevel(self):
    lvl = 0
    while random.random()<self.P and \
          lvl<self.MAXLVL:lvl += 1
    return lvl

# insert given key in skip list
def insertElement(self, key):
    # create update array and initialize it
    update = [None]*(self.MAXLVL+1)
    current = self.header

    """
    start from highest level of skip list
    move the current reference forward while key
    is greater than key of node next to current
    Otherwise inserted current in update and
    move one level down and continue search
    """
    for i in range(self.level, -1, -1):
        while current.forward[i] and \
              current.forward[i].key < key:
            current = current.forward[i]
            update[i] = current

    """
    reached level 0 and forward reference to
    right, which is desired position to
    insert key.
    """
    current = current.forward[0]

    """
    if current is NULL that means we have reached
    to end of the level or current's key is not equal
    to key to insert that means we have to insert
    node between update[0] and current node
    """
    if current == None or current.key != key:
        # Generate a random level for node
        rlevel = self.randomLevel()
```

```
    ...
    If random level is greater than list's current
    level (node with highest level inserted in
    list so far), initialize update value with reference
    to header for further use
    ...
    if rlevel > self.level:
        for i in range(self.level+1, rlevel+1):
            update[i] = self.header
        self.level = rlevel

    # create new node with random level generated
    n = self.createNode(rlevel, key)

    # insert node by rearranging references
    for i in range(rlevel+1):
        n.forward[i] = update[i].forward[i]
        update[i].forward[i] = n

    print("Successfully inserted key {}".format(key))

def deleteElement(self, search_key):

    # create update array and initialize it
    update = [None]*(self.MAXLVL+1)
    current = self.header

    ...
    start from highest level of skip list
    move the current reference forward while key
    is greater than key of node next to current
    Otherwise inserted current in update and
    move one level down and continue search
    ...
    for i in range(self.level, -1, -1):
        while(current.forward[i] and \
              current.forward[i].key < search_key):
            current = current.forward[i]
            update[i] = current

    ...
    reached level 0 and advance reference to
    right, which is possibly our desired node
    ...
    current = current.forward[0]

    # If current node is target node
```

```
if current != None and current.key == search_key:  
    ...  
    start from lowest level and rearrange references  
    just like we do in singly linked list  
    to remove target node  
    ...  
    for i in range(self.level+1):  
        ...  
        If at level i, next node is not target  
        node, break the loop, no need to move  
        further level  
        ...  
        if update[i].forward[i] != current:  
            break  
        update[i].forward[i] = current.forward[i]  
  
    # Remove levels having no elements  
    while(self.level>0 and\  
          self.header.forward[self.level] == None):  
        self.level -= 1  
    print("Successfully deleted {}".format(search_key))  
  
def searchElement(self, key):  
    current = self.header  
  
    ...  
    start from highest level of skip list  
    move the current reference forward while key  
    is greater than key of node next to current  
    Otherwise inserted current in update and  
    move one level down and continue search  
    ...  
    for i in range(self.level, -1, -1):  
        while(current.forward[i] and\  
              current.forward[i].key < key):  
            current = current.forward[i]  
  
    # reached level 0 and advance reference to  
    # right, which is possibly our desired node  
    current = current.forward[0]  
  
    # If current node have key equal to  
    # search key, we have found our target node  
    if current and current.key == key:  
        print("Found key ", key)
```

```
# Display skip list level wise
def displayList(self):
    print("\n*****Skip List*****")
    head = self.header
    for lvl in range(self.level+1):
        print("Level {}: ".format(lvl), end=" ")
        node = head.forward[lvl]
        while(node != None):
            print(node.key, end=" ")
            node = node.forward[lvl]
        print("")

# Driver to test above code
def main():
    lst = SkipList(3, 0.5)
    lst.insertElement(3)
    lst.insertElement(6)
    lst.insertElement(7)
    lst.insertElement(9)
    lst.insertElement(12)
    lst.insertElement(19)
    lst.insertElement(17)
    lst.insertElement(26)
    lst.insertElement(21)
    lst.insertElement(25)
    lst.displayList()

    # Search for node 19
    lst.searchElement(19)

    # Delete node 19
    lst.deleteElement(19)
    lst.displayList()

main()
```

Output:

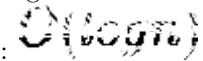
```
Successfully Inserted key 3
Successfully Inserted key 6
Successfully Inserted key 7
Successfully Inserted key 9
Successfully Inserted key 12
Successfully Inserted key 19
Successfully Inserted key 17
Successfully Inserted key 26
```

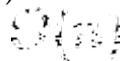
```
Successfully Inserted key 21
Successfully Inserted key 25
```

```
*****Skip List*****
Level 0: 3 6 7 9 12 17 19 21 25 26
Level 1: 3 17 19 21 26
Level 2: 17 19 21
Found key: 19
Successfully deleted key 19
```

```
*****Skip List*****
Level 0: 3 6 7 9 12 17 21 25 26
Level 1: 3 17 21 26
Level 2: 17 21
```

Time complexity of both searching and deletion is same –

Time complexity (Average): 

Time complexity (Worst): 

#### References

<ftp://ftp.cs.umd.edu/pub/skipLists/skiplists.pdf>

#### Source

<https://www.geeksforgeeks.org/skip-list-set-3-searching-deletion/>

## Chapter 233

# Sort a k sorted doubly linked list

Sort a k sorted doubly linked list - GeeksforGeeks

Given a doubly linked list containing **n** nodes, where each node is at most **k** away from its target position in the list. The problem is to sort the given doubly linked list.

For example, let us consider **k** is 2, a node at position 7 in the sorted doubly linked list, can be at positions 5, 6, 7, 8, 9 in the given doubly linked list.

Examples:

```
Input : DLL: 3 <-> 6 <-> 2 <-> 12 <-> 56 <-> 8  
       k = 2  
Output : 2 <-> 3 <-> 6 <-> 8 <-> 12 <-> 56
```

**Naive Approach:** Sort the given doubly linked list using insertion sort technique.

Time Complexity:  $O(nk)$

Auxiliary Space:  $O(1)$

**Efficient Approach:** We can sort the list using the MIN HEAP data structure. The approach has been explained in [Sort a nearly sorted \(or K sorted\) array](#). We only have to be careful while traversing the input doubly linked list and adjusting the required **next** and **previous** links in the final sorted list.

```
// C++ implementation to sort a k sorted doubly  
// linked list  
#include <bits/stdc++.h>  
using namespace std;  
  
// a node of the doubly linked list  
struct Node {  
    int data;
```

```
struct Node* next;
struct Node* prev;
};

// 'compare' function used to build up the
// priority queue
struct compare {
    bool operator()(struct Node* p1, struct Node* p2)
    {
        return p1->data > p2->data;
    }
};

// function to sort a k sorted doubly linked list
struct Node* sortAKSortedDLL(struct Node* head, int k)
{
    // if list is empty
    if (head == NULL)
        return head;

    // priority_queue 'pq' implemented as min heap with the
    // help of 'compare' function
    priority_queue<Node*, vector<Node*>, compare> pq;

    struct Node* newHead = NULL, *last;

    // Create a Min Heap of first (k+1) elements from
    // input doubly linked list
    for (int i = 0; head != NULL && i <= k; i++) {
        // push the node on to 'pq'
        pq.push(head);

        // move to the next node
        head = head->next;
    }

    // loop till there are elements in 'pq'
    while (!pq.empty()) {

        // place root or top of 'pq' at the end of the
        // result sorted list so far having the first node
        // pointed to by 'newHead'
        // and adjust the required links
        if (newHead == NULL) {
            newHead = pq.top();
            newHead->prev = NULL;

            // 'last' points to the last node
            last = newHead;
        } else {
            last->next = pq.top();
            pq.top()->prev = last;
            last = pq.top();
        }
    }
}
```

```
// of the result sorted list so far
last = newHead;
}

else {
    last->next = pq.top();
    pq.top()->prev = last;
    last = pq.top();
}

// remove element from 'pq'
pq.pop();

// if there are more nodes left in the input list
if (head != NULL) {
    // push the node on to 'pq'
    pq.push(head);

    // move to the next node
    head = head->next;
}
}

// making 'next' of last node point to NULL
last->next = NULL;

// new head of the required sorted DLL
return newHead;
}

// Function to insert a node at the beginning
// of the Doubly Linked List
void push(struct Node** head_ref, int new_data)
{
    // allocate node
    struct Node* new_node =
        (struct Node*)malloc(sizeof(struct Node));

    // put in the data
    new_node->data = new_data;

    // since we are adding at the begining,
    // prev is always NULL
    new_node->prev = NULL;

    // link the old list off the new node
    new_node->next = (*head_ref);
```

```
// change prev of head node to new node
if ((*head_ref) != NULL)
    (*head_ref)->prev = new_node;

// move the head to point to the new node
(*head_ref) = new_node;
}

// Function to print nodes in a given doubly linked list
void printList(struct Node* head)
{
    // if list is empty
    if (head == NULL)
        cout << "Doubly Linked list empty";

    while (head != NULL) {
        cout << head->data << " ";
        head = head->next;
    }
}

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

    // Create the doubly linked list:
    // 3<->6<->2<->12<->56<->8
    push(&head, 8);
    push(&head, 56);
    push(&head, 12);
    push(&head, 2);
    push(&head, 6);
    push(&head, 3);

    int k = 2;

    cout << "Original Doubly linked list:n";
    printList(head);

    // sort the bitonic DLL
    head = sortAKSortedDLL(head, k);

    cout << "\nDoubly linked list after sorting:n";
    printList(head);

    return 0;
}
```

Output:

```
Original Doubly linked list:  
3 6 2 12 56 8  
Doubly linked list after sorting:  
2 3 6 8 12 56
```

Time Complexity:  $O(n \log k)$   
Auxiliary Space:  $O(k)$

## Source

<https://www.geeksforgeeks.org/sort-k-sorted-doubly-linked-list/>

## Chapter 234

# Sort a linked list of 0s, 1s and 2s

Sort a linked list of 0s, 1s and 2s - GeeksforGeeks

Given a linked list of 0s, 1s and 2s, sort it.

Source: [Microsoft Interview | Set 1](#)

Following steps can be used to sort the given linked list.

- 1) Traverse the list and count the number of 0s, 1s and 2s. Let the counts be n1, n2 and n3 respectively.
- 2) Traverse the list again, fill the first n1 nodes with 0, then n2 nodes with 1 and finally n3 nodes with 2.

**C/C++**

```
// C Program to sort a linked list 0s, 1s or 2s
#include<stdio.h>
#include<stdlib.h>

/* Link list node */
struct Node
{
    int data;
    struct Node* next;
};

// Function to sort a linked list of 0s, 1s and 2s
void sortList(struct Node *head)
{
    int count[3] = {0, 0, 0}; // Initialize count of '0', '1' and '2' as 0
    struct Node *ptr = head;

    /* count total number of '0', '1' and '2'
     * count[0] will store total number of '0's
     * count[1] will store total number of '1's
     * count[2] will store total number of '2's
    */
    for(ptr = head; ptr != NULL; ptr = ptr->next)
    {
        if(ptr->data == 0)
            count[0]++;
        else if(ptr->data == 1)
            count[1]++;
        else
            count[2]++;
    }

    ptr = head;
    for(int i=0; i<count[0]; i++)
        ptr->data = 0;
    for(int i=0; i<count[1]; i++)
        ptr->data = 1;
    for(int i=0; i<count[2]; i++)
        ptr->data = 2;
}
```

```
* count[2] will store total number of '2's */
while (ptr != NULL)
{
    count[ptr->data] += 1;
    ptr = ptr->next;
}

int i = 0;
ptr = head;

/* Let say count[0] = n1, count[1] = n2 and count[2] = n3
 * now start traversing list from head node,
 * 1) fill the list with 0, till n1 > 0
 * 2) fill the list with 1, till n2 > 0
 * 3) fill the list with 2, till n3 > 0 */
while (ptr != NULL)
{
    if (count[i] == 0)
        ++i;
    else
    {
        ptr->data = i;
        --count[i];
        ptr = ptr->next;
    }
}
}

/* Function to push a node */
void push (struct Node** head_ref, int new_data)
{
    /* allocate node */
    struct Node* new_node =
        (struct Node*) malloc(sizeof(struct Node));

    /* put in the data */
    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;
}

/* Function to print linked list */
void printList(struct Node *node)
{
```

```
while (node != NULL)
{
    printf("%d ", node->data);
    node = node->next;
}
printf("\n");

/* Drier program to test above function*/
int main(void)
{
    struct Node *head = NULL;
    push(&head, 0);
    push(&head, 1);
    push(&head, 0);
    push(&head, 2);
    push(&head, 1);
    push(&head, 1);
    push(&head, 2);
    push(&head, 1);
    push(&head, 2);

    printf("Linked List Before Sortingn");
    printList(head);

    sortList(head);

    printf("Linked List After Sortingn");
    printList(head);

    return 0;
}
```

### Java

```
// Java program to sort a linked list of 0, 1 and 2
class LinkedList
{
    Node head; // head of list

    /* Linked list Node*/
    class Node
    {
        int data;
        Node next;
        Node(int d) {data = d; next = null; }
    }
```

```
void sortList()
{
    // initialise count of 0 1 and 2 as 0
    int count[] = {0, 0, 0};

    Node ptr = head;

    /* count total number of '0', '1' and '2'
     * count[0] will store total number of '0's
     * count[1] will store total number of '1's
     * count[2] will store total number of '2's */
    while (ptr != null)
    {
        count[ptr.data]++;
        ptr = ptr.next;
    }

    int i = 0;
    ptr = head;

    /* Let say count[0] = n1, count[1] = n2 and count[2] = n3
     * now start traversing list from head node,
     * 1) fill the list with 0, till n1 > 0
     * 2) fill the list with 1, till n2 > 0
     * 3) fill the list with 2, till n3 > 0 */
    while (ptr != null)
    {
        if (count[i] == 0)
            i++;
        else
        {
            ptr.data= i;
            --count[i];
            ptr = ptr.next;
        }
    }
}

/* Utility functions */

/* Inserts a new Node at front of the list. */
public void push(int new_data)
{
    /* 1 & 2: Allocate the Node &
       Put in the data*/
    Node new_node = new Node(new_data);
```

```
/* 3. Make next of new Node as head */
new_node.next = head;

/* 4. Move the head to point to new Node */
head = new_node;
}

/* Function to print linked list */
void printList()
{
    Node temp = head;
    while (temp != null)
    {
        System.out.print(temp.data+" ");
        temp = temp.next;
    }
    System.out.println();
}

/* Drier program to test above functions */
public static void main(String args[])
{
    LinkedList llist = new LinkedList();

    /* Constructed Linked List is 1->2->3->4->5->6->7->
       8->8->9->null */
    llist.push(0);
    llist.push(1);
    llist.push(0);
    llist.push(2);
    llist.push(1);
    llist.push(1);
    llist.push(2);
    llist.push(1);
    llist.push(2);

    System.out.println("Linked List before sorting");
    llist.printList();

    llist.sortList();

    System.out.println("Linked List after sorting");
    llist.printList();
}
}

/* This code is contributed by Rajat Mishra */
```

Python

```
# Python program to sort a linked list of 0, 1 and 2
class LinkedList(object):
    def __init__(self):

        # head of list
        self.head = None

    # Linked list Node
    class Node(object):
        def __init__(self, d):
            self.data = d
            self.next = None

    def sortList(self):

        # initialise count of 0 1 and 2 as 0
        count = [0, 0, 0]

        ptr = self.head

        # count total number of '0', '1' and '2'
        # * count[0] will store total number of '0's
        # * count[1] will store total number of '1's
        # * count[2] will store total number of '2's
        while ptr != None:
            count[ptr.data]+=1
            ptr = ptr.next

        i = 0
        ptr = self.head

        # Let say count[0] = n1, count[1] = n2 and count[2] = n3
        # * now start traversing list from head node,
        # * 1) fill the list with 0, till n1 > 0
        # * 2) fill the list with 1, till n2 > 0
        # * 3) fill the list with 2, till n3 > 0
        while ptr != None:
            if count[i] == 0:
                i+=1
            else:
                ptr.data = i
                count[i]-=1
                ptr = ptr.next

    # Utility functions
    # Inserts a new Node at front of the list.
    def push(self, new_data):
```

```
# 1 & 2: Allocate the Node &
# Put in the data
new_node = self.Node(new_data)

# 3. Make next of new Node as head
new_node.next = self.head

# 4. Move the head to point to new Node
self.head = new_node

# Function to print linked list
def printList(self):
    temp = self.head
    while temp != None:
        print str(temp.data),
        temp = temp.next
    print ''

# Drier program to test above functions
llist = LinkedList()
llist.push(0)
llist.push(1)
llist.push(0)
llist.push(2)
llist.push(1)
llist.push(1)
llist.push(2)
llist.push(1)
llist.push(2)

print "Linked List before sorting"
llist.printList()

llist.sortList()

print "Linked List after sorting"
llist.printList()

# This code is contributed by BHAVYA JAIN
```

Output:

```
Linked List Before Sorting
2 1 2 1 1 2 0 1 0
Linked List After Sorting
0 0 1 1 1 2 2 2
```

Time Complexity:  $O(n)$  where  $n$  is number of nodes in linked list.

Auxiliary Space:  $O(1)$

### **Sort a linked list of 0s, 1s and 2s by changing links**

This article is compiled by **Narendra Kangalkar**. 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/sort-a-linked-list-of-0s-1s-or-2s/>

## Chapter 235

# Sort a linked list of 0s, 1s and 2s by changing links

Sort a linked list of 0s, 1s and 2s by changing links - GeeksforGeeks

Given a linked list of 0s, 1s and 2s, sort it.

Examples:

Input : 2->1->2->1->1->2->0->1->0  
Output : 0->0->1->1->1->1->2->2->2

Input : 2->1->0  
Output : 0->1->2

We have discussed a solution in below post that works by changing data of nodes.

[Sort a linked list of 0s, 1s and 2s](#)

The above solution does not work when these values have associated data with them. For example, these three represent three colors and different types of objects associated with the colors and we want to sort objects (connected with a linked list) based on colors.

In this post, a new solution is discussed that works by changing links.

Iterate through the linked list. Maintain 3 pointers named zero, one and two to point to current ending nodes of linked lists containing 0, 1, and 2 respectively. For every traversed node, we attach it to the end of its corresponding list. Finally we link all three lists. To avoid many null checks, we use three dummy pointers zeroD, oneD and twoD that work as dummy headers of three lists.

```
// CPP Program to sort a linked list 0s, 1s
// or 2s by changing links
#include <stdio.h>
```

```
/* Link list node */
struct Node {
    int data;
    struct Node* next;
};

Node* newNode(int data);

// Sort a linked list of 0s, 1s and 2s
// by changing pointers.
Node* sortList(Node* head)
{
    if (!head || !(head->next))
        return head;

    // Create three dummy nodes to point to
    // beginning of three linked lists. These
    // dummy nodes are created to avoid many
    // null checks.
    Node* zeroD = newNode(0);
    Node* oneD = newNode(0);
    Node* twoD = newNode(0);

    // Initialize current pointers for three
    // lists and whole list.
    Node* zero = zeroD, *one = oneD, *two = twoD;

    // Traverse list
    Node* curr = head;
    while (curr) {
        if (curr->data == 0) {
            zero->next = curr;
            zero = zero->next;
            curr = curr->next;
        } else if (curr->data == 1) {
            one->next = curr;
            one = one->next;
            curr = curr->next;
        } else {
            two->next = curr;
            two = two->next;
            curr = curr->next;
        }
    }

    // Attach three lists
    zero->next = (oneD->next) ? (oneD->next) : (twoD->next);
}
```

```
one->next = twoD->next;
two->next = NULL;

// Updated head
head = zeroD->next;

// Delete dummy nodes
delete zeroD;
delete oneD;
delete twoD;

return head;
}

// function to create and return a node
Node* newNode(int data)
{
    // allocating space
    Node* newNode = new Node;

    // inserting the required data
    newNode->data = data;
    newNode->next = NULL;
}

/* Function to print linked list */
void printList(struct Node* node)
{
    while (node != NULL) {
        printf("%d ", node->data);
        node = node->next;
    }
    printf("\n");
}

/* Drier program to test above function*/
int main(void)
{
    // Creating the list 1->2->4->5
    Node* head = newNode(1);
    head->next = newNode(2);
    head->next->next = newNode(0);
    head->next->next->next = newNode(1);

    printf("Linked List Before Sorting\n");
    printList(head);

    head = sortList(head);
```

```
printf("Linked List After Sorting\n");
printList(head);

return 0;
}
```

Output :

```
Linked List Before Sorting
1 2 0 1
Linked List After Sorting
0 1 1 2
```

Thanks to Musarrat\_123 for suggesting above solution in a comment [here](#).

Time Complexity:  $O(n)$  where  $n$  is number of nodes in linked list.

Auxiliary Space:  $O(1)$

## Source

<https://www.geeksforgeeks.org/sort-linked-list-0s-1s-2s-changing-links/>

## Chapter 236

# Sort a linked list that is sorted alternating ascending and descending orders?

Sort a linked list that is sorted alternating ascending and descending orders? - Geeks-forGeeks

Given a Linked List. The Linked List is in alternating ascending and descending orders. Sort the list efficiently.

**Example:**

Input List: 10->40->53->30->67->12->89->NULL  
Output List: 10->12->30->43->53->67->89->NULL

A **Simple Solution** is to use [Merge Sort for linked List](#). This solution takes  $O(n \log n)$  time.

An **Efficient Solution** works in  $O(n)$  time. Below are all steps.

1. Separate two lists.
2. Reverse the one with descending order
3. Merge both lists.

Below are C++ and Java implementations of above algorithm.

**C++**

```
// C++ program to sort a linked list that is alternatively
// sorted in increasing and decreasing order
#include<bits/stdc++.h>
using namespace std;
```

```
// Linked list node
struct Node
{
    int data;
    struct Node *next;
};

Node *mergelist(Node *head1, Node *head2);
void splitList(Node *head, Node **Ahead, Node **Dhead);
void reverselist(Node *&head);

// This is the main function that sorts the
// linked list
void sort(Node **head)
{
    // Split the list into lists
    Node *Ahead, *Dhead;
    splitList(*head, &Ahead, &Dhead);

    // Reverse the descending linked list
    reverselist(Dhead);

    // Merge the two linked lists
    *head = mergelist(Ahead, Dhead);
}

// A utility function to create a new node
Node* newNode(int key)
{
    Node *temp = new Node;
    temp->data = key;
    temp->next = NULL;
    return temp;
}

// A utility function to reverse a linked list
void reverselist(Node *&head)
{
    Node* prev = NULL, *curr = head, *next;
    while (curr)
    {
        next = curr->next;
        curr->next = prev;
        prev = curr;
        curr = next;
    }
    head = prev;
}
```

```
// A utility function to print a linked list
void printlist(Node *head)
{
    while (head != NULL)
    {
        cout << head->data << " ";
        head = head->next;
    }
    cout << endl;
}

// A utility function to merge two sorted linked lists
Node *mergelist(Node *head1, Node *head2)
{
    // Base cases
    if (!head1) return head2;
    if (!head2) return head1;

    Node *temp = NULL;
    if (head1->data < head2->data)
    {
        temp = head1;
        head1->next = mergelist(head1->next, head2);
    }
    else
    {
        temp = head2;
        head2->next = mergelist(head1, head2->next);
    }
    return temp;
}

// This function alternatively splits a linked list with head
// as head into two:
// For example, 10->20->30->15->40->7 is splitted into 10->30->40
// and 20->15->7
// "Ahead" is reference to head of ascending linked list
// "Dhead" is reference to head of descending linked list
void splitList(Node *head, Node **Ahead, Node **Dhead)
{
    // Create two dummy nodes to initialize heads of two linked list
    *Ahead = newNode(0);
    *Dhead = newNode(0);

    Node *ascn = *Ahead;
    Node *dscn = *Dhead;
    Node *curr = head;
```

```
// Link alternate nodes
while (curr)
{
    // Link alternate nodes of ascending linked list
    ascn->next = curr;
    ascn = ascn->next;
    curr = curr->next;

    // Link alternate nodes of descending linked list
    if (curr)
    {
        dscn->next = curr;
        dscn = dscn->next;
        curr = curr->next;
    }
}

ascn->next = NULL;
dscn->next = NULL;
*Ahead = (*Ahead)->next;
*Dhead = (*Dhead)->next;
}

// Driver program to test above function
int main()
{
    Node *head = newNode(10);
    head->next = newNode(40);
    head->next->next = newNode(53);
    head->next->next->next = newNode(30);
    head->next->next->next->next = newNode(67);
    head->next->next->next->next->next = newNode(12);
    head->next->next->next->next->next = newNode(89);

    cout << "Given Linked List is " << endl;
    printlist(head);

    sort(&head);

    cout << "Sorted Linked List is " << endl;
    printlist(head);

    return 0;
}
```

Java

```
// Java program to sort a linked list that is alternatively
// sorted in increasing and decreasing order
class LinkedList
{
    Node head; // head of list

    /* Linked list Node*/
    class Node
    {
        int data;
        Node next;
        Node(int d) { data = d; next = null; }
    }

    Node newNode(int key)
    {
        return new Node(key);
    }

    /* This is the main function that sorts
       the linked list.*/
    void sort()
    {
        /* Create 2 dummy nodes and initialise as
           heads of linked lists */
        Node Ahead = new Node(0), Dhead = new Node(0);

        // Split the list into lists
        splitList(Ahead, Dhead);

        Ahead = Ahead.next;
        Dhead = Dhead.next;

        // reverse the descending list
        Dhead = reverseList(Dhead);

        // merge the 2 linked lists
        head = mergeList(Ahead,Dhead);
    }

    /* Function to reverse the linked list */
    Node reverseList(Node Dhead)
    {
        Node current = Dhead;
        Node prev = null;
        Node next;
        while (current != null)
        {
```

```
        next = current.next;
        current.next = prev;
        prev = current;
        current = next;
    }
    Dhead = prev;
    return Dhead;
}

/* Function to print linked list */
void printList()
{
    Node temp = head;
    while (temp != null)
    {
        System.out.print(temp.data+" ");
        temp = temp.next;
    }
    System.out.println();
}

// A utility function to merge two sorted linked lists
Node mergeList(Node head1, Node head2)
{
    // Base cases
    if (head1 == null) return head2;
    if (head2 == null) return head1;

    Node temp = null;
    if (head1.data < head2.data)
    {
        temp = head1;
        head1.next = mergeList(head1.next, head2);
    }
    else
    {
        temp = head2;
        head2.next = mergeList(head1, head2.next);
    }
    return temp;
}

// This function alternatively splits a linked list with head
// as head into two:
// For example, 10->20->30->15->40->7 is splitted into 10->30->40
// and 20->15->7
// "Ahead" is reference to head of ascending linked list
// "Dhead" is reference to head of descending linked list
```

```
void splitList(Node Ahead, Node Dhead)
{
    Node ascn = Ahead;
    Node dscn = Dhead;
    Node curr = head;

    // Link alternate nodes

    while (curr != null)
    {
        // Link alternate nodes in ascending order
        ascn.next = curr;
        ascn = ascn.next;
        curr = curr.next;

        if (curr != null)
        {
            dscn.next = curr;
            dscn = dscn.next;
            curr = curr.next;
        }
    }

    ascn.next = null;
    dscn.next = null;
}

/* Drier program to test above functions */
public static void main(String args[])
{
    LinkedList llist = new LinkedList();
    llist.head = llist.newNode(10);
    llist.head.next = llist.newNode(40);
    llist.head.next.next = llist.newNode(53);
    llist.head.next.next.next = llist.newNode(30);
    llist.head.next.next.next.next = llist.newNode(67);
    llist.head.next.next.next.next.next = llist.newNode(12);
    llist.head.next.next.next.next.next = llist.newNode(89);

    System.out.println("Given linked list");
    llist.printList();

    llist.sort();

    System.out.println("Sorted linked list");
    llist.printList();
}
```

```
} /* This code is contributed by Rajat Mishra */
```

### Python

```
# Python program to sort a linked list that is alternatively
# sorted in increasing and decreasing order
class LinkedList(object):
    def __init__(self):
        self.head = None

    # Linked list Node
    class Node(object):
        def __init__(self, d):
            self.data = d
            self.next = None

    def newNode(self, key):
        return self.Node(key)

    # This is the main function that sorts
    # the linked list.
    def sort(self):
        # Create 2 dummy nodes and initialise as
        # heads of linked lists
        Ahead = self.Node(0)
        Dhead = self.Node(0)
        # Split the list into lists
        self.splitList(Ahead, Dhead)
        Ahead = Ahead.next
        Dhead = Dhead.next
        # reverse the descending list
        Dhead = self.reverseList(Dhead)
        # merge the 2 linked lists
        self.head = self.mergeList(Ahead, Dhead)

    # Function to reverse the linked list
    def reverseList(self, Dhead):
        current = Dhead
        prev = None
        while current != None:
            self._next = current.next
            current.next = prev
            prev = current
            current = self._next
        Dhead = prev
        return Dhead

    # Function to print linked list
```

```
def printList(self):
    temp = self.head
    while temp != None:
        print temp.data,
        temp = temp.next
    print ''

# A utility function to merge two sorted linked lists
def mergeList(self, head1, head2):
    # Base cases
    if head1 == None:
        return head2
    if head2 == None:
        return head1
    temp = None
    if head1.data < head2.data:
        temp = head1
        head1.next = self.mergeList(head1.next, head2)
    else:
        temp = head2
        head2.next = self.mergeList(head1, head2.next)
    return temp

# This function alternatively splits a linked list with head
# as head into two:
# For example, 10->20->30->15->40->7 is splitted into 10->30->40
# and 20->15->7
# "Ahead" is reference to head of ascending linked list
# "Dhead" is reference to head of descending linked list
def splitList(self, Ahead, Dhead):
    ascn = Ahead
    dscn = Dhead
    curr = self.head
    # Link alternate nodes
    while curr != None:
        # Link alternate nodes in ascending order
        ascn.next = curr
        ascn = ascn.next
        curr = curr.next
        if curr != None:
            dscn.next = curr
            dscn = dscn.next
            curr = curr.next
    ascn.next = None
    dscn.next = None

# Driver program
llist = LinkedList()
```

*Chapter 236. Sort a linked list that is sorted alternating ascending and descending orders?*

---

```
llist.head = llist.newNode(10)
llist.head.next = llist.newNode(40)
llist.head.next.next = llist.newNode(53)
llist.head.next.next.next = llist.newNode(30)
llist.head.next.next.next.next = llist.newNode(67)
llist.head.next.next.next.next.next = llist.newNode(12)
llist.head.next.next.next.next.next.next = llist.newNode(89)

print 'Given linked list'
llist.printList()

llist.sort()

print 'Sorted linked list'
llist.printList()

# This code is contributed by BHAVYA JAIN
```

Output:

```
Given Linked List is
10 40 53 30 67 12 89
Sorted Linked List is
10 12 30 40 53 67 89
```

Thanks to Gaurav Ahirwar for suggesting this method.

## Source

<https://www.geeksforgeeks.org/how-to-sort-a-linked-list-that-is-sorted-alternating-ascending-and-descending-order/>

## Chapter 237

# Sort linked list which is already sorted on absolute values

Sort linked list which is already sorted on absolute values - GeeksforGeeks

Given a linked list which is sorted based on absolute values. Sort the list based on actual values.

Examples:

Input : 1 → -10  
output: -10 → 1

Input : 1 → -2 → -3 → 4 → -5  
output: -5 → -3 → -2 → 1 → 4

Input : -5 → -10  
Output: -10 → -5

Input : 5 → 10  
output: 5 → 10

Source : [Amazon Interview](#)

A simple solution is to traverse the linked list from beginning to end. For every visited node, check if it is out of order. If it is, remove it from its current position and insert at correct position. This is implementation of [insertion sort for linked list](#) and time complexity of this solution is  $O(n^2)$ .

A better solution is to [sort the linked list using merge sort](#). Time complexity of this solution is  $O(n \log n)$ .

An efficient solution can work in  $O(n)$  time. An important observation is, all negative elements are present in reverse order. So we traverse the list, whenever we find an element that is out of order, we move it to the front of linked list.

Below is C++ implementation of above idea.

C++

```
// C++ program to sort a linked list, already
// sorted by absolute values
#include <bits/stdc++.h>
using namespace std;

// Linked List Node
struct Node
{
    Node* next;
    int data;
};

// Utility function to insert a node at the
// beginning
void push(Node** head, int data)
{
    Node* newNode = new Node;
    newNode->next = (*head);
    newNode->data = data;
    (*head) = newNode;
}

// Utility function to print a linked list
void printList(Node* head)
{
    while (head != NULL)
    {
        cout << head->data;
        if (head->next != NULL)
            cout << " -> ";
        head = head->next;
    }
    cout<<endl;
}

// To sort a linked list by actual values.
// The list is assumed to be sorted by absolute
// values.
void sortList(Node** head)
{
    // Initialize previous and current nodes
    Node* prev = (*head);
    Node* curr = (*head)->next;
```

```
// Traverse list
while (curr != NULL)
{
    // If curr is smaller than prev, then
    // it must be moved to head
    if (curr->data < prev->data)
    {
        // Detach curr from linked list
        prev->next = curr->next;

        // Move current node to beginning
        curr->next = (*head);
        (*head) = curr;

        // Update current
        curr = prev;
    }

    // Nothing to do if current element
    // is at right place
    else
        prev = curr;

    // Move current
    curr = curr->next;
}
}

// Driver code
int main()
{
    Node* head = NULL;
    push(&head, -5);
    push(&head, 5);
    push(&head, 4);
    push(&head, 3);
    push(&head, -2);
    push(&head, 1);
    push(&head, 0);

    cout << "Original list :\n";
    printList(head);

    sortList(&head);

    cout << "\nSorted list :\n";
    printList(head);
```

```
    return 0;
}
```

**Java**

```
// Java program to sort a linked list, already
// sorted by absolute values
class SortList
{
    static Node head; // head of list

    /* Linked list Node*/
    static class Node
    {
        int data;
        Node next;
        Node(int d) {data = d; next = null; }
    }

    // To sort a linked list by actual values.
    // The list is assumed to be sorted by absolute
    // values.
    Node sortedList(Node head)
    {
        // Initialize previous and current nodes
        Node prev = head;
        Node curr = head.next;

        // Traverse list
        while(curr != null)
        {
            // If curr is smaller than prev, then
            // it must be moved to head
            if(curr.data < prev.data)
            {
                // Detach curr from linked list
                prev.next = curr.next;

                // Move current node to beginning
                curr.next = head;
                head = curr;

                // Update current
                curr = prev;
            }

            // Nothing to do if current element
            // is at right place
        }
    }
}
```

```
        else
            prev = curr;

        // Move current
        curr = curr.next;
    }
    return head;
}

/* Inserts a new Node at front of the list. */
public void push(int new_data)
{
    /* 1 & 2: Allocate the Node &
       Put in the data*/
    Node new_node = new Node(new_data);

    /* 3. Make next of new Node as head */
    new_node.next = head;

    /* 4. Move the head to point to new Node */
    head = new_node;
}

/* Function to print linked list */
void printList(Node head)
{
    Node temp = head;
    while (temp != null)
    {
        System.out.print(temp.data+" ");
        temp = temp.next;
    }
    System.out.println();
}

/* Driver program to test above functions */
public static void main(String args[])
{
    SortList llist = new SortList();

    /* Constructed Linked List is 1->2->3->4->5->6->
       7->8->8->9->null */
    llist.push(-5);
    llist.push(5);
    llist.push(4);
    llist.push(3);
    llist.push(-2);
    llist.push(1);
```

```
llist.push(0);

System.out.println("Original List :");
llist.printList(llist.head);

llist.head = llist.sortedList(head);

System.out.println("Sorted list :");
llist.printList(llist.head);
}

}

// This code has been contributed by Amit Khandelwal(Amit Khandelwal 1).
```

Output:

```
Original list :
0 -> 1 -> -2 -> 3 -> 4 -> 5 -> -5

Sorted list :
-5 -> -2 -> 0 -> 1 -> 3 -> 4 -> 5
```

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

## Source

<https://www.geeksforgeeks.org/sort-linked-list-already-sorted-absolute-values/>

## Chapter 238

# Sort the biotonic doubly linked list

Sort the biotonic doubly linked list - GeeksforGeeks

Sort the given biotonic doubly linked list. A biotonic doubly linked list is a doubly linked list which is first increasing and then decreasing. A strictly increasing or a strictly decreasing list is also a biotonic doubly linked list.

Examples:

```
Input : DLL: 2<->5<->7<->12<->10<->6<->4<->1
Output : 1<->2<->4<->5<->6<->7<->10<->12
```

```
Input : DLL: 20<->17<->14<->8<->3
Output : 3<->8<->14<->17<->20
```

**Approach:** Find the first node in the list which is smaller than its previous node. Let it be **current**. If no such node is present then list is already sorted. Else split the list into two lists, **first** starting from **head** node till the **current**'s previous node and **second** starting from **current** node till the end of the list. Reverse the **second** doubly linked list. Refer [this](#) post. Now merge the **first** and **second** sorted doubly linked list. Refer merge procedure of [this](#) post. The final merged list is the required sorted doubly linked list.

```
// C++ implementation to sort the biotonic doubly linked list
#include <bits/stdc++.h>

using namespace std;

// a node of the doubly linked list
struct Node {
    int data;
    struct Node* next;
    struct Node* prev;
}
```

```
    struct Node* prev;
};

// Function to reverse a Doubly Linked List
void reverse(struct Node** head_ref)
{
    struct Node* temp = NULL;
    struct Node* current = *head_ref;

    // swap next and prev for all nodes
    // of doubly linked list
    while (current != NULL) {
        temp = current->prev;
        current->prev = current->next;
        current->next = temp;
        current = current->prev;
    }

    // Before changing head, check for the cases
    // like empty list and list with only one node
    if (temp != NULL)
        *head_ref = temp->prev;
}

// Function to merge two sorted doubly linked lists
struct Node* merge(struct Node* first, struct 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->next = merge(first->next, second);
        first->next->prev = first;
        first->prev = NULL;
        return first;
    } else {
        second->next = merge(first, second->next);
        second->next->prev = second;
        second->prev = NULL;
        return second;
    }
}
```

```
// function to sort a biotonic doubly linked list
struct Node* sort(struct Node* head)
{
    // if list is empty or if it contains a single
    // node only
    if (head == NULL || head->next == NULL)
        return head;

    struct Node* current = head->next;

    while (current != NULL) {

        // if true, then 'current' is the first node
        // which is smaller than its previous node
        if (current->data < current->prev->data)
            break;

        // move to the next node
        current = current->next;
    }

    // if true, then list is already sorted
    if (current == NULL)
        return head;

    // spilt into two lists, one starting with 'head'
    // and other starting with 'current'
    current->prev->next = NULL;
    current->prev = NULL;

    // reverse the list starting with 'current'
    reverse(&current);

    // merge the two lists and return the
    // final merged doubly linked list
    return merge(head, current);
}

// Function to insert a node at the beginning
// of the Doubly Linked List
void push(struct Node** head_ref, int new_data)
{
    // allocate node
    struct Node* new_node =
        (struct Node*)malloc(sizeof(struct Node));

    // put in the data
```

```
new_node->data = new_data;

// since we are adding at the begining,
// prev is always NULL
new_node->prev = NULL;

// link the old list off the new node
new_node->next = (*head_ref);

// change prev of head node to new node
if ((*head_ref) != NULL)
    (*head_ref)->prev = new_node;

// move the head to point to the new node
(*head_ref) = new_node;
}

// Function to print nodes in a given doubly
// linked list
void printList(struct Node* head)
{
    // if list is empty
    if (head == NULL)
        cout << "Doubly Linked list empty";

    while (head != NULL) {
        cout << head->data << " ";
        head = head->next;
    }
}

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

    // Create the doubly linked list:
    // 2<->5<->7<->12<->10<->6<->4<->1
    push(&head, 1);
    push(&head, 4);
    push(&head, 6);
    push(&head, 10);
    push(&head, 12);
    push(&head, 7);
    push(&head, 5);
    push(&head, 2);

    cout << "Original Doubly linked list:n";
```

```
printList(head);

// sort the biotonic DLL
head = sort(head);

cout << "\nDoubly linked list after sorting:";
printList(head);

return 0;
}
```

Output:

```
Original Doubly linked list:
2 5 7 12 10 6 4 1
Doubly linked list after sorting:
1 2 4 5 6 7 10 12
```

Time Complexity: O(n)

## Source

<https://www.geeksforgeeks.org/sort-biotonic-doubly-linked-list/>

## Chapter 239

# Sort the bitonic doubly linked list | Set-2

Sort the bitonic doubly linked list | Set-2 - GeeksforGeeks

Sort the given bitonic doubly linked list. A bitonic doubly linked list is a doubly linked list which is first increasing and then decreasing. A strictly increasing or a strictly decreasing list is also a bitonic doubly linked list.

Examples:

Input : 2 5 7 12 10 6 4 1  
Output : 1 2 4 5 6 7 10 12

Input : 20 17 14 8 3  
Output : 3 8 14 17 20

In the [previous post](#), we split the bitonic doubly linked list, reverse the second half and then merge both halves. In this post another alternative method is discussed. The idea is to maintain two pointers, one pointing to head element initially and other pointing to last element of doubly linked list. Compare both the elements and add the smaller element to result list. Advance pointer of that element to next adjacent element. Repeat this until all elements of input doubly linked list are added to result list.

Below is C++ implementation of above algorithm:

```
// C++ implementation to sort the bitonic
// doubly linked list

#include <bits/stdc++.h>
using namespace std;
```

```
// structure of node of the doubly linked list
struct Node {
    int data;
    struct Node* next;
    struct Node* prev;
};

// function to sort a biotonic doubly linked list
struct Node* sort(struct Node* head)
{
    // If number of elements are less than or
    // equal to 1 then return.
    if (head == NULL || head->next == NULL) {
        return head;
    }

    // Pointer to first element of doubly
    // linked list.
    Node* front = head;

    // Pointer to last element of doubly
    // linked list.
    Node* last = head;

    // Dummy node to which resultant
    // sorted list is added.
    Node* res = new Node;

    // Node after which next element
    // of sorted list is added.
    Node* resEnd = res;

    // Node to store next element to
    // which pointer is moved after
    // element pointed by that pointer
    // is added to result list.
    Node* next;

    // Find last element of input list.
    while (last->next != NULL) {
        last = last->next;
    }

    // Compare first and last element
    // until both pointers are not equal.
    while (front != last) {

        // If last element data is less than
```

```
// or equal to front element data,
// then add last element to
// result list and change the
// last pointer to its previous
// element.
if (last->data <= front->data) {
    resEnd->next = last;
    next = last->prev;
    last->prev->next = NULL;
    last->prev = resEnd;
    last = next;
    resEnd = resEnd->next;
}

// If front element is smaller, then
// add it to result list and change
// front pointer to its next element.
else {
    resEnd->next = front;
    next = front->next;
    front->next = NULL;
    front->prev = resEnd;
    front = next;
    resEnd = resEnd->next;
}
}

// Add the single element left to the
// result list.
resEnd->next = front;
front->prev = resEnd;

// The head of required sorted list is
// next to dummy node res.
return res->next;
}

// Function to insert a node at the beginning
// of the Doubly Linked List
void push(struct Node** head_ref, int new_data)
{
    // allocate node
    struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));

    // put in the data
    new_node->data = new_data;

    // since we are adding at the begining,
```

```
// prev is always NULL
new_node->prev = NULL;

// link the old list off the new node
new_node->next = (*head_ref);

// change prev of head node to new node
if ((*head_ref) != NULL)
    (*head_ref)->prev = new_node;

// move the head to point to the new node
(*head_ref) = new_node;
}

// Function to print nodes in a given doubly
// linked list
void printList(struct Node* head)
{
    // if list is empty
    if (head == NULL)
        cout << "Doubly Linked list empty";

    while (head != NULL) {
        cout << head->data << " ";
        head = head->next;
    }
}

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

    // Create the doubly linked list:
    // 2<->5<->7<->12<->10<->6<->4<->1
    push(&head, 1);
    push(&head, 4);
    push(&head, 6);
    push(&head, 10);
    push(&head, 12);
    push(&head, 7);
    push(&head, 5);
    push(&head, 2);

    cout << "Original Doubly linked list:\n";
    printList(head);

    // sort the biotonic DLL
```

```
head = sort(head);

cout << "\nDoubly linked list after sorting:\n";
printList(head);

return 0;
}
```

**Output:**

```
Original Doubly linked list:
2 5 7 12 10 6 4 1
Doubly linked list after sorting:
1 2 4 5 6 7 10 12
```

**Time Complexity:** O(n)

**Auxiliary Space:** O(1)

## Source

<https://www.geeksforgeeks.org/sort-the-biotonic-doubly-linked-list-set-2/>

## Chapter 240

# Sort the linked list in the order of elements appearing in the array

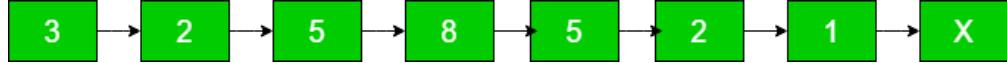
Sort the linked list in the order of elements appearing in the array - GeeksforGeeks

Given an array of size N and a Linked List where elements will be from the array but can also be duplicated, sort the linked list in the order, elements are appearing in the array. It may be assumed that the array covers all elements of the linked list.

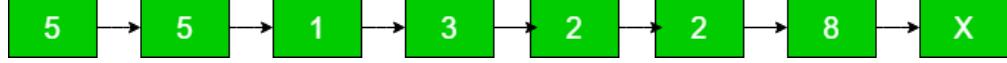
arr[] =

5	1	3	2	8
---	---	---	---	---

list =



Sorted list =



Asked in [Amazon](#)

First, make a hash table that stores the frequencies of elements in linked list. Then, simply traverse list and for each element of arr[i] check the frequency in the has table and modify the data of list by arr[i] element upto its frequency and at last Print the list.

```
// Efficient CPP program to sort given list in order
// elements are appearing in an array
#include <bits/stdc++.h>
using namespace std;
```

```
// Linked list node
struct Node {
    int data;
    struct Node* next;
};

// function prototype for printing the list
void printList(struct Node*);

// Function to insert a node at the
// beginning of the linked list
void push(struct Node** head_ref, int new_data)
{
    struct Node* new_node = new Node;
    new_node -> data = new_data;
    new_node -> next = *head_ref;
    *head_ref = new_node;
}

// function to print the linked list
void printList(struct Node* head)
{
    while (head != NULL) {
        cout << head -> data << " -> ";
        head = head -> next;
    }
}

// Function that sort list in order of apperaing
// elements in an array
void sortlist(int arr[], int N, struct Node* head)
{
    // Store frequencies of elements in a
    // hash table.
    unordered_map<int, int> hash;
    struct Node* temp = head;
    while (temp) {
        hash[temp -> data]++;
        temp = temp -> next;
    }

    temp = head;

    // One by one put elements in lis according
    // to their appearance in array
    for (int i = 0; i < N; i++) {

        // Update 'frequency' nodes with value
```

```
// equal to arr[i]
int frequency = hash[arr[i]];
while (frequency--) {

    // Modify list data as element
    // appear in an array
    temp -> data = arr[i];
    temp = temp -> next;
}
}

// Driver Code
int main()
{
    struct Node* head = NULL;
    int arr[] = { 5, 1, 3, 2, 8 };
    int N = sizeof(arr) / sizeof(arr[0]);

    // creating the linked list
    push(&head, 3);
    push(&head, 2);
    push(&head, 5);
    push(&head, 8);
    push(&head, 5);
    push(&head, 2);
    push(&head, 1);

    // Function call to sort the list in order
    // elements are apperaing in an array
    sortlist(arr, N, head);

    // print the modified linked list
    cout << "Sorted List:" << endl;
    printList(head);
    return 0;
}
```

Output :

```
Sort list:
5 -> 5 -> 1 -> 3 -> 2 -> 2 -> 8
```

## Source

<https://www.geeksforgeeks.org/sort-linked-list-order-elements-appearing-array/>

## Chapter 241

# Sorted Linked List to Balanced BST

Sorted Linked List to Balanced BST - GeeksforGeeks

Given a Singly Linked List which has data members sorted in ascending order. Construct a [Balanced Binary Search Tree](#) which has same data members as the given Linked List.

Examples:

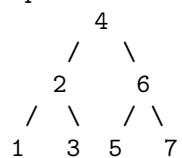
Input: Linked List 1->2->3

Output: A Balanced BST



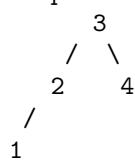
Input: Linked List 1->2->3->4->5->6->7

Output: A Balanced BST



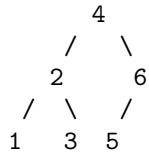
Input: Linked List 1->2->3->4

Output: A Balanced BST



Input: Linked List 1->2->3->4->5->6

Output: A Balanced BST



### Method 1 (Simple)

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 linked list 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.

Time complexity:  $O(n\log n)$  where  $n$  is the number of nodes in Linked List.

### Method 2 (Tricky)

The method 1 constructs the tree from root to leaves. In this method, we construct from leaves to root. The idea is to insert nodes in BST in the same order as they appear in Linked List, so that the tree can be constructed in  $O(n)$  time complexity. We first count the number of nodes in the given Linked List. Let the count be  $n$ . After counting nodes, we take left  $n/2$  nodes and recursively construct the left subtree. After left subtree is constructed, we allocate memory for root and link the left subtree with root. Finally, we recursively construct the right subtree and link it with root.

While constructing the BST, we also keep moving the list head pointer to next so that we have the appropriate pointer in each recursive call.

Following is C implementation of method 2. The main code which creates Balanced BST is highlighted.

C

```

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

/* Link list node */
struct LNode
{
    int data;
    struct LNode* next;
};
  
```

```
/* A Binary Tree node */
struct TNode
{
    int data;
    struct TNode* left;
    struct TNode* right;
};

struct TNode* newNode(int data);
int countLNodes(struct LNode *head);
struct TNode* sortedListToBSTRecur(struct LNode **head_ref, int n);

/* This function counts the number of nodes in Linked List and then calls
   sortedListToBSTRecur() to construct BST */
struct TNode* sortedListToBST(struct LNode *head)
{
    /*Count the number of nodes in Linked List */
    int n = countLNodes(head);

    /* Construct BST */
    return sortedListToBSTRecur(&head, n);
}

/* The main function that constructs balanced BST and returns root of it.
   head_ref --> Pointer to pointer to head node of linked list
   n --> No. of nodes in Linked List */
struct TNode* sortedListToBSTRecur(struct LNode **head_ref, int n)
{
    /* Base Case */
    if (n <= 0)
        return NULL;

    /* Recursively construct the left subtree */
    struct TNode *left = sortedListToBSTRecur(head_ref, n/2);

    /* Allocate memory for root, and link the above constructed left
       subtree with root */
    struct TNode *root = newNode((*head_ref)->data);
    root->left = left;

    /* Change head pointer of Linked List for parent recursive calls */
    *head_ref = (*head_ref)->next;

    /* Recursively construct the right subtree and link it with root
       The number of nodes in right subtree is total nodes - nodes in
       left subtree - 1 (for root) which is n-n/2-1*/
}
```

```
root->right = sortedListToBSTRecur(head_ref, n-n/2-1);

return root;
}

/* UTILITY FUNCTIONS */

/* A utility function that returns count of nodes in a given Linked List */
int countLNodes(struct LNode *head)
{
    int count = 0;
    struct LNode *temp = head;
    while(temp)
    {
        temp = temp->next;
        count++;
    }
    return count;
}

/* Function to insert a node at the beginning of the linked list */
void push(struct LNode** head_ref, int new_data)
{
    /* allocate node */
    struct LNode* new_node =
        (struct LNode*) malloc(sizeof(struct LNode));
    /* put in the data */
    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;
}

/* Function to print nodes in a given linked list */
void printList(struct LNode *node)
{
    while(node!=NULL)
    {
        printf("%d ", node->data);
        node = node->next;
    }
}
```

```
/* 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()
{
    /* Start with the empty list */
    struct LNode* head = NULL;

    /* Let us create a sorted linked list to test the functions
       Created linked list will be 1->2->3->4->5->6->7 */
    push(&head, 7);
    push(&head, 6);
    push(&head, 5);
    push(&head, 4);
    push(&head, 3);
    push(&head, 2);
    push(&head, 1);

    printf("\n Given Linked List ");
    printList(head);

    /* Convert List to BST */
    struct TNode *root = sortedListToBST(head);
    printf("\n PreOrder Traversal of constructed BST ");
    preOrder(root);

    return 0;
}
```

}

**Java**

```
class LinkedList {

    /* head node of link list */
    static LNode head;

    /* Link list Node */
    class LNode
    {
        int data;
        LNode next, prev;

        LNode(int d)
        {
            data = d;
            next = prev = null;
        }
    }

    /* A Binary Tree Node */
    class TNode
    {
        int data;
        TNode left, right;

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

    /* This function counts the number of nodes in Linked List
       and then calls sortedListToBSTRecur() to construct BST */
    TNode sortedListToBST()
    {
        /*Count the number of nodes in Linked List */
        int n = countNodes(head);

        /* Construct BST */
        return sortedListToBSTRecur(n);
    }

    /* The main function that constructs balanced BST and
       returns root of it.
```

```
n --> No. of nodes in the Doubly Linked List */
TNode sortedListToBSTRecur(int n)
{
    /* Base Case */
    if (n <= 0)
        return null;

    /* Recursively construct the left subtree */
    TNode left = sortedListToBSTRecur(n / 2);

    /* head_ref now refers to middle node,
       make middle node as root of BST*/
    TNode root = new TNode(head.data);

    // Set pointer to left subtree
    root.left = left;

    /* Change head pointer of Linked List for parent
       recursive calls */
    head = head.next;

    /* Recursively construct the right subtree and link it
       with root. The number of nodes in right subtree is
       total nodes - nodes in left subtree - 1 (for root) */
    root.right = sortedListToBSTRecur(n - n / 2 - 1);

    return root;
}

/* UTILITY FUNCTIONS */
/* A utility function that returns count of nodes in a
   given Linked List */
int countNodes(LNode head)
{
    int count = 0;
    LNode temp = head;
    while (temp != null)
    {
        temp = temp.next;
        count++;
    }
    return count;
}

/* Function to insert a node at the beginning of
   the Doubly Linked List */
void push(int new_data)
{
```

```
/* allocate node */
LNode new_node = new LNode(new_data);

/* since we are adding at the begining,
   prev is always NULL */
new_node.prev = null;

/* link the old list off the new node */
new_node.next = head;

/* change prev of head node to new node */
if (head != null)
    head.prev = new_node;

/* move the head to point to the new node */
head = new_node;
}

/* Function to print nodes in a given linked list */
void printList(LNode node)
{
    while (node != null)
    {
        System.out.print(node.data + " ");
        node = node.next;
    }
}

/* A utility function to print preorder traversal of BST */
void preOrder(TNode node)
{
    if (node == null)
        return;
    System.out.print(node.data + " ");
    preOrder(node.left);
    preOrder(node.right);
}

/* Drier program to test above functions */
public static void main(String[] args) {
    LinkedList llist = new LinkedList();

    /* Let us create a sorted linked list to test the functions
       Created linked list will be 7->6->5->4->3->2->1 */
    llist.push(7);
    llist.push(6);
    llist.push(5);
    llist.push(4);
```

```
llist.push(3);
llist.push(2);
llist.push(1);

System.out.println("Given Linked List ");
llist.printList(head);

/* Convert List to BST */
TNode root = llist.sortedListToBST();
System.out.println("");
System.out.println("Pre-Order Traversal of constructed BST ");
llist.preOrder(root);
}

}

// This code has been contributed by Mayank Jaiswal(mayank_24)
```

**Output:**

```
Given Linked List 1 2 3 4 5 6 7
PreOrder Traversal of constructed BST 4 2 1 3 6 5 7
```

Time Complexity: O(n)

Improved By : [Sulav Timsina](#)

**Source**

<https://www.geeksforgeeks.org/sorted-linked-list-to-balanced-bst/>

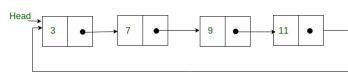
## Chapter 242

# Sorted insert for circular linked list

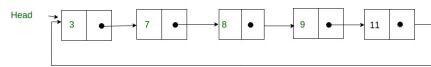
Sorted insert for circular linked list - GeeksforGeeks

**Difficulty Level:** Rookie

Write a C function to insert a new value in a sorted Circular Linked List (CLL). For example, if the input CLL is following.



After inserting 8, the above CLL should be changed to the following



### Algorithm:

Allocate memory for the newly inserted node and put data in the newly allocated node. Let the pointer to the new node be new\_node. After memory allocation, following are the three cases that need to be handled.

- 1) Linked List is empty:
  - a) since new\_node is the only node in CLL, make a self loop.  
`new_node->next = new_node;`
  - b) change the head pointer to point to new node.  
`*head_ref = new_node;`
- 2) New node is to be inserted just before the head node:
  - (a) Find out the last node using a loop.  
`while(current->next != *head_ref)  
 current = current->next;`
  - (b) Change the next of last node.  
`current->next = new_node;`

```

(c) Change next of new node to point to head.
    new_node->next = *head_ref;
(d) change the head pointer to point to new node.
    *head_ref = new_node;
3) New node is to be inserted somewhere after the head:
(a) Locate the node after which new node is to be inserted.
    while ( current->next != *head_ref &&
            current->next->data < new_node->data)
    {
        current = current->next;
    }
(b) Make next of new_node as next of the located pointer
    new_node->next = current->next;
(c) Change the next of the located pointer
    current->next = new_node;

```

C

```

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

/* structure for a node */
struct Node
{
    int data;
    struct Node *next;
};

/* function to insert a new_node in a list in sorted way.
   Note that this function expects a pointer to head node
   as this can modify the head of the input linked list */
void sortedInsert(struct Node** head_ref, struct Node* new_node)
{
    struct Node* current = *head_ref;

    // Case 1 of the above algo
    if (current == NULL)
    {
        new_node->next = new_node;
        *head_ref = new_node;
    }

    // Case 2 of the above algo
    else if (current->data >= new_node->data)
    {
        /* If value is smaller than head's value then
           we need to change next of last node */
        while(current->next != *head_ref)
            current = current->next;
        current->next = new_node;
    }
}

```

```
new_node->next = *head_ref;
*head_ref = new_node;
}

// Case 3 of the above algo
else
{
    /* Locate the node before the point of insertion */
    while (current->next != *head_ref &&
           current->next->data < new_node->data)
        current = current->next;

    new_node->next = current->next;
    current->next = new_node;
}
}

/* Function to print nodes in a given linked list */
void printList(struct Node *start)
{
    struct Node *temp;

    if(start != NULL)
    {
        temp = start;
        printf("\n");
        do {
            printf("%d ", temp->data);
            temp = temp->next;
        } while(temp != start);
    }
}

/* Driver program to test above functions */
int main()
{
    int arr[] = {12, 56, 2, 11, 1, 90};
    int list_size, i;

    /* start with empty linked list */
    struct Node *start = NULL;
    struct Node *temp;

    /* Create linked list from the array arr[].
     * Created linked list will be 1->2->11->12->56->90 */
    for (i = 0; i < 6; i++)
    {
        temp = (struct Node *)malloc(sizeof(struct Node));
        temp->
```

```
    temp->data = arr[i];
    sortedInsert(&start, temp);
}

printList(start);

return 0;
}
```

**Java**

```
// Java program for sorted insert in circular linked list

class Node
{
    int data;
    Node next;

    Node(int d)
    {
        data = d;
        next = null;
    }
}

class LinkedList
{
    Node head;

    // Constructor
    LinkedList() { head = null; }

    /* function to insert a new_node in a list in sorted way.
     Note that this function expects a pointer to head node
     as this can modify the head of the input linked list */
    void sortedInsert(Node new_node)
    {
        Node current = head;

        // Case 1 of the above algo
        if (current == null)
        {
            new_node.next = new_node;
            head = new_node;
        }

        // Case 2 of the above algo
```

```
else if (current.data >= new_node.data)
{
    /* If value is smaller than head's value then
       we need to change next of last node */
    while (current.next != head)
        current = current.next;

    current.next = new_node;
    new_node.next = head;
    head = new_node;
}

// Case 3 of the above algo
else
{
    /* Locate the node before the point of insertion */
    while (current.next != head &&
           current.next.data < new_node.data)
        current = current.next;

    new_node.next = current.next;
    current.next = new_node;
}
}

// Utility method to print a linked list
void printList()
{
    if (head != null)
    {
        Node temp = head;
        do
        {
            System.out.print(temp.data + " ");
            temp = temp.next;
        } while (temp != head);
    }
}

// Driver code to test above
public static void main(String[] args)
{
    LinkedList list = new LinkedList();

    // Creating the linkedlist
    int arr[] = new int[] {12, 56, 2, 11, 1, 90};
```

```
/* start with empty linked list */
Node temp = null;

/* Create linked list from the array arr[] .
Created linked list will be 1->2->11->12->56->90*/
for (int i = 0; i < 6; i++)
{
    temp = new Node(arr[i]);
    list.sortedInsert(temp);
}

list.printList();
}
}

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

### Python

```
# Node class
class Node:

    # Constructor to initialize the node object
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:

    # Function to initialize head
    def __init__(self):
        self.head = None

    # Function to insert a new node at the beginning
    def push(self, new_data):
        new_node = Node(new_data)
        new_node.next = self.head
        self.head = new_node

    # Utility function to print the linked LinkedList
    def printList(self):
        temp = self.head
        print temp.data,
        temp = temp.next
        while(temp != self.head):
            print temp.data,
            temp = temp.next
```

```
""" function to insert a new_node in a list in sorted way.
Note that this function expects a pointer to head node
as this can modify the head of the input linked list """
def sortedInsert(self, new_node):

    current = self.head

    # Case 1 of the above algo
    if current is None:
        new_node.next = new_node
        self.head = new_node

    # Case 2 of the above algo
    elif (current.data >= new_node.data):

        # If value is smaller than head's value then we
        # need to change next of last node
        while current.next != self.head :
            current = current.next
        current.next = new_node
        new_node.next = self.head
        self.head = new_node

    # Case 3 of the above algo
    else:

        # Locate the node before the point of insertion
        while (current.next != self.head and
               current.next.data < new_node.data):
            current = current.next

        new_node.next = current.next
        current.next = new_node

# Driver program to test the above function
#llist = LinkedList()
arr = [12, 56, 2, 11, 1, 90]

list_size = len(arr)

# start with empty linked list
start = LinkedList()

# Create linked list from the array arr[]
# Created linked list will be 1->2->11->12->56->90
```

```
for i in range(list_size):
    temp = Node(arr[i])
    start.sortedInsert(temp)

start.printList()

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

Output:

```
1 2 11 12 56 90
```

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

Case 2 of the above algorithm/code can be optimized. To implement the suggested change we need to modify the case 2 to following.

```
// Case 2 of the above algo
else if (current->data >= new_node->data)
{
    // swap the data part of head node and new node
    // assuming that we have a function swap(int *, int *)
    swap(&(current->data), &(new_node->data));

    new_node->next = (*head_ref)->next;
    (*head_ref)->next = new_node;
}
```

## Source

<https://www.geeksforgeeks.org/sorted-insert-for-circular-linked-list/>

## Chapter 243

# Sorted insert in a doubly linked list with head and tail pointers

Sorted insert in a doubly linked list with head and tail pointers - GeeksforGeeks

A [Doubly linked list](#) is a linked list that consists of a set of sequentially linked records called nodes. Each node contains two fields that are references to the previous and to the next node in the sequence of nodes.

The task is to create a doubly linked list by inserting nodes such that list remains in ascending order on printing from left to right. Also, we need to maintain two pointers, head (points to first node) and tail (points to last node).

### Examples:

Input : 40 50 10 45 90 100 95  
Output :10 40 45 50 90 95 100

Input : 30 10 50 43 56 12  
Output :10 12 30 43 50 56

### Algorithm:

The task can be accomplished as:

1. If Linked list is empty then make both the left and right pointers point to the node to be inserted and make its previous and next field point to NULL.
2. If node to be inserted has value less than the value of first node of linked list then connect that node from previous field of first node.
3. If node to be inserted has value more than the value of last node of linked list then connect that node from next field of last node.
4. If node to be inserted has value in between the value of first and last node, then check for appropriate position and make connections.

```
/* C program to insetail nodes in doubly
linked list such that list remains in
ascending order on printing from left
to right */
#include<stdio.h>
#include<stdlib.h>

// A linked list node
struct Node
{
    struct Node *prev;
    int info;
    struct Node *next;
};

// Function to insetail new node
void nodeInsetail(struct Node **head,
                  struct Node **tail,
                  int key)
{

    struct Node *p = new Node;
    p->info = key;
    p->next = NULL;

    // If first node to be insetailed in doubly
    // linked list
    if ((*head) == NULL)
    {
        (*head) = p;
        (*tail) = p;
        (*head)->prev = NULL;
        return;
    }

    // If node to be insetailed has value less
    // than first node
    if ((p->info) < ((*head)->info))
    {
        p->prev = NULL;
        (*head)->prev = p;
        p->next = (*head);
        (*head) = p;
        return;
    }

    // If node to be insetailed has value more
    // than last node
```

```
if ((p->info) > ((*tail)->info))
{
    p->prev = (*tail);
    (*tail)->next = p;
    (*tail) = p;
    return;
}

// Find the node before which we need to
// insert p.
temp = (*head)->next;
while ((temp->info) < (p->info))
    temp = temp->next;

// Insert new node before temp
(temp->prev)->next = p;
p->prev = temp->prev;
temp->prev = p;
p->next = temp;
}

// Function to print nodes in from left to right
void printList(struct Node *temp)
{
    while (temp != NULL)
    {
        printf("%d ", temp->info);
        temp = temp->next;
    }
}

// Driver program to test above functions
int main()
{
    struct Node *left = NULL, *right = NULL;
    nodeInsetail(&left, &right, 30);
    nodeInsetail(&left, &right, 50);
    nodeInsetail(&left, &right, 90);
    nodeInsetail(&left, &right, 10);
    nodeInsetail(&left, &right, 40);
    nodeInsetail(&left, &right, 110);
    nodeInsetail(&left, &right, 60);
    nodeInsetail(&left, &right, 95);
    nodeInsetail(&left, &right, 23);

    printf("\nDoubly linked list on printing"
           " from left to right\n");
    printList(left);
```

```
    return 0;  
}
```

**Output:**

```
Doubly linked list on printing from left to right  
10 23 30 40 50 60 90 95 110
```

**Source**

<https://www.geeksforgeeks.org/create-doubly-linked-list-using-double-pointer-inserting-nodes-list-remains-ascending/>

## Chapter 244

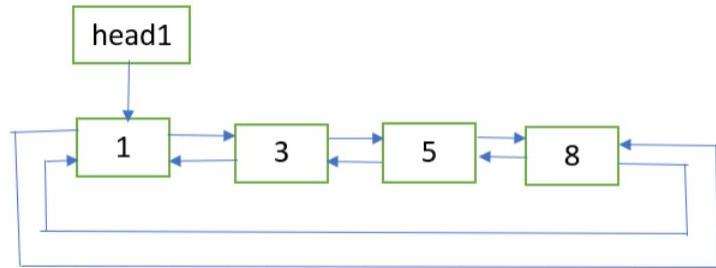
# Sorted merge of two sorted doubly circular linked lists

Sorted merge of two sorted doubly circular linked lists - GeeksforGeeks

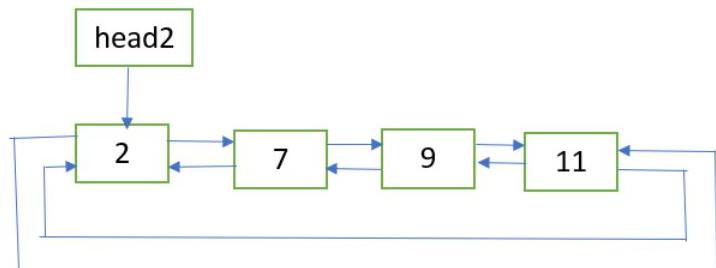
Given two sorted [Doubly circular Linked List](#) containing **n1** and **n2** nodes respectively. The problem is to merge the two lists such that resultant list is also in sorted order.

Example:

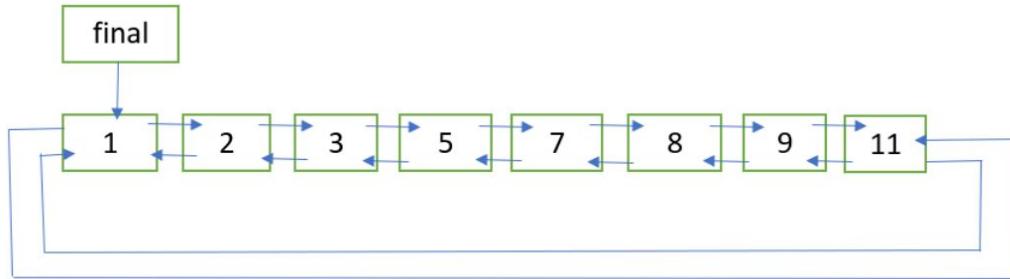
List 1:



List 2:



Final list:



**Approach:** Following are the steps:

1. If `head1 == NULL`, return `head2`.
2. If `head2 == NULL`, return `head1`.
3. Let `last1` and `last2` be the last nodes of the two lists respectively. They can be obtained with the help of the previous links of the first nodes.
4. Get pointer to the node which will be the last node of the final list. If `last1.data < last2.data`, then `last_node = last2`, Else `last_node = last1`.
5. Update `last1.next = last2.next = NULL`.
6. Now merge the two lists as two sorted doubly linked list are being merged. Refer `merge` procedure of [this](#) post. Let the first node of the final list be `finalHead`.
7. Update `finalHead.prev = last_node` and `last_node.next = finalHead`.
8. Return `finalHead`.

```
// C++ implementation for Sorted merge of two
// sorted doubly circular linked list
#include <bits/stdc++.h>
using namespace std;

struct Node {
    int data;
    Node *next, *prev;
};

// A utility function to insert a new node at the
// beginning of doubly circular linked list
void insert(Node** head_ref, int data)
{
    // allocate space
    Node* new_node = new Node;

    // put in the data
    new_node->data = data;

    // if list is empty
    if (*head_ref == NULL)
```

```
if (*head_ref == NULL) {
    new_node->next = new_node;
    new_node->prev = new_node;
}

else {

    // pointer points to last Node
    Node* last = (*head_ref)->prev;

    // setting up previous and next of new node
    new_node->next = *head_ref;
    new_node->prev = last;

    // update next and previous pointers of head_ref
    // and last.
    last->next = (*head_ref)->prev = new_node;
}

// update head_ref pointer
*head_ref = new_node;
}

// function for Sorted merge of two
// sorted doubly linked list
Node* merge(Node* first, Node* second)
{
    // If first list is empty
    if (!first)
        return second;

    // If second list is empty
    if (!second)
        return first;

    // Pick the smaller value and adjust
    // the links
    if (first->data < second->data) {
        first->next = merge(first->next, second);
        first->next->prev = first;
        first->prev = NULL;
        return first;
    }
    else {
        second->next = merge(first, second->next);
        second->next->prev = second;
        second->prev = NULL;
        return second;
    }
}
```

```
        }
    }

// function for Sorted merge of two sorted
// doubly circular linked list
Node* mergeUtil(Node* head1, Node* head2)
{
    // if 1st list is empty
    if (!head1)
        return head2;

    // if 2nd list is empty
    if (!head2)
        return head1;

    // get pointer to the node which will be the
    // last node of the final list
    Node* last_node;
    if (head1->prev->data < head2->prev->data)
        last_node = head2->prev;
    else
        last_node = head1->prev;

    // store NULL to the 'next' link of the last nodes
    // of the two lists
    head1->prev->next = head2->prev->next = NULL;

    // sorted merge of head1 and head2
    Node* finalHead = merge(head1, head2);

    // 'prev' of 1st node pointing the last node
    // 'next' of last node pointing to 1st node
    finalHead->prev = last_node;
    last_node->next = finalHead;

    return finalHead;
}

// function to print the list
void printList(Node* head)
{
    Node* temp = head;

    while (temp->next != head) {
        cout << temp->data << " ";
        temp = temp->next;
    }
    cout << temp->data << " ";
```

```
}

// Driver program to test above
int main()
{
    Node *head1 = NULL, *head2 = NULL;

    // list 1:
    insert(&head1, 8);
    insert(&head1, 5);
    insert(&head1, 3);
    insert(&head1, 1);

    // list 2:
    insert(&head2, 11);
    insert(&head2, 9);
    insert(&head2, 7);
    insert(&head2, 2);

    Node* newHead = mergeUtil(head1, head2);

    cout << "Final Sorted List: ";
    printList(newHead);

    return 0;
}
```

Output:

```
Final Sorted List: 1 2 3 5 7 8 9 11
```

Time Complexity:  $O(n_1 + n_2)$ .

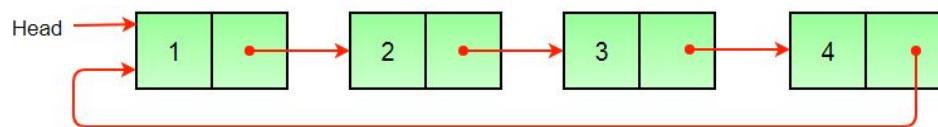
## Source

<https://www.geeksforgeeks.org/sorted-merge-of-two-sorted-doubly-circular-linked-lists/>

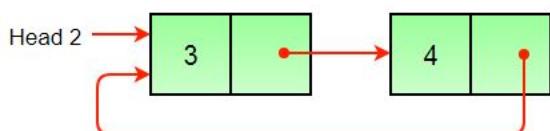
## Chapter 245

# Split a Circular Linked List into two halves

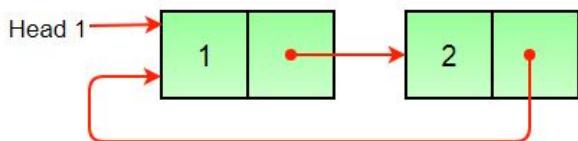
Split a Circular Linked List into two halves - GeeksforGeeks



Original Linked List



Result Linked List 1



Result Linked List 2

If there are **odd number of nodes**, then first list should contain one extra.

Thanks to [Geek4u](#) for suggesting the algorithm.

- 1) Store the mid and last pointers of the circular linked list using tortoise and hare algorithm.
- 2) Make the second half circular.
- 3) Make the first half circular.
- 4) Set head (or start) pointers of the two linked lists.

In the below implementation, if there are odd nodes in the given circular linked list then the first result list has 1 more node than the second result list.

## C

```
/* Program to split a circular linked list into two halves */
#include<stdio.h>
#include<stdlib.h>

/* structure for a node */
struct Node
{
    int data;
    struct Node *next;
};

/* Function to split a list (starting with head) into two lists.
   head1_ref and head2_ref are references to head nodes of
   the two resultant linked lists */
void splitList(struct Node *head, struct Node **head1_ref,
               struct Node **head2_ref)
{
    struct Node *slow_ptr = head;
    struct Node *fast_ptr = head;

    if(head == NULL)
        return;

    /* If there are odd nodes in the circular list then
       fast_ptr->next becomes head and for even nodes
       fast_ptr->next->next becomes head */
    while(fast_ptr->next != head &&
          fast_ptr->next->next != head)
    {
        fast_ptr = fast_ptr->next->next;
        slow_ptr = slow_ptr->next;
    }

    /* If there are even elements in list then move fast_ptr */
    if(fast_ptr->next->next == head)
```

```
fast_ptr = fast_ptr->next;

/* Set the head pointer of first half */
*head1_ref = head;

/* Set the head pointer of second half */
if(head->next != head)
    *head2_ref = slow_ptr->next;

/* Make second half circular */
fast_ptr->next = slow_ptr->next;

/* Make first half circular */
slow_ptr->next = head;
}

/* UTILITY FUNCTIONS */
/* Function to insert a node at the begining of a Circular
   linked lsit */
void push(struct Node **head_ref, int data)
{
    struct Node *ptr1 = (struct Node *)malloc(sizeof(struct Node));
    struct Node *temp = *head_ref;
    ptr1->data = data;
    ptr1->next = *head_ref;

    /* If linked list is not NULL then set the next of
       last node */
    if(*head_ref != NULL)
    {
        while(temp->next != *head_ref)
            temp = temp->next;
        temp->next = ptr1;
    }
    else
        ptr1->next = ptr1; /*For the first node */

    *head_ref = ptr1;
}

/* Function to print nodes in a given Circular linked list */
void printList(struct Node *head)
{
    struct Node *temp = head;
    if(head != NULL)
    {
        printf("\n");
        do {
```

```
    printf("%d ", temp->data);
    temp = temp->next;
} while(temp != head);
}

/* Driver program to test above functions */
int main()
{
    int list_size, i;

    /* Initialize lists as empty */
    struct Node *head = NULL;
    struct Node *head1 = NULL;
    struct Node *head2 = NULL;

    /* Created linked list will be 12->56->2->11 */
    push(&head, 12);
    push(&head, 56);
    push(&head, 2);
    push(&head, 11);

    printf("Original Circular Linked List");
    printList(head);

    /* Split the list */
    splitList(head, &head1, &head2);

    printf("\nFirst Circular Linked List");
    printList(head1);

    printf("\nSecond Circular Linked List");
    printList(head2);

    getchar();
    return 0;
}
```

**Java**

```
// Java program to delete a node from doubly linked list

class LinkedList {

    static Node head, head1, head2;

    static class Node {
```

```
int data;
Node next, prev;

Node(int d) {
    data = d;
    next = prev = null;
}
}

/* Function to split a list (starting with head) into two lists.
head1_ref and head2_ref are references to head nodes of
the two resultant linked lists */
void splitList() {
    Node slow_ptr = head;
    Node fast_ptr = head;

    if (head == null) {
        return;
    }

    /* If there are odd nodes in the circular list then
    fast_ptr->next becomes head and for even nodes
    fast_ptr->next->next becomes head */
    while (fast_ptr.next != head
        && fast_ptr.next.next != head) {
        fast_ptr = fast_ptr.next.next;
        slow_ptr = slow_ptr.next;
    }

    /* If there are even elements in list then move fast_ptr */
    if (fast_ptr.next.next == head) {
        fast_ptr = fast_ptr.next;
    }

    /* Set the head pointer of first half */
    head1 = head;

    /* Set the head pointer of second half */
    if (head.next != head) {
        head2 = slow_ptr.next;
    }
    /* Make second half circular */
    fast_ptr.next = slow_ptr.next;

    /* Make first half circular */
    slow_ptr.next = head;
}
```

```
/* Function to print nodes in a given singly linked list */
void printList(Node node) {
    Node temp = node;
    if (node != null) {
        do {
            System.out.print(temp.data + " ");
            temp = temp.next;
        } while (temp != node);
    }
}

public static void main(String[] args) {
    LinkedList list = new LinkedList();

    //Created linked list will be 12->56->2->11
    list.head = new Node(12);
    list.head.next = new Node(56);
    list.head.next.next = new Node(2);
    list.head.next.next.next = new Node(11);
    list.head.next.next.next.next = list.head;

    System.out.println("Original Circular Linked list ");
    list.printList(head);

    // Split the list
    list.splitList();
    System.out.println("");
    System.out.println("First Circular List ");
    list.printList(head1);
    System.out.println("");
    System.out.println("Second Circular List ");
    list.printList(head2);

}
}

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

### Python

```
# Python program to split circular linked list into two halves

# A node structure
class Node:

    # Constructor to create a new node
    def __init__(self, data):
        self.data = data
```

```
        self.next = None

# Class to create a new Circular Linked list
class CircularLinkedList:

    # Constructor to create a empty circular linked list
    def __init__(self):
        self.head = None

    # Function to insert a node at the beginning of a
    # circular linked list
    def push(self, data):
        ptr1 = Node(data)
        temp = self.head

        ptr1.next = self.head

        # If linked list is not None then set the next of
        # last node
        if self.head is not None:
            while(temp.next != self.head):
                temp = temp.next
            temp.next = ptr1

        else:
            ptr1.next = ptr1 # For the first node

        self.head = ptr1

    # Function to print nodes in a given circular linked list
    def printList(self):
        temp = self.head
        if self.head is not None:
            while(True):
                print "%d" %(temp.data),
                temp = temp.next
                if (temp == self.head):
                    break

    # Function to split a list (starting with head) into
    # two lists. head1 and head2 are the head nodes of the
    # two resultant linked lists
    def splitList(self, head1, head2):
        slow_ptr = self.head
        fast_ptr = self.head
```

```
if self.head is None:  
    return  
  
# If htere are odd nodes in the circular list then  
# fast_ptr->next becomes head and for even nodes  
# fast_ptr->next->next becomes head  
while(fast_ptr.next != self.head and  
      fast_ptr.next.next != self.head ):  
    fast_ptr = fast_ptr.next.next  
    slow_ptr = slow_ptr.next  
  
# If there are event elements in list then  
# move fast_ptr  
if fast_ptr.next.next == self.head:  
    fast_ptr = fast_ptr.next  
  
# Set the head pointer of first half  
head1.head = self.head  
  
# Set the head pointer of second half  
if self.head.next != self.head:  
    head2.head = slow_ptr.next  
  
# Make second half circular  
fast_ptr.next = slow_ptr.next  
  
# Make first half circular  
slow_ptr.next = self.head  
  
# Driver program to test above functions  
  
# Initialize lists as empty  
head = CircularLinkedList()  
head1 = CircularLinkedList()  
head2 = CircularLinkedList()  
  
head.push(12)  
head.push(56)  
head.push(2)  
head.push(11)  
  
print "Original Circular Linked List"  
head.printList()  
  
# Split the list  
head.splitList(head1 , head2)
```

```
print "\nFirst Circular Linked List"
head1.printList()

print "\nSecond Circular Linked List"
head2.printList()

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

Output:

```
Original Circular Linked List
11 2 56 12
First Circular Linked List
11 2
Second Circular Linked List
56 12
```

Time Complexity: **O(n)**

## Source

<https://www.geeksforgeeks.org/split-a-circular-linked-list-into-two-halves/>

## Chapter 246

# SquareRoot(n)-th node in a Linked List

SquareRoot(n)-th node in a Linked List - GeeksforGeeks

Given a Linked List, write a function that accepts the head node of the linked list as a parameter and returns the value of node present at  $(\text{floor}(\sqrt{n}))$ th position in the Linked List, where n is the length of the linked list or the total number of nodes in the list.

Examples:

Input : 1->2->3->4->5->NULL  
Output : 2

Input : 10->20->30->40->NULL  
Output : 20

Input : 10->20->30->40->50->60->70->80->90->NULL  
Output : 30

**Simple method:** The simple method is to first find the total number of nodes present in the linked list, then find the value of  $\text{floor}(\text{squareRoot}(n))$  where n is the total number of nodes. Then traverse from the first node in the list to this position and return the node at this position.

This method traverses the linked list 2 times.

**Optimized approach:** In this method, we can get the required node by traversing the linked list once only. Below is the step by step algorithm for this approach.

1. Initialize two counters i and j both to 1 and a pointer sq rtn to NULL to traverse till the required position is reached.
2. Start traversing the list using head node until the last node is reached.

3. While traversing check if the value of j is equal to  $\sqrt{i}$ . If the value is equal increment both i and j and sqrtn to point sqrtn->next otherwise increment only i.
4. Now, when we will reach the last node of list i will contain value of n, j will contain value of  $\sqrt{i}$  and sqrtn will point to node at jth position.

```

// C program to find sqrt(n)'th node
// of a linked list

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

// Linked list node
struct Node
{
    int data;
    struct Node* next;
};

// Function to get the sqrt(n)th
// node of a linked list
int printsqrtn(struct Node* head)
{
    struct Node* sqrtn = NULL;
    int i = 1, j = 1;

    // Traverse the list
    while (head!=NULL)
    {
        // check if j = sqrt(i)
        if (i == j*j)
        {
            // for first node
            if (sqrtn == NULL)
                sqrtn = head;
            else
                sqrtn = sqrtn->next;

            // increment j if j = sqrt(i)
            j++;
        }
        i++;

        head=head->next;
    }

    // return node's data
    return sqrtn->data;
}

```

```
void print(struct Node* head)
{
    while (head != NULL)
    {
        printf("%d ", head->data);
        head = head->next;
    }
    printf("\n");
}

// function to add a new node at the
// begining of the list
void push(struct Node** head_ref, int new_data)
{
    // allocate node
    struct Node* new_node =
        (struct Node*) malloc(sizeof(struct Node));

    // put in the data
    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;
}

/* Driver program to test above function*/
int main()
{
    /* Start with the empty list */
    struct Node* head = NULL;
    push(&head, 40);
    push(&head, 30);
    push(&head, 20);
    push(&head, 10);
    printf("Given linked list is:");
    print(head);
    printf("sqrt(n)th node is %d ", printsqrtn(head));

    return 0;
}
```

Output:

Given linked list is:10 20 30 40  
 $\text{sqrt}(n)$ th node is 20

### Source

<https://www.geeksforgeeks.org/squarerootnth-node-in-a-linked-list/>

## Chapter 247

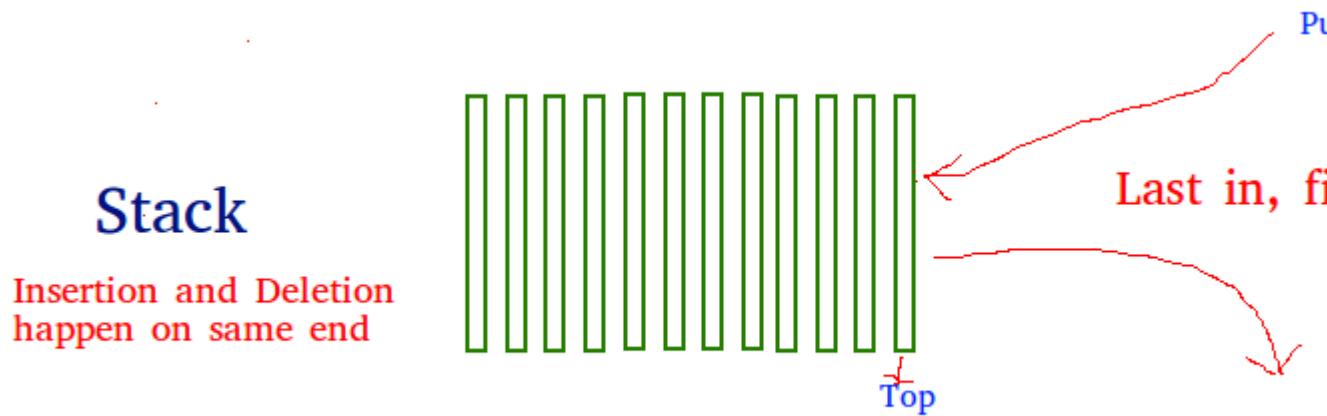
# Stack Data Structure (Introduction and Program)

Stack Data Structure (Introduction and Program) - GeeksforGeeks

Stack is a linear data structure which follows a particular order in which the operations are performed. The order may be LIFO(Last In First Out) or FILO(First In Last Out).

Mainly the following three basic operations are performed in the stack:

- **Push:** Adds an item in the stack. If the stack is full, then it is said to be an Overflow condition.
- **Pop:** Removes an item from the stack. The items are popped in the reversed order in which they are pushed. If the stack is empty, then it is said to be an Underflow condition.
- **Peek or Top:** Returns top element of stack.
- **isEmpty:** Returns true if stack is empty, else false.



### How to understand a stack practically?

There are many real life examples of stack. Consider the simple example of plates stacked over one another in canteen. The plate which is at the top is the first one to be removed, i.e. the plate which has been placed at the bottommost position remains in the stack for the longest period of time. So, it can be simply seen to follow LIFO/FILO order.

### Time Complexities of operations on stack:

push(), pop(), isEmpty() and peek() all take O(1) time. We do not run any loop in any of these operations.

### Applications of stack:

- Balancing of symbols
- Infix to Postfix /Prefix conversion
- Redo-undo features at many places like editors, photoshop.
- Forward and backward feature in web browsers
- Used in many algorithms like [Tower of Hanoi](#), tree traversals, stock span problem, [histogram problem](#).
- Other applications can be Backtracking, [Knight tour problem](#), rat in a maze,[N queen problem](#) and [sudoku solver](#)
- In Graph Algorithms like [Topological Sorting](#) and [Strongly Connected Components](#)

### Implementation:

There are two ways to implement a stack:

- Using array
- Using linked list

### Implementing Stack using Arrays

C++

```
/* C++ program to implement basic stack
operations */
#include<bits/stdc++.h>

using namespace std;

#define MAX 1000

class Stack
{
    int top;
public:
    int a[MAX]; //Maximum size of Stack

    Stack() { top = -1; }
    bool push(int x);
    int pop();
    bool isEmpty();
};

bool Stack::push(int x)
{
    if (top >= (MAX-1))
    {
        cout << "Stack Overflow";
        return false;
    }
    else
    {
        a[++top] = x;
        cout << x << " pushed into stack\n";
        return true;
    }
}

int Stack::pop()
{
    if (top < 0)
    {
        cout << "Stack Underflow";
        return 0;
    }
    else
    {
        int x = a[top--];
        return x;
    }
}
```

```
bool Stack::isEmpty()
{
    return (top < 0);
}

// Driver program to test above functions
int main()
{
    struct Stack s;
    s.push(10);
    s.push(20);
    s.push(30);
    cout<<s.pop() << " Popped from stack\n";

    return 0;
}
```

C

```
// C program for array implementation of stack
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

// A structure to represent a stack
struct Stack
{
    int top;
    unsigned capacity;
    int* array;
};

// function to create a stack of given capacity. It initializes size of
// stack as 0
struct Stack* createStack(unsigned capacity)
{
    struct Stack* stack = (struct Stack*) malloc(sizeof(struct Stack));
    stack->capacity = capacity;
    stack->top = -1;
    stack->array = (int*) malloc(stack->capacity * sizeof(int));
    return stack;
}

// Stack is full when top is equal to the last index
int isFull(struct Stack* stack)
{   return stack->top == stack->capacity - 1; }
```

```
// Stack is empty when top is equal to -1
int isEmpty(struct Stack* stack)
{   return stack->top == -1; }

// Function to add an item to stack. It increases top by 1
void push(struct Stack* stack, int item)
{
    if (isFull(stack))
        return;
    stack->array[+stack->top] = item;
    printf("%d pushed to stack\n", item);
}

// Function to remove an item from stack. It decreases top by 1
int pop(struct Stack* stack)
{
    if (isEmpty(stack))
        return INT_MIN;
    return stack->array[stack->top--];
}

// Driver program to test above functions
int main()
{
    struct Stack* stack = createStack(100);

    push(stack, 10);
    push(stack, 20);
    push(stack, 30);

    printf("%d popped from stack\n", pop(stack));

    return 0;
}
```

### Java

```
/* Java program to implement basic stack
operations */
class Stack
{
    static final int MAX = 1000;
    int top;
    int a[] = new int[MAX]; // Maximum size of Stack

    boolean isEmpty()
    {
        return (top < 0);
    }
```

```
Stack()
{
    top = -1;
}

boolean push(int x)
{
    if (top >= (MAX-1))
    {
        System.out.println("Stack Overflow");
        return false;
    }
    else
    {
        a[++top] = x;
        System.out.println(x + " pushed into stack");
        return true;
    }
}

int pop()
{
    if (top < 0)
    {
        System.out.println("Stack Underflow");
        return 0;
    }
    else
    {
        int x = a[top--];
        return x;
    }
}

// Driver code
class Main
{
    public static void main(String args[])
    {
        Stack s = new Stack();
        s.push(10);
        s.push(20);
        s.push(30);
        System.out.println(s.pop() + " Popped from stack");
    }
}
```

## Python

```
# Python program for implementation of stack

# import maxsize from sys module
# Used to return -infinite when stack is empty
from sys import maxsize

# Function to create a stack. It initializes size of stack as 0
def createStack():
    stack = []
    return stack

# Stack is empty when stack size is 0
def isEmpty(stack):
    return len(stack) == 0

# Function to add an item to stack. It increases size by 1
def push(stack, item):
    stack.append(item)
    print(item + " pushed to stack ")

# Function to remove an item from stack. It decreases size by 1
def pop(stack):
    if (isEmpty(stack)):
        return str(-maxsize -1) #return minus infinite

    return stack.pop()

# Driver program to test above functions
stack = createStack()
push(stack, str(10))
push(stack, str(20))
push(stack, str(30))
print(pop(stack) + " popped from stack")
```

## C#

```
// C# program to implement basic stack
// operations
using System;

namespace ImplementStack
{
    class Stack
    {
        private int[] ele;
```

```
private int top;
private int max;
public Stack(int size)
{
    ele = new int[size];//Maximum size of Stack
    top = -1;
    max = size;
}

public void push(int item)
{
    if (top == max-1)
    {
        Console.WriteLine("Stack Overflow");
        return;
    }
    else
    {
        ele[++top] = item;
    }
}

public int pop()
{
    if(top == -1)
    {
        Console.WriteLine("Stack is Empty");
        return -1;
    }
    else
    {
        Console.WriteLine("{0} popped from stack ",ele[top]);
        return ele[top--];
    }
}

public void printStack()
{
    if (top == -1)
    {
        Console.WriteLine("Stack is Empty");
        return;
    }
    else
    {
        for (int i = 0; i <= top; i++)
        {
            Console.WriteLine("{0} pushed into stack", ele[i]);
        }
    }
}
```

```
        }
    }
}

// Driver program to test above functions
class Program
{
    static void Main()
    {
        Stack p = new Stack(5);

        p.push(10);
        p.push(20);
        p.push(30);
        p.printStack();
        p.pop();
    }
}
```

**Pros:** Easy to implement. Memory is saved as pointers are not involved.

**Cons:** It is not dynamic. It doesn't grow and shrink depending on needs at runtime.

**Output :**

```
10 pushed into stack
20 pushed into stack
30 pushed into stack
30 popped from stack
```

### Implementing Stack using Linked List

C

```
// C program for linked list implementation of stack
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

// A structure to represent a stack
struct StackNode
{
    int data;
    struct StackNode* next;
};

struct StackNode* newNode(int data)
```

```
{  
    struct StackNode* stackNode =  
        (struct StackNode*) malloc(sizeof(struct StackNode));  
    stackNode->data = data;  
    stackNode->next = NULL;  
    return stackNode;  
}  
  
int isEmpty(struct StackNode *root)  
{  
    return !root;  
}  
  
void push(struct StackNode** root, int data)  
{  
    struct StackNode* stackNode = newNode(data);  
    stackNode->next = *root;  
    *root = stackNode;  
    printf("%d pushed to stack\n", data);  
}  
  
int pop(struct StackNode** root)  
{  
    if (isEmpty(*root))  
        return INT_MIN;  
    struct StackNode* temp = *root;  
    *root = (*root)->next;  
    int popped = temp->data;  
    free(temp);  
  
    return popped;  
}  
  
int peek(struct StackNode* root)  
{  
    if (isEmpty(root))  
        return INT_MIN;  
    return root->data;  
}  
  
int main()  
{  
    struct StackNode* root = NULL;  
  
    push(&root, 10);  
    push(&root, 20);  
    push(&root, 30);
```

```
printf("%d popped from stack\n", pop(&root));  
  
printf("Top element is %d\n", peek(root));  
  
    return 0;  
}
```

### Python

```
# Python program for linked list implementation of stack  
  
# Class to represent a node  
class StackNode:  
  
    # Constructor to initialize a node  
    def __init__(self, data):  
        self.data = data  
        self.next = None  
  
class Stack:  
  
    # Constructor to initialize the root of linked list  
    def __init__(self):  
        self.root = None  
  
    def isEmpty(self):  
        return True if self.root is None else False  
  
    def push(self, data):  
        newNode = StackNode(data)  
        newNode.next = self.root  
        self.root = newNode  
        print "%d pushed to stack" %(data)  
  
    def pop(self):  
        if (self.isEmpty()):  
            return float("-inf")  
        temp = self.root  
        self.root = self.root.next  
        popped = temp.data  
        return popped  
  
    def peek(self):  
        if self.isEmpty():  
            return float("-inf")  
        return self.root.data  
  
# Driver program to test above class
```

```
stack = Stack()
stack.push(10)
stack.push(20)
stack.push(30)

print "%d popped from stack" %(stack.pop())
print "Top element is %d " %(stack.peek())

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

Output:

```
10 pushed to stack
20 pushed to stack
30 pushed to stack
30 popped from stack
Top element is 20
```

**Pros:** The linked list implementation of stack can grow and shrink according to the needs at runtime.

**Cons:** Requires extra memory due to involvement of pointers.

We will cover the implementation of applications of stack in separate posts.

[Stack Set -2 \(Infix to Postfix\)](#)

[Quiz: Stack Questions](#)

**References:**

[http://en.wikipedia.org/wiki/Stack\\_%28abstract\\_data\\_type%29#Problem\\_Description](http://en.wikipedia.org/wiki/Stack_%28abstract_data_type%29#Problem_Description)

[Improved By : SUNILKUMAR19, SoumikMondal](#)

## Source

<https://www.geeksforgeeks.org/stack-data-structure-introduction-program/>

## Chapter 248

# Sublist Search (Search a linked list in another list)

Sublist Search (Search a linked list in another list) - GeeksforGeeks

Given two linked lists, the task is to check whether the first list is present in 2nd list or not.

```
Input  : list1 = 10->20
        list2 = 5->10->20
Output : LIST FOUND
```

```
Input  : list1 = 1->2->3->4
        list2 = 1->2->1->2->3->4
Output : LIST FOUND
```

```
Input  : list1 = 1->2->3->4
        list2 = 1->2->2->1->2->3
Output : LIST NOT FOUND
```

Algorithm:

- 1- Take first node of second list.
- 2- Start matching the first list from this first node.
- 3- If whole lists match return true.
- 4- Else break and take first list to the first node again.
- 5- And take second list to its second node.
- 6- Repeat these steps until any of linked lists becomes empty.
- 7- If first list becomes empty then list found else not.

Below is C++ implementation.

```
// C++ program to find a list in second list
```

```
#include <bits/stdc++.h>
using namespace std;

// A Linked List node
struct Node
{
    int data;
    Node* next;
};

// Returns true if first list is present in second
// list
bool findList(Node* first, Node* second)
{
    Node* ptr1 = first, *ptr2 = second;

    // If both linked lists are empty, return true
    if (first == NULL && second == NULL)
        return true;

    // Else If one is empty and other is not return
    // false
    if (first == NULL ||
        (first != NULL && second == NULL))
        return false;

    // Traverse the second list by picking nodes
    // one by one
    while (second != NULL)
    {
        // Initialize ptr2 with current node of second
        ptr2 = second;

        // Start matching first list with second list
        while (ptr1 != NULL)
        {
            // If second list becomes empty and first
            // not then return false
            if (ptr2 == NULL)
                return false;

            // If data part is same, go to next
            // of both lists
            else if (ptr1->data == ptr2->data)
            {
                ptr1 = ptr1->next;
                ptr2 = ptr2->next;
            }
        }
    }
}
```

```
// If not equal then break the loop
else break;
}

// Return true if first list gets traversed
// completely that means it is matched.
if (ptr1 == NULL)
    return true;

// Initialize ptr1 with first again
ptr1 = first;

// And go to next node of second list
second = second->next;
}

return false;
}

/* Function to print nodes in a given linked list */
void printList(Node* node)
{
    while (node != NULL)
    {
        printf("%d ", node->data);
        node = node->next;
    }
}

// Function to add new node to linked lists
Node *newNode(int key)
{
    Node *temp = new Node;
    temp-> data= key;
    temp->next = NULL;
    return temp;
}

/* Driver program to test above functions*/
int main()
{
    /* Let us create two linked lists to test
the above functions. Created lists shall be
    a: 1->2->3->4
    b: 1->2->1->2->3->4*/
    Node *a = newNode(1);
    a->next = newNode(2);
```

```
a->next->next = newNode(3);
a->next->next->next = newNode(4);

Node *b = newNode(1);
b->next = newNode(2);
b->next->next = newNode(1);
b->next->next->next = newNode(2);
b->next->next->next->next = newNode(3);
b->next->next->next->next = newNode(4);

findList(a,b) ? cout << "LIST FOUND" :
               cout << "LIST NOT FOUND";

return 0;
}
```

Output:

LIST FOUND

Time Complexity :  $O(m*n)$  where m is the number of nodes in second list and n in first.

**Optimization :**

Above code can be optimized by using extra space i.e. stores the list into two strings and apply [KMP algorithm](#). Refer <https://ide.geeksforgeeks.org/3fXUaV> for implementation provided by [Nishant Singh](#).

**Source**

<https://www.geeksforgeeks.org/sublist-search-search-a-linked-list-in-another-list/>

## Chapter 249

# Subtract Two Numbers represented as Linked Lists

Subtract Two Numbers represented as Linked Lists - GeeksforGeeks

Given two linked lists that represent two large positive numbers. Subtract the smaller number from larger one and return the difference as a linked list. Note that the input lists may be in any order, but we always need to subtract smaller from larger one.

It may be assumed that there are no extra leading zeros in input lists.

Examples

Input : 11 = 1 → 0 → 0 → NULL, 12 = 1 → NULL  
Output : 0->9->9->NULL

Input : 11 = 1 → 0 → 0 → NULL, 12 = 1 → NULL  
Output : 0->9->9->NULL

Input : 11 = 7->8->6->NULL, 12 = 7->8->9 NULL  
Output : 3->NULL

Following are the steps.

- 1) Calculate sizes of given two linked lists.
- 2) If sizes not same, then append zeros in smaller linked list.
- 3) If sizes same, then follow below steps:
  - ....a) Find the smaller valued linked list.
  - ....b) One by one subtract nodes of smaller sized linked list from larger size. Keep track of borrow while subtracting.

Following is the implementation of the above approach.

C++

```
// C++ program to subtract smaller valued list from
// larger valued list and return result as a list.
#include<bits/stdc++.h>
using namespace std;

// A linked List Node
struct Node
{
    int data;
    struct Node* next;
};

// A utility which creates Node.
Node* newNode(int data)
{
    Node* temp = new Node;
    temp->data = data;
    temp->next = NULL;
    return temp;
}

/* A utility function to get length of linked list */
int getLength(Node *Node)
{
    int size = 0;
    while (Node != NULL)
    {
        Node = Node->next;
        size++;
    }
    return size;
}

/* A Utility that pads zeros in front of the
   Node, with the given diff */
Node* paddZeros(Node* sNode, int diff)
{
    if (sNode == NULL)
        return NULL;

    Node* zHead = newNode(0);
    diff--;
    Node* temp = zHead;
    while (diff--)
    {
        temp->next = newNode(0);
        temp = temp->next;
    }
}
```

```
temp->next = sNode;
return zHead;
}

/* Subtract LinkedList Helper is a recursive function,
move till the last Node, and subtract the digits and
create the Node and return the Node. If d1 < d2, we
borrow the number from previous digit. */
Node* subtractLinkedListHelper(Node* l1, Node* l2, bool& borrow)
{
    if (l1 == NULL && l2 == NULL && borrow == 0)
        return NULL;

    Node* previous = subtractLinkedListHelper(l1 ? l1->next : NULL,
                                                l2 ? l2->next : NULL, borrow);

    int d1 = l1->data;
    int d2 = l2->data;
    int sub = 0;

    /* if you have given the value value to next digit then
       reduce the d1 by 1 */
    if (borrow)
    {
        d1--;
        borrow = false;
    }

    /* If d1 < d2 , then borrow the number from previous digit.
       Add 10 to d1 and set borrow = true; */
    if (d1 < d2)
    {
        borrow = true;
        d1 = d1 + 10;
    }

    /* subtract the digits */
    sub = d1 - d2;

    /* Create a Node with sub value */
    Node* current = newNode(sub);

    /* Set the Next pointer as Previous */
    current->next = previous;

    return current;
}
```

```
/* This API subtracts two linked lists and returns the
   linked list which shall have the subtracted result. */
Node* subtractLinkedList(Node* l1, Node* l2)
{
    // Base Case.
    if (l1 == NULL && l2 == NULL)
        return NULL;

    // In either of the case, get the lengths of both
    // Linked list.
    int len1 = getLength(l1);
    int len2 = getLength(l2);

    Node *lNode = NULL, *sNode = NULL;

    Node* temp1 = l1;
    Node* temp2 = l2;

    // If lengths differ, calculate the smaller Node
    // and padd zeros for smaller Node and ensure both
    // larger Node and smaller Node has equal length.
    if (len1 != len2)
    {
        lNode = len1 > len2 ? l1 : l2;
        sNode = len1 > len2 ? l2 : l1;
        sNode = paddZeros(sNode, abs(len1 - len2));
    }

    else
    {
        // If both list lengths are equal, then calculate
        // the larger and smaller list. If 5-6-7 & 5-6-8
        // are linked list, then walk through linked list
        // at last Node as 7 < 8, larger Node is 5-6-8
        // and smaller Node is 5-6-7.
        while (l1 && l2)
        {
            if (l1->data != l2->data)
            {
                lNode = l1->data > l2->data ? temp1 : temp2;
                sNode = l1->data > l2->data ? temp2 : temp1;
                break;
            }
            l1 = l1->next;
            l2 = l2->next;
        }
    }
}
```

```
// After calculating larger and smaller Node, call
// subtractLinkedListHelper which returns the subtracted
// linked list.
bool borrow = false;
return subtractLinkedListHelper(lNode, sNode, borrow);
}

/* A utility function to print linked list */
void printList(struct Node *Node)
{
    while (Node != NULL)
    {
        printf("%d ", Node->data);
        Node = Node->next;
    }
    printf("\n");
}

// Driver program to test above functions
int main()
{
    Node* head1 = newNode(1);
    head1->next = newNode(0);
    head1->next->next = newNode(0);

    Node* head2 = newNode(1);

    Node* result = subtractLinkedList(head1, head2);

    printList(result);

    return 0;
}
```

### Java

```
// Java program to subtract smaller valued
// list from larger valued list and return
// result as a list.
import java.util.*;
import java.lang.*;
import java.io.*;

class LinkedList
{
    static Node head; // head of list
    boolean borrow;
```

```
/* Node Class */
static class Node {
    int data;
    Node next;

    // Constructor to create a new node
    Node(int d) {
        data = d;
        next = null;
    }
}

/* A utility function to get length of
linked list */
int getLength(Node node)
{
    int size = 0;
    while (node != null)
    {
        node = node.next;
        size++;
    }
    return size;
}

/* A Utility that padds zeros in front
of the Node, with the given diff */
Node paddZeros(Node sNode, int diff)
{
    if (sNode == null)
        return null;

    Node zHead = new Node(0);
    diff--;
    Node temp = zHead;
    while ((diff--) != 0)
    {
        temp.next = new Node(0);
        temp = temp.next;
    }
    temp.next = sNode;
    return zHead;
}

/* Subtract LinkedList Helper is a recursive
function, move till the last Node, and
subtract the digits and create the Node and
```

```
return the Node. If d1 < d2, we borrow the
number from previous digit. */
Node subtractLinkedListHelper(Node l1, Node l2)
{
    if (l1 == null && l2 == null && borrow == false)
        return null;

    Node previous = subtractLinkedListHelper((l1 != null) ?
                                              l1.next : null, (l2 != null) ?
                                              l2.next : null);

    int d1 = l1.data;
    int d2 = l2.data;
    int sub = 0;

    /* if you have given the value value to
     next digit then reduce the d1 by 1 */
    if (borrow)
    {
        d1--;
        borrow = false;
    }

    /* If d1 < d2 , then borrow the number from
     previous digit. Add 10 to d1 and set
     borrow = true; */
    if (d1 < d2)
    {
        borrow = true;
        d1 = d1 + 10;
    }

    /* subtract the digits */
    sub = d1 - d2;

    /* Create a Node with sub value */
    Node current = new Node(sub);

    /* Set the Next pointer as Previous */
    current.next = previous;

    return current;
}

/* This API subtracts two linked lists and
returns the linked list which shall have the
subtracted result. */
Node subtractLinkedList(Node l1, Node l2)
```

```
{  
    // Base Case.  
    if (l1 == null && l2 == null)  
        return null;  
  
    // In either of the case, get the lengths  
    // of both Linked list.  
    int len1 = getLength(l1);  
    int len2 = getLength(l2);  
  
    Node lNode = null, sNode = null;  
  
    Node temp1 = l1;  
    Node temp2 = l2;  
  
    // If lengths differ, calculate the smaller  
    // Node and padd zeros for smaller Node and  
    // ensure both larger Node and smaller Node  
    // has equal length.  
    if (len1 != len2)  
    {  
        lNode = len1 > len2 ? l1 : l2;  
        sNode = len1 > len2 ? l2 : l1;  
        sNode = paddZeros(sNode, Math.abs(len1 - len2));  
    }  
  
    else  
    {  
        // If both list lengths are equal, then  
        // calculate the larger and smaller list.  
        // If 5-6-7 & 5-6-8 are linked list, then  
        // walk through linked list at last Node  
        // as 7 < 8, larger Node is 5-6-8 and  
        // smaller Node is 5-6-7.  
        while (l1 != null && l2 != null)  
        {  
            if (l1.data != l2.data)  
            {  
                lNode = l1.data > l2.data ? temp1 : temp2;  
                sNode = l1.data > l2.data ? temp2 : temp1;  
                break;  
            }  
            l1 = l1.next;  
            l2 = l2.next;  
        }  
    }  
  
    // After calculating larger and smaller Node,
```

```
// call subtractLinkedListHelper which returns
// the subtracted linked list.
borrow = false;
return subtractLinkedListHelper(lNode, sNode);
}

// function to display the linked list
static void printList(Node head)
{
    Node temp = head;
    while (temp != null)
    {
        System.out.print(temp.data + " ");
        temp = temp.next;
    }
}

// Driver program to test above
public static void main (String[] args)
{
    Node head = new Node(1);
    head.next = new Node(0);
    head.next.next = new Node(0);

    Node head2 = new Node(1);

    LinkedList ob = new LinkedList();
    Node result = ob.subtractLinkedList(head, head2);

    printList(result);
}

}

// This article is contributed by Chhavi
```

Output :

0 9 9

## Source

<https://www.geeksforgeeks.org/subtract-two-numbers-represented-as-linked-lists/>

## Chapter 250

# Sudo Placement[1.4] | K Sum

Sudo Placement[1.4] | K Sum - GeeksforGeeks

Given head of a linked list of integers and an integer k, your task is to modify the linked list as follows:

- Consider nodes in groups of size k. In every group, replace value of first node with group sum.
- Also, delete the elements of group except the first node.
- During traversal, if the remaining nodes in linked list are less than k then also do the above considering remaining nodes.

**Examples:**

**Input:** N = 6, K = 2

1->2->3->4->5->6

**Output:** 3 7 11

We have denoted grouping of k elements by (). The elements inside () are summed.

```
1 -> 2 -> 3 -> 4 -> 5 -> 6 -> null  
(1 -> 2) -> 3 -> 4 -> 5 -> 6 -> null  
(3) -> 3 -> 4 -> 5 -> 6 -> null  
3 -> (3 -> 4) -> 5 -> 6 -> null  
3 -> (7) -> 5 -> 6 -> null  
3 -> 7 -> (5 -> 6) -> null  
3 -> 7 -> (11) -> null  
3 -> 7 -> 11 -> null
```

**Approach:** Insert the given nodes in the Linked list. The approach of insertion has been discussed in [this](#) post. Once the nodes are inserted, iterate from the beginning of the list. Mark the first node as **temp** node. Iterate for the next k-1 nodes and sum up the nodes in **sum** variable. If the number of nodes is less than K, then replace the temp node's data

with sum and point temp to NULL. If there is a group with K nodes, replace the temp's data with sum and move temp to the node which is just after K nodes. Continue the above operation till temp points to NULL. Once temp reaches the end, it means all groups of size K has been traversed and the node has been replaced with the sum of nodes of size K. Once the operation of replacements have been done, the linked list thus formed can be printed. The method of printing the linked list has been discussed in [this](#) post.

Below is the implementation of the above approach:

C++

```
// C++ program for
// SP - K Sum

#include <iostream>
using namespace std;

// structure for linked list
struct Node {
    long long data;
    Node* next;
};

// allocated new node
Node* newNode(int key)
{
    Node* temp = new Node;
    temp->data = key;
    temp->next = NULL;
    return temp;
}

// function to insert node in a Linked List
Node* insertNode(Node* head, int key)
{
    if (head == NULL)
        head = newNode(key);
    else
        head->next = insertNode(head->next, key);

    return head;
}

// traverse the node
// to print it
void print(Node* head)
{
    // traverse the linked list
```

```
while (head) {
    cout << head->data << " ";
    head = head->next;
}
cout << endl;

// function to perform the following operation
void KSum(Node* head, int k)
{
    // initially pointing to start
    Node* temp = head;

    // iterate till end
    while (temp) {

        // dummy variable to store k
        long long count = k;

        // summation of nodes
        long long sum = 0;

        // currently at node from
        // which iteration is done
        Node* curr = temp;

        // till k nodes are visited and
        // last node is not visited
        while (count-- && curr) {
            // add to sum the node data
            sum = sum + curr->data;

            // move to the next node
            curr = curr->next;
        }

        // change pointer's data of temp to sum
        temp->data = sum;

        // move temp to next pointer
        temp->next = curr;

        // move temp to the next pointer
        temp = temp->next;
    }
}

// Driver Code
```

```
int main()
{
    Node* head = NULL;

    // inserts nodes in the linked list
    head = insertNode(head, 1);
    head = insertNode(head, 2);
    head = insertNode(head, 3);
    head = insertNode(head, 4);
    head = insertNode(head, 5);
    head = insertNode(head, 6);

    int k = 2;
    // Function call to perform the
    // given operations
    KSum(head, k);

    // print the linked list
    print(head);

    return 0;
}
```

**Java**

```
// Java program for
// SP - K Sum

import java.io.*;
import java.util.*;

// structure for linked list
class Node
{
    int data;
    Node next;
    Node(int key)
    {
        data = key;
        next = null;
    }
}

class GFG
{

    // allocated new node
```

```
static Node head;

// function to insert node
// in a Linked List
public static Node insertNode(Node head, int key)
{
    if (head == null)
        head = new Node(key);
    else
        head.next = insertNode(head.next, key);

    return head;
}

// traverse the node
// to print it
public static void print(Node head)
{
    // travere the linked list
    while (head != null)
    {
        System.out.print(head.data + " ");
        head = head.next;
    }
    System.out.println();
}

// function to perform
// the following operation
public static void KSum(Node head, int k)
{
    // initially pointing
    // to start
    Node temp = head;

    // iterate till end
    while (temp != null)
    {

        // dummy variable
        // to store k
        int count = k;

        // summation of nodes
        int sum = 0;

        // currently at node from
        // which iteration is done
```

```
Node curr = temp;

// till k nodes are visited and
// last node is not visited
while (count > 0 && curr != null)
{
    // add to sum the node data
    sum = sum + curr.data;

    // move to the next node
    curr = curr.next;
    count--;
}

// change pointer's data
// of temp to sum
temp.data = sum;

// move temp to
// next pointer
temp.next = curr;

// move temp to
// the next pointer
temp = temp.next;
}

//return temp;
}

// Driver Code
public static void main(String args[])
{
    head = null;

    // inserts nodes in
    // the linked list
    head = insertNode(head, 1);
    head = insertNode(head, 2);
    head = insertNode(head, 3);
    head = insertNode(head, 4);
    head = insertNode(head, 5);
    head = insertNode(head, 6);

    int k = 2;

    // Function call to perform
    // the given operations
```

```
KSum(head, k);  
  
    // print the linked list  
    print(head);  
}  
}
```

**Output:**

3 7 11

**Source**

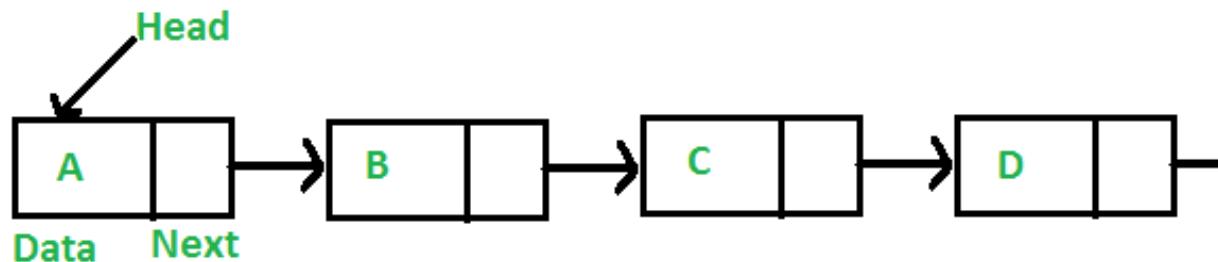
<https://www.geeksforgeeks.org/sudo-placement1-4-k-sum/>

## Chapter 251

### Sum of the nodes of a Singly Linked List

Sum of the nodes of a Singly Linked List - GeeksforGeeks

Given a singly linked list. The task is to find the sum of nodes of the given linked list.



Task is to do  $A + B + C + D$ .

**Examples:**

Input: 7->6->8->4->1

Output: 26

Sum of nodes:

$$7 + 6 + 8 + 4 + 1 = 26$$

Input: 1->7->3->9->11->5

Output: 36

**Recursive Solution:**

- Call a function by passing the head and variable to store the sum.
  - Then recursively call the function by passing the next of current node and sum variable.
    - \* Add the value of the current node to the sum.

Below is the implementation of above approach:

```
// C++ implementation to find the sum of
// nodes of the Linked List
#include <bits/stdc++.h>
using namespace std;

/* A Linked list node */
struct Node {
    int data;
    struct Node* next;
};

// function to insert a node at the
// beginning of the linked list
void push(struct Node** head_ref, int new_data)
{
    /* allocate node */
    struct Node* new_node = new Node;

    /* put in the data */
    new_node->data = new_data;

    /* link the old list to the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref) = new_node;
}

// function to recursively find the sum of
// nodes of the given linked list
void sumOfNodes(struct Node* head, int* sum)
{
    // if head = NULL
    if (!head)
        return;

    // recursively traverse the remaining nodes
    sumOfNodes(head->next, sum);

    // accumulate sum
```

```
*sum = *sum + head->data;
}

// utility function to find the sum of nodes
int sumOfNodesUtil(struct Node* head)
{

    int sum = 0;

    // find the sum of nodes
    sumOfNodes(head, &sum);

    // required sum
    return sum;
}

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

    // create linked list 7->6->8->4->1
    push(&head, 7);
    push(&head, 6);
    push(&head, 8);
    push(&head, 4);
    push(&head, 1);

    cout << "Sum of nodes = "
         << sumOfNodesUtil(head);
    return 0;
}
```

**Output:**

```
Sum of nodes = 26
```

**Time Complexity:** O(N) , N is the number of nodes in a linked list.

**Auxiliary Space:** O(N), only if the stack size is considered during recursive calls.

**Iterative Solution:**

1. Initialise a pointer ***ptr*** with the head of the linked list and a ***sum*** variable with 0.
2. Start traversing the linked list using a loop until all the nodes get traversed.
  - Add the value of current node to the sum i.e. ***sum += ptr -> data*** .
  - Increment the pointer to the next node of linked list i.e. ***ptr = ptr ->next*** .

3. Return the *sum*.

Below is the implementation of above approach:

```
// C++ implementation to find the sum of
// nodes of the Linked List
#include <bits/stdc++.h>
using namespace std;

/* A Linked list node */
struct Node {
    int data;
    struct Node* next;
};

// function to insert a node at the
// beginning of the linked list
void push(struct Node** head_ref, int new_data)
{
    /* allocate node */
    struct Node* new_node = new Node;

    /* put in the data */
    new_node->data = new_data;

    /* link the old list to the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref) = new_node;
}

// function to find the sum of
// nodes of the given linked list
int sumOfNodes(struct Node* head)
{
    struct Node* ptr = head;
    int sum = 0;
    while (ptr != NULL) {

        sum += ptr->data;
        ptr = ptr->next;
    }

    return sum;
}

// Driver program to test above
```

```
int main()
{
    struct Node* head = NULL;

    // create linked list 7->6->8->4->1
    push(&head, 7);
    push(&head, 6);
    push(&head, 8);
    push(&head, 4);
    push(&head, 1);

    cout << "Sum of nodes = "
        << sumOfNodes(head);
    return 0;
}
```

**Output:**

Sum of nodes = 26

**Time Complexity:**  $O(N)$  , N is the number of nodes in a linked list.  
**Auxiliary Space:**  $O(1)$

**Source**

<https://www.geeksforgeeks.org/sum-of-the-nodes-of-a-singly-linked-list/>

## Chapter 252

# Swap Kth node from beginning with Kth node from end in a Linked List

Swap Kth node from beginning with Kth node from end in a Linked List - GeeksforGeeks

Given a singly linked list, swap kth node from beginning with kth node from end. **Swapping of data is not allowed, only pointers should be changed.** This requirement may be logical in many situations where the linked list data part is huge (For example student details like Name, RollNo, Address, ..etc). The pointers are always fixed (4 bytes for most of the compilers).

The problem seems simple at first look, but it has many interesting cases.

Let X be the kth node from beginning and Y be the kth node from end. Following are the interesting cases that must be handled.

- 1) Y is next to X
- 2) X is next to Y
- 3) X and Y are same
- 4) X and Y don't exist (k is more than number of nodes in linked list)

C++

```
// A C++ program to swap Kth node from beginning with kth node from end
#include <iostream>
#include <stdlib.h>
using namespace std;

// A Linked List node
struct Node
{
    int data;
    struct Node *next;
}
```

```
};

/* Utility function to insert a node at the beginning */
void push(struct Node **head_ref, int new_data)
{
    struct Node *new_node = (struct Node *) malloc(sizeof(struct Node));
    new_node->data = new_data;
    new_node->next = (*head_ref);
    (*head_ref) = new_node;
}

/* Utility function for displaying linked list */
void printList(struct Node *node)
{
    while (node != NULL)
    {
        cout << node->data << " ";
        node = node->next;
    }
    cout << endl;
}

/* Utility function for calculating length of linked list */
int countNodes(struct Node *s)
{
    int count = 0;
    while (s != NULL)
    {
        count++;
        s = s->next;
    }
    return count;
}

/* Function for swapping kth nodes from both ends of linked list */
void swapKth(struct Node **head_ref, int k)
{
    // Count nodes in linked list
    int n = countNodes(*head_ref);

    // Check if k is valid
    if (n < k)  return;

    // If x (kth node from start) and y(kth node from end) are same
    if (2*k - 1 == n) return;

    // Find the kth node from beginning of linked list. We also find
    // previous of kth node because we need to update next pointer of
```

```
// the previous.
Node *x = *head_ref;
Node *x_prev = NULL;
for (int i = 1; i < k; i++)
{
    x_prev = x;
    x = x->next;
}

// Similarly, find the kth node from end and its previous. kth node
// from end is (n-k+1)th node from beginning
Node *y = *head_ref;
Node *y_prev = NULL;
for (int i = 1; i < n-k+1; i++)
{
    y_prev = y;
    y = y->next;
}

// If x_prev exists, then new next of it will be y. Consider the case
// when y->next is x, in this case, x_prev and y are same. So the statement
// "x_prev->next = y" creates a self loop. This self loop will be broken
// when we change y->next.
if (x_prev)
    x_prev->next = y;

// Same thing applies to y_prev
if (y_prev)
    y_prev->next = x;

// Swap next pointers of x and y. These statements also break self
// loop if x->next is y or y->next is x
Node *temp = x->next;
x->next = y->next;
y->next = temp;

// Change head pointers when k is 1 or n
if (k == 1)
    *head_ref = y;
if (k == n)
    *head_ref = x;
}

// Driver program to test above functions
int main()
{
    // Let us create the following linked list for testing
    // 1->2->3->4->5->6->7->8
```

```
struct Node *head = NULL;
for (int i = 8; i >= 1; i--)
    push(&head, i);

cout << "Original Linked List: ";
printList(head);

for (int k = 1; k < 9; k++)
{
    swapKth(&head, k);
    cout << "\nModified List for k = " << k << endl;
    printList(head);
}

return 0;
}
```

**Java**

```
// A Java program to swap kth node from the beginning with
// kth node from the end

class Node
{
    int data;
    Node next;
    Node(int d) { data = d;  next = null; }
}

class LinkedList
{
    Node head;

    /* Utility function to insert a node at the beginning */
    void push(int new_data)
    {
        Node new_node = new Node(new_data);
        new_node.next = head;
        head = new_node;
    }

    /* Utility function for displaying linked list */
    void printList()
    {
        Node node = head;
        while (node != null)
        {
            System.out.print(node.data + " ");
        }
    }
}
```

```
        node = node.next;
    }
    System.out.println("");
}

/* Utility function for calculating length of linked list */
int countNodes()
{
    int count = 0;
    Node s = head;
    while (s != null)
    {
        count++;
        s = s.next;
    }
    return count;
}

/* Function for swapping kth nodes from both ends of
   linked list */
void swapKth(int k)
{
    // Count nodes in linked list
    int n = countNodes();

    // Check if k is valid
    if (n < k)
        return;

    // If x (kth node from start) and y(kth node from end)
    // are same
    if (2*k - 1 == n)
        return;

    // Find the kth node from beginning of linked list.
    // We also find previous of kth node because we need
    // to update next pointer of the previous.
    Node x = head;
    Node x_prev = null;
    for (int i = 1; i < k; i++)
    {
        x_prev = x;
        x = x.next;
    }

    // Similarly, find the kth node from end and its
    // previous. kth node from end is (n-k+1)th node
    // from beginning
```

```
Node y = head;
Node y_prev = null;
for (int i = 1; i < n - k + 1; i++)
{
    y_prev = y;
    y = y.next;
}

// If x_prev exists, then new next of it will be y.
// Consider the case when y->next is x, in this case,
// x_prev and y are same. So the statement
// "x_prev->next = y" creates a self loop. This self
// loop will be broken when we change y->next.
if (x_prev != null)
    x_prev.next = y;

// Same thing applies to y_prev
if (y_prev != null)
    y_prev.next = x;

// Swap next pointers of x and y. These statements
// also break self loop if x->next is y or y->next
// is x
Node temp = x.next;
x.next = y.next;
y.next = temp;

// Change head pointers when k is 1 or n
if (k == 1)
    head = y;

if (k == n)
    head = x;
}

// Driver code to test above
public static void main(String[] args)
{
    LinkedList llist = new LinkedList();
    for (int i = 8; i >= 1; i--)
        llist.push(i);

    System.out.print("Original linked list: ");
    llist.printList();
    System.out.println("");

    for (int i = 1; i < 9; i++)
    {
```

```
        llist.swapKth(i);
        System.out.println("Modified List for k = " + i);
        llist.printList();
        System.out.println("");
    }
}
}
```

Output:

Original Linked List: 1 2 3 4 5 6 7 8

Modified List for k = 1  
8 2 3 4 5 6 7 1

Modified List for k = 2  
8 7 3 4 5 6 2 1

Modified List for k = 3  
8 7 6 4 5 3 2 1

Modified List for k = 4  
8 7 6 5 4 3 2 1

Modified List for k = 5  
8 7 6 4 5 3 2 1

Modified List for k = 6  
8 7 3 4 5 6 2 1

Modified List for k = 7  
8 2 3 4 5 6 7 1

Modified List for k = 8  
1 2 3 4 5 6 7 8

Please note that the above code runs three separate loops to count nodes, find x and x prev, and to find y and y\_prev. These three things can be done in a single loop. The code uses three loops to keep things simple and readable.

Thanks to [Chandra Prakash](#) for 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/swap-kth-node-from-beginning-with-kth-node-from-end-in-a-linked-list/>

## Chapter 253

# Swap nodes in a linked list without swapping data

Swap nodes in a linked list without swapping data - GeeksforGeeks

Given a linked list and two keys in it, swap nodes for two given keys. Nodes should be swapped by changing links. Swapping data of nodes may be expensive in many situations when data contains many fields.

It may be assumed that all keys in linked list are distinct.

**Examples:**

Input: 10->15->12->13->20->14, x = 12, y = 20  
Output: 10->15->20->13->12->14

Input: 10->15->12->13->20->14, x = 10, y = 20  
Output: 20->15->12->13->10->14

Input: 10->15->12->13->20->14, x = 12, y = 13  
Output: 10->15->13->12->20->14

This may look a simple problem, but is interesting question as it has following cases to be handled.

- 1) x and y may or may not be adjacent.
- 2) Either x or y may be a head node.
- 3) Either x or y may be last node.
- 4) x and/or y may not be present in linked list.

How to write a clean working code that handles all of the above possibilities.

**We strongly recommend to minimize your browser and try this yourself first.**

The idea is to first search x and y in given linked list. If any of them is not present, then return. While searching for x and y, keep track of current and previous pointers. First change next of previous pointers, then change next of current pointers. Following are C and Java implementations of this approach.

C

```
/* This program swaps the nodes of linked list rather
   than swapping the field from the nodes.*/

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

/* A linked list node */
struct Node
{
    int data;
    struct Node *next;
};

/* Function to swap nodes x and y in linked list by
   changing links */
void swapNodes(struct Node **head_ref, int x, int y)
{
    // Nothing to do if x and y are same
    if (x == y) return;

    // Search for x (keep track of prevX and CurrX
    struct Node *prevX = NULL, *currX = *head_ref;
    while (currX && currX->data != x)
    {
        prevX = currX;
        currX = currX->next;
    }

    // Search for y (keep track of prevY and CurrY
    struct Node *prevY = NULL, *currY = *head_ref;
    while (currY && currY->data != y)
    {
        prevY = currY;
        currY = currY->next;
    }

    // If either x or y is not present, nothing to do
    if (currX == NULL || currY == NULL)
        return;

    // If x is not head of linked list
    if (prevX != NULL)
```

```
    prevX->next = currY;
else // Else make y as new head
    *head_ref = currY;

// If y is not head of linked list
if (prevY != NULL)
    prevY->next = currX;
else // Else make x as new head
    *head_ref = currX;

// Swap next pointers
struct Node *temp = currY->next;
currY->next = currX->next;
currX->next = temp;
}

/* Function to add a node at the begining of List */
void push(struct Node** head_ref, int new_data)
{
    /* allocate node */
    struct Node* new_node =
        (struct Node*) malloc(sizeof(struct Node));

    /* put in the data */
    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;
}

/* Function to print nodes in a given linked list */
void printList(struct Node *node)
{
    while(node != NULL)
    {
        printf("%d ", node->data);
        node = node->next;
    }
}

/* Druver program to test above function */
int main()
{
    struct Node *start = NULL;
```

```
/* The constructed linked list is:  
1->2->3->4->5->6->7 */  
push(&start, 7);  
push(&start, 6);  
push(&start, 5);  
push(&start, 4);  
push(&start, 3);  
push(&start, 2);  
push(&start, 1);  
  
printf("\n Linked list before calling swapNodes() ");  
printList(start);  
  
swapNodes(&start, 4, 3);  
  
printf("\n Linked list after calling swapNodes() ");  
printList(start);  
  
return 0;  
}
```

### Java

```
// Java program to swap two given nodes of a linked list  
  
class Node  
{  
    int data;  
    Node next;  
    Node(int d)  
    {  
        data = d;  
        next = null;  
    }  
}  
  
class LinkedList  
{  
    Node head; // head of list  
  
    /* Function to swap Nodes x and y in linked list by  
       changing links */  
    public void swapNodes(int x, int y)  
    {  
        // Nothing to do if x and y are same  
        if (x == y) return;  
  
        // Search for x (keep track of prevX and CurrX)
```

```
Node prevX = null, currX = head;
while (currX != null && currX.data != x)
{
    prevX = currX;
    currX = currX.next;
}

// Search for y (keep track of prevY and currY)
Node prevY = null, currY = head;
while (currY != null && currY.data != y)
{
    prevY = currY;
    currY = currY.next;
}

// If either x or y is not present, nothing to do
if (currX == null || currY == null)
    return;

// If x is not head of linked list
if (prevX != null)
    prevX.next = currY;
else //make y the new head
    head = currY;

// If y is not head of linked list
if (prevY != null)
    prevY.next = currX;
else // make x the new head
    head = currX;

// Swap next pointers
Node temp = currX.next;
currX.next = currY.next;
currY.next = temp;
}

/* Function to add Node at beginning of list. */
public void push(int new_data)
{
    /* 1. alloc the Node and put the data */
    Node new_Node = new Node(new_data);

    /* 2. Make next of new Node as head */
    new_Node.next = head;

    /* 3. Move the head to point to new Node */
    head = new_Node;
```

```
}

/* This function prints contents of linked list starting
   from the given Node */
public void printList()
{
    Node tNode = head;
    while (tNode != null)
    {
        System.out.print(tNode.data+" ");
        tNode = tNode.next;
    }
}

/* Druver program to test above function */
public static void main(String[] args)
{
    LinkedList llist = new LinkedList();

    /* The constructed linked list is:
       1->2->3->4->5->6->7 */
    llist.push(7);
    llist.push(6);
    llist.push(5);
    llist.push(4);
    llist.push(3);
    llist.push(2);
    llist.push(1);

    System.out.print("\n Linked list before calling swapNodes() ");
    llist.printList();

    llist.swapNodes(4, 3);

    System.out.print("\n Linked list after calling swapNodes() ");
    llist.printList();
}
}

// This code is contributed by Rajat Mishra
```

### Python

```
# Python program to swap two given nodes of a linked list
class LinkedList(object):
    def __init__(self):
        self.head = None

    # head of list
```

```
class Node(object):
    def __init__(self, d):
        self.data = d
        self.next = None

    # Function to swap Nodes x and y in linked list by
    # changing links
    def swapNodes(self, x, y):

        # Nothing to do if x and y are same
        if x == y:
            return

        # Search for x (keep track of prevX and CurrX)
        prevX = None
        currX = self.head
        while currX != None and currX.data != x:
            prevX = currX
            currX = currX.next

        # Search for y (keep track of prevY and currY)
        prevY = None
        currY = self.head
        while currY != None and currY.data != y:
            prevY = currY
            currY = currY.next

        # If either x or y is not present, nothing to do
        if currX == None or currY == None:
            return
        # If x is not head of linked list
        if prevX != None:
            prevX.next = currY
        else: #make y the new head
            self.head = currY

        # If y is not head of linked list
        if prevY != None:
            prevY.next = currX
        else: # make x the new head
            self.head = currX

        # Swap next pointers
        temp = currX.next
        currX.next = currY.next
        currY.next = temp

    # Function to add Node at beginning of list.
```

```
def push(self, new_data):

    # 1. alloc the Node and put the data
    new_Node = self.Node(new_data)

    # 2. Make next of new Node as head
    new_Node.next = self.head

    # 3. Move the head to point to new Node
    self.head = new_Node

# This function prints contents of linked list starting
# from the given Node
def printList(self):
    tNode = self.head
    while tNode != None:
        print tNode.data,
        tNode = tNode.next

# Driver program to test above function
llist = LinkedList()

# The constructed linked list is:
# 1->2->3->4->5->6->7
llist.push(7)
llist.push(6)
llist.push(5)
llist.push(4)
llist.push(3)
llist.push(2)
llist.push(1)
print "Linked list before calling swapNodes() "
llist.printList()
llist.swapNodes(4, 3)
print "\nLinked list after calling swapNodes() "
llist.printList()

# This code is contributed by BHAVYA JAIN
```

Output:

```
Linked list before calling swapNodes() 1 2 3 4 5 6 7
Linked list after calling swapNodes() 1 2 4 3 5 6 7
```

**Optimizations:** The above code can be optimized to search x and y in single traversal. Two loops are used to keep program simple.

**Simpler approach –**

```
#include <iostream>

using namespace std;

// A linked list node class
class Node {

public:
    int data;
    class Node* next;

    // constructor
    Node(int val, Node* next)
        : data(val), next(next)
    {
    }

    // print list from this
    // to last till null
    void printList()
    {

        Node* node = this;

        while (node != NULL) {

            cout << node->data;
            node = node->next;
        }

        cout << endl;
    }
};

// Function to add a node
// at the beginning of List
void push(Node** head_ref, int new_data)
{

    // allocate node
    (*head_ref) = new Node(new_data, *head_ref);
}

void swap(Node*& a, Node*& b)
{

    Node* temp = a;
    a = b;
    b = temp;
}
```

```
b = temp;
}

void swapNodes(Node** head_ref, int x, int y)
{

    // Nothing to do if x and y are same
    if (x == y)
        return;

    Node **a = NULL, **b = NULL;

    // search for x and y in the linked list
    // and store their pointer in a and b
    while (*head_ref) {

        if ((*head_ref)->data == x) {
            a = head_ref;
        }

        else if ((*head_ref)->data == y) {
            b = head_ref;
        }

        head_ref = &((*head_ref)->next);
    }

    // if we have found both a and b
    // in the linked list swap current
    // pointer and next pointer of these
    if (a && b) {

        swap(*a, *b);
        swap((*a)->next, (*b)->next);
    }
}

int main()
{

    Node* start = NULL;

    // The constructed linked list is:
    // 1->2->3->4->5->6->7
    push(&start, 7);
    push(&start, 6);
    push(&start, 5);
    push(&start, 4);
```

```
push(&start, 3);
push(&start, 2);
push(&start, 1);

cout << "Linked list before calling swapNodes() ";
start->printList();

swapNodes(&start, 6, 3);

cout << "Linked list after calling swapNodes() ";
start->printList();
}
```

Output:

```
Linked list before calling swapNodes() 1 2 3 4 5 6 7
Linked list after calling swapNodes() 1 2 6 4 5 3 7
```

This article is contributed by **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/swap-nodes-in-a-linked-list-without-swapping-data/>

## Chapter 254

# 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 255

# Union and Intersection of two Linked Lists

Union and Intersection of two Linked Lists - GeeksforGeeks

Given two Linked Lists, create union and intersection lists that contain union and intersection of the elements present in the given lists. Order of elements in output lists doesn't matter.

Example:

**Input:**

```
List1: 10->15->4->20  
List2: 8->4->2->10
```

**Output:**

```
Intersection List: 4->10  
Union List: 2->8->20->4->15->10
```

### Method 1 (Simple)

Following are simple algorithms to get union and intersection lists respectively.

*Intersection (list1, list2)*

Initialize result list as NULL. Traverse list1 and look for its each element in list2, if the element is present in list2, then add the element to result.

*Union (list1, list2):*

Initialize result list as NULL. Traverse list1 and add all of its elements to the result. Traverse list2. If an element of list2 is already present in result then do not insert it to result, otherwise insert.

This method assumes that there are no duplicates in the given lists.

Thanks to Shekhu for suggesting this method. Following are C and Java implementations of this method.

C/C++

```
// C/C++ program to find union and intersection of two unsorted
// linked lists
#include<stdio.h>
#include<stdlib.h>
#include<stdbool.h>
/* Link list node */
struct Node
{
    int data;
    struct Node* next;
};

/* A utility function to insert a node at the beginning of
   a linked list*/
void push(struct Node** head_ref, int new_data);

/* A utility function to check if given data is present in a list */
bool isPresent(struct Node *head, int data);

/* Function to get union of two linked lists head1 and head2 */
struct Node *getUnion(struct Node *head1, struct Node *head2)
{
    struct Node *result = NULL;
    struct Node *t1 = head1, *t2 = head2;

    // Insert all elements of list1 to the result list
    while (t1 != NULL)
    {
        push(&result, t1->data);
        t1 = t1->next;
    }

    // Insert those elements of list2 which are not
    // present in result list
    while (t2 != NULL)
    {
        if (!isPresent(result, t2->data))
            push(&result, t2->data);
        t2 = t2->next;
    }

    return result;
}

/* Function to get intersection of two linked lists
   head1 and head2 */
```

```
struct Node *getIntersection(struct Node *head1,
                             struct Node *head2)
{
    struct Node *result = NULL;
    struct Node *t1 = head1;

    // Traverse list1 and search each element of it in
    // list2. If the element is present in list 2, then
    // insert the element to result
    while (t1 != NULL)
    {
        if (isPresent(head2, t1->data))
            push (&result, t1->data);
        t1 = t1->next;
    }

    return result;
}

/* A utility function to insert a node at the begining of a linked list*/
void push (struct Node** head_ref, int new_data)
{
    /* allocate node */
    struct Node* new_node =
        (struct Node*) malloc(sizeof(struct Node));

    /* put in the data */
    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;
}

/* A utility function to print a linked list*/
void printList (struct Node *node)
{
    while (node != NULL)
    {
        printf ("%d ", node->data);
        node = node->next;
    }
}

/* A utility function that returns true if data is
   present in linked list else return false */
```

```
bool isPresent (struct Node *head, int data)
{
    struct Node *t = head;
    while (t != NULL)
    {
        if (t->data == data)
            return 1;
        t = t->next;
    }
    return 0;
}

/* Drier program to test above function*/
int main()
{
    /* Start with the empty list */
    struct Node* head1 = NULL;
    struct Node* head2 = NULL;
    struct Node* intersecn = NULL;
    struct Node* unin = NULL;

    /*create a linked lists 10->15->5->20 */
    push (&head1, 20);
    push (&head1, 4);
    push (&head1, 15);
    push (&head1, 10);

    /*create a linked lists 8->4->2->10 */
    push (&head2, 10);
    push (&head2, 2);
    push (&head2, 4);
    push (&head2, 8);

    intersecn = getIntersection (head1, head2);
    unin = getUnion (head1, head2);

    printf ("\n First list is \n");
    printList (head1);

    printf ("\n Second list is \n");
    printList (head2);

    printf ("\n Intersection list is \n");
    printList (intersecn);

    printf ("\n Union list is \n");
    printList (unin);
```

```
    return 0;
}
```

**Java**

```
// Java program to find union and intersection of two unsorted
// linked lists
class LinkedList
{
    Node head; // head of list

    /* Linked list Node*/
    class Node
    {
        int data;
        Node next;
        Node(int d)
        {
            data = d;
            next = null;
        }
    }

    /* Function to get Union of 2 Linked Lists */
    void getUnion(Node head1, Node head2)
    {
        Node t1 = head1, t2 = head2;

        //insert all elements of list1 in the result
        while (t1 != null)
        {
            push(t1.data);
            t1 = t1.next;
        }

        // insert those elements of list2 that are not present
        while (t2 != null)
        {
            if (!isPresent(head, t2.data))
                push(t2.data);
            t2 = t2.next;
        }
    }

    void getIntersection(Node head1, Node head2)
    {
        Node result = null;
        Node t1 = head1;
```

```
// Traverse list1 and search each element of it in list2.  
// If the element is present in list 2, then insert the  
// element to result  
while (t1 != null)  
{  
    if (isPresent(head2, t1.data))  
        push(t1.data);  
    t1 = t1.next;  
}  
  
/* Utility function to print list */  
void printList()  
{  
    Node temp = head;  
    while(temp != null)  
    {  
        System.out.print(temp.data+" ");  
        temp = temp.next;  
    }  
    System.out.println();  
}  
  
/* Inserts a node at start of linked list */  
void push(int new_data)  
{  
    /* 1 & 2: Allocate the Node &  
       Put in the data*/  
    Node new_node = new Node(new_data);  
  
    /* 3. Make next of new Node as head */  
    new_node.next = head;  
  
    /* 4. Move the head to point to new Node */  
    head = new_node;  
}  
  
/* A utility function that returns true if data is present  
   in linked list else return false */  
boolean isPresent (Node head, int data)  
{  
    Node t = head;  
    while (t != null)  
    {  
        if (t.data == data)
```

```
        return true;
    t = t.next;
}
return false;
}

/* Drier program to test above functions */
public static void main(String args[])
{
    LinkedList llist1 = new LinkedList();
    LinkedList llist2 = new LinkedList();
    LinkedList unin = new LinkedList();
    LinkedList intersecn = new LinkedList();

    /*create a linked lists 10->15->5->20 */
    llist1.push(20);
    llist1.push(4);
    llist1.push(15);
    llist1.push(10);

    /*create a linked lists 8->4->2->10 */
    llist2.push(10);
    llist2.push(2);
    llist2.push(4);
    llist2.push(8);

    intersecn.getIntersection(llist1.head, llist2.head);
    unin.getUnion(llist1.head, llist2.head);

    System.out.println("First List is");
    llist1.printList();

    System.out.println("Second List is");
    llist2.printList();

    System.out.println("Intersection List is");
    intersecn.printList();

    System.out.println("Union List is");
    unin.printList();
}
} /* This code is contributed by Rajat Mishra */
```

Output:

```
First list is
```

```
10 15 4 20
Second list is
8 4 2 10
Intersection list is
4 10
Union list is
2 8 20 4 15 10
```

Time Complexity:  $O(mn)$  for both union and intersection operations. Here  $m$  is the number of elements in first list and  $n$  is the number of elements in second list.

### **Method 2 (Use Merge Sort)**

In this method, algorithms for Union and Intersection are very similar. First we sort the given lists, then we traverse the sorted lists to get union and intersection.

Following are the steps to be followed to get union and intersection lists.

- 1) Sort the first Linked List using merge sort. This step takes  $O(m\log m)$  time. Refer [this post](#) for details of this step.
- 2) Sort the second Linked List using merge sort. This step takes  $O(n\log n)$  time. Refer [this post](#) for details of this step.
- 3) Linearly scan both sorted lists to get the union and intersection. This step takes  $O(m + n)$  time. This step can be implemented using the same algorithm as sorted arrays algorithm discussed [here](#).

Time complexity of this method is  $O(m\log m + n\log n)$  which is better than method 1's time complexity.

### **Method 3 (Use Hashing)**

*Union (list1, list2)*

Initialize the result list as NULL and create an empty hash table. Traverse both lists one by one, for each element being visited, look the element in hash table. If the element is not present, then insert the element to result list. If the element is present, then ignore it.

*Intersection (list1, list2)*

Initialize the result list as NULL and create an empty hash table. Traverse list1. For each element being visited in list1, insert the element in hash table. Traverse list2, for each element being visited in list2, look the element in hash table. If the element is present, then insert the element to result list. If the element is not present, then ignore it.

Both of the above methods assume that there are no duplicates.

```
// Java code for Union and Intersection of two
// Linked Lists
import java.util.HashMap;
import java.util.HashSet;

class LinkedList {
    Node head; // head of list

    /* Linked list Node*/
    class Node
```

```
{  
    int data;  
    Node next;  
    Node(int d)  
    {  
        data = d;  
        next = null;  
    }  
}  
  
/* Utility function to print list */  
void printList()  
{  
    Node temp = head;  
    while(temp != null)  
    {  
        System.out.print(temp.data+" ");  
        temp = temp.next;  
    }  
    System.out.println();  
}  
  
/* Inserts a node at start of linked list */  
void push(int new_data)  
{  
    /* 1 & 2: Allocate the Node &  
    Put in the data*/  
    Node new_node = new Node(new_data);  
  
    /* 3. Make next of new Node as head */  
    new_node.next = head;  
  
    /* 4. Move the head to point to new Node */  
    head = new_node;  
}  
  
public void append(int new_data)  
{  
    if(this.head == null)  
    {  
        Node n = new Node(new_data);  
        this.head = n;  
        return;  
    }  
    Node n1 = this.head;  
    Node n2 = new Node(new_data);  
    while(n1.next != null)  
    {  
        n1 = n1.next;  
    }  
    n1.next = n2;  
}
```

```
n1 = n1.next;
}

n1.next = n2;
n2.next = null;
}

/* A utility function that returns true if data is
present in linked list else return false */
boolean isPresent (Node head, int data)
{
    Node t = head;
    while (t != null)
    {
        if (t.data == data)
            return true;
        t = t.next;
    }
    return false;
}

LinkedList getIntersection(Node head1, Node head2)
{
    HashSet<Integer> hset = new HashSet<>();
    Node n1 = head1;
    Node n2 = head2;
    LinkedList result = new LinkedList();

    // loop stores all the elements of list1 in hset
    while(n1 != null)
    {
        if(hset.contains(n1.data))
        {
            hset.add(n1.data);
        }
        else
        {
            hset.add(n1.data);
        }
        n1 = n1.next;
    }

    //For every element of list2 present in hset
    //loop inserts the element into the result
    while(n2 != null)
    {
        if(hset.contains(n2.data))
        {
```

```
        result.push(n2.data);
    }
    n2 = n2.next;
}
return result;
}

LinkedList getUnion(Node head1, Node head2)
{
    // HashMap that will store the
    // elements of the lists with their counts
    HashMap<Integer, Integer> hmap = new HashMap<>();
    Node n1 = head1;
    Node n2 = head2;
    LinkedList result = new LinkedList();

    // loop inserts the elements and the count of
    // that element of list1 into the hmap
    while(n1 != null)
    {
        if(hmap.containsKey(n1.data))
        {
            int val = hmap.get(n1.data);
            hmap.put(n1.data, val + 1);
        }
        else
        {
            hmap.put(n1.data, 1);
        }
        n1 = n1.next;
    }

    // loop further adds the elements of list2 with
    // their counts into the hmap
    while(n2 != null)
    {
        if(hmap.containsKey(n2.data))
        {
            int val = hmap.get(n2.data);
            hmap.put(n2.data, val + 1);
        }
        else
        {
            hmap.put(n2.data, 1);
        }
        n2 = n2.next;
    }
}
```

```
// Eventually add all the elements
// into the result that are present in the hmap
for(int a:hmap.keySet())
{
    result.append(a);
}
return result;
}

/* Driver program to test above functions */
public static void main(String args[])
{
    LinkedList llist1 = new LinkedList();
    LinkedList llist2 = new LinkedList();
    LinkedList union = new LinkedList();
    LinkedList intersection = new LinkedList();

    /*create a linked list 10->15->4->20 */
    llist1.push(20);
    llist1.push(4);
    llist1.push(15);
    llist1.push(10);

    /*create a linked list 8->4->2->10 */
    llist2.push(10);
    llist2.push(2);
    llist2.push(4);
    llist2.push(8);

    intersection = intersection.getIntersection(llist1.head,
                                                llist2.head);
    union=union.getUnion(llist1.head, llist2.head);

    System.out.println("First List is");
    llist1.printList();

    System.out.println("Second List is");
    llist2.printList();

    System.out.println("Intersection List is");
    intersection.printList();

    System.out.println("Union List is");
    union.printList();
}

}

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

Output:

```
First List is
10 15 4 20
Second List is
8 4 2 10
Intersection List is
10 4
Union List is
2 4 20 8 10 15
```

Time complexity of this method depends on the hashing technique used and the distribution of elements in input lists. In practical, this approach may turn out to be better than above 2 methods.

## Source

<https://www.geeksforgeeks.org/union-and-intersection-of-two-linked-lists/>

## Chapter 256

# Union and Intersection of two linked lists | Set-2 (Using Merge Sort)

Union and Intersection of two linked lists | Set-2 (Using Merge Sort) - GeeksforGeeks

Given two Linked Lists, create union and intersection lists that contain union and intersection of the elements present in the given lists. Order of elements in output lists doesn't matter.

Examples:

Input:

```
List1: 10 -> 15 -> 4 -> 20  
List2: 8 -> 4 -> 2 -> 10
```

Output:

```
Intersection List: 4 -> 10  
Union List: 2 -> 8 -> 20 -> 4 -> 15 -> 10
```

There were three methods discussed in this [post](#) with an implementation of Method 1. In this post, we will see an implementation of Method 2 i.e. Using [Merge sort](#).

Implementation:

Following are the steps to be followed to get union and intersection lists.

- 1) Sort both Linked Lists using merge sort.  
This step takes  $O(m \log m)$  time.
- 2) Linearly scan both sorted lists to get the union and intersection.  
This step takes  $O(m + n)$  time.

Just like Method 1, This method also assumes that there are distinct elements in the lists.

```
// C++ program to find union and intersection of
// two unsorted linked lists in O(n Log n) time.
#include<iostream>
using namespace std;

/* Link list node */
struct Node
{
    int data;
    struct Node* next;
};

/* A utility function to insert a node at the begining
   of a linked list*/
void push(struct Node** head_ref, int new_data)
{
    /* allocate node */
    struct Node* new_node =
        (struct Node*) malloc(sizeof(struct Node));

    /* put in the data */
    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;
}

/* UTILITY FUNCTIONS */
/* Split the nodes of the given list into front and back halves,
   and return the two lists using the reference parameters.
   If the length is odd, the extra node should go in the front list.
   Uses the fast/slow pointer strategy. */
void FrontBackSplit(struct Node* source, struct Node** frontRef,
                    struct Node** backRef)
{
    struct Node* fast;
    struct Node* slow;
    if (source==NULL || source->next==NULL)
    {
        /* length < 2 cases */
        *frontRef = source;
        *backRef = NULL;
    }
}
```

```
else
{
    slow = source;
    fast = source->next;

    /* Advance 'fast' two nodes, and advance 'slow' one node */
    while (fast != NULL)
    {
        fast = fast->next;
        if (fast != NULL)
        {
            slow = slow->next;
            fast = fast->next;
        }
    }

    /* 'slow' is before the midpoint in the list,
       so split it in two at that point. */
    *frontRef = source;
    *backRef = slow->next;
    slow->next = NULL;
}

/*
 * See https://www.geeksforgeeks.org/?p=3622 for details
 * of this function */
struct Node* SortedMerge(struct Node* a, struct Node* b)
{
    struct Node* result = NULL;

    /* Base cases */
    if (a == NULL)
        return(b);
    else if (b==NULL)
        return(a);

    /* Pick either a or b, and recur */
    if (a->data <= b->data)
    {
        result = a;
        result->next = SortedMerge(a->next, b);
    }
    else
    {
        result = b;
        result->next = SortedMerge(a, b->next);
    }
    return(result);
}
```

```
}

/* sorts the linked list by changing next pointers
 * (not data) */
void mergeSort(struct Node** headRef)
{
    struct Node *head = *headRef;
    struct Node *a, *b;

    /* Base case -- length 0 or 1 */
    if ((head == NULL) || (head->next == NULL))
        return;

    /* Split head into 'a' and 'b' sublists */
    FrontBackSplit(head, &a, &b);

    /* Recursively sort the sublists */
    mergeSort(&a);
    mergeSort(&b);

    /* answer = merge the two sorted lists together */
    *headRef = SortedMerge(a, b);
}

/* Function to get union of two linked lists head1 and head2 */
struct Node *getUnion(struct Node *head1, struct Node *head2)
{
    struct Node *result = NULL;
    struct Node *t1 = head1, *t2 = head2;

    // Traverse both lists and store the element in
    // the resultant list
    while (t1 != NULL && t2 != NULL)
    {
        // Move to the next of first list
        // if its element is smaller
        if (t1->data < t2->data)
        {
            push(&result, t1->data);
            t1 = t1->next;
        }

        // Else move to the next of second list
        else if (t1->data > t2->data)
        {
            push(&result, t2->data);
            t2 = t2->next;
        }
    }
}
```

```
}

// If same then move to the next node
// in both lists
else
{
    push(&result, t2->data);
    t1 = t1->next;
    t2 = t2->next;
}
}

/* Print remaining elements of the lists */
while (t1 != NULL)
{
    push(&result, t1->data);
    t1 = t1->next;
}
while (t2 != NULL)
{
    push(&result, t2->data);
    t2 = t2->next;
}

return result;
}

/* Function to get intersection of two linked lists
head1 and head2 */
struct Node *getIntersection(struct Node *head1,
                             struct Node *head2)
{
    struct Node *result = NULL;
    struct Node *t1 = head1, *t2 = head2;

    // Traverse both lists and store the same element
    // in the resultant list
    while (t1 != NULL && t2 != NULL)
    {
        // Move to the next of first list if smaller
        if (t1->data < t2->data)
            t1 = t1->next;

        // Move to the next of second list if it is smaller
        else if (t1->data > t2->data)
            t2 = t2->next;

        // If both are same
    }
}
```

```
    else
    {
        // Store current element in the list
        push(&result, t2->data);

        // Move to the next node of both lists
        t1 = t1->next;
        t2 = t2->next;
    }
}

//return the resultant list
return result;
}

/* A utility function to print a linked list*/
void printList (struct Node *node)
{
    while (node != NULL)
    {
        printf ("%d ", node->data);
        node = node->next;
    }
}

/* Drier program to test above function*/
int main()
{
    /* Start with the empty list */
    struct Node* head1 = NULL;
    struct Node* head2 = NULL;
    struct Node* intersection_list = NULL;
    struct Node* union_list = NULL;

    /*create a linked lits 11->10->15->4->20 */
    push(&head1, 20);
    push(&head1, 4);
    push(&head1, 15);
    push(&head1, 10);
    push(&head1, 11);

    /*create a linked lits 8->4->2->10 */
    push(&head2, 10);
    push(&head2, 2);
    push(&head2, 4);
    push(&head2, 8);
```

```
/* Sort the above created Linked List */
mergeSort(&head1);
mergeSort(&head2);

intersection_list = getIntersection(head1, head2);
union_list = getUnion(head1, head2);

printf("First list is \n");
printList(head1);

printf("\nSecond list is \n");
printList(head2);

printf("\nIntersection list is \n");
printList(intersection_list);

printf("\nUnion list is \n");
printList(union_list);

return 0;
}
```

Output:

```
First list is
4 10 11 15 20
Second list is
2 4 8 10
Intersection list is
10 4
Union list is
20 15 11 10 8 4 2
```

Time complexity of this method is  $O(m \log m + n \log n)$ .

In the next post, Method-3 will be discussed i.e. using hashing.

## Source

<https://www.geeksforgeeks.org/union-intersection-two-linked-lists-set-2-using-merge-sort/>

## Chapter 257

# Union and Intersection of two linked lists | Set-3 (Hashing)

Union and Intersection of two linked lists | Set-3 (Hashing) - GeeksforGeeks

Given two Linked Lists, create union and intersection lists that contain union and intersection of the elements present in the given lists. Order of elements in output lists doesn't matter.

Examples:

Input:

List1: 10 → 15 → 4 → 20  
List2: 8 → 4 → 2 → 10

Output:

Intersection List: 4 → 10  
Union List: 2 → 8 → 20 → 4 → 15 → 10

We have already discussed [Method-1](#) and [Method-2](#) of this question.

In this post, its Method-3 (Using Hashing) is discussed with a Time Complexity of  $O(m+n)$  i.e. better than both methods discussed earlier.

Implementation:

- 1- Start traversing both the lists.
  - a) Store the current element of both lists with its occurrence in the map.
- 2- For Union: Store all the elements of the map in the resultant list.
- 3- For Intersection: Store all the elements only with an occurrence of 2 as 2 denotes that they are present in both the lists.

Below is C++ implementation of above steps.

```
// C++ program to find union and intersection of
// two unsorted linked lists in O(m+n) time.
#include<bits/stdc++.h>
using namespace std;

/* Link list node */
struct Node
{
    int data;
    struct Node* next;
};

/* A utility function to insert a node at the
   begining of a linked list*/
void push(struct Node** head_ref, int new_data)
{
    /* allocate node */
    struct Node* new_node =
        (struct Node*) malloc(sizeof(struct Node));

    /* put in the data */
    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;
}

/* Utility function to store the elements of both list */
void storeEle(struct Node* head1, struct Node *head2,
              unordered_map<int, int> &eleOcc)
{
    struct Node* ptr1 = head1;
    struct Node* ptr2 = head2;

    // Traverse both lists
    while (ptr1 != NULL || ptr2 != NULL)
    {
        // store element in the map
        if (ptr1!=NULL)
        {
            eleOcc[ptr1->data]++;
            ptr1=ptr1->next;
        }
    }
}
```

```
// store element in the map
if (ptr2 != NULL)
{
    eleOcc[ptr2->data]++;
    ptr2=ptr2->next;
}
}

/* Function to get union of two linked lists head1
   and head2 */
struct Node *getUnion(unordered_map<int, int> eleOcc)
{
    struct Node *result = NULL;

    // Push all the elements into the resultant list
    for (auto it=eleOcc.begin(); it!=eleOcc.end(); it++)
        push(&result, it->first);

    return result;
}

/* Function to get intersection of two linked lists
   head1 and head2 */
struct Node *getIntersection(unordered_map<int, int> eleOcc)
{
    struct Node *result = NULL;

    // Push a node with an element having occurrence
    // of 2 as that means the current element is present
    // in both the lists
    for (auto it=eleOcc.begin(); it!=eleOcc.end(); it++)
        if (it->second == 2)
            push(&result, it->first);

    // return resultant list
    return result;
}

/* A utility function to print a linked list*/
void printList(struct Node *node)
{
    while (node != NULL)
    {
        printf ("%d ", node->data);
        node = node->next;
    }
}
```

```
}

// Prints union and intersection of lists with head1
// and head2.
void printUnionIntersection(Node *head1, Node *head2)
{
    // Store all the elements of both lists in the map
    unordered_map<int, int> eleOcc;
    storeEle(head1, head2, eleOcc);

    Node *intersection_list = getIntersection(eleOcc);
    Node *union_list = getUnion(eleOcc);

    printf("\nIntersection list is \n");
    printList(intersection_list);

    printf("\nUnion list is \n");
    printList(union_list);
}

/* Drier program to test above function*/
int main()
{
    /* Start with the empty list */
    struct Node* head1 = NULL;
    struct Node* head2 = NULL;

    /* create a linked lits 11->10->15->4->20 */
    push(&head1, 1);
    push(&head1, 2);
    push(&head1, 3);
    push(&head1, 4);
    push(&head1, 5);

    /* create a linked lits 8->4->2->10 */
    push(&head2, 1);
    push(&head2, 3);
    push(&head2, 5);
    push(&head2, 6);

    printf("First list is \n");
    printList(head1);

    printf("\nSecond list is \n");
    printList(head2);

    printUnionIntersection(head1, head2);
```

```
    return 0;  
}
```

Output:

```
First list is  
5 4 3 2 1  
Second list is  
6 5 3 1  
Intersection list is  
3 5 1  
Union list is  
3 4 6 5 2 1
```

We can also handle the case of duplicates by maintaining separate Hash for both the lists.

Time Complexity :  $O(m + n)$

Auxiliary Space :  $O(m + n)$

## Source

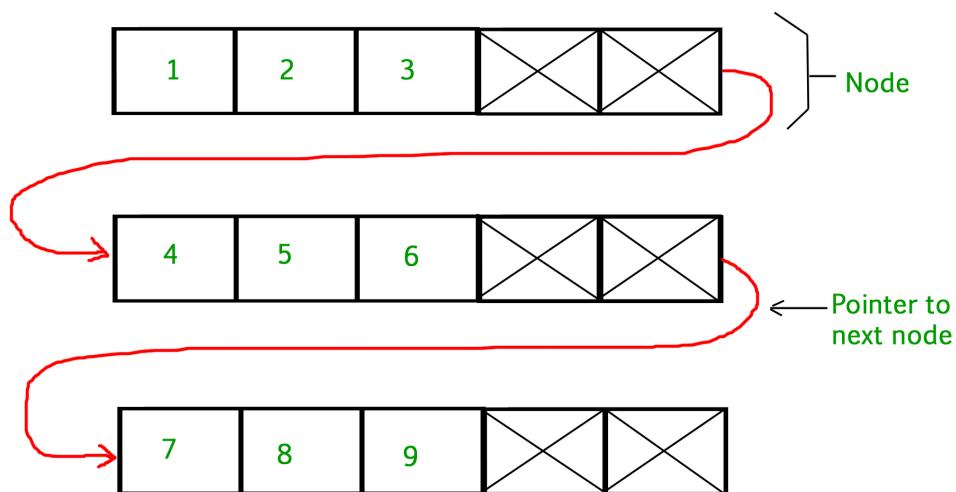
<https://www.geeksforgeeks.org/union-intersection-two-linked-lists-set-3-hashing/>

## Chapter 258

# Unrolled Linked List | Set 1 (Introduction)

Unrolled Linked List | Set 1 (Introduction) - GeeksforGeeks

Like array and linked list, unrolled Linked List is also a linear data structure and is a variant of linked list. Unlike simple linked list, it stores multiple elements at each node. That is, instead of storing single element at a node, unrolled linked lists store an array of elements at a node. Unrolled linked list covers advantages of both array and linked list as it reduces the memory overhead in comparison to simple linked lists by storing multiple elements at each node and it also has the advantage of fast insertion and deletion as that of a linked list.



Advantages:

- Because of the Cache behavior, linear search is much faster in unrolled linked lists.
- In comparison to ordinary linked list, it requires less storage space for pointers/references.
- It performs operations like insertion, deletion and traversal more quickly than ordinary linked lists (because search is faster).

**Disadvantages:**

- The overhead per node is comparatively high than singly linked lists. Refer an example node in below code.

**Simple Implementation in C**

The below program creates a simple unrolled linked list with 3 nodes containing variable number of elements in each. It also traverses the created list.

```
// C program to implement unrolled linked list
// and traversing it.
#include<stdio.h>
#include<stdlib.h>
#define maxElements 4

// Unrolled Linked List Node
struct Node
{
    int numElements;
    int array[maxElements];
    struct Node *next;
};

/* Function to traverse am unrolled linked list
   and print all the elements*/
void printUnrolledList(struct Node *n)
{
    while (n != NULL)
    {
        // Print elements in current node
        for (int i=0; i<n->numElements; i++)
            printf("%d ", n->array[i]);

        // Move to next node
        n = n->next;
    }
}

// Program to create an unrolled linked list
// with 3 Nodes
int main()
```

```
{  
    struct Node* head = NULL;  
    struct Node* second = NULL;  
    struct Node* third = NULL;  
  
    // allocate 3 Nodes  
    head = (struct Node*)malloc(sizeof(struct Node));  
    second = (struct Node*)malloc(sizeof(struct Node));  
    third = (struct Node*)malloc(sizeof(struct Node));  
  
    // Let us put some values in second node (Number  
    // of values must be less than or equal to  
    // maxElement)  
    head->numElements = 3;  
    head->array[0] = 1;  
    head->array[1] = 2;  
    head->array[2] = 3;  
  
    // Link first Node with the second Node  
    head->next = second;  
  
    // Let us put some values in second node (Number  
    // of values must be less than or equal to  
    // maxElement)  
    second->numElements = 3;  
    second->array[0] = 4;  
    second->array[1] = 5;  
    second->array[2] = 6;  
  
    // Link second Node with the third Node  
    second->next = third;  
  
    // Let us put some values in third node (Number  
    // of values must be less than or equal to  
    // maxElement)  
    third->numElements = 3;  
    third->array[0] = 7;  
    third->array[1] = 8;  
    third->array[2] = 9;  
    third->next = NULL;  
  
    printUnrolledList(head);  
  
    return 0;  
}
```

Output:

1 2 3 4 5 6 7 8 9

In this article, we have introduced unrolled list and advantages of it. We have also shown how to traverse the list. In the next article, we will be discussing insertion, deletion and values of maxElements/numElements in detail.

[Insertion in Unrolled Linked List](#)

### Source

<https://www.geeksforgeeks.org/unrolled-linked-list-set-1-introduction/>

## Chapter 259

# Write a function that counts the number of times a given int occurs in a Linked List

Write a function that counts the number of times a given int occurs in a Linked List - GeeksforGeeks

Given a singly linked list and a key, count number of occurrences of given key in linked list. For example, if given linked list is 1->2->1->2->1->3->1 and given key is 1, then output should be 4.

### Method 1- Without Recursion

#### Algorithm:

1. Initialize count as zero.
2. Loop through each element of linked list:
  - a) If element data is equal to the passed number then increment the count.
3. Return count.

#### Implementation:

#### C/C++

```
// C/C++ program to count occurrences in a linked list
#include<stdio.h>
#include<stdlib.h>

/* Link list node */
struct Node
```

```
{  
    int data;  
    struct Node* next;  
};  
  
/* Given a reference (pointer to pointer) to the head  
   of a list and an int, push a new node on the front  
   of the list. */  
void push(struct Node** head_ref, int new_data)  
{  
    /* allocate node */  
    struct Node* new_node =  
        (struct Node*) malloc(sizeof(struct Node));  
  
    /* put in the data */  
    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;  
}  
  
/* Counts the no. of occurrences of a node  
   (search_for) in a linked list (head)*/  
int count(struct Node* head, int search_for)  
{  
    struct Node* current = head;  
    int count = 0;  
    while (current != NULL)  
    {  
        if (current->data == search_for)  
            count++;  
        current = current->next;  
    }  
    return count;  
}  
  
/* Driver program to test count function*/  
int main()  
{  
    /* Start with the empty list */  
    struct Node* head = NULL;  
  
    /* Use push() to construct below list  
       1->2->1->3->1 */  
    push(&head, 1);
```

```
    push(&head, 3);
    push(&head, 1);
    push(&head, 2);
    push(&head, 1);

    /* Check the count function */
    printf("count of 1 is %d", count(head, 1));
    return 0;
}
```

**Java**

```
// Java program to count occurrences in a linked list
class LinkedList
{
    Node head; // head of list

    /* Linked list Node*/
    class Node
    {
        int data;
        Node next;
        Node(int d) {data = d; next = null; }
    }

    /* Inserts a new Node at front of the list. */
    public void push(int new_data)
    {
        /* 1 & 2: Allocate the Node &
           Put in the data*/
        Node new_node = new Node(new_data);

        /* 3. Make next of new Node as head */
        new_node.next = head;

        /* 4. Move the head to point to new Node */
        head = new_node;
    }

    /* Counts the no. of occurrences of a node
       (search_for) in a linked list (head)*/
    int count(int search_for)
    {
        Node current = head;
        int count = 0;
        while (current != null)
        {
            if (current.data == search_for)
```

```
        count++;
        current = current.next;
    }
    return count;
}

/* Drier function to test the above methods */
public static void main(String args[])
{
    LinkedList llist = new LinkedList();

    /* Use push() to construct below list
       1->2->1->3->1 */
    llist.push(1);
    llist.push(2);
    llist.push(1);
    llist.push(3);
    llist.push(1);

    /*Checking count function*/
    System.out.println("Count of 1 is "+llist.count(1));
}
}
// This code is contributed by Rajat Mishra
```

### Python

```
# Python program to count the number of time a given
# int occurs in a linked list

# Node class
class Node:

    # Constructor to initialize the node object
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:

    # Function to initialize head
    def __init__(self):
        self.head = None

    # Counts the no . of occurrences of a node
    # (search_for) in a linked list (head)
    def count(self, search_for):
        current = self.head
```

```
count = 0
while(current is not None):
    if current.data == search_for:
        count += 1
    current = current.next
return count

# Function to insert a new node at the beginning
def push(self, new_data):
    new_node = Node(new_data)
    new_node.next = self.head
    self.head = new_node

# Utility function to print the linked LinkedList
def printList(self):
    temp = self.head
    while(temp):
        print temp.data,
        temp = temp.next

# Driver program
llist = LinkedList()
llist.push(1)
llist.push(3)
llist.push(1)
llist.push(2)
llist.push(1)

# Check for the count function
print "count of 1 is %d" %(llist.count(1))

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

Output:

```
count of 1 is 3
```

**Time Complexity:** O(n)

**Auxiliary Space:** O(1)

### Method 2- With Recursion

This method is contributed by [MY\\_DOOM](#).

#### Algorithm:

```
Algorithm
count(head, key);
if head is NULL
```

```
return frequency
if(head->data==key)
increase frequency by 1
count(head->next,key)
```

**Implementation:**

C++

```
// C++ program to count occurrences in
// a linked list using recursion
#include <bits/stdc++.h>
using namespace std;

/* Link list node */
struct Node {
    int data;
    struct Node* next;
};

// global variable for counting frequency of
// given element k
int frequency = 0;

/* Given a reference (pointer to pointer) to the head
of a list and an int, push a new node on the front
of the list. */
void push(struct Node** head_ref, int new_data)
{
    /* allocate node */
    struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));

    /* put in the data */
    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;
}

/* Counts the no. of occurrences of a node
(search_for) in a linked list (head)*/
int count(struct Node* head, int key)
{
    if (head == NULL)
        return frequency;
```

```
if (head->data == key)
    frequency++;
return count(head->next, key);
}

/* Driver program to test count function*/
int main()
{
    /* Start with the empty list */
    struct Node* head = NULL;

    /* Use push() to construct below list
     * 1->2->1->3->1 */
    push(&head, 1);
    push(&head, 3);
    push(&head, 1);
    push(&head, 2);
    push(&head, 1);

    /* Check the count function */
    cout<<"count of 1 is " <<count(head, 1);
    return 0;
}
```

Output:

```
count of 1 is 3
```

The above method implements head recursion. Below given is the tail recursive implementation for the same. Thanks to **Puneet Jain** for suggesting this approach :

```
int count(struct Node* head, int key)
{
    if(head == NULL)
        return 0;

    int frequency = count(head->next, key);
    if(head->data == key)
        return 1 + frequency;

    // else
    return frequency;
}
```

**Time Complexity:** O(n)

**Improved By :** [Puneet Jain 1](#), [Shreyash Sharma 3](#)

*Chapter 259. Write a function that counts the number of times a given int occurs in a  
Linked List*

---

## Source

<https://www.geeksforgeeks.org/write-a-function-that-counts-the-number-of-times-a-given-int-occurs-in-a-linked-list/>

## Chapter 260

# Write a function to delete a Linked List

Write a function to delete a Linked List - GeeksforGeeks

**Algorithm For C/C++:** Iterate through the linked list and delete all the nodes one by one. Main point here is not to access next of the current pointer if current pointer is deleted.

In **Java**, automatic garbage collection happens, so deleting a linked list is easy. We just need to change head to null.

**Implementation:**

C/C++

```
// C program to delete a linked list
#include<stdio.h>
#include<stdlib.h>
#include<assert.h>

/* Link list node */
struct Node
{
    int data;
    struct Node* next;
};

/* Function to delete the entire linked list */
void deleteList(struct Node** head_ref)
{
    /* deref head_ref to get the real head */
    struct Node* current = *head_ref;
    struct Node* next;
```

```
while (current != NULL)
{
    next = current->next;
    free(current);
    current = next;
}

/* deref head_ref to affect the real head back
   in the caller. */
*head_ref = NULL;
}

/* Given a reference (pointer to pointer) to the head
   of a list and an int, push a new node on the front
   of the list. */
void push(struct Node** head_ref, int new_data)
{
    /* allocate node */
    struct Node* new_node =
        (struct Node*) malloc(sizeof(struct Node));

    /* put in the data */
    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;
}

/* Driver program to test count function*/
int main()
{
    /* Start with the empty list */
    struct Node* head = NULL;

    /* Use push() to construct below list
       1->12->1->4->1 */
    push(&head, 1);
    push(&head, 4);
    push(&head, 1);
    push(&head, 12);
    push(&head, 1);

    printf("\n Deleting linked list");
    deleteList(&head);
```

```
    printf("\n Linked list deleted");
}
```

**Java**

```
// Java program to delete a linked list
class LinkedList
{
    Node head; // head of the list

    /* Linked List node */
    class Node
    {
        int data;
        Node next;
        Node(int d) { data = d; next = null; }
    }

    /* Function deletes the entire linked list */
    void deleteList()
    {
        head = null;
    }

    /* Inserts a new Node at front of the list. */
    public void push(int new_data)
    {
        /* 1 & 2: Allocate the Node &
           Put in the data*/
        Node new_node = new Node(new_data);

        /* 3. Make next of new Node as head */
        new_node.next = head;

        /* 4. Move the head to point to new Node */
        head = new_node;
    }

    public static void main(String [] args)
    {
        LinkedList llist = new LinkedList();
        /* Use push() to construct below list
           1->12->1->4->1 */

        llist.push(1);
        llist.push(4);
        llist.push(1);
```

```
llist.push(12);
llist.push(1);

System.out.println("Deleting the list");
llist.deleteList();

System.out.println("Linked list deleted");
}

}

// This code is contributed by Rajat Mishra
```

### Python3

```
# Python3 program to delete all
# the nodes of singly linked list

# Node class
class Node:

    # Function to initialise the node object
    def __init__(self, data):
        self.data = data # Assign data
        self.next = None # Initialize next as null

    # Constructor to initialize the node object
class LinkedList:

    # Function to initialize head
    def __init__(self):
        self.head = None

    def deleteList(self):

        # initialize the current node
        current = self.head
        while current:
            prev = current.next # move next node

            # delete the current node
            del current.data

            # set current equals prev node
            current = prev

    # push function to add node in front of llist
    def push(self, new_data):
```

```
# Allocate the Node &
# Put in the data
new_node = Node(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

# Use push() to construct below
# list 1-> 12-> 1-> 4-> 1
if __name__ == '__main__':

    llist = LinkedList()
    llist.push(1)
    llist.push(4)
    llist.push(1)
    llist.push(12)
    llist.push(1)

    print("Deleting linked list")
    llist.deleteList()

    print("Linked list deleted")

# This article is provided by Shrikant13
```

Output:

```
Deleting linked list
Linked list deleted
```

**Time Complexity:** O(n)  
**Auxiliary Space:** O(1)

## Source

<https://www.geeksforgeeks.org/write-a-function-to-delete-a-linked-list/>

## Chapter 261

# Write a function to get Nth node in a Linked List

Write a function to get Nth node in a Linked List - GeeksforGeeks

Write a GetNth() function that takes a linked list and an integer index and returns the data value stored in the node at that index position.

Example:

```
Input: 1->10->30->14, index = 2  
Output: 30  
The node at index 2 is 30
```

**Algorithm:**

1. Initialize count = 0
2. Loop through the link list
  - a. if count is equal to the passed index then return current node
  - b. Increment count
  - c. change current to point to next of the current.

**Implementation:**

C

```
// C program to find n'th
```

```
// node in linked list
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

// Link list node
struct Node
{
    int data;
    struct Node* next;
};

/* Given a reference (pointer to
   pointer) to the head of a list
   and an int, push a new node on
   the front of the list. */
void push(struct Node** head_ref,
          int new_data)
{

    // allocate node
    struct Node* new_node =
        (struct Node*) malloc(sizeof(struct Node));

    // put in the data
    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;
}

// Takes head pointer of
// the linked list and index
// as arguments and return
// data at index
int GetNth(struct Node* head,
           int index)
{

    struct Node* current = head;

    // the index of the
    // node we're currently
```

```
// looking at
int count = 0;
while (current != NULL)
{
    if (count == index)
        return(current->data);
    count++;
    current = current->next;
}

/* if we get to this line,
   the caller was asking
   for a non-existent element
   so we assert fail */
assert(0);
}

// Driver Code
int main()
{

    // Start with the
    // empty list
    struct Node* head = NULL;

    // Use push() to construct
    // below list
    // 1->12->1->4->1
    push(&head, 1);
    push(&head, 4);
    push(&head, 1);
    push(&head, 12);
    push(&head, 1);

    // Check the count
    // function
    printf("Element at index 3 is %d",
           GetNth(head, 3));
    getchar();
}
```

### Java

```
// Java program to find n'th node in linked list

class Node
{
    int data;
```

```
Node next;
Node(int d)
{
    data = d;
    next = null;
}
}

class LinkedList
{
    Node head; //the head of list

    /* Takes index as argument and return data at index*/
    public int GetNth(int index)
    {
        Node current = head;
        int count = 0; /* index of Node we are
                           currently looking at */
        while (current != null)
        {
            if (count == index)
                return current.data;
            count++;
            current = current.next;
        }

        /* if we get to this line, the caller was asking
           for a non-existent element so we assert fail */
        assert(false);
        return 0;
    }

    /* Given a reference to the head of a list and an int,
       inserts a new Node on the front of the list. */
    public void push(int new_data)
    {

        /* 1. alloc the Node and put data*/
        Node new_Node = new Node(new_data);

        /* 2. Make next of new Node as head */
        new_Node.next = head;

        /* 3. Move the head to point to new Node */
        head = new_Node;
    }

    /* Drier program to test above functions*/
}
```

```
public static void main(String[] args)
{
    /* Start with empty list */
    LinkedList llist = new LinkedList();

    /* Use push() to construct below list
       1->12->1->4->1 */
    llist.push(1);
    llist.push(4);
    llist.push(1);
    llist.push(12);
    llist.push(1);

    /* Check the count function */
    System.out.println("Element at index 3 is "+llist.GetNth(3));
}
}
```

### Python

```
# A complete working Python program to find n'th node
# in a linked list

# Node class
class Node:
    # Function to initialise the node object
    def __init__(self, data):
        self.data = data # Assign data
        self.next = None # Initialize next as null

# Linked List class contains a Node object
class LinkedList:

    # Function to initialize head
    def __init__(self):
        self.head = None

    # This function is in LinkedList class. It inserts
    # a new node at the beginning of Linked List.
    def push(self, new_data):

        # 1 & 2: Allocate the Node &
        #         Put in the data
        new_node = Node(new_data)

        # 3. Make next of new Node as head
```

```
new_node.next = self.head

# 4. Move the head to point to new Node
self.head = new_node

# Returns data at given index in linked list
def getNth(self, index):
    current = self.head # Initialise temp
    count = 0 # Index of current node

    # Loop while end of linked list is not reached
    while (current):
        if (count == index):
            return current.data
        count += 1
        current = current.next

    # if we get to this line, the caller was asking
    # for a non-existent element so we assert fail
    assert(false)
    return 0;

# Code execution starts here
if __name__=='__main__':
    llist = LinkedList()

    # Use push() to construct below list
    # 1->12->1->4->1
    llist.push(1);
    llist.push(4);
    llist.push(1);
    llist.push(12);
    llist.push(1);

    n = 3
    print ("Element at index 3 is :", llist.getNth(n))
```

Output:

```
Element at index 3 is 4
```

**Time Complexity:** O(n)

**Method 2- With Recursion**

This method is contributed by [MY\\_DOOM](#).

**Algorithm:**

```
Algorithm
getnth(node,n)
1. Initialize count = 1
2. if count==n
    return node->data
3. else
    return getnth(node->next,n-1)
```

**Implementation:**

C++

```
// C program to find n'th node in linked list
// using recursion
#include <bits/stdc++.h>
using namespace std;

/* Link list node */
struct Node
{
    int data;
    struct Node* next;
};

/* Given a reference (pointer to pointer) to
the head of a list and an int, push a
new node on the front of the list. */
void push(struct Node** head_ref, int new_data)
{
    /* allocate node */
    struct Node* new_node =
        (struct Node*) malloc(sizeof(struct Node));

    /* put in the data */
    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;
}

/* Takes head pointer of the linked list and index
as arguments and return data at index*/
```

```
int GetNth(struct Node *head,int n)
{
    int count = 1;

    //if count equal too n return node->data
    if(count == n)
        return head->data;

    //recursively decrease n and increase
    // head to next pointer
    return GetNth(head->next, n-1);
}

/* Drier program to test above function*/
int main()
{
    /* Start with the empty list */
    struct Node* head = NULL;

    /* Use push() to construct below list
     * 1->12->1->4->1  */
    push(&head, 1);
    push(&head, 4);
    push(&head, 1);
    push(&head, 12);
    push(&head, 1);

    /* Check the count function */
    printf("Element at index 3 is %d", GetNth(head, 3));
    getchar();
}
```

Output:

```
Element at index 3 is 1
```

**Time Complexity:** O(n)

## Source

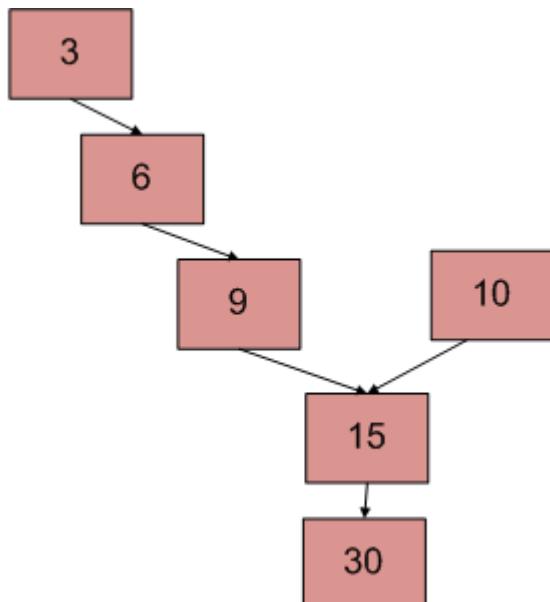
<https://www.geeksforgeeks.org/write-a-function-to-get-nth-node-in-a-linked-list/>

## Chapter 262

**Write a function to get the intersection point of two Linked Lists.**

Write a function to get the intersection point of two Linked Lists. - GeeksforGeeks

There are two singly linked lists in a system. By some programming error, the end node of one of the linked list got linked to the second list, forming an inverted Y shaped list. Write a program to get the point where two linked list merge.



Above diagram shows an example with two linked list having 15 as intersection point.

**Method 1(Simply use two loops)**

Use 2 nested for loops. The outer loop will be for each node of the 1st list and inner loop

will be for 2nd list. In the inner loop, check if any of nodes of the 2nd list is same as the current node of the first linked list. The time complexity of this method will be  $O(mn)$  where m and n are the numbers of nodes in two lists.

### **Method 2 (Mark Visited Nodes)**

This solution requires modifications to basic linked list data structure. Have a visited flag with each node. Traverse the first linked list and keep marking visited nodes. Now traverse the second linked list, If you see a visited node again then there is an intersection point, return the intersecting node. This solution works in  $O(m+n)$  but requires additional information with each node. A variation of this solution that doesn't require modification to the basic data structure can be implemented using a hash. Traverse the first linked list and store the addresses of visited nodes in a hash. Now traverse the second linked list and if you see an address that already exists in the hash then return the intersecting node.

### **Method 3(Using difference of node counts)**

- 1) Get count of the nodes in the first list, let count be c1.
- 2) Get count of the nodes in the second list, let count be c2.
- 3) Get the difference of counts  $d = \text{abs}(c1 - c2)$
- 4) Now traverse the bigger list from the first node till  $d$  nodes so that from here onwards both the lists have equal no of nodes.
- 5) Then we can traverse both the lists in parallel till we come across a common node. (Note that getting a common node is done by comparing the address of the nodes)

C

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

/* Link list node */
struct Node
{
    int data;
    struct Node* next;
};

/* Function to get the counts of node in a linked list */
int getCount(struct Node* head);

/* function to get the intersection point of two linked
   lists head1 and head2 where head1 has d more nodes than
   head2 */
int _getIntersectionNode(int d, struct Node* head1, struct Node* head2);

/* function to get the intersection point of two linked
   lists head1 and head2 */
int getIntersectionNode(struct Node* head1, struct Node* head2)
{
    int c1 = getCount(head1);
    int c2 = getCount(head2);
```

```
int d;

if(c1 > c2)
{
    d = c1 - c2;
    return _getIntersectionNode(d, head1, head2);
}
else
{
    d = c2 - c1;
    return _getIntersectionNode(d, head2, head1);
}

/* function to get the intersection point of two linked
lists head1 and head2 where head1 has d more nodes than
head2 */
int _getIntersectionNode(int d, struct Node* head1, struct Node* head2)
{
    int i;
    struct Node* current1 = head1;
    struct Node* current2 = head2;

    for(i = 0; i < d; i++)
    {
        if(current1 == NULL)
        {   return -1; }
        current1 = current1->next;
    }

    while(current1 != NULL && current2 != NULL)
    {
        if(current1 == current2)
            return current1->data;
        current1= current1->next;
        current2= current2->next;
    }

    return -1;
}

/* Takes head pointer of the linked list and
returns the count of nodes in the list */
int getCount(struct Node* head)
{
    struct Node* current = head;
    int count = 0;
```

```
while (current != NULL)
{
    count++;
    current = current->next;
}

return count;
}

/* IGNORE THE BELOW LINES OF CODE. THESE LINES
   ARE JUST TO QUICKLY TEST THE ABOVE FUNCTION */
int main()
{
/*
Create two linked lists

1st 3->6->9->15->30
2nd 10->15->30

15 is the intersection point
*/
struct Node* newNode;
struct Node* head1 =
    (struct Node*) malloc(sizeof(struct Node));
head1->data = 10;

struct Node* head2 =
    (struct Node*) malloc(sizeof(struct Node));
head2->data = 3;

newNode = (struct Node*) malloc (sizeof(struct Node));
newNode->data = 6;
head2->next = newNode;

newNode = (struct Node*) malloc (sizeof(struct Node));
newNode->data = 9;
head2->next->next = newNode;

newNode = (struct Node*) malloc (sizeof(struct Node));
newNode->data = 15;
head1->next = newNode;
head2->next->next->next = newNode;

newNode = (struct Node*) malloc (sizeof(struct Node));
newNode->data = 30;
head1->next->next= newNode;
```

```
head1->next->next->next = NULL;  
  
printf("\n The node of intersection is %d \n",  
      getIntesectionNode(head1, head2));  
  
getchar();  
}
```

**Java**

```
// Java program to get intersection point of two linked list  
  
class LinkedList {  
  
    static Node head1, head2;  
  
    static class Node {  
  
        int data;  
        Node next;  
  
        Node(int d) {  
            data = d;  
            next = null;  
        }  
    }  
  
    /*function to get the intersection point of two linked  
     lists head1 and head2 */  
    int getNode() {  
        int c1 = getCount(head1);  
        int c2 = getCount(head2);  
        int d;  
  
        if (c1 > c2) {  
            d = c1 - c2;  
            return _getIntesectionNode(d, head1, head2);  
        } else {  
            d = c2 - c1;  
            return _getIntesectionNode(d, head2, head1);  
        }  
    }  
  
    /* function to get the intersection point of two linked  
     lists head1 and head2 where head1 has d more nodes than  
     head2 */  
    int _getIntesectionNode(int d, Node node1, Node node2) {  
        int i;
```

```
Node current1 = node1;
Node current2 = node2;
for (i = 0; i < d; i++) {
    if (current1 == null) {
        return -1;
    }
    current1 = current1.next;
}
while (current1 != null && current2 != null) {
    if (current1.data == current2.data) {
        return current1.data;
    }
    current1 = current1.next;
    current2 = current2.next;
}

return -1;
}

/*Takes head pointer of the linked list and
returns the count of nodes in the list */
int getCount(Node node) {
    Node current = node;
    int count = 0;

    while (current != null) {
        count++;
        current = current.next;
    }

    return count;
}

public static void main(String[] args) {
    LinkedList list = new LinkedList();

    // creating first linked list
    list.head1 = new Node(3);
    list.head1.next = new Node(6);
    list.head1.next.next = new Node(15);
    list.head1.next.next.next = new Node(15);
    list.head1.next.next.next.next = new Node(30);

    // creating second linked list
    list.head2 = new Node(10);
    list.head2.next = new Node(15);
    list.head2.next.next = new Node(30);
```

```
System.out.println("The node of intersection is " + list.getNode());  
}  
}  
  
// This code has been contributed by Mayank Jaiswal
```

**Time Complexity:** O(m+n)

**Auxiliary Space:** O(1)

**Method 4(Make circle in first list)**

Thanks to **Saravanan Man** for providing below solution.

1. Traverse the first linked list(count the elements) and make a circular linked list. (Remember the last node so that we can break the circle later on).
2. Now view the problem as finding the loop in the second linked list. So the problem is solved.
3. Since we already know the length of the loop(size of the first linked list) we can traverse those many numbers of nodes in the second list, and then start another pointer from the beginning of the second list. we have to traverse until they are equal, and that is the required intersection point.
4. remove the circle from the linked list.

**Time Complexity:** O(m+n)

**Auxiliary Space:** O(1)

**Method 5 (Reverse the first list and make equations)**

Thanks to **Saravanan Mani** for providing this method.

- 1) Let X be the length of the first linked list until intersection point.  
Let Y be the length of the second linked list until the intersection point.  
Let Z be the length of the linked list from the intersection point to End of the linked list including the intersection node.  
We Have  
$$X + Z = C_1;$$
$$Y + Z = C_2;$$
- 2) Reverse first linked list.
- 3) Traverse Second linked list. Let C3 be the length of second list - 1.  
Now we have  
$$X + Y = C_3$$
  
We have 3 linear equations. By solving them, we get  
$$X = (C_1 + C_3 - C_2)/2;$$
$$Y = (C_2 + C_3 - C_1)/2;$$
$$Z = (C_1 + C_2 - C_3)/2;$$
  
WE GOT THE INTERSECTION POINT.
- 4) Reverse first linked list.

Advantage: No Comparison of pointers.

Disadvantage : Modifying linked list(Reversing list).

**Time complexity:**  $O(m+n)$

**Auxiliary Space:**  $O(1)$

**Method 6 (Traverse both lists and compare addresses of last nodes)** This method is only to detect if there is an intersection point or not. (Thanks to NeoTheSaviour for suggesting this)

- 1) Traverse the list 1, store the last node address
- 2) Traverse the list 2, store the last node address.
- 3) If nodes stored in 1 and 2 are same then they are intersecting.

The time complexity of this method is  $O(m+n)$  and used Auxiliary space is  $O(1)$

**Method 7 (Use Hashing)**

Basically, we need to find a common node of two linked lists. So we hash all nodes of the first list and then check the second list.

- 1) Create an empty hash table such that node address is used as key and a binary value present/absent is used as the value.
- 2) Traverse the first linked list and insert all nodes' addresses in the hash table.
- 3) Traverse the second list. For every node check if it is present in the hash table. If we find a node in the hash table, return the node.

Please write comments if you find any bug in the above algorithm or a better way to solve the same problem.

## Source

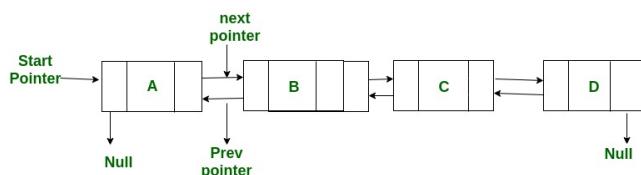
<https://www.geeksforgeeks.org/write-a-function-to-get-the-intersection-point-of-two-linked-lists/>

## Chapter 263

# XOR Linked List – A Memory Efficient Doubly Linked List | Set 1

XOR Linked List - A Memory Efficient Doubly Linked List | Set 1 - GeeksforGeeks

An ordinary Doubly Linked List requires space for two address fields to store the addresses of previous and next nodes. A memory efficient version of Doubly Linked List can be created using only one space for address field with every node. This memory efficient Doubly Linked List is called XOR Linked List or Memory Efficient as the list uses bitwise XOR operation to save space for one address. In the XOR linked list, instead of storing actual memory addresses, every node stores the XOR of addresses of previous and next nodes.



Consider the above Doubly Linked List. Following are the Ordinary and XOR (or Memory Efficient) representations of the Doubly Linked List.

#### Ordinary Representation:

Node A:

```
prev = NULL, next = add(B) // previous is NULL and next is address of B
```

Node B:

```
prev = add(A), next = add(C) // previous is address of A and next is address of C
```

Node C:

```
prev = add(B), next = add(D) // previous is address of B and next is address of D
```

Node D:

prev = add(C), next = NULL // previous is address of C and next is NULL

**XOR List Representation:**

Let us call the address variable in XOR representation npx (XOR of next and previous)

Node A:

npx = 0 XOR add(B) // bitwise XOR of zero and address of B

Node B:

npx = add(A) XOR add(C) // bitwise XOR of address of A and address of C

Node C:

npx = add(B) XOR add(D) // bitwise XOR of address of B and address of D

Node D:

npx = add(C) XOR 0 // bitwise XOR of address of C and 0

**Traversal of XOR Linked List:**

We can traverse the XOR list in both forward and reverse direction. While traversing the list we need to remember the address of the previously accessed node in order to calculate the next node's address. For example when we are at node C, we must have address of B. XOR of add(B) and *npx* of C gives us the add(D). The reason is simple: *npx*(C) is “add(B) XOR add(D)”. If we do xor of *npx*(C) with add(B), we get the result as “add(B) XOR add(D) XOR add(B)” which is “add(D) XOR 0” which is “add(D)”. So we have the address of next node. Similarly we can traverse the list in backward direction.

We have covered more on XOR Linked List in the following post.

[XOR Linked List – A Memory Efficient Doubly Linked List | Set 2](#)

**References:**

[http://en.wikipedia.org/wiki/XOR\\_linked\\_list](http://en.wikipedia.org/wiki/XOR_linked_list)

<http://www.linuxjournal.com/article/6828?page=0,0>

**Source**

<https://www.geeksforgeeks.org/xor-linked-list-a-memory-efficient-doubly-linked-list-set-1/>

## Chapter 264

# XOR Linked List – A Memory Efficient Doubly Linked List | Set 2

XOR Linked List – A Memory Efficient Doubly Linked List | Set 2 - GeeksforGeeks

In the [previous post](#), we discussed how a Doubly Linked can be created using only one space for address field with every node. In this post, we will discuss implementation of memory efficient doubly linked list. We will mainly discuss following two simple functions.

- 1) A function to insert a new node at the beginning.
- 2) A function to traverse the list in forward direction.

In the following code, *insert()* function inserts a new node at the beginning. We need to change the head pointer of Linked List, that is why a double pointer is used (See [this](#)). Let us first discuss few things again that have been discussed in the [previous post](#). We store XOR of next and previous nodes with every node and we call it npx, which is the only address member we have with every node. When we insert a new node at the beginning, npx of new node will always be XOR of NULL and current head. And npx of current head must be changed to XOR of new node and node next to current head.

*printList()* traverses the list in forward direction. It prints data values from every node. To traverse the list, we need to get pointer to the next node at every point. We can get the address of next node by keeping track of current node and previous node. If we do XOR of curr->npx and prev, we get the address of next node.

```
/* C/C++ Implementation of Memory
   efficient Doubly Linked List */
#include <stdio.h>
#include <stdlib.h>
#include <inttypes.h>

// Node structure of a memory
```

```
// efficient doubly linked list
struct Node
{
    int data;
    struct Node* npx; /* XOR of next and previous node */
};

/* returns XORed value of the node addresses */
struct Node* XOR (struct Node *a, struct Node *b)
{
    return (struct Node*) ((uintptr_t) (a) ^ (uintptr_t) (b));
}

/* Insert a node at the begining of the
   XORed linked list and makes the newly
   inserted node as head */
void insert(struct Node **head_ref, int data)
{
    // Allocate memory for new node
    struct Node *new_node = (struct Node *) malloc (sizeof (struct Node) );
    new_node->data = data;

    /* Since new node is being inserted at the
       begining, npx of new node will always be
       XOR of current head and NULL */
    new_node->npx = XOR(*head_ref, NULL);

    /* If linked list is not empty, then npx of
       current head node will be XOR of new node
       and node next to current head */
    if (*head_ref != NULL)
    {
        // *(head_ref)->npx is XOR of NULL and next.
        // So if we do XOR of it with NULL, we get next
        struct Node* next = XOR((*head_ref)->npx, NULL);
        (*head_ref)->npx = XOR(new_node, next);
    }

    // Change head
    *head_ref = new_node;
}

// prints contents of doubly linked
// list in forward direction
void printList (struct Node *head)
{
    struct Node *curr = head;
    struct Node *prev = NULL;
```

```
struct Node *next;

printf ("Following are the nodes of Linked List: \n");

while (curr != NULL)
{
    // print current node
    printf ("%d ", curr->data);

    // get address of next node: curr->npx is
    // next^prev, so curr->npx^prev will be
    // next^prev^prev which is next
    next = XOR (prev, curr->npx);

    // update prev and curr for next iteration
    prev = curr;
    curr = next;
}
}

// Driver program to test above functions
int main ()
{
    /* Create following Doubly Linked List
    head-->40<-->30<-->20<-->10 */
    struct Node *head = NULL;
    insert(&head, 10);
    insert(&head, 20);
    insert(&head, 30);
    insert(&head, 40);

    // print the created list
    printList (head);

    return (0);
}
```

Output:

```
Following are the nodes of Linked List:
40 30 20 10
```

Note that XOR of pointers is not defined by C/C++ standard. So the above implementation may not work on all platforms.

Improved By : [piyush02](#)

## Source

<https://www.geeksforgeeks.org/xor-linked-list-a-memory-efficient-doubly-linked-list-set-2/>