

Contents

1 AA Trees Set 1 (Introduction)	15
Source	16
2 AVL Tree Set 1 (Insertion)	17
Source	34
3 AVL Tree Set 2 (Deletion)	35
Source	56
4 Advantages of Trie Data Structure	57
Source	58
5 Agents in Artificial Intelligence	59
Source	65
6 Array range queries for elements with frequency same as value	66
Source	73
7 Auto-complete feature using Trie	74
Source	79
8 B-Tree Set 1 (Introduction)	80
Source	83
9 B-Tree Set 2 (Insert)	84
Source	92
10 B-Tree Set 3 (Delete)	93
Source	111
11 BK-Tree Introduction & Implementation	112
Source	120
12 Barabasi Albert Graph (for Scale Free Models)	121
Source	125
13 Binary Indexed Tree : Range Update and Range Queries	126
Source	130

14 Binary Indexed Tree : Range Updates and Point Queries	131
Source	138
15 Binary Indexed Tree or Fenwick Tree	139
Source	148
16 Binomial Heap	149
Source	154
17 Boggle Set 2 (Using Trie)	155
Source	164
18 Burrows – Wheeler Data Transform Algorithm	165
Source	169
19 C++ Program to implement Symbol Table	170
Source	177
20 Cartesian Tree	178
Source	183
21 Cartesian Tree Sorting	184
Source	191
22 Centroid Decomposition of Tree	192
Source	200
23 Check if the given string of words can be formed from words present in the dictionary	201
Source	206
24 Coalesced hashing	207
Source	210
25 Comparisons involved in Modified Quicksort Using Merge Sort Tree	211
Source	216
26 Convert an Array to a Circular Doubly Linked List	217
Source	220
27 Count Inversions of size three in a given array	221
Source	231
28 Count and Toggle Queries on a Binary Array	232
Source	236
29 Count elements which divide all numbers in range L-R	237
Source	243
30 Count greater nodes in AVL tree	244
Source	252

31 Count inversion pairs in a matrix	253
Source	257
32 Count inversions in an array Set 3 (Using BIT)	258
Source	263
33 Count inversions of size k in a given array	264
Source	270
34 Count number of smallest elements in given range	271
Source	275
35 Count of distinct substrings of a string using Suffix Array	276
Source	281
36 Count of distinct substrings of a string using Suffix Trie	282
Source	288
37 Counting Triangles in a Rectangular space using BIT	289
Source	293
38 Counting k-mers via Suffix Array	294
Source	298
39 Counting the number of words in a Trie	299
Source	303
40 Data Structure for Dictionary and Spell Checker?	304
Source	305
41 Decision Tree Introduction with example	306
Source	312
42 Decision Trees – Fake (Counterfeit) Coin Puzzle (12 Coin Puzzle)	313
Source	318
43 Design a data structure that supports insert, delete, search and getRandom in constant time	319
Source	324
44 Design an efficient data structure for given operations	325
Source	337
45 Difference Array Range update query in O(1)	338
Source	345
46 Difference between Inverted Index and Forward Index	346
Source	347
47 Disjoint Set Data Structures (Java Implementation)	348
Source	355

48 Dope Vector	356
Source	357
49 Dynamic Connectivity Set 1 (Incremental)	358
Source	362
50 Dynamic Disjoint Set Data Structure for large range values	363
Source	368
51 Dynamic Programming on Trees Set 2	369
Source	376
52 Dynamic Programming on Trees Set-1	377
Source	381
53 Efficiently design Insert, Delete and Median queries on a set	382
Source	388
54 Euler Tour Subtree Sum using Segment Tree	389
Source	396
55 Euler tour of Binary Tree	397
Source	400
56 Extended Mo's Algorithm with O(1) time complexity	401
Source	412
57 Fibonacci Heap Set 1 (Introduction)	413
Source	414
58 Find LCA in Binary Tree using RMQ	415
Source	427
59 Find all possible interpretations of an array of digits	428
Source	432
60 Find last unique URL from long list of URLs in single traversal	433
Source	437
61 Find maximum XOR of given integer in a stream of integers	438
Source	442
62 Find pair of rows in a binary matrix that has maximum bit difference	443
Source	447
63 Find shortest unique prefix for every word in a given list Set 1 (Using Trie)	448
Source	454
64 Find the k most frequent words from a file	455
Source	461

65 Find the maximum subarray XOR in a given array	462
Source	472
66 Find the number of Islands Set 2 (Using Disjoint Set)	473
Source	477
67 Find whether a subarray is in form of a mountain or not	478
Source	484
68 Flatten a binary tree into linked list Set-2	485
Source	488
69 GCDs of given index ranges in an array	489
Source	495
70 Generalized Suffix Tree 1	496
Source	507
71 Generic Linked List in C	508
Source	510
72 Given a sequence of words, print all anagrams together Set 2	511
Source	517
73 Gomory-Hu Tree Set 1 (Introduction)	518
Source	520
74 Hashtables Chaining with Doubly Linked Lists	521
Source	525
75 Heavy Light Decomposition Set 1 (Introduction)	526
Source	532
76 Heavy Light Decomposition Set 2 (Implementation)	533
Source	544
77 How to Implement Forward DNS Look Up Cache?	545
Source	548
78 How to Implement Reverse DNS Look Up Cache?	549
Source	552
79 How to design a tiny URL or URL shortener?	553
Source	555
80 Image Manipulation Using Quadtrees	556
Source	560
81 Implement a Phone Directory	561
Source	570

82 Implementation of Binomial Heap Set – 2 (delete() and decreaseKey())	571
Source	578
83 Inclusion Exclusion principle and programming applications	579
Source	586
84 Insertion at Specific Position in a Circular Doubly Linked List	587
Source	592
85 Insertion in Unrolled Linked List	593
Source	599
86 Interval Tree	600
Source	605
87 Iterative Preorder Traversal of an N-ary Tree	606
Source	609
88 Iterative Segment Tree (Range Maximum Query with Node Update)	610
Source	613
89 Iterative Segment Tree (Range Minimum Query)	614
Source	617
90 K Dimensional Tree Set 1 (Search and Insert)	618
Source	629
91 K Dimensional Tree Set 2 (Find Minimum)	630
Source	634
92 K Dimensional Tree Set 3 (Delete)	635
Source	642
93 LCA for general or n-ary trees (Sparse Matrix DP approach < O(nlogn), O(logn)>)	643
Source	649
94 LCA for n-ary Tree Constant Query O(1)	650
Source	656
95 LRU Cache Implementation	657
Source	667
96 Largest Rectangular Area in a Histogram Set 1	668
Source	672
97 Lazy Propagation in Segment Tree	673
Source	686
98 Left-Child Right-Sibling Representation of Tree	687
Source	693

99 Leftist Tree / Leftist Heap	694
Source	705
100 Leftover element after performing alternate Bitwise OR and Bitwise XOR operations on adjacent pairs	706
Source	719
101 Level Ancestor Problem	720
Source	727
102 Levelwise Alternating GCD and LCM of nodes in Segment Tree	728
Source	736
103 Levelwise Alternating OR and XOR operations in Segment Tree	737
Source	746
104 Longest Common Extension / LCE Set 2 (Reduction to RMQ)	747
Source	754
105 Longest Common Prefix using Trie	755
Source	762
106 Longest prefix matching – A Trie based solution in Java	763
Source	766
107 Longest word in ternary search tree	767
Source	772
108 Maximum Occurrence in a Given Range	773
Source	779
109 Maximum Subarray Sum in a given Range	780
Source	787
110 Maximum Sum Increasing Subsequence using Binary Indexed Tree	788
Source	791
111 Merge Sort Tree (Smaller or equal elements in given row range)	792
Source	795
112 Merge Sort Tree for Range Order Statistics	796
Source	800
113 Min-Max Range Queries in Array	801
Source	805
114 Minimum Word Break	806
Source	811
115 Number of elements greater than K in the range L to R using Fenwick Tree (Offline queries)	812

Source	817
116 Number of elements less than or equal to a given number in a given subarray	818
Source	821
117 Number of elements less than or equal to a given number in a given subarray Set 2 (Including Updates)	822
Source	829
118 Number of primes in a subarray (with updates)	830
Source	836
119 Order statistic tree using fenwick tree (BIT)	837
Source	840
120 Ordered Set and GNU C++ PBDS	841
Source	844
121 Overview of Data Structures Set 3 (Graph, Trie, Segment Tree and Suffix Tree)	845
Source	849
122 Palindrome pair in an array of words (or strings)	850
Source	860
123 Palindromic Tree Introduction & Implementation	861
Source	874
124 Pattern Searching using Suffix Tree	875
Source	879
125 Pattern Searching using a Trie of all Suffixes	880
Source	888
126 Persistent Segment Tree Set 1 (Introduction)	889
Source	895
127 Persistent data structures	896
Source	899
128 Print Kth character in sorted concatenated substrings of a string	900
Source	905
129 Print all valid words that are possible using Characters of Array	906
Source	912
130 Print all words matching a pattern in CamelCase Notation Dictionary	913
Source	920
131 Print the DFS traversal step-wise (Backtracking also)	921

Source	924
132 Print unique rows in a given boolean matrix	925
Source	929
133 Program for assigning usernames using Trie	930
Source	934
134 Quad Tree	935
Source	941
135 Queries for number of distinct elements in a subarray	942
Source	945
136 Queries on substring palindrome formation	946
Source	951
137 Queries to find distance between two nodes of a Binary tree	952
Source	954
138 Queries to find distance between two nodes of a Binary tree – O(logn) method	955
Source	963
139 Queries to find maximum product pair in range with updates	964
Source	968
140 Querying the number of distinct colors in a subtree of a colored tree using BIT	969
Source	975
141 Range LCM Queries	976
Source	979
142 Range Minimum Query (Square Root Decomposition and Sparse Table)	980
Source	988
143 Range Queries for Longest Correct Bracket Subsequence	989
Source	994
144 Range and Update Query for Chessboard Pieces	995
Source	1003
145 Range query for Largest Sum Contiguous Subarray	1004
Source	1010
146 Range sum query using Sparse Table	1011
Source	1016
147 Reconstructing Segment Tree	1017
Source	1019

148 Red-Black Tree Set 1 (Introduction)	1020
Source1022
149 Red-Black Tree Set 2 (Insert)	1023
Source1026
150 Red-Black Tree Set 3 (Delete)	1027
Source1031
151 Remove edges connected to a node such that the three given nodes are in different trees	1032
Source1036
152 Ropes Data Structure (Fast String Concatenation)	1037
Source1042
153 ScapeGoat Tree Set 1 (Introduction and Insertion)	1043
Source1054
154 Search an Element in Doubly Circular Linked List	1055
Source1059
155 Second minimum element using minimum comparisons	1060
Source1064
156 Segment Tree (XOR of a given range)	1065
Source1069
157 Segment Tree Set 1 (Sum of given range)	1070
Source1079
158 Segment Tree Set 2 (Range Maximum Query with Node Update)	1080
Source1085
159 Segment Tree Set 2 (Range Minimum Query)	1086
Source1093
160 Segment Tree Set 3 (XOR of given range)	1094
Source1098
161 Segment Trees (Product of given Range Modulo m)	1099
Source1104
162 Segment tree Efficient implementation	1105
Source1112
163 Self Organizing List : Count Method	1113
Source1118
164 Self Organizing List : Move to Front Method	1119
Source1123

165 Self Organizing List Set 1 (Introduction)	1124
Source1125
166 Self-Balancing-Binary-Search-Trees (Comparisons)	1126
Source1128
167 Skew Heap	1129
Source1134
168 Skip List Set 1 (Introduction)	1135
Source1136
169 Skip List Set 2 (Insertion)	1137
Source1146
170 Skip List Set 3 (Searching and Deletion)	1147
Source1160
171 Smallest Subarray with given GCD	1161
Source1167
172 Sort numbers stored on different machines	1168
Source1172
173 Sorting array of strings (or words) using Trie Set-2 (Handling Duplicates)	1173
Source1179
174 Spaghetti Stack	1180
Source1181
175 Sparse Set	1182
Source1190
176 Sparse Table	1191
Source1200
177 Splay Tree Set 1 (Search)	1201
Source1207
178 Splay Tree Set 2 (Insert)	1208
Source1212
179 Splay Tree Set 3 (Delete)	1213
Source1218
180 Sqrt (or Square Root) Decomposition Set 2 (LCA of Tree in O(sqrt(height)) time)	1219
Source1229
181 Substring with highest frequency length product	1230
Source1236

182 Suffix Array Set 1 (Introduction)	1237
Source1241
183 Suffix Array Set 2 (nLogn Algorithm)	1242
Source1247
184 Suffix Tree Application 1 – Substring Check	1248
Source1258
185 Suffix Tree Application 2 – Searching All Patterns	1259
Source1273
186 Suffix Tree Application 3 – Longest Repeated Substring	1274
Source1285
187 Suffix Tree Application 4 – Build Linear Time Suffix Array	1286
Source1297
188 Suffix Tree Application 5 – Longest Common Substring	1298
Source1310
189 Suffix Tree Application 6 – Longest Palindromic Substring	1311
Source1325
190 Sum of K largest elements in BST using O(1) Extra space	1326
Source1330
191 Sum over Subsets Dynamic Programming	1331
Source1343
192 Summed Area Table – Submatrix Summation	1344
Source1345
193 Tango Tree Data Structure	1346
Source1348
194 Tarjan's off-line lowest common ancestors algorithm	1349
Source1359
195 Ternary Search Tree	1360
Source1365
196 Ternary Search Tree (Deletion)	1366
Source1375
197 Total nodes traversed in Euler Tour Tree	1376
Source1380
198 Tournament Tree (Winner Tree) and Binary Heap	1381
Source1384

199 Treap (A Randomized Binary Search Tree)	1385
Source1388
200 Treap Set 2 (Implementation of Search, Insert and Delete)	1389
Source1398
201 Trie memory optimization using hash map	1399
Source1401
202 Trie (Delete)	1402
Source1410
203 Trie (Display Content)	1411
Source1414
204 Trie (Insert and Search)	1415
Source1426
205 Two Dimensional Binary Indexed Tree or Fenwick Tree	1427
Source1433
206 Two Dimensional Segment Tree Sub-Matrix Sum	1434
Source1441
207 Ukkonen's Suffix Tree Construction – Part 1	1442
Source1454
208 Ukkonen's Suffix Tree Construction – Part 2	1455
Source1461
209 Ukkonen's Suffix Tree Construction – Part 3	1462
Source1471
210 Ukkonen's Suffix Tree Construction – Part 4	1472
Source1480
211 Ukkonen's Suffix Tree Construction – Part 5	1481
Source1493
212 Ukkonen's Suffix Tree Construction – Part 6	1494
Source1504
213 Union-Find Algorithm (Union By Rank and Find by Optimized Path Compression)	1505
Source1510
214 Unrolled Linked List Set 1 (Introduction)	1511
Source1514
215 Wavelet Trees Introduction	1515
Source1522

216 Weighted Prefix Search	1523
Source1531
217 Word formation using concatenation of two dictionary words	1532
Source1538
218 XOR Linked List – A Memory Efficient Doubly Linked List Set 1	1539
Source1540
219 XOR Linked List – A Memory Efficient Doubly Linked List Set 2	1541
Source1544
220 XOR of numbers that appeared even number of times in given Range	1545
Source1550
221 proto van Emde Boas Trees Set 1 (Background and Introduction)	1551
Source1554
222 sklearn.Binarizer() in Python	1555
Source1557
223 kasai's Algorithm for Construction of LCP array from Suffix Array	1558
Source1564

Contents

Chapter 1

AA Trees Set 1 (Introduction)

AA Trees Set 1 (Introduction) - GeeksforGeeks

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

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

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

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

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

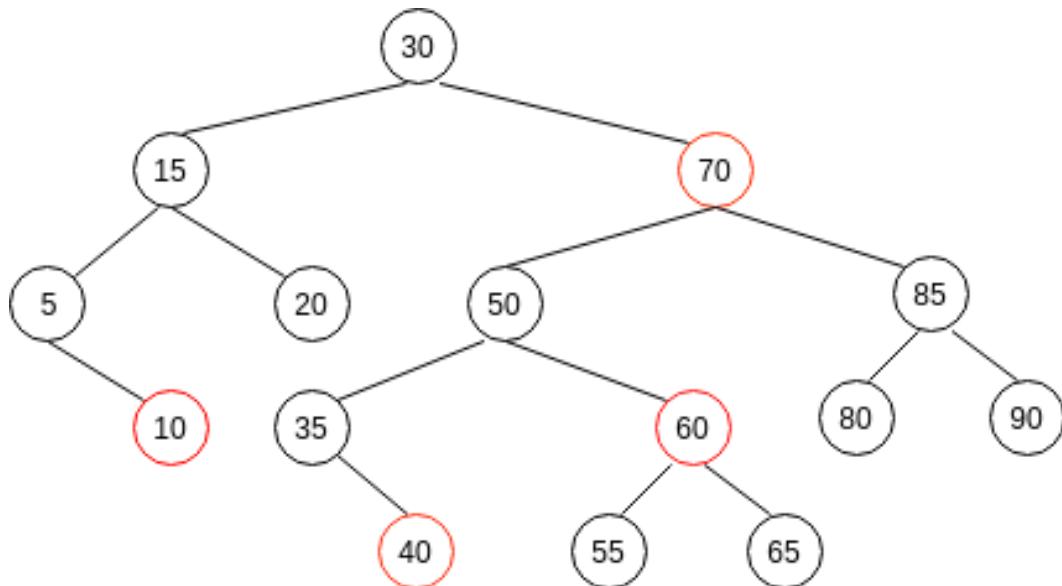
1. Every node can be either red (linked horizontally) or black.
2. There are no two adjacent red nodes (or horizontal links).
3. Every path from root to a NULL node has same number of black nodes (ot black links).
4. **Left link cannot NOT be red (horizontal).** (*New added rule*)

Why AA trees :

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

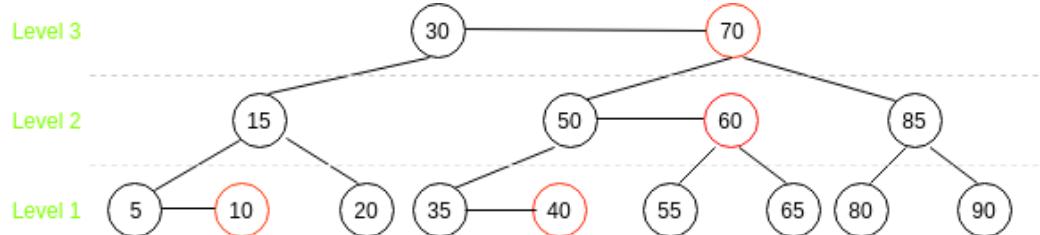
- It eliminates half of the restructuring process by eliminating half of the rotation cases, which is easier to code.
- It simplifies the deletion process by removing multiple cases.

Below tree is the example of AA tree :



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

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



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

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

Source

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

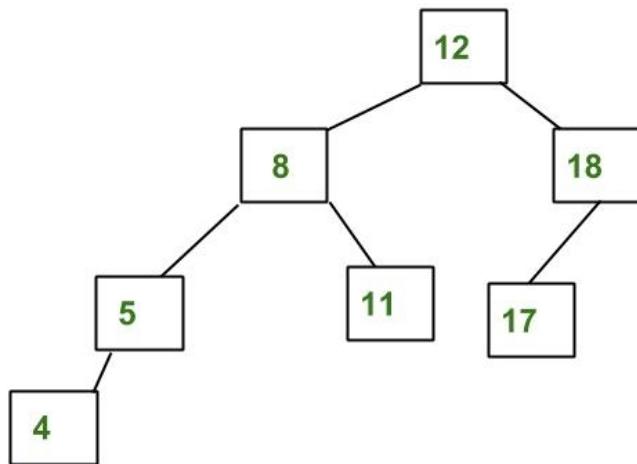
Chapter 2

AVL Tree Set 1 (Insertion)

AVL Tree Set 1 (Insertion) - GeeksforGeeks

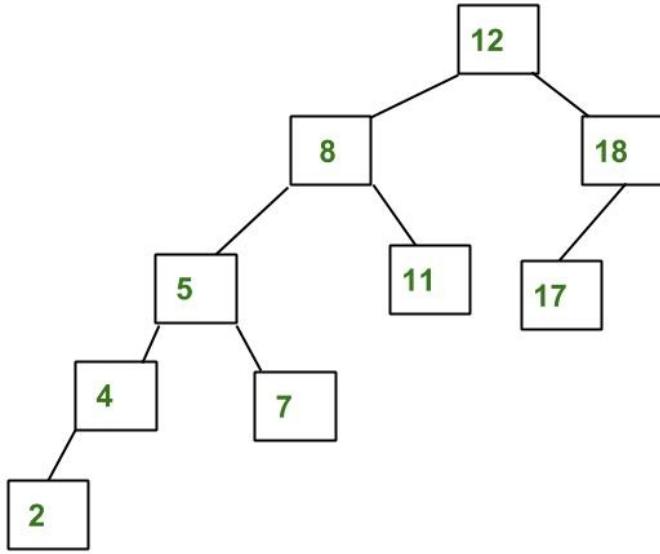
AVL tree is a self-balancing Binary Search Tree (BST) where the difference between heights of left and right subtrees cannot be more than one for all nodes.

An Example Tree that is an AVL Tree



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

An Example Tree that is NOT an AVL Tree



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

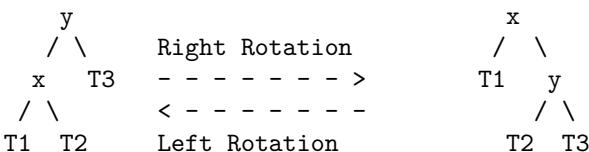
Why AVL Trees?

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

Insertion

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

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



Keys in both of the above trees follow the following order

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

Steps to follow for insertion

Let the newly inserted node be w

1) Perform standard BST insert for w.

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

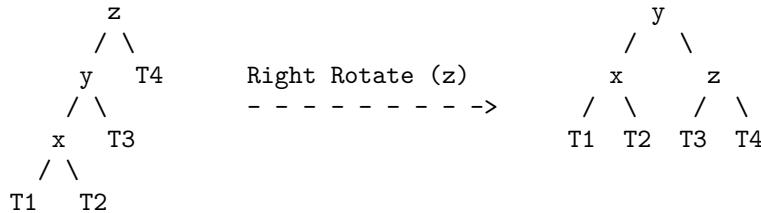
3) Re-balance the tree by performing appropriate rotations on the subtree rooted with z. There can be 4 possible cases that needs to be handled as x, y and z can be arranged in 4 ways. Following are the possible 4 arrangements:

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

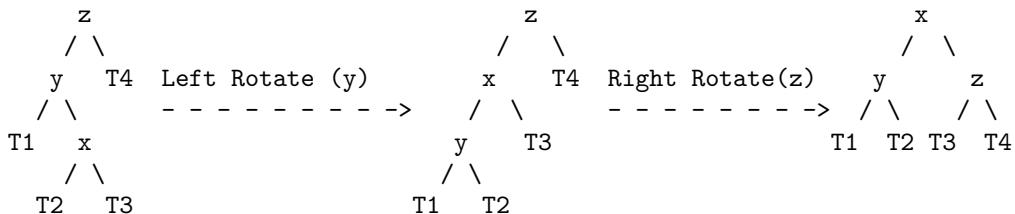
Following are the operations to be performed in above mentioned 4 cases. In all of the cases, we only need to re-balance the subtree rooted with z and the complete tree becomes balanced as the height of subtree (After appropriate rotations) rooted with z becomes same as it was before insertion. (See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

a) Left Left Case

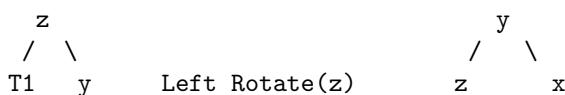
T1, T2, T3 and T4 are subtrees.

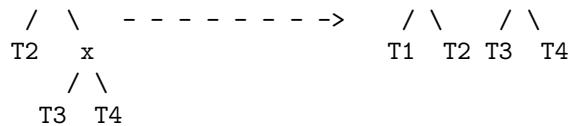


b) Left Right Case

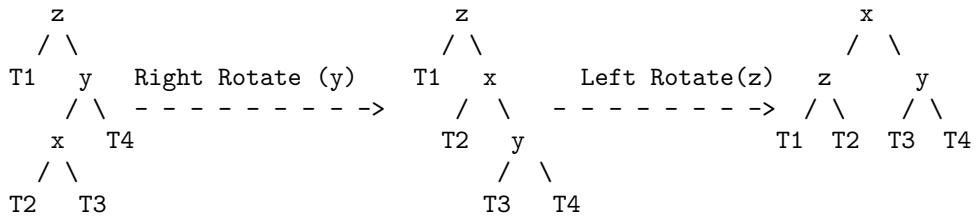


c) Right Right Case

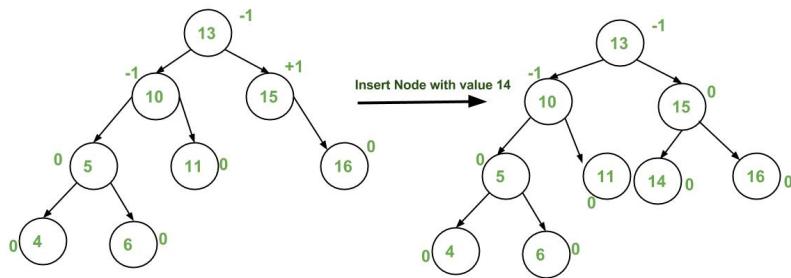


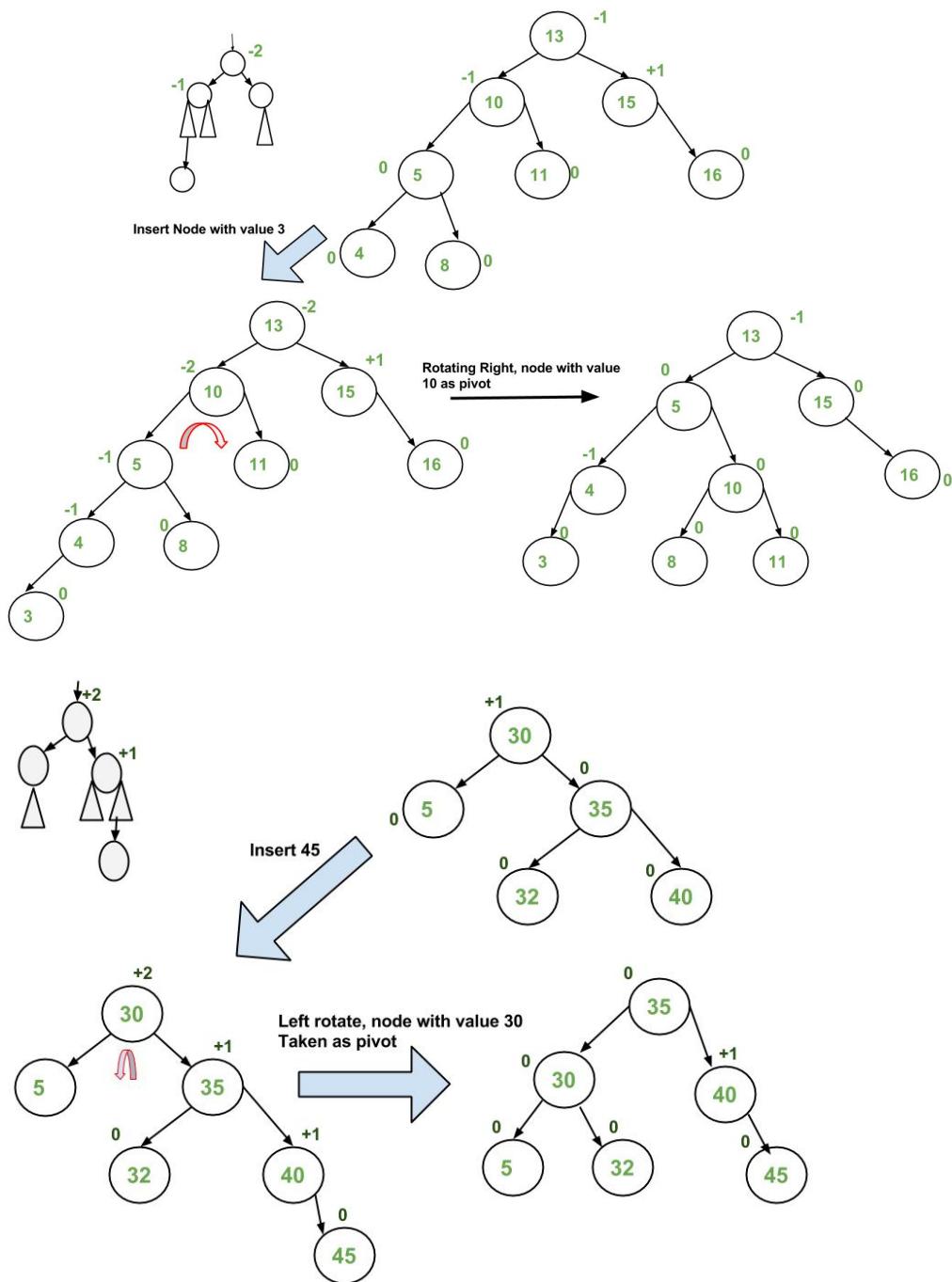


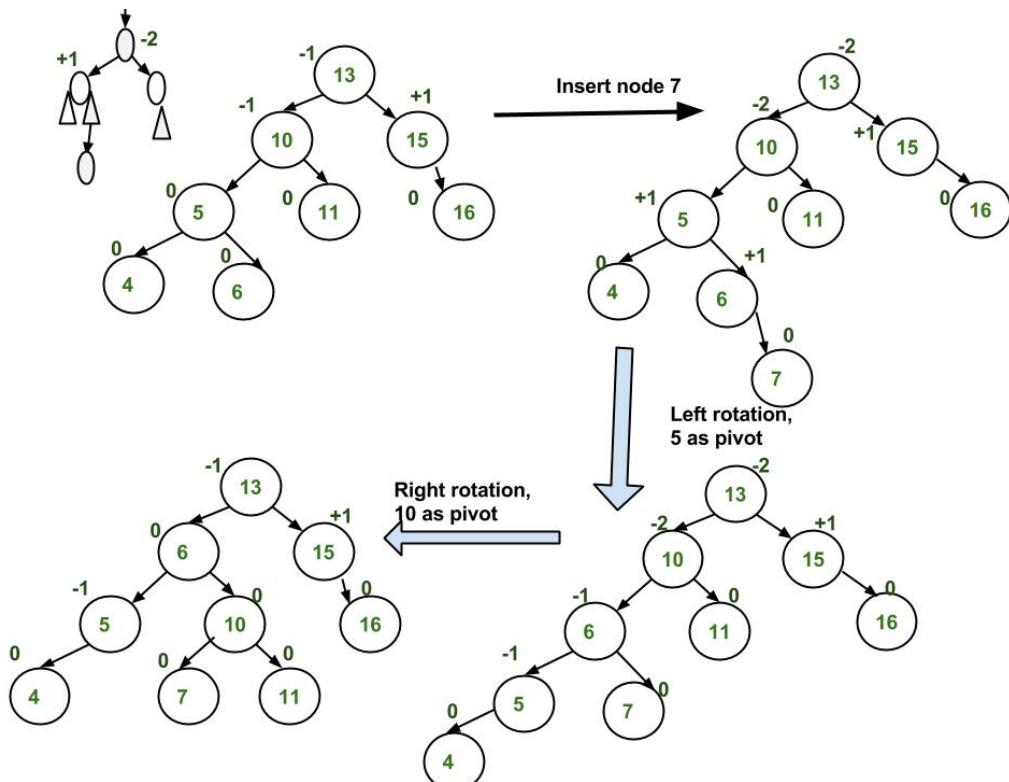
d) Right Left Case

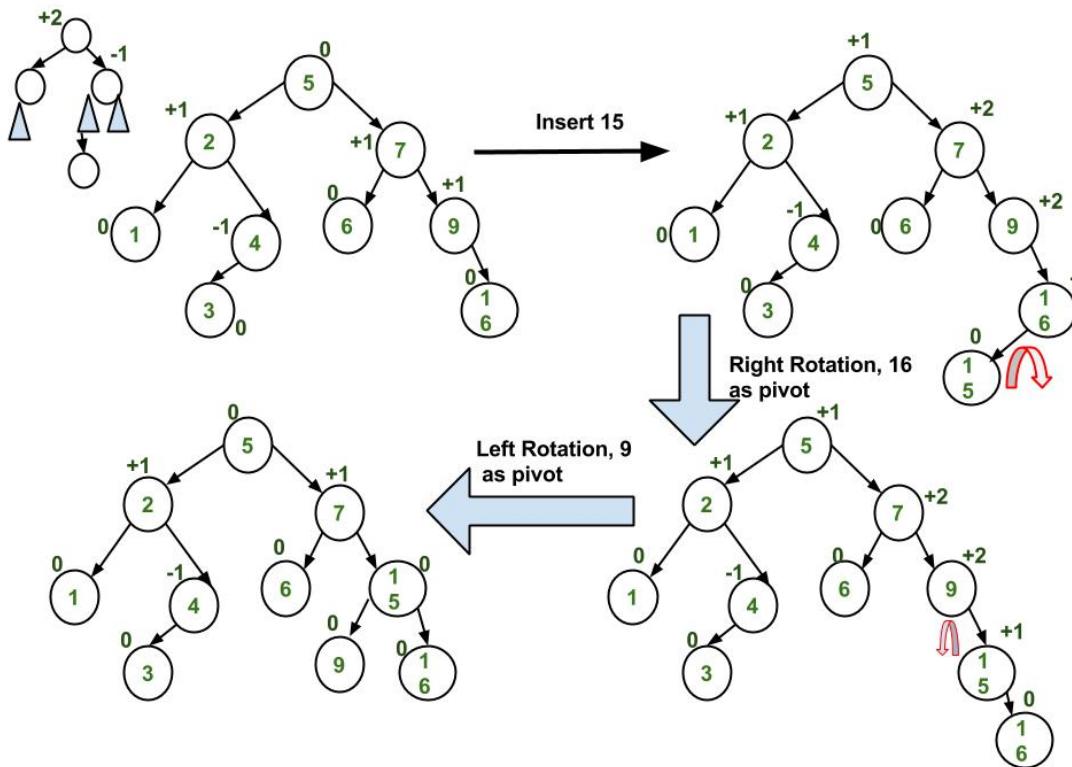


Insertion Examples:









implementation

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

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

C

```

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

```

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

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

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

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

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

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

    // Perform rotation

```

```

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

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

// Return new root
return x;
}

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

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

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

    // Return new root
    return y;
}

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

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

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

```

```

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

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

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

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

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

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

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

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

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

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

```

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

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

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

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

    return 0;
}
```

Java

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

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

```
class AVLTree {  
  
    Node root;  
  
    // A utility function to get the height of the tree  
    int height(Node N) {  
        if (N == null)  
            return 0;  
  
        return N.height;  
    }  
  
    // A utility function to get maximum of two integers  
    int max(int a, int b) {  
        return (a > b) ? a : b;  
    }  
  
    // A utility function to right rotate subtree rooted with y  
    // See the diagram given above.  
    Node rightRotate(Node y) {  
        Node x = y.left;  
        Node T2 = x.right;  
  
        // Perform rotation  
        x.right = y;  
        y.left = T2;  
  
        // Update heights  
        y.height = max(height(y.left), height(y.right)) + 1;  
        x.height = max(height(x.left), height(x.right)) + 1;  
  
        // Return new root  
        return x;  
    }  
  
    // A utility function to left rotate subtree rooted with x  
    // See the diagram given above.  
    Node leftRotate(Node x) {  
        Node y = x.right;  
        Node T2 = y.left;  
  
        // Perform rotation  
        y.left = x;  
        x.right = T2;  
  
        // Update heights  
        x.height = max(height(x.left), height(x.right)) + 1;
```

```

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

// Return new root
return y;
}

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

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

Node insert(Node node, int key) {

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

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

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

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

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

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

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

```

```

        return rightRotate(node);
    }

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

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

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

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

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

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

// This code has been contributed by Mayank Jaiswal

```

Python3

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

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

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

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

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

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

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

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

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

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

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

        return root
```

```

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

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

return root

def leftRotate(self, z):

    y = z.right
    T2 = y.left

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

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

    # Return the new root
    return y

def rightRotate(self, z):

    y = z.left
    T3 = y.right

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

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

    # Return the new root
    return y

def getHeight(self, root):

```

```

if not root:
    return 0

return root.height

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

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

def preOrder(self, root):

    if not root:
        return

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

# Driver program to test above function
myTree = AVL_Tree()
root = None

root = myTree.insert(root, 10)
root = myTree.insert(root, 20)
root = myTree.insert(root, 30)
root = myTree.insert(root, 40)
root = myTree.insert(root, 50)
root = myTree.insert(root, 25)

"""The constructed AVL Tree would be
      30
      / \
     20   40
    /   \   \
   10   25   50"""

# Preorder Traversal
print("Preorder traversal of the",
      "constructed AVL tree is")
myTree.preOrder(root)
print()

# This code is contributed by Ajitesh Pathak

```

Output:

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

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

Comparison with Red Black Tree

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

Following is the post for delete.

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

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

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

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

[Count smaller elements on right side](#)

References:

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

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

Source

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

Chapter 3

AVL Tree Set 2 (Deletion)

AVL Tree Set 2 (Deletion) - GeeksforGeeks

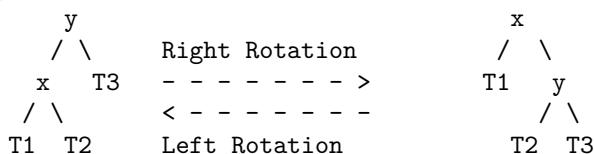
We have discussed AVL insertion in the [previous post](#). In this post, we will follow a similar approach for deletion.

Steps to follow for deletion.

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

- 1) Left Rotation
- 2) Right Rotation

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



Keys in both of the above trees follow the following order

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

So BST property is not violated anywhere.

Let w be the node to be deleted

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

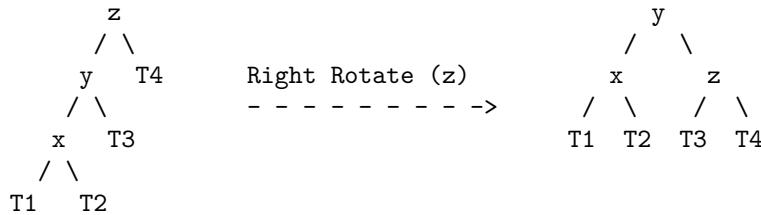
ways. Following are the possible 4 arrangements:

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

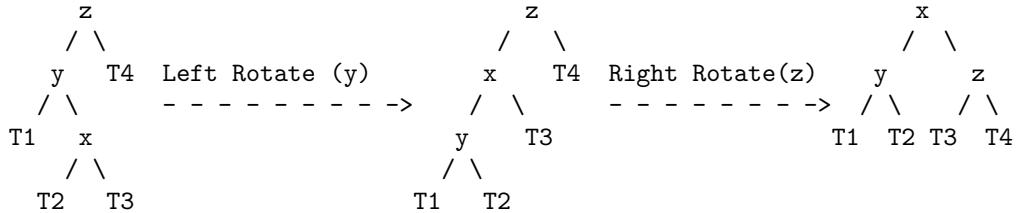
Like insertion, following are the operations to be performed in above mentioned 4 cases. Note that, unlike insertion, fixing the node z won't fix the complete AVL tree. After fixing z, we may have to fix ancestors of z as well (See [this video lecture](#) for proof)

a) Left Left Case

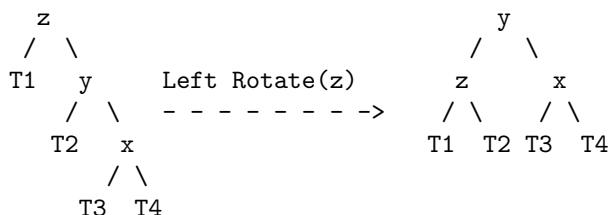
T1, T2, T3 and T4 are subtrees.



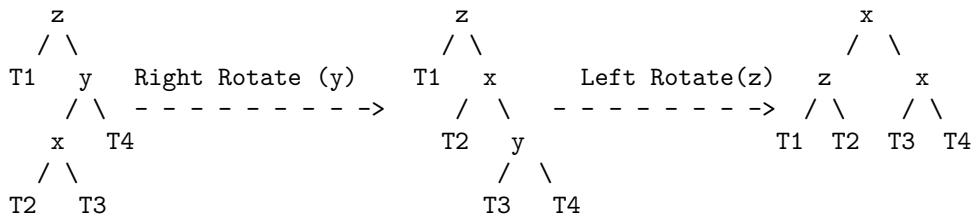
b) Left Right Case



c) Right Right Case



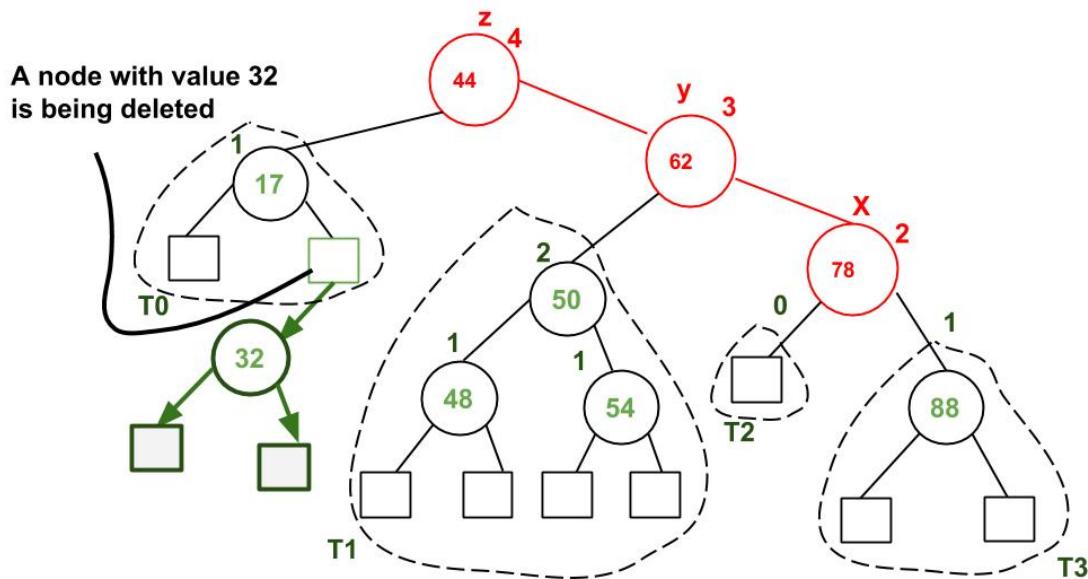
d) Right Left Case

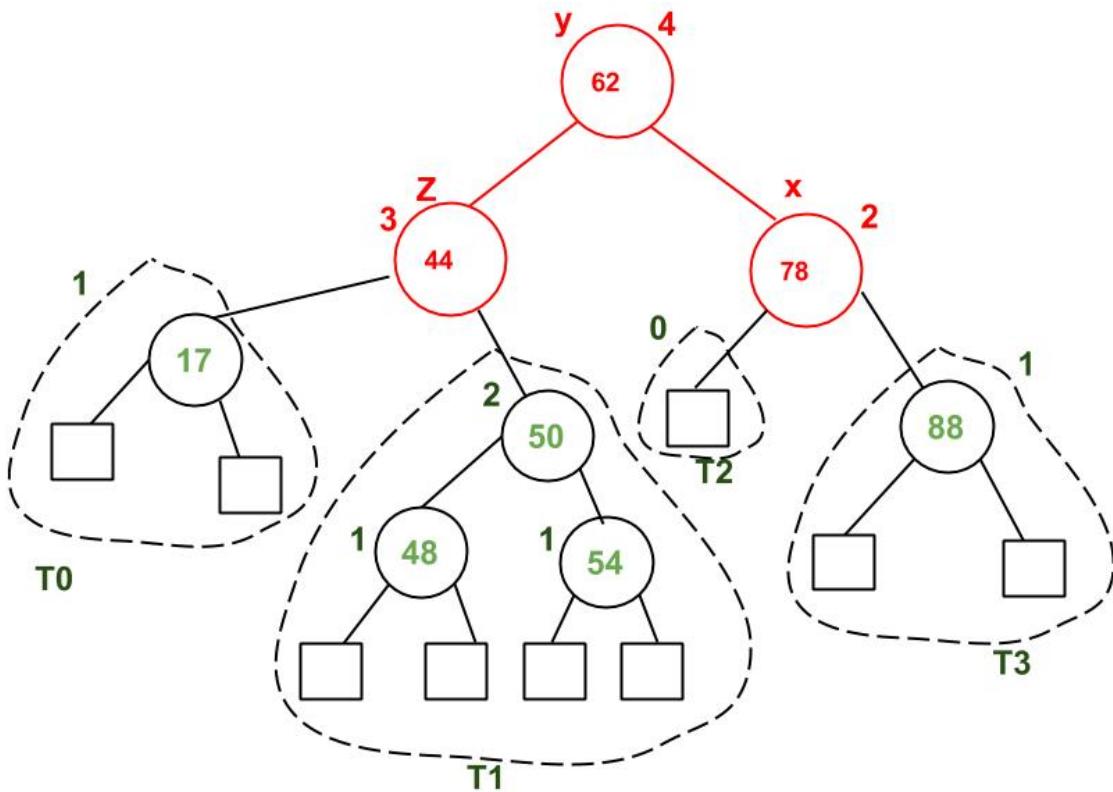


Unlike insertion, in deletion, after we perform a rotation at z , we may have to perform a rotation at ancestors of z . Thus, we must continue to trace the path until we reach the root.

Example:

Example of deletion from an AVL Tree:





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

C implementation

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

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

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

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

C

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

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

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

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

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

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

```

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

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

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

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

    // Return new root
    return x;
}

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

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

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

    // Return new root
    return y;
}

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

```

```

}

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

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

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

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

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

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

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

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

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

```

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

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

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

    return current;
}

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

    if (root == NULL)
        return root;

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

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

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

```

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

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

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

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

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

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

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

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

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

```

```

        return rightRotate(root);
    }

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

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

    return root;
}

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

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

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

    /* The constructed AVL Tree would be

```

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

root = deleteNode(root, 10);

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

return 0;
}
```

Java

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

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

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

class AVLTree
{
```

```
Node root;

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

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

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

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

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

    // Return new root
    return x;
}

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

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

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

```

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

// Return new root
return y;
}

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

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

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

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

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

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

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

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

```

```

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

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

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

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

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

    return current;
}

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

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

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

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

```

```

else
{

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

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

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

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

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

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

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

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

```

```

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

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

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

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

return root;
}

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

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

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

```

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

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

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

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

}

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

Python3

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

```

        self.height = 1

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

    def insert(self, root, key):

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

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

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

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

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

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

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

    return root

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

```

```

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

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

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

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

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

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

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

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

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

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

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

```

```

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

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

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

return root

def leftRotate(self, z):

    y = z.right
    T2 = y.left

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

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

    # Return the new root
    return y

def rightRotate(self, z):

    y = z.left
    T3 = y.right

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

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

```

```
# Return the new root
return y

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

    return root.height

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

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

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

    return self.getMinValueNode(root.left)

def preOrder(self, root):

    if not root:
        return

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

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

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

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

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

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

# This code is contributed by Ajitesh Pathak
```

Output:

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

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

References:

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

Improved By : [AnkushRodewad, KP1975](#)

Source

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

Chapter 4

Advantages of Trie Data Structure

Advantages of Trie Data Structure - GeeksforGeeks

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

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

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

Why Trie? :-

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

Issues with Trie :-

The main disadvantage of tries is that they need lot of memory for storing the strings. For each node we have too many node pointers(equal to number of characters of the alphabet),

If space is concern, then **Ternary Search Tree** can be preferred for dictionary implementations. In Ternary Search Tree, time complexity of search operation is $O(h)$ where h is height of the tree. Ternary Search Trees also supports other operations supported by Trie like prefix search, alphabetical order printing and nearest neighbor search.

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

Improved By : [Aashutosh Rathi](#)

Source

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

Chapter 5

Agents in Artificial Intelligence

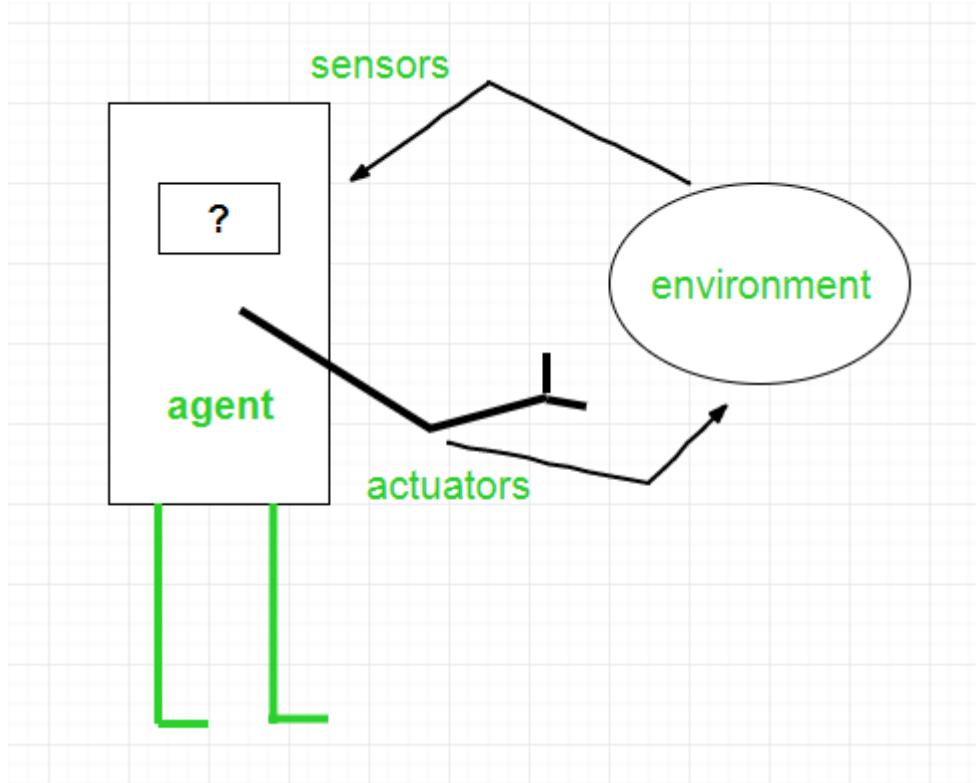
Agents in Artificial Intelligence - GeeksforGeeks

Artificial intelligence is defined as study of rational agents. A rational agent could be anything which makes decisions, like a person, firm, machine, or software. It carries out an action with the best outcome after considering past and current percepts(agent's perceptual inputs at a given instance).

An AI system is composed of an **agent and its environment**. The agents act in their environment. The environment may contain other agents. An agent is anything that can be viewed as :

- perceiving its environment through **sensors** and
- acting upon that environment through **actuators**

Note : Every agent can perceive its own actions (but not always the effects)



To understand the structure of Intelligent Agents, we should be familiar with *Architecture* and *Agent Program*. **Architecture** is the machinery that the agent executes on. It is a device with sensors and actuators, for example : a robotic car, a camera, a PC. **Agent program** is an implementation of an agent function. An **agent function** is a map from the percept sequence(history of all that an agent has perceived till date) to an action.

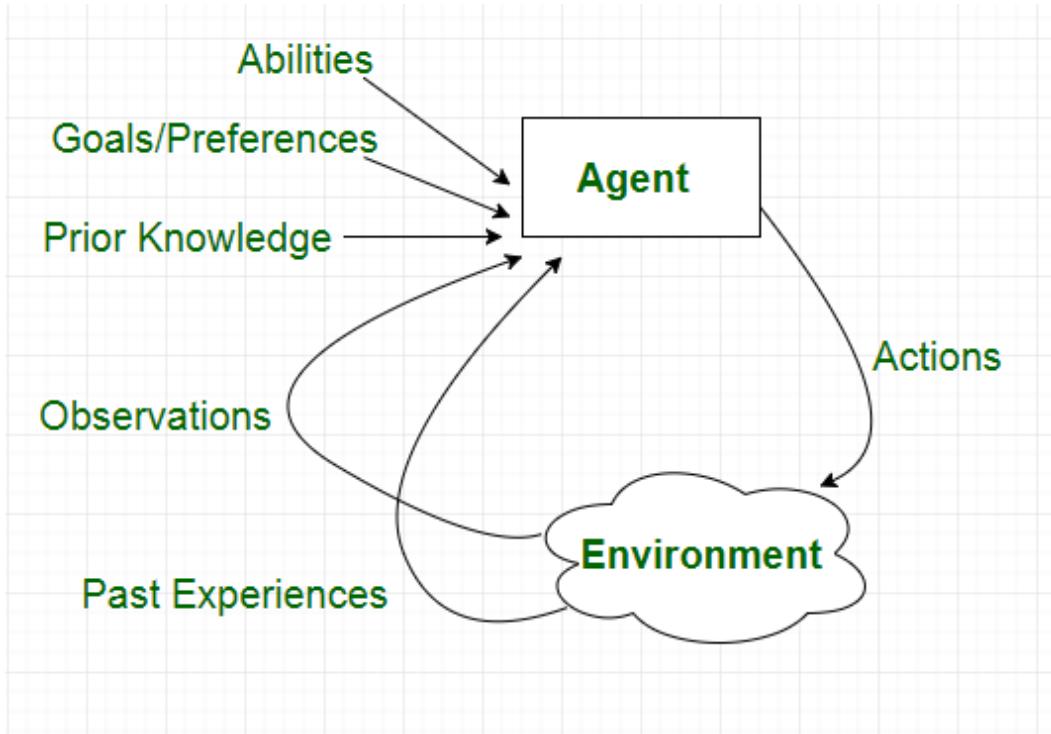
$$\text{Agent} = \text{Architecture} + \text{Agent Program}$$

Examples of Agent:-

A **software agent** has Keystrokes, file contents, received network packages which act as sensors and displays on the screen, files, sent network packets acting as actuators.

A **Human agent** has eyes, ears, and other organs which act as sensors and hands, legs, mouth, and other body parts acting as actuators.

A **Robotic agent** has Cameras and infrared range finders which act as sensors and various motors acting as actuators.



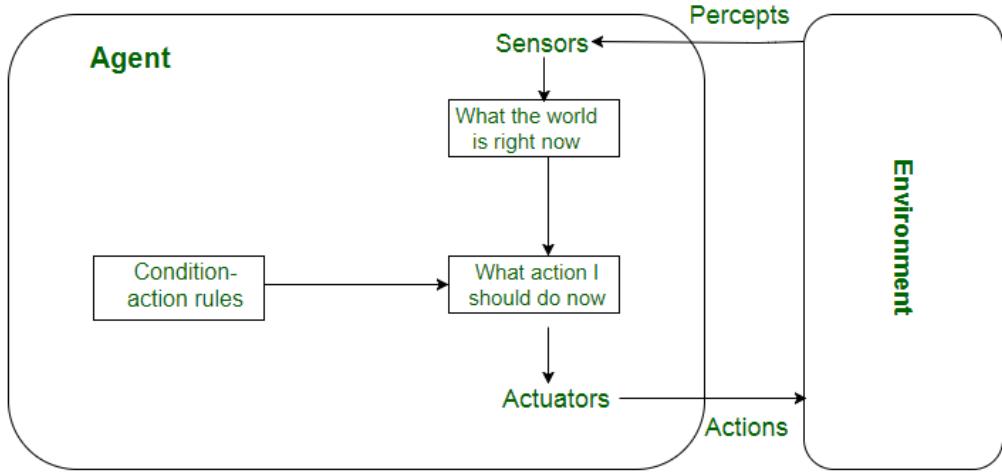
Agents can be grouped into four classes based on their degree of perceived intelligence and capability :

- Simple Reflex Agents
- Model-Based Reflex Agents
- Goal-Based Agents
- Utility-Based Agents

Simple reflex agents ignore the rest of the percept history and act only on the basis of the **current percept**. Percept history is the history of all that an agent has perceived till date. The agent function is based on the **condition-action rule**. A condition-action rule is a rule that maps a state i.e, condition to an action. If the condition is true, then the action is taken, else not. This agent function only succeeds when the environment is fully observable. For simple reflex agents operating in partially observable environments, infinite loops are often unavoidable. It may be possible to escape from infinite loops if the agent can randomize its actions. Problems with Simple reflex agents are :

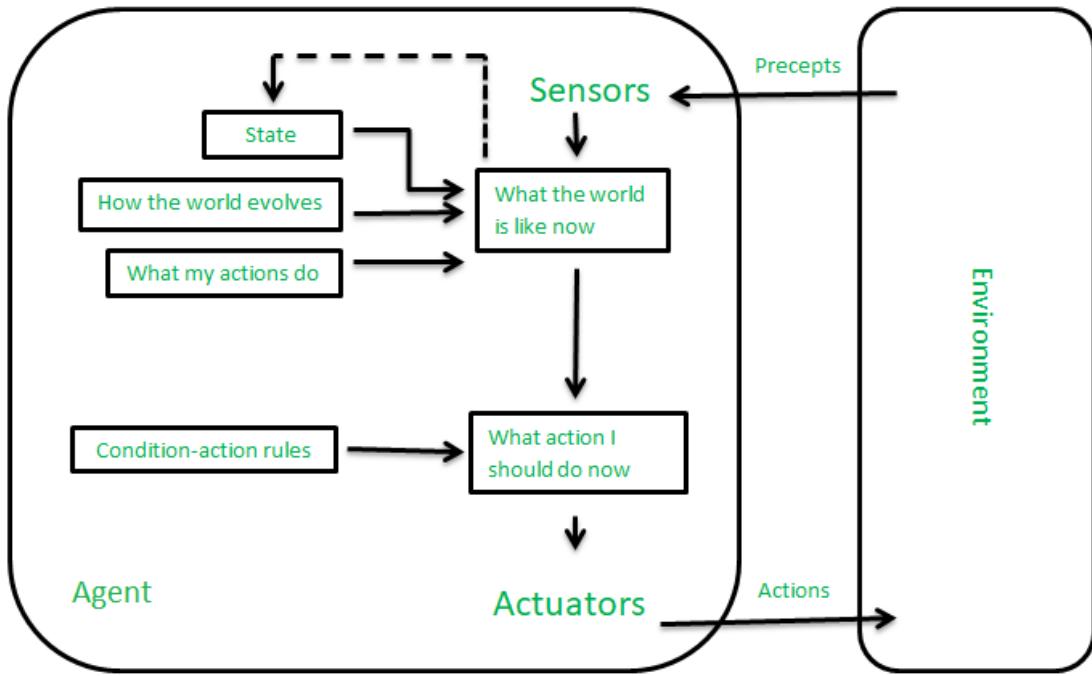
- Very limited intelligence.
- No knowledge of non-perceptual parts of state.
- Usually too big to generate and store.

- If there occurs any change in the environment, then the collection of rules need to be updated.

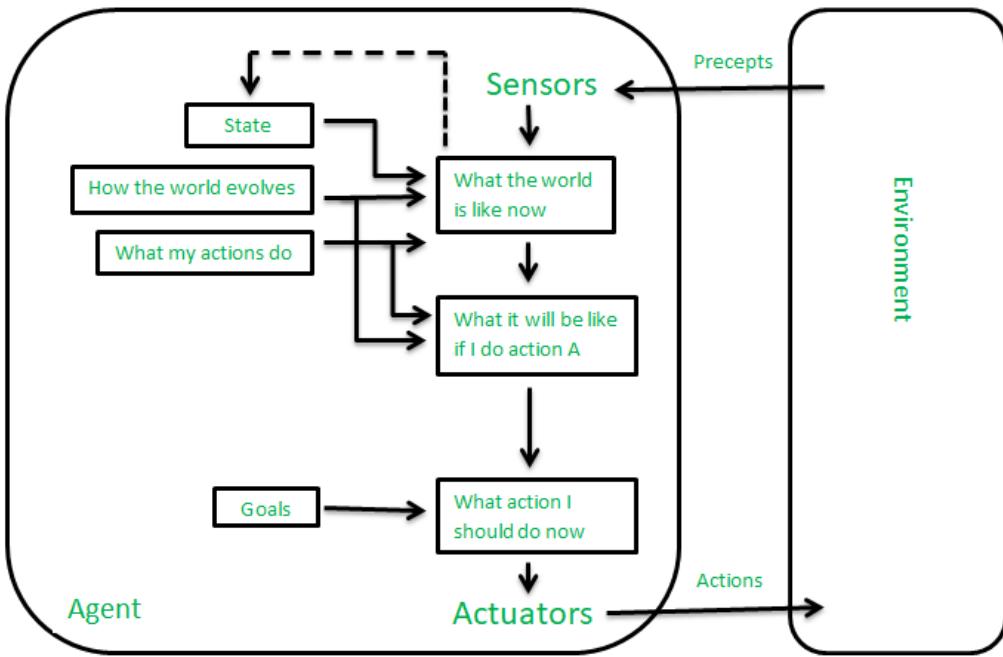


It works by finding a rule whose condition matches the current situation. A model-based agent can handle **partially observable environments** by use of model about the world. The agent has to keep track of **internal state** which is adjusted by each percept and that depends on the percept history. The current state is stored inside the agent which maintains some kind of structure describing the part of the world which cannot be seen. Updating the state requires the information about :

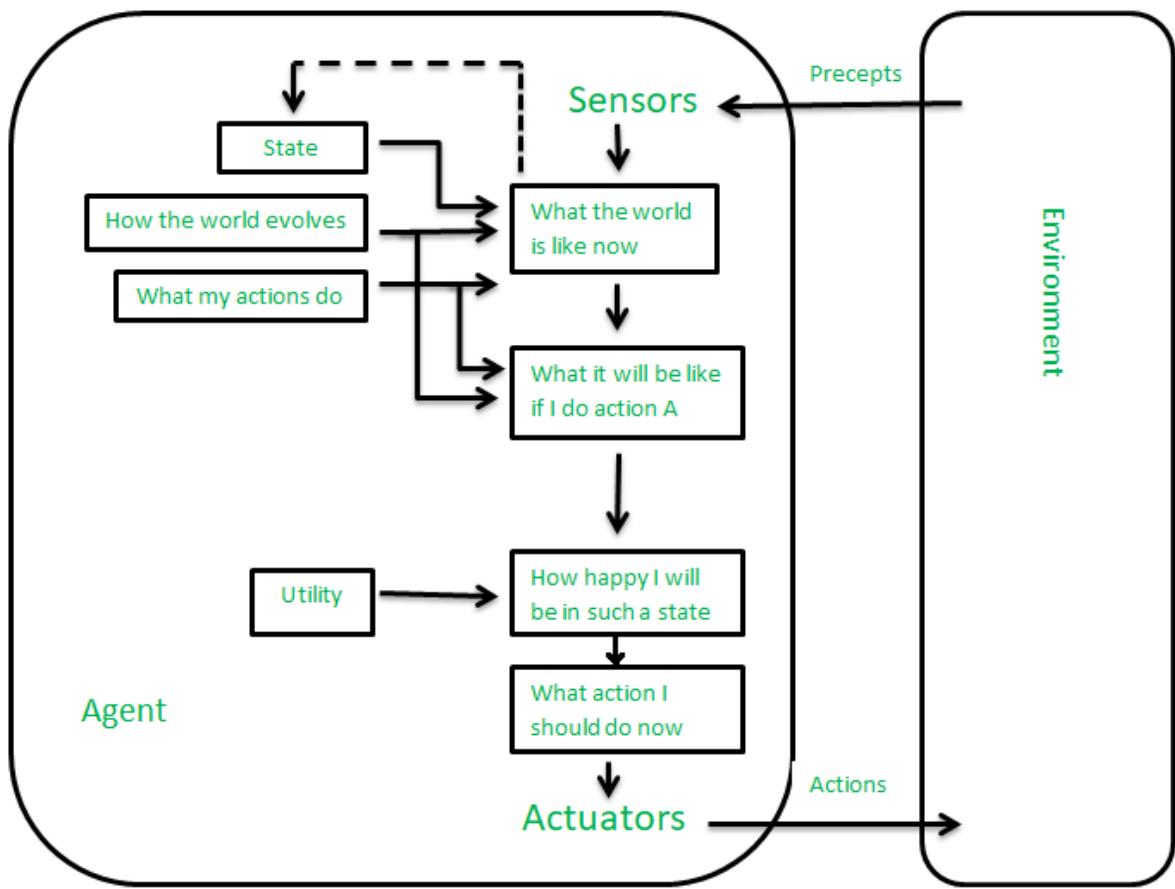
- how the world evolves in-dependently from the agent, and
- how the agent actions affects the world.



These kind of agents take decision based on how far they are currently from their goal(description of desirable situations). Their every action is intended to reduce its distance from goal. This allows the agent a way to choose among multiple possibilities, selecting the one which reaches a goal state. The knowledge that supports its decisions is represented explicitly and can be modified, which makes these agents more flexible. They usually require search and planning. The goal based agent's behavior can easily be changed.



The agents which are developed having their end uses as building blocks are called utility based agents. When there are multiple possible alternatives, then to decide which one is best, utility based agents are used. They choose actions based on a **preference (utility)** for each state. Sometimes achieving the desired goal is not enough. We may look for quicker, safer, cheaper trip to reach a destination. Agent happiness should be taken into consideration. Utility describes how “**happy**” the agent is. Because of the uncertainty in the world, a utility agent chooses the action that maximizes the expected utility. A utility function maps a state onto a real number which describes the associated degree of happiness.



Source

<https://www.geeksforgeeks.org/agents-artificial-intelligence/>

Chapter 6

Array range queries for elements with frequency same as value

Array range queries for elements with frequency same as value - GeeksforGeeks

Given an array of N numbers, the task is to answer Q queries of the following type:-

```
query(start, end) = Number of times a  
number x occurs exactly x times in a  
subarray from start to end
```

Examples:

Input : arr = {1, 2, 2, 3, 3, 3}

Query 1: start = 0, end = 1,
Query 2: start = 1, end = 1,
Query 3: start = 0, end = 2,
Query 4: start = 1, end = 3,
Query 5: start = 3, end = 5,
Query 6: start = 0, end = 5

Output : 1 0 2 1 1 3

Explanation

In Query 1, Element 1 occurs once in subarray [1, 2];

In Query 2, No Element satisfies the required condition in subarray [2];

In Query 3, Element 1 occurs once and 2 occurs twice in subarray [1, 2, 2];

In Query 4, Element 2 occurs twice in subarray [2, 2, 3];

In Query 5, Element 3 occurs thrice in subarray [3, 3, 3];

In Query 6, Element 1 occurs once, 2 occurs twice and 3 occurs thrice in subarray [1, 2, 2, 3, 3, 3]

Method 1 (Brute Force)

Calculate frequency of every element in the subarray under each query. If any number x has frequency x in the subarray covered under each query, we increment the counter.

```
/* C++ Program to answer Q queries to
   find number of times an element x
   appears x times in a Query subarray */
#include <bits/stdc++.h>
using namespace std;

/* Returns the count of number x with
   frequency x in the subarray from
   start to end */
int solveQuery(int start, int end, int arr[])
{
    // map for frequency of elements
    unordered_map<int, int> frequency;

    // store frequency of each element
    // in arr[start; end]
    for (int i = start; i <= end; i++)
        frequency[arr[i]]++;

    // Count elements with same frequency
    // as value
    int count = 0;
    for (auto x : frequency)
        if (x.first == x.second)
            count++;
    return count;
}

int main()
{
    int A[] = { 1, 2, 2, 3, 3, 3 };
    int n = sizeof(A) / sizeof(A[0]);

    // 2D array of queries with 2 columns
    int queries[][][3] = { { 0, 1 },
                           { 1, 1 },
                           { 0, 2 },
                           { 1, 3 },
                           { 3, 5 },
                           { 0, 5 } };

    // calculating number of queries
    int q = sizeof(queries) / sizeof(queries[0]);
```

```
for (int i = 0; i < q; i++) {
    int start = queries[i][0];
    int end = queries[i][1];
    cout << "Answer for Query " << (i + 1)
        << " = " << solveQuery(start,
        end, A) << endl;
}

return 0;
}
```

Output:

```
Answer for Query 1 = 1
Answer for Query 2 = 0
Answer for Query 3 = 2
Answer for Query 4 = 1
Answer for Query 5 = 1
Answer for Query 6 = 3
```

Time Complexity of this method is $O(Q * N)$

Method 2 (Efficient)

We can solve this problem using the [MO's Algorithm](#).

We assign starting index, ending index and query number to each query, Each query takes the following form-

Starting Index(L): Starting Index of the subarray covered under the query;
Ending Index(R) : Ending Index of the subarray covered under the query;
Query Number(Index) : Since queries are sorted, this tells us original position of the query so that we answer the queries in the original order

Firstly, we divide the queries into blocks and sort the queries using a custom comparator. Now we process the queries offline where we keep two pointers i.e. **MO_RIGHT** and **MO_LEFT** with each incoming query, we move these pointers forward and backward and insert and delete elements according to the starting and ending indices of the current query.

Let the current running answer be **current_ans**.

Whenever we **insert** an element we increment the frequency of the included element, if this frequency is equal to the element we just included, we increment the **current_ans**. If the frequency of this element becomes (**current_element + 1**) this means that earlier this element was counted in the **current_ans** when it was equal to its frequency, thus we need to decrement **current_ans** in this case.

Whenever we **delete/remove** an element we decrement the frequency of the excluded element, if this frequency is equal to the element we just excluded, we increment the **current_ans**. If the frequency of this element becomes (**current_element - 1**) this means that

earlier this element was counted in the current_ans when it was equal to its frequency, thus we need to decrement current_ans in this case.

```
/* C++ Program to answer Q queries to
   find number of times an element x
   appears x times in a Query subarray */
#include <bits/stdc++.h>
using namespace std;

// Variable to represent block size.
// This is made global so compare()
// of sort can use it.
int block;

// Structure to represent a query range
struct Query {
    int L, R, index;
};

/* Function used to sort all queries
   so that all queries of same block
   are arranged together and within
   a block, queries are sorted in
   increasing order of R values. */
bool compare(Query x, Query y)
{
    // Different blocks, sort by block.
    if (x.L / block != y.L / block)
        return x.L / block < y.L / block;

    // Same block, sort by R value
    return x.R < y.R;
}

/* Inserts element (x) into current range
   and updates current answer */
void add(int x, int& currentAns,
         unordered_map<int, int>& freq)
{
    // increment frequency of this element
    freq[x]++;

    // if this element was previously
    // contributing to the currentAns,
    // decrement currentAns
    if (freq[x] == (x + 1))
        currentAns--;
}
```

```
// if this element has frequency
// equal to its value, increment
// currentAns
else if (freq[x] == x)
    currentAns++;
}

/* Removes element (x) from current
range btw L and R and updates
current Answer */
void remove(int x, int& currentAns,
            unordered_map<int, int>& freq)
{

    // decrement frequency of this element
    freq[x]--;

    // if this element has frequency equal
    // to its value, increment currentAns
    if (freq[x] == x)
        currentAns++;

    // if this element was previously
    // contributing to the currentAns
    // decrement currentAns
    else if (freq[x] == (x - 1))
        currentAns--;
}

/* Utility Function to answer all queries
and build the ans array in the original
order of queries */
void queryResultsUtil(int a[], Query q[],
                      int ans[], int m)
{

    // map to store freq of each element
    unordered_map<int, int> freq;

    // Initialize current L, current R
    // and current sum
    int currL = 0, currR = 0;
    int currentAns = 0;

    // Traverse through all queries
    for (int i = 0; i < m; i++) {
        // L and R values of current range
```

```

int L = q[i].L, R = q[i].R;
int index = q[i].index;

// Remove extra elements of previous
// range. For example if previous
// range is [0, 3] and current range
// is [2, 5], then a[0] and a[1] are
// removed
while (currL < L) {
    remove(a[currL], currentAns, freq);
    currL++;
}

// Add Elements of current Range
while (currL > L) {
    currL--;
    add(a[currL], currentAns, freq);
}
while (currR <= R) {
    add(a[currR], currentAns, freq);
    currR++;
}

// Remove elements of previous range. For example
// when previous range is [0, 10] and current range
// is [3, 8], then a[9] and a[10] are Removed
while (currR > R + 1) {
    currR--;
    remove(a[currR], currentAns, freq);
}

// Store current ans as the Query ans for
// Query number index
ans[index] = currentAns;
}

}

/* Wrapper for queryResultsUtil() and outputs the
   ans array constructed by answering all queries */
void queryResults(int a[], int n, Query q[], int m)
{
    // Find block size
    block = (int)sqrt(n);

    // Sort all queries so that queries of same blocks
    // are arranged together.
    sort(q, q + m, compare);
}

```

```

int* ans = new int[m];
queryResultsUtil(a, q, ans, m);

for (int i = 0; i < m; i++) {
    cout << "Answer for Query " << (i + 1)
        << " = " << ans[i] << endl;
}
}

// Driver program
int main()
{
    int A[] = { 1, 2, 2, 3, 3, 3 };

    int n = sizeof(A) / sizeof(A[0]);

    // 2D array of queries with 2 columns
    Query queries[] = { { 0, 1, 0 },
                        { 1, 1, 1 },
                        { 0, 2, 2 },
                        { 1, 3, 3 },
                        { 3, 5, 4 },
                        { 0, 5, 5 } };

    // calculating number of queries
    int q = sizeof(queries) / sizeof(queries[0]);

    // Print result for each Query
    queryResults(A, n, queries, q);

    return 0;
}

```

Output:

```

Answer for Query 1 = 1
Answer for Query 2 = 0
Answer for Query 3 = 2
Answer for Query 4 = 1
Answer for Query 5 = 1
Answer for Query 6 = 3

```

Time Complexity of this approach using MO's Algorithm is **O(Q * sqrt(N) * logA)** where logA is the complexity to insert an element A into the unordered_map for each query.

Source

<https://www.geeksforgeeks.org/array-range-queries-elements-frequency-value/>

Chapter 7

Auto-complete feature using Trie

Auto-complete feature using Trie - GeeksforGeeks

We are given a Trie with a set of strings stored in it. Now the user types in a prefix of his search query, we need to give him all recommendations to auto-complete his query based on the strings stored in the Trie. We assume that the Trie stores past searches by the users.

For example if the Trie store {"abc", "abcd", "aa", "abbbaba"} and the User types in "ab" then he must be shown {"abc", "abcd", "abbbaba"}.

[Prerequisite Trie Search and Insert.](#)

Given a query prefix, we search for all words having this query.

1. Search for given query using standard Trie search algorithm.
2. If query prefix itself is not present, return -1 to indicate the same.
3. If query is present and is end of word in Trie, print query. This can quickly checked by seeing if last matching node has isEndWord flag set. We use this flag in Trie to mark end of word nodes for purpose of searching.
4. If last matching node of query has no children, return.
5. Else recursively print all nodes under subtree of last matching node.

Below is C++ implementation of above steps.

```
// C++ program to demonstrate auto-complete feature
// using Trie data structure.
#include<bits/stdc++.h>
using namespace std;
```

```

// Alphabet size (# of symbols)
#define ALPHABET_SIZE (26)

// Converts key current character into index
// use only 'a' through 'z' and lower case
#define CHAR_TO_INDEX(c) ((int)c - (int)'a')

// trie node
struct TrieNode
{
    struct TrieNode *children[ALPHABET_SIZE];

    // isWordEnd is true if the node represents
    // end of a word
    bool isWordEnd;
};

// Returns new trie node (initialized to NULLs)
struct TrieNode *getNode(void)
{
    struct TrieNode *pNode = new TrieNode;
    pNode->isWordEnd = false;

    for (int i = 0; i < ALPHABET_SIZE; i++)
        pNode->children[i] = NULL;

    return pNode;
}

// If not present, inserts key into trie.  If the
// key is prefix of trie node, just marks leaf node
void insert(struct TrieNode *root, const string key)
{
    struct TrieNode *pCrawl = root;

    for (int level = 0; level < key.length(); level++)
    {
        int index = CHAR_TO_INDEX(key[level]);
        if (!pCrawl->children[index])
            pCrawl->children[index] = getNode();

        pCrawl = pCrawl->children[index];
    }

    // mark last node as leaf
    pCrawl->isWordEnd = true;
}

```

```

// Returns true if key presents in trie, else false
bool search(struct TrieNode *root, const string key)
{
    int length = key.length();
    struct TrieNode *pCrawl = root;
    for (int level = 0; level < length; level++)
    {
        int index = CHAR_TO_INDEX(key[level]);

        if (!pCrawl->children[index])
            return false;

        pCrawl = pCrawl->children[index];
    }

    return (pCrawl != NULL && pCrawl->isWordEnd);
}

// Returns 0 if current node has a child
// If all children are NULL, return 1.
bool isLastNode(struct TrieNode* root)
{
    for (int i = 0; i < ALPHABET_SIZE; i++)
        if (root->children[i])
            return 0;
    return 1;
}

// Recursive function to print auto-suggestions for given
// node.
void suggestionsRec(struct TrieNode* root, string currPrefix)
{
    // found a string in Trie with the given prefix
    if (root->isWordEnd)
    {
        cout << currPrefix;
        cout << endl;
    }

    // All children struct node pointers are NULL
    if (isLastNode(root))
        return;

    for (int i = 0; i < ALPHABET_SIZE; i++)
    {
        if (root->children[i])
        {
            // append current character to currPrefix string

```

```

        currPrefix.push_back(97 + i);

        // recur over the rest
        suggestionsRec(root->children[i], currPrefix);
    }
}

// print suggestions for given query prefix.
int printAutoSuggestions(TrieNode* root, const string query)
{
    struct TrieNode* pCrawl = root;

    // Check if prefix is present and find the
    // the node (of last level) with last character
    // of given string.
    int level;
    int n = query.length();
    for (level = 0; level < n; level++)
    {
        int index = CHAR_TO_INDEX(query[level]);

        // no string in the Trie has this prefix
        if (!pCrawl->children[index])
            return 0;

        pCrawl = pCrawl->children[index];
    }

    // If prefix is present as a word.
    bool isWord = (pCrawl->isWordEnd == true);

    // If prefix is last node of tree (has no
    // children)
    bool isLast = isLastNode(pCrawl);

    // If prefix is present as a word, but
    // there is no subtree below the last
    // matching node.
    if (isWord && isLast)
    {
        cout << query << endl;
        return -1;
    }

    // If there are are nodes below last
    // matching character.
    if (!isLast)

```

```

    {
        string prefix = query;
        suggestionsRec(pCrawl, prefix);
        return 1;
    }
}

// Driver Code
int main()
{
    struct TrieNode* root = getNode();
    insert(root, "hello");
    insert(root, "dog");
    insert(root, "hell");
    insert(root, "cat");
    insert(root, "a");
    insert(root, "hel");
    insert(root, "help");
    insert(root, "helps");
    insert(root, "helping");
    int comp = printAutoSuggestions(root, "hel");

    if (comp == -1)
        cout << "No other strings found with this prefix\n";

    else if (comp == 0)
        cout << "No string found with this prefix\n";

    return 0;
}

```

Output:

```

hel
hell
hello
help
helping
helps

```

How can we improve this?

The number of matches might just be too large so we have to be selective while displaying them. We can restrict ourselves to display only the relevant results. By relevant, we can consider the past search history and show only the most searched matching strings as relevant results.

Store another value for the each node where isleaf=True which contains the number of hits

for that query search. For example if “hat” is searched 10 times, then we store this 10 at the last node for “hat”. Now when we want to show the recommendations, we display the top k matches with the highest hits. Try to implement this on your own.

Source

<https://www.geeksforgeeks.org/auto-complete-feature-using-trie/>

Chapter 8

B-Tree Set 1 (Introduction)

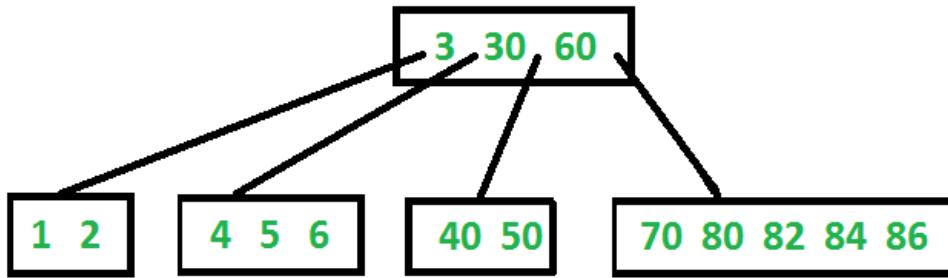
B-Tree Set 1 (Introduction) - GeeksforGeeks

B-Tree is a self-balancing search tree. In most of the other self-balancing search trees (like [AVL](#) and Red-Black Trees), it is assumed that everything is in main memory. To understand the use of B-Trees, we must think of the huge amount of data that cannot fit in main memory. When the number of keys is high, the data is read from disk in the form of blocks. Disk access time is very high compared to main memory access time. The main idea of using B-Trees is to reduce the number of disk accesses. Most of the tree operations (search, insert, delete, max, min, ..etc) require $O(h)$ disk accesses where h is the height of the tree. B-tree is a fat tree. The height of B-Trees is kept low by putting maximum possible keys in a B-Tree node. Generally, a B-Tree node size is kept equal to the disk block size. Since h is low for B-Tree, total disk accesses for most of the operations are reduced significantly compared to balanced Binary Search Trees like AVL Tree, Red-Black Tree, ..etc.

Properties of B-Tree

- 1) All leaves are at same level.
- 2) A B-Tree is defined by the term *minimum degree 't'*. The value of t depends upon disk block size.
- 3) Every node except root must contain at least $t-1$ keys. Root may contain minimum 1 key.
- 4) All nodes (including root) may contain at most $2t - 1$ keys.
- 5) Number of children of a node is equal to the number of keys in it plus 1.
- 6) All keys of a node are sorted in increasing order. The child between two keys k_1 and k_2 contains all keys in the range from k_1 and k_2 .
- 7) B-Tree grows and shrinks from the root which is unlike Binary Search Tree. Binary Search Trees grow downward and also shrink from downward.
- 8) Like other balanced Binary Search Trees, time complexity to search, insert and delete is $O(\log n)$.

Following is an example B-Tree of minimum degree 3. Note that in practical B-Trees, the value of minimum degree is much more than 3.



Search

Search is similar to the search in Binary Search Tree. Let the key to be searched be k . We start from the root and recursively traverse down. For every visited non-leaf node, if the node has the key, we simply return the node. Otherwise, we recur down to the appropriate child (The child which is just before the first greater key) of the node. If we reach a leaf node and don't find k in the leaf node, we return NULL.

Traverse

Traversal is also similar to Inorder traversal of Binary Tree. We start from the leftmost child, recursively print the leftmost child, then repeat the same process for remaining children and keys. In the end, recursively print the rightmost child.

```
// C++ implementation of search() and traverse() methods
#include<iostream>
using namespace std;

// A BTREE node
class BTREENode
{
    int *keys; // An array of keys
    int t; // Minimum degree (defines the range for number of keys)
    BTREENode **C; // An array of child pointers
    int n; // Current number of keys
    bool leaf; // Is true when node is leaf. Otherwise false
public:
    BTREENode(int _t, bool _leaf); // Constructor

    // A function to traverse all nodes in a subtree rooted with this node
    void traverse();

    // A function to search a key in the subtree rooted with this node.
    BTREENode *search(int k); // returns NULL if k is not present.

    // Make the BTREE friend of this so that we can access private members of this
    // class in BTREE functions
    friend class BTREE;
};


```

```

// A BTree
class BTree
{
    BTreeNode *root; // Pointer to root node
    int t; // Minimum degree
public:
    // Constructor (Initializes tree as empty)
    BTree(int _t)
    { root = NULL; t = _t; }

    // function to traverse the tree
    void traverse()
    { if (root != NULL) root->traverse(); }

    // function to search a key in this tree
    BTreeNode* search(int k)
    { return (root == NULL)? NULL : root->search(k); }
};

// Constructor for BTreeNode class
BTreeNode::BTreeNode(int _t, bool _leaf)
{
    // Copy the given minimum degree and leaf property
    t = _t;
    leaf = _leaf;

    // Allocate memory for maximum number of possible keys
    // and child pointers
    keys = new int[2*t-1];
    C = new BTreeNode *[2*t];

    // Initialize the number of keys as 0
    n = 0;
}

// Function to traverse all nodes in a subtree rooted with this node
void BTreeNode::traverse()
{
    // There are n keys and n+1 children, travers through n keys
    // and first n children
    int i;
    for (i = 0; i < n; i++)
    {
        // If this is not leaf, then before printing key[i],
        // traverse the subtree rooted with child C[i].
        if (leaf == false)
            C[i]->traverse();
}

```

```
        cout << " " << keys[i];
    }

    // Print the subtree rooted with last child
    if (leaf == false)
        C[i]->traverse();
}

// Function to search key k in subtree rooted with this node
BTreeNode *BTreeNode::search(int k)
{
    // Find the first key greater than or equal to k
    int i = 0;
    while (i < n && k > keys[i])
        i++;

    // If the found key is equal to k, return this node
    if (keys[i] == k)
        return this;

    // If the key is not found here and this is a leaf node
    if (leaf == true)
        return NULL;

    // Go to the appropriate child
    return C[i]->search(k);
}
```

The above code doesn't contain driver program. We will be covering the complete program in our next post on B-Tree Insertion.

There are two conventions to define a B-Tree, one is to define by minimum degree (followed in [Cormen book](#)), second is define by order. We have followed the minimum degree convention and will be following same in coming posts on B-Tree. The variable names used in the above program are also kept same as Cormen book for better readability.

Insertion and Deletion

[B-Tree Insertion](#)

[B-Tree Deletion](#)

References:

[Introduction to Algorithms 3rd Edition by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest](#)

Source

<https://www.geeksforgeeks.org/b-tree-set-1-introduction-2/>

Chapter 9

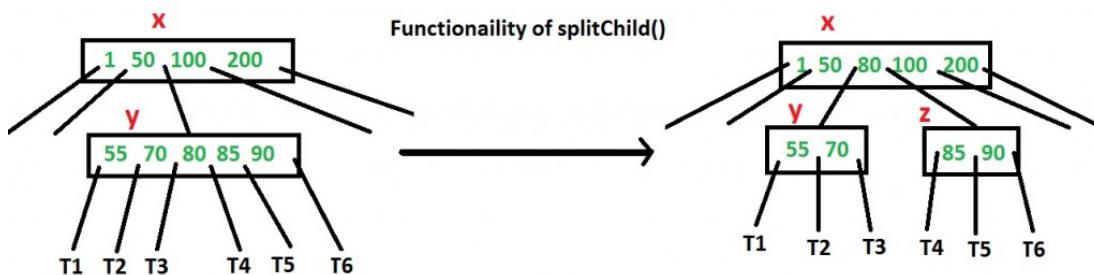
B-Tree Set 2 (Insert)

B-Tree Set 2 (Insert) - GeeksforGeeks

In the [previous post](#), we introduced B-Tree. We also discussed search() and traverse() functions.

In this post, insert() operation is discussed. A new key is always inserted at leaf node. Let the key to be inserted be k. Like BST, we start from root and traverse down till we reach a leaf node. Once we reach a leaf node, we insert the key in that leaf node. Unlike BSTs, we have a predefined range on number of keys that a node can contain. So before inserting a key to node, we make sure that the node has extra space.

How to make sure that a node has space available for key before the key is inserted? We use an operation called splitChild() that is used to split a child of a node. See the following diagram to understand split. In the following diagram, child y of x is being split into two nodes y and z. Note that the splitChild operation moves a key up and this is the reason B-Trees grow up unlike BSTs which grow down.



As discussed above, to insert a new key, we go down from root to leaf. Before traversing down to a node, we first check if the node is full. If the node is full, we split it to create space. Following is complete algorithm.

Insertion

- 1) Initialize x as root.
- 2) While x is not leaf, do following

- ..a) Find the child of x that is going to be traversed next. Let the child be y.
- ..b) If y is not full, change x to point to y.
- ..c) If y is full, split it and change x to point to one of the two parts of y. If k is smaller than mid key in y, then set x as first part of y. Else second part of y. When we split y, we move a key from y to its parent x.
- 3) The loop in step 2 stops when x is leaf. x must have space for 1 extra key as we have been splitting all nodes in advance. So simply insert k to x.

Note that the algorithm follows the Cormen book. It is actually a proactive insertion algorithm where before going down to a node, we split it if it is full. The advantage of splitting before is, we never traverse a node twice. If we don't split a node before going down to it and split it only if new key is inserted (reactive), we may end up traversing all nodes again from leaf to root. This happens in cases when all nodes on the path from root to leaf are full. So when we come to the leaf node, we split it and move a key up. Moving a key up will cause a split in parent node (because parent was already full). This cascading effect never happens in this proactive insertion algorithm. There is a disadvantage of this proactive insertion though, we may do unnecessary splits.

Let us understand the algorithm with an example tree of minimum degree 't' as 3 and a sequence of integers 10, 20, 30, 40, 50, 60, 70, 80 and 90 in an initially empty B-Tree.

Initially root is NULL. Let us first insert 10.

Insert 10



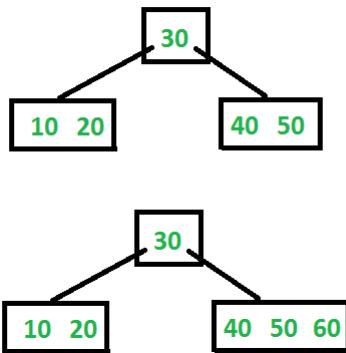
Let us now insert 20, 30, 40 and 50. They all will be inserted in root because maximum number of keys a node can accommodate is $2*t - 1$ which is 5.

Insert 20, 30, 40 and 50

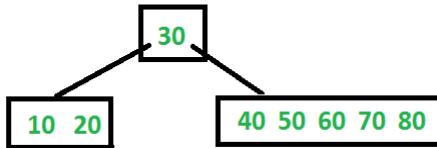


Let us now insert 60. Since root node is full, it will first split into two, then 60 will be inserted into the appropriate child.

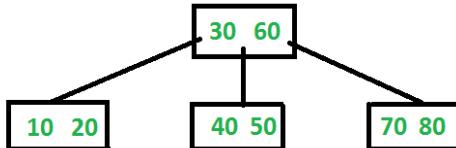
Insert 60



Let us now insert 70 and 80. These new keys will be inserted into the appropriate leaf without any split.

Insert 70 and 80


Let us now insert 90. This insertion will cause a split. The middle key will go up to the parent.

Insert 90


See [this](#) for more examples.

Following is C++ implementation of the above proactive algorithm.

```

// C++ program for B-Tree insertion
#include<iostream>
using namespace std;

// A BTREE node
class BTREENode
{
    int *keys; // An array of keys
    int t; // Minimum degree (defines the range for number of keys)
    BTREENode **C; // An array of child pointers
    int n; // Current number of keys
    bool leaf; // Is true when node is leaf. Otherwise false
public:
    BTREENode(int _t, bool _leaf); // Constructor

    // A utility function to insert a new key in the subtree rooted with
    // this node. The assumption is, the node must be non-full when this
    // function is called
    void insertNonFull(int k);

    // A utility function to split the child y of this node. i is index of y in
    // child array C[]. The Child y must be full when this function is called
    void splitChild(int i, BTREENode *y);

    // A function to traverse all nodes in a subtree rooted with this node
    void traverse();

    // A function to search a key in subtree rooted with this node.
}
  
```

```
BTreenode *search(int k); // returns NULL if k is not present.

// Make BTreenode friend of this so that we can access private members of this
// class in BTreenode functions
friend class BTreenode;
};

// A BTreenode
class BTreenode
{
    BTreenode *root; // Pointer to root node
    int t; // Minimum degree
public:
    // Constructor (Initializes tree as empty)
    BTreenode(int _t)
    { root = NULL; t = _t; }

    // function to traverse the tree
    void traverse()
    { if (root != NULL) root->traverse(); }

    // function to search a key in this tree
    BTreenode* search(int k)
    { return (root == NULL)? NULL : root->search(k); }

    // The main function that inserts a new key in this BTreenode
    void insert(int k);
};

// Constructor for BTreenode class
BTreenode::BTreenode(int t1, bool leaf1)
{
    // Copy the given minimum degree and leaf property
    t = t1;
    leaf = leaf1;

    // Allocate memory for maximum number of possible keys
    // and child pointers
    keys = new int[2*t-1];
    C = new BTreenode *[2*t];

    // Initialize the number of keys as 0
    n = 0;
}

// Function to traverse all nodes in a subtree rooted with this node
void BTreenode::traverse()
{
```

```

// There are n keys and n+1 children, travers through n keys
// and first n children
int i;
for (i = 0; i < n; i++)
{
    // If this is not leaf, then before printing key[i],
    // traverse the subtree rooted with child C[i].
    if (leaf == false)
        C[i]->traverse();
    cout << " " << keys[i];
}

// Print the subtree rooted with last child
if (leaf == false)
    C[i]->traverse();
}

// Function to search key k in subtree rooted with this node
BTreeNode *BTreeNode::search(int k)
{
    // Find the first key greater than or equal to k
    int i = 0;
    while (i < n && k > keys[i])
        i++;

    // If the found key is equal to k, return this node
    if (keys[i] == k)
        return this;

    // If key is not found here and this is a leaf node
    if (leaf == true)
        return NULL;

    // Go to the appropriate child
    return C[i]->search(k);
}

// The main function that inserts a new key in this B-Tree
void BTREE::insert(int k)
{
    // If tree is empty
    if (root == NULL)
    {
        // Allocate memory for root
        root = new BTreeNode(t, true);
        root->keys[0] = k; // Insert key
        root->n = 1; // Update number of keys in root
    }
}

```

```

else // If tree is not empty
{
    // If root is full, then tree grows in height
    if (root->n == 2*t-1)
    {
        // Allocate memory for new root
        BTreenode *s = new BTreenode(t, false);

        // Make old root as child of new root
        s->C[0] = root;

        // Split the old root and move 1 key to the new root
        s->splitChild(0, root);

        // New root has two children now. Decide which of the
        // two children is going to have new key
        int i = 0;
        if (s->keys[0] < k)
            i++;
        s->C[i]->insertNonFull(k);

        // Change root
        root = s;
    }
    else // If root is not full, call insertNonFull for root
        root->insertNonFull(k);
}
}

// A utility function to insert a new key in this node
// The assumption is, the node must be non-full when this
// function is called
void BTreenode::insertNonFull(int k)
{
    // Initialize index as index of rightmost element
    int i = n-1;

    // If this is a leaf node
    if (leaf == true)
    {
        // The following loop does two things
        // a) Finds the location of new key to be inserted
        // b) Moves all greater keys to one place ahead
        while (i >= 0 && keys[i] > k)
        {
            keys[i+1] = keys[i];
            i--;
        }
    }
}

```

```

        // Insert the new key at found location
        keys[i+1] = k;
        n = n+1;
    }
    else // If this node is not leaf
    {
        // Find the child which is going to have the new key
        while (i >= 0 && keys[i] > k)
            i--;
        // See if the found child is full
        if (C[i+1]->n == 2*t-1)
        {
            // If the child is full, then split it
            splitChild(i+1, C[i+1]);
            // After split, the middle key of C[i] goes up and
            // C[i] is splitted into two. See which of the two
            // is going to have the new key
            if (keys[i+1] < k)
                i++;
        }
        C[i+1]->insertNonFull(k);
    }
}

// A utility function to split the child y of this node
// Note that y must be full when this function is called
void BTreenode::splitChild(int i, BTreenode *y)
{
    // Create a new node which is going to store (t-1) keys
    // of y
    BTreenode *z = new BTreenode(y->t, y->leaf);
    z->n = t - 1;

    // Copy the last (t-1) keys of y to z
    for (int j = 0; j < t-1; j++)
        z->keys[j] = y->keys[j+t];

    // Copy the last t children of y to z
    if (y->leaf == false)
    {
        for (int j = 0; j < t; j++)
            z->C[j] = y->C[j+t];
    }

    // Reduce the number of keys in y
}

```

```
y->n = t - 1;

// Since this node is going to have a new child,
// create space of new child
for (int j = n; j >= i+1; j--)
    C[j+1] = C[j];

// Link the new child to this node
C[i+1] = z;

// A key of y will move to this node. Find location of
// new key and move all greater keys one space ahead
for (int j = n-1; j >= i; j--)
    keys[j+1] = keys[j];

// Copy the middle key of y to this node
keys[i] = y->keys[t-1];

// Increment count of keys in this node
n = n + 1;
}

// Driver program to test above functions
int main()
{
    BTREE t(3); // A B-Tree with minium degree 3
    t.insert(10);
    t.insert(20);
    t.insert(5);
    t.insert(6);
    t.insert(12);
    t.insert(30);
    t.insert(7);
    t.insert(17);

    cout << "Traversal of the constucted tree is ";
    t.traverse();

    int k = 6;
    (t.search(k) != NULL)? cout << "\nPresent" : cout << "\nNot Present";

    k = 15;
    (t.search(k) != NULL)? cout << "\nPresent" : cout << "\nNot Present";

    return 0;
}
```

Output:

Traversal of the constructed tree is 5 6 7 10 12 17 20 30
Present
Not Present

References:

Introduction to Algorithms 3rd Edition by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest
<http://www.cs.utexas.edu/users/djimenez/utsa/cs3343/lecture17.html>

Source

<https://www.geeksforgeeks.org/b-tree-set-1-insert-2/>

Chapter 10

B-Tree Set 3 (Delete)

B-Tree Set 3 (Delete) - GeeksforGeeks

It is recommended to refer following posts as prerequisite of this post.

[B-Tree Set 1 \(Introduction\)](#)

[B-Tree Set 2 \(Insert\)](#)

B-Tree is a type of a multi-way search tree. So, if you are not familiar with multi-way search trees in general, it is better to take a look at [this video lecture from IIT-Delhi](#), before proceeding further. Once you get the basics of a multi-way search tree clear, B-Tree operations will be easier to understand.

Source of the following explanation and algorithm is [Introduction to Algorithms 3rd Edition by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest](#)

Deletion process:

Deletion from a B-tree is more complicated than insertion, because we can delete a key from any node—not just a leaf—and when we delete a key from an internal node, we will have to rearrange the node’s children.

As in insertion, we must make sure the deletion doesn’t violate the [B-tree properties](#). Just as we had to ensure that a node didn’t get too big due to insertion, we must ensure that a node doesn’t get too small during deletion (except that the root is allowed to have fewer than the minimum number $t-1$ of keys). Just as a simple insertion algorithm might have to back up if a node on the path to where the key was to be inserted was full, a simple approach to deletion might have to back up if a node (other than the root) along the path to where the key is to be deleted has the minimum number of keys.

The deletion procedure deletes the key k from the subtree rooted at x . This procedure guarantees that whenever it calls itself recursively on a node x , the number of keys in x is at least the minimum degree t . Note that this condition requires one more key than the minimum required by the usual B-tree conditions, so that sometimes a key may have to be moved into a child node before recursion descends to that child. This strengthened condition allows us to delete a key from the tree in one downward pass without having to “back up” (with one exception, which we’ll explain). You should interpret the following specification

for deletion from a B-tree with the understanding that if the root node x ever becomes an internal node having no keys (this situation can occur in cases 2c and 3b then we delete x , and x 's only child $x.c_1$ becomes the new root of the tree, decreasing the height of the tree by one and preserving the property that the root of the tree contains at least one key (unless the tree is empty).

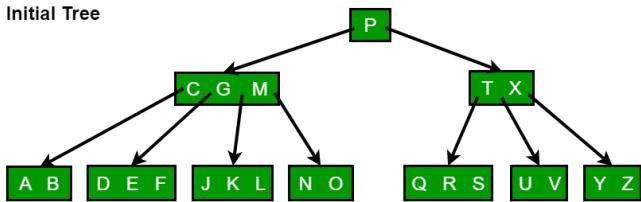
We sketch how deletion works with various cases of deleting keys from a B-tree.

- 1.** If the key k is in node x and x is a leaf, delete the key k from x .
- 2.** If the key k is in node x and x is an internal node, do the following.
 - a)** If the child y that precedes k in node x has at least t keys, then find the predecessor k_0 of k in the sub-tree rooted at y . Recursively delete k_0 , and replace k by k_0 in x . (We can find k_0 and delete it in a single downward pass.)
 - b)** If y has fewer than t keys, then, symmetrically, examine the child z that follows k in node x . If z has at least t keys, then find the successor k_0 of k in the subtree rooted at z . Recursively delete k_0 , and replace k by k_0 in x . (We can find k_0 and delete it in a single downward pass.)
 - c)** Otherwise, if both y and z have only $t-1$ keys, merge k and all of z into y , so that x loses both k and the pointer to z , and y now contains $2t-1$ keys. Then free z and recursively delete k from y .
- 3.** If the key k is not present in internal node x , determine the root $x.c(i)$ of the appropriate subtree that must contain k , if k is in the tree at all. If $x.c(i)$ has only $t-1$ keys, execute step 3a or 3b as necessary to guarantee that we descend to a node containing at least t keys. Then finish by recursing on the appropriate child of x .
 - a)** If $x.c(i)$ has only $t-1$ keys but has an immediate sibling with at least t keys, give $x.c(i)$ an extra key by moving a key from x down into $x.c(i)$, moving a key from $x.c(i)$'s immediate left or right sibling up into x , and moving the appropriate child pointer from the sibling into $x.c(i)$.
 - b)** If $x.c(i)$ and both of $x.c(i)$'s immediate siblings have $t-1$ keys, merge $x.c(i)$ with one sibling, which involves moving a key from x down into the new merged node to become the median key for that node.

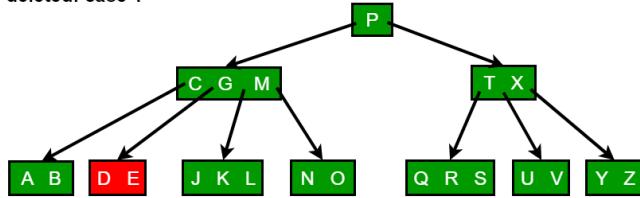
Since most of the keys in a B-tree are in the leaves, deletion operations are most often used to delete keys from leaves. The recursive delete procedure then acts in one downward pass through the tree, without having to back up. When deleting a key in an internal node, however, the procedure makes a downward pass through the tree but may have to return to the node from which the key was deleted to replace the key with its predecessor or successor (cases 2a and 2b).

The following figures explain the deletion process.

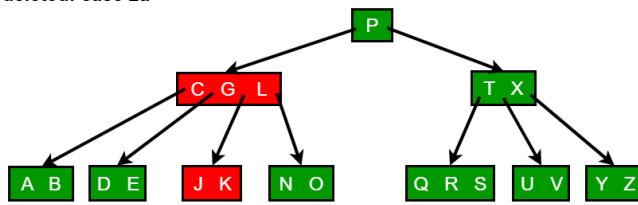
(a) Initial Tree



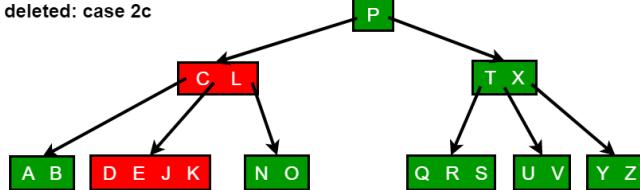
(b) F deleted: case 1



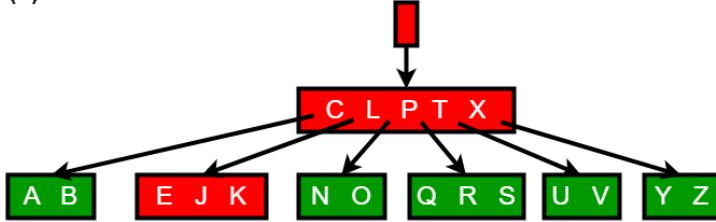
(c) M deleted: case 2a



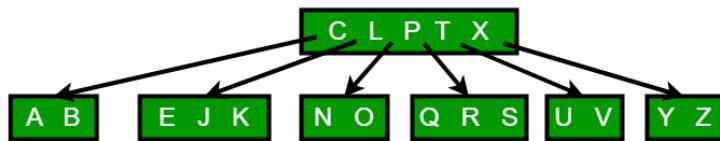
(d) G deleted: case 2c



(e) D deleted: case 3b



(e') tree shrinks in height



(f) B deleted: case 3a



Implementation:

Following is C++ implementation of deletion process.

```
/* The following program performs deletion on a B-Tree. It contains functions
specific for deletion along with all the other functions provided in the
previous articles on B-Trees. See https://www.geeksforgeeks.org/b-tree-set-1-introduction-2/
for previous article.
```

The deletion function has been compartmentalized into 8 functions for ease of understanding and clarity

The following functions are exclusive for deletion

In class BTreenode:

- 1) remove
- 2) removeFromLeaf
- 3) removeFromNonLeaf
- 4) getPred
- 5) getSucc
- 6) borrowFromPrev
- 7) borrowFromNext
- 8) merge
- 9) findKey

In class BTree:

- 1) remove

The removal of a key from a B-Tree is a fairly complicated process. The program handles all the 6 different cases that might arise while removing a key.

Testing: The code has been tested using the B-Tree provided in the CLRS book(included in the main function) along with other cases.

Reference: CLRS3 - Chapter 18 - (499-502)

It is advised to read the material in CLRS before taking a look at the code. */

```
#include<iostream>
using namespace std;

// A BTREE node
class BTREENode
{
    int *keys; // An array of keys
    int t;      // Minimum degree (defines the range for number of keys)
    BTREENode **C; // An array of child pointers
    int n;      // Current number of keys
    bool leaf; // Is true when node is leaf. Otherwise false

public:
    BTREENode(int _t, bool _leaf); // Constructor

    // A function to traverse all nodes in a subtree rooted with this node
    void traverse();

    // A function to search a key in subtree rooted with this node.
    BTREENode *search(int k); // returns NULL if k is not present.

    // A function that returns the index of the first key that is greater
    // or equal to k
    int findKey(int k);

    // A utility function to insert a new key in the subtree rooted with
    // this node. The assumption is, the node must be non-full when this
    // function is called
    void insertNonFull(int k);

    // A utility function to split the child y of this node. i is index
    // of y in child array C[]. The Child y must be full when this
    // function is called
    void splitChild(int i, BTREENode *y);

    // A wrapper function to remove the key k in subtree rooted with
    // this node.
```

```

void remove(int k);

// A function to remove the key present in idx-th position in
// this node which is a leaf
void removeFromLeaf(int idx);

// A function to remove the key present in idx-th position in
// this node which is a non-leaf node
void removeFromNonLeaf(int idx);

// A function to get the predecessor of the key- where the key
// is present in the idx-th position in the node
int getPred(int idx);

// A function to get the successor of the key- where the key
// is present in the idx-th position in the node
int getSucc(int idx);

// A function to fill up the child node present in the idx-th
// position in the C[] array if that child has less than t-1 keys
void fill(int idx);

// A function to borrow a key from the C[idx-1]-th node and place
// it in C[idx]th node
void borrowFromPrev(int idx);

// A function to borrow a key from the C[idx+1]-th node and place it
// in C[idx]th node
void borrowFromNext(int idx);

// A function to merge idx-th child of the node with (idx+1)th child of
// the node
void merge(int idx);

// Make BTee friend of this so that we can access private members of
// this class in BTee functions
friend class BTee;
};

class BTee
{
    BTeeNode *root; // Pointer to root node
    int t; // Minimum degree
public:

    // Constructor (Initializes tree as empty)
    BTee(int _t)
    {

```

```

        root = NULL;
        t = _t;
    }

    void traverse()
    {
        if (root != NULL) root->traverse();
    }

    // function to search a key in this tree
    BTreenode* search(int k)
    {
        return (root == NULL)? NULL : root->search(k);
    }

    // The main function that inserts a new key in this B-Tree
    void insert(int k);

    // The main function that removes a new key in thie B-Tree
    void remove(int k);

};

BTreenode::BTreenode(int t1, bool leaf1)
{
    // Copy the given minimum degree and leaf property
    t = t1;
    leaf = leaf1;

    // Allocate memory for maximum number of possible keys
    // and child pointers
    keys = new int[2*t-1];
    C = new BTreenode *[2*t];

    // Initialize the number of keys as 0
    n = 0;
}

// A utility function that returns the index of the first key that is
// greater than or equal to k
int BTreenode::findKey(int k)
{
    int idx=0;
    while (idx<n && keys[idx] < k)
        ++idx;
    return idx;
}

```

```

// A function to remove the key k from the sub-tree rooted with this node
void BTeeNode::remove(int k)
{
    int idx = findKey(k);

    // The key to be removed is present in this node
    if (idx < n && keys[idx] == k)
    {

        // If the node is a leaf node - removeFromLeaf is called
        // Otherwise, removeFromNonLeaf function is called
        if (leaf)
            removeFromLeaf(idx);
        else
            removeFromNonLeaf(idx);
    }
    else
    {

        // If this node is a leaf node, then the key is not present in tree
        if (leaf)
        {
            cout << "The key "<< k << " is does not exist in the tree\n";
            return;
        }

        // The key to be removed is present in the sub-tree rooted with this node
        // The flag indicates whether the key is present in the sub-tree rooted
        // with the last child of this node
        bool flag = ( (idx==n)? true : false );

        // If the child where the key is supposed to exist has less than t keys,
        // we fill that child
        if (C[idx]->n < t)
            fill(idx);

        // If the last child has been merged, it must have merged with the previous
        // child and so we recurse on the (idx-1)th child. Else, we recurse on the
        // (idx)th child which now has atleast t keys
        if (flag && idx > n)
            C[idx-1]->remove(k);
        else
            C[idx]->remove(k);
    }
    return;
}

// A function to remove the idx-th key from this node - which is a leaf node

```

```

void BTreenode::removeFromLeaf (int idx)
{
    // Move all the keys after the idx-th pos one place backward
    for (int i=idx+1; i<n; ++i)
        keys[i-1] = keys[i];

    // Reduce the count of keys
    n--;

    return;
}

// A function to remove the idx-th key from this node - which is a non-leaf node
void BTreenode::removeFromNonLeaf(int idx)
{
    int k = keys[idx];

    // If the child that precedes k (C[idx]) has atleast t keys,
    // find the predecessor 'pred' of k in the subtree rooted at
    // C[idx]. Replace k by pred. Recursively delete pred
    // in C[idx]
    if (C[idx]->n >= t)
    {
        int pred = getPred(idx);
        keys[idx] = pred;
        C[idx]->remove(pred);
    }

    // If the child C[idx] has less than t keys, examine C[idx+1].
    // If C[idx+1] has atleast t keys, find the successor 'succ' of k in
    // the subtree rooted at C[idx+1]
    // Replace k by succ
    // Recursively delete succ in C[idx+1]
    else if (C[idx+1]->n >= t)
    {
        int succ = getSucc(idx);
        keys[idx] = succ;
        C[idx+1]->remove(succ);
    }

    // If both C[idx] and C[idx+1] has less than t keys, merge k and all of C[idx+1]
    // into C[idx]
    // Now C[idx] contains 2t-1 keys
    // Free C[idx+1] and recursively delete k from C[idx]
    else
    {

```

```

        merge(idx);
        C[idx]->remove(k);
    }
    return;
}

// A function to get predecessor of keys[idx]
int BTreenode::getPred(int idx)
{
    // Keep moving to the right most node until we reach a leaf
    BTreenode *cur=C[idx];
    while (!cur->leaf)
        cur = cur->C[cur->n];

    // Return the last key of the leaf
    return cur->keys[cur->n-1];
}

int BTreenode::getSucc(int idx)
{
    // Keep moving the left most node starting from C[idx+1] until we reach a leaf
    BTreenode *cur = C[idx+1];
    while (!cur->leaf)
        cur = cur->C[0];

    // Return the first key of the leaf
    return cur->keys[0];
}

// A function to fill child C[idx] which has less than t-1 keys
void BTreenode::fill(int idx)
{
    // If the previous child(C[idx-1]) has more than t-1 keys, borrow a key
    // from that child
    if (idx!=0 && C[idx-1]->n>=t)
        borrowFromPrev(idx);

    // If the next child(C[idx+1]) has more than t-1 keys, borrow a key
    // from that child
    else if (idx!=n && C[idx+1]->n>=t)
        borrowFromNext(idx);

    // Merge C[idx] with its sibling
    // If C[idx] is the last child, merge it with with its previous sibling
    // Otherwise merge it with its next sibling
    else

```

```

{
    if (idx != n)
        merge(idx);
    else
        merge(idx-1);
}
return;
}

// A function to borrow a key from C[idx-1] and insert it
// into C[idx]
void BTreenode::borrowFromPrev(int idx)
{
    BTreenode *child=C[idx];
    BTreenode *sibling=C[idx-1];

    // The last key from C[idx-1] goes up to the parent and key[idx-1]
    // from parent is inserted as the first key in C[idx]. Thus, the loses
    // sibling one key and child gains one key

    // Moving all key in C[idx] one step ahead
    for (int i=child->n-1; i>=0; --i)
        child->keys[i+1] = child->keys[i];

    // If C[idx] is not a leaf, move all its child pointers one step ahead
    if (!child->leaf)
    {
        for(int i=child->n; i>=0; --i)
            child->C[i+1] = child->C[i];
    }

    // Setting child's first key equal to keys[idx-1] from the current node
    child->keys[0] = keys[idx-1];

    // Moving sibling's last child as C[idx]'s first child
    if(!child->leaf)
        child->C[0] = sibling->C[sibling->n];

    // Moving the key from the sibling to the parent
    // This reduces the number of keys in the sibling
    keys[idx-1] = sibling->keys[sibling->n-1];

    child->n += 1;
    sibling->n -= 1;

    return;
}

```

```

// A function to borrow a key from the C[idx+1] and place
// it in C[idx]
void BTreenode::borrowFromNext(int idx)
{
    BTreenode *child=C[idx];
    BTreenode *sibling=C[idx+1];

    // keys[idx] is inserted as the last key in C[idx]
    child->keys[(child->n)] = keys[idx];

    // Sibling's first child is inserted as the last child
    // into C[idx]
    if (!(child->leaf))
        child->C[(child->n)+1] = sibling->C[0];

    //The first key from sibling is inserted into keys[idx]
    keys[idx] = sibling->keys[0];

    // Moving all keys in sibling one step behind
    for (int i=1; i<sibling->n; ++i)
        sibling->keys[i-1] = sibling->keys[i];

    // Moving the child pointers one step behind
    if (!sibling->leaf)
    {
        for(int i=1; i<=sibling->n; ++i)
            sibling->C[i-1] = sibling->C[i];
    }

    // Increasing and decreasing the key count of C[idx] and C[idx+1]
    // respectively
    child->n += 1;
    sibling->n -= 1;

    return;
}

// A function to merge C[idx] with C[idx+1]
// C[idx+1] is freed after merging
void BTreenode::merge(int idx)
{
    BTreenode *child = C[idx];
    BTreenode *sibling = C[idx+1];

    // Pulling a key from the current node and inserting it into (t-1)th
    // position of C[idx]

```

```

child->keys[t-1] = keys[idx];

// Copying the keys from C[idx+1] to C[idx] at the end
for (int i=0; i<sibling->n; ++i)
    child->keys[i+t] = sibling->keys[i];

// Copying the child pointers from C[idx+1] to C[idx]
if (!child->leaf)
{
    for(int i=0; i<=sibling->n; ++i)
        child->C[i+t] = sibling->C[i];
}

// Moving all keys after idx in the current node one step before -
// to fill the gap created by moving keys[idx] to C[idx]
for (int i=idx+1; i<n; ++i)
    keys[i-1] = keys[i];

// Moving the child pointers after (idx+1) in the current node one
// step before
for (int i=idx+2; i<=n; ++i)
    C[i-1] = C[i];

// Updating the key count of child and the current node
child->n += sibling->n+1;
n--;

// Freeing the memory occupied by sibling
delete(sibling);
return;
}

// The main function that inserts a new key in this B-Tree
void BTee::insert(int k)
{
    // If tree is empty
    if (root == NULL)
    {
        // Allocate memory for root
        root = new BTeeNode(t, true);
        root->keys[0] = k; // Insert key
        root->n = 1; // Update number of keys in root
    }
    else // If tree is not empty
    {
        // If root is full, then tree grows in height
        if (root->n == 2*t-1)
        {

```

```

// Allocate memory for new root
BTreeNode *s = new BTreeNode(t, false);

// Make old root as child of new root
s->C[0] = root;

// Split the old root and move 1 key to the new root
s->splitChild(0, root);

// New root has two children now. Decide which of the
// two children is going to have new key
int i = 0;
if (s->keys[0] < k)
    i++;
s->C[i]->insertNonFull(k);

// Change root
root = s;
}
else // If root is not full, call insertNonFull for root
    root->insertNonFull(k);
}

// A utility function to insert a new key in this node
// The assumption is, the node must be non-full when this
// function is called
void BTreeNode::insertNonFull(int k)
{
    // Initialize index as index of rightmost element
    int i = n-1;

    // If this is a leaf node
    if (leaf == true)
    {
        // The following loop does two things
        // a) Finds the location of new key to be inserted
        // b) Moves all greater keys to one place ahead
        while (i >= 0 && keys[i] > k)
        {
            keys[i+1] = keys[i];
            i--;
        }

        // Insert the new key at found location
        keys[i+1] = k;
        n = n+1;
    }
}

```

```

else // If this node is not leaf
{
    // Find the child which is going to have the new key
    while (i >= 0 && keys[i] > k)
        i--;

    // See if the found child is full
    if (C[i+1]->n == 2*t-1)
    {
        // If the child is full, then split it
        splitChild(i+1, C[i+1]);

        // After split, the middle key of C[i] goes up and
        // C[i] is splitted into two. See which of the two
        // is going to have the new key
        if (keys[i+1] < k)
            i++;
    }
    C[i+1]->insertNonFull(k);
}

// A utility function to split the child y of this node
// Note that y must be full when this function is called
void BTreenode::splitChild(int i, BTreenode *y)
{
    // Create a new node which is going to store (t-1) keys
    // of y
    BTreenode *z = new BTreenode(y->t, y->leaf);
    z->n = t - 1;

    // Copy the last (t-1) keys of y to z
    for (int j = 0; j < t-1; j++)
        z->keys[j] = y->keys[j+t];

    // Copy the last t children of y to z
    if (y->leaf == false)
    {
        for (int j = 0; j < t; j++)
            z->C[j] = y->C[j+t];
    }

    // Reduce the number of keys in y
    y->n = t - 1;

    // Since this node is going to have a new child,
    // create space of new child
    for (int j = n; j >= i+1; j--)
}

```

```

C[j+1] = C[j];

// Link the new child to this node
C[i+1] = z;

// A key of y will move to this node. Find location of
// new key and move all greater keys one space ahead
for (int j = n-1; j >= i; j--)
    keys[j+1] = keys[j];

// Copy the middle key of y to this node
keys[i] = y->keys[t-1];

// Increment count of keys in this node
n = n + 1;
}

// Function to traverse all nodes in a subtree rooted with this node
void BTreenode::traverse()
{
    // There are n keys and n+1 children, travers through n keys
    // and first n children
    int i;
    for (i = 0; i < n; i++)
    {
        // If this is not leaf, then before printing key[i],
        // traverse the subtree rooted with child C[i].
        if (leaf == false)
            C[i]->traverse();
        cout << " " << keys[i];
    }

    // Print the subtree rooted with last child
    if (leaf == false)
        C[i]->traverse();
}

// Function to search key k in subtree rooted with this node
BTreenode *BTreenode::search(int k)
{
    // Find the first key greater than or equal to k
    int i = 0;
    while (i < n && k > keys[i])
        i++;

    // If the found key is equal to k, return this node
    if (keys[i] == k)
        return this;
}

```

```
// If key is not found here and this is a leaf node
if (leaf == true)
    return NULL;

// Go to the appropriate child
return C[i]->search(k);
}

void BTREE::remove(int k)
{
    if (!root)
    {
        cout << "The tree is empty\n";
        return;
    }

    // Call the remove function for root
    root->remove(k);

    // If the root node has 0 keys, make its first child as the new root
    // if it has a child, otherwise set root as NULL
    if (root->n==0)
    {
        BTREENode *tmp = root;
        if (root->leaf)
            root = NULL;
        else
            root = root->C[0];

        // Free the old root
        delete tmp;
    }
    return;
}

// Driver program to test above functions
int main()
{
    BTREE t(3); // A B-Tree with minium degree 3

    t.insert(1);
    t.insert(3);
    t.insert(7);
    t.insert(10);
    t.insert(11);
    t.insert(13);
    t.insert(14);
```

```
t.insert(15);
t.insert(18);
t.insert(16);
t.insert(19);
t.insert(24);
t.insert(25);
t.insert(26);
t.insert(21);
t.insert(4);
t.insert(5);
t.insert(20);
t.insert(22);
t.insert(2);
t.insert(17);
t.insert(12);
t.insert(6);

cout << "Traversal of tree constructed is\n";
t.traverse();
cout << endl;

t.remove(6);
cout << "Traversal of tree after removing 6\n";
t.traverse();
cout << endl;

t.remove(13);
cout << "Traversal of tree after removing 13\n";
t.traverse();
cout << endl;

t.remove(7);
cout << "Traversal of tree after removing 7\n";
t.traverse();
cout << endl;

t.remove(4);
cout << "Traversal of tree after removing 4\n";
t.traverse();
cout << endl;

t.remove(2);
cout << "Traversal of tree after removing 2\n";
t.traverse();
cout << endl;

t.remove(16);
cout << "Traversal of tree after removing 16\n";
```

```
t.traverse();
cout << endl;

return 0;
}
```

Output:

```
Traversal of tree constructed is
1 2 3 4 5 6 7 10 11 12 13 14 15 16 17 18 19 20 21 22 24 25 26
Traversal of tree after removing 6
1 2 3 4 5 7 10 11 12 13 14 15 16 17 18 19 20 21 22 24 25 26
Traversal of tree after removing 13
1 2 3 4 5 7 10 11 12 14 15 16 17 18 19 20 21 22 24 25 26
Traversal of tree after removing 7
1 2 3 4 5 10 11 12 14 15 16 17 18 19 20 21 22 24 25 26
Traversal of tree after removing 4
1 2 3 5 10 11 12 14 15 16 17 18 19 20 21 22 24 25 26
Traversal of tree after removing 2
1 3 5 10 11 12 14 15 16 17 18 19 20 21 22 24 25 26
Traversal of tree after removing 16
1 3 5 10 11 12 14 15 17 18 19 20 21 22 24 25 26
```

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/b-tree-set-3delete/>

Chapter 11

BK-Tree Introduction & Implementation

BK-Tree Introduction & Implementation - GeeksforGeeks

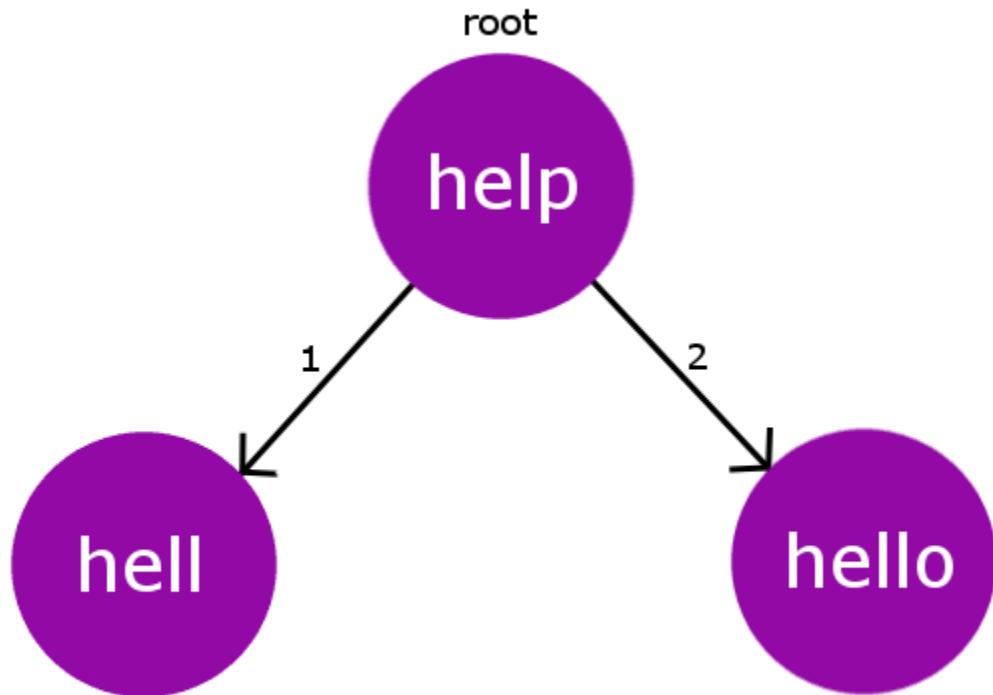
BK Tree or Burkhard Keller Tree is a data structure that is used to perform spell check based on Edit Distance (Levenshtein distance) concept. BK trees are also used for approximate string matching. Various auto correct feature in many softwares can be implemented based on this data structure.

Pre-requisites : Edit distance Problem
Metric tree

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

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

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

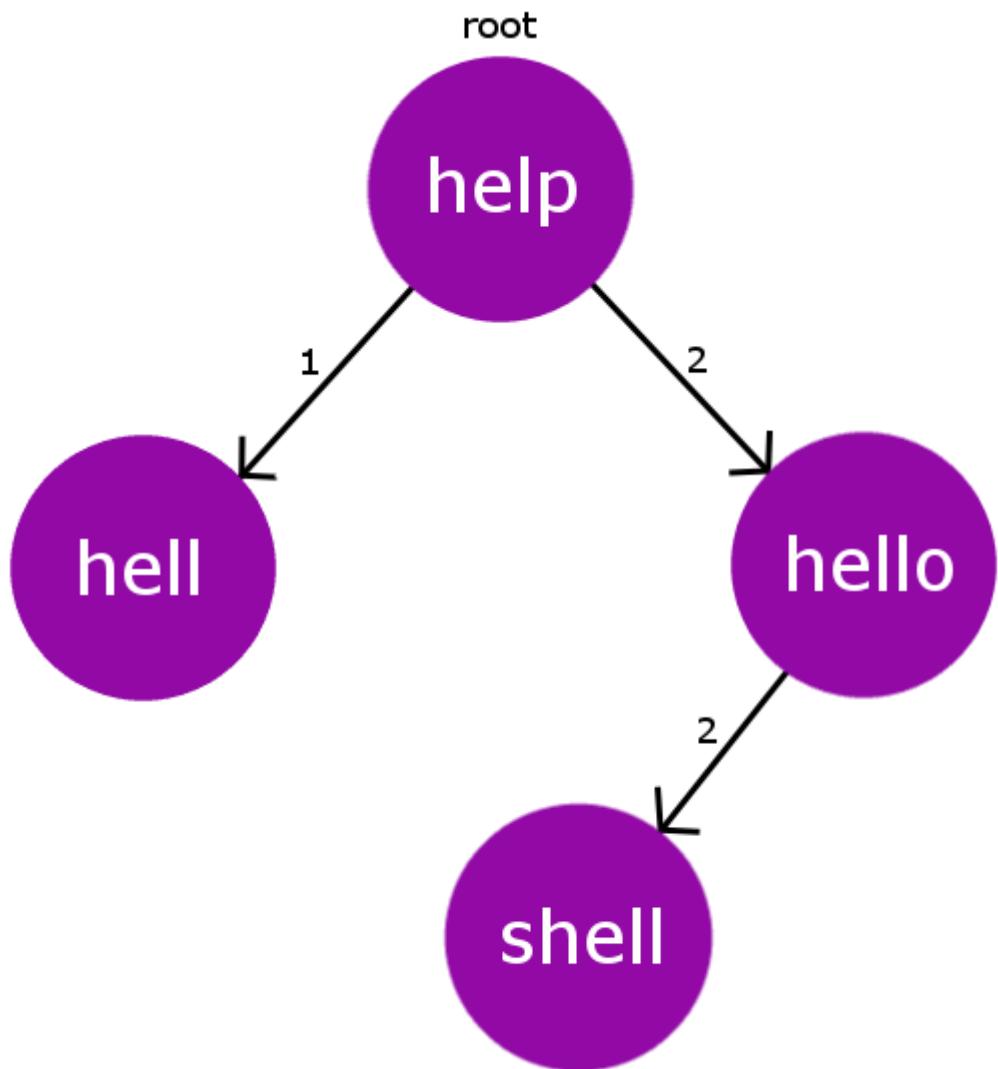


Every node in the BK Tree will have exactly one child with same edit-distance. In case, if we encounter some collision for edit-distance while inserting, we will then propagate the insertion process down the children until we find an appropriate parent for the string node.

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

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

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

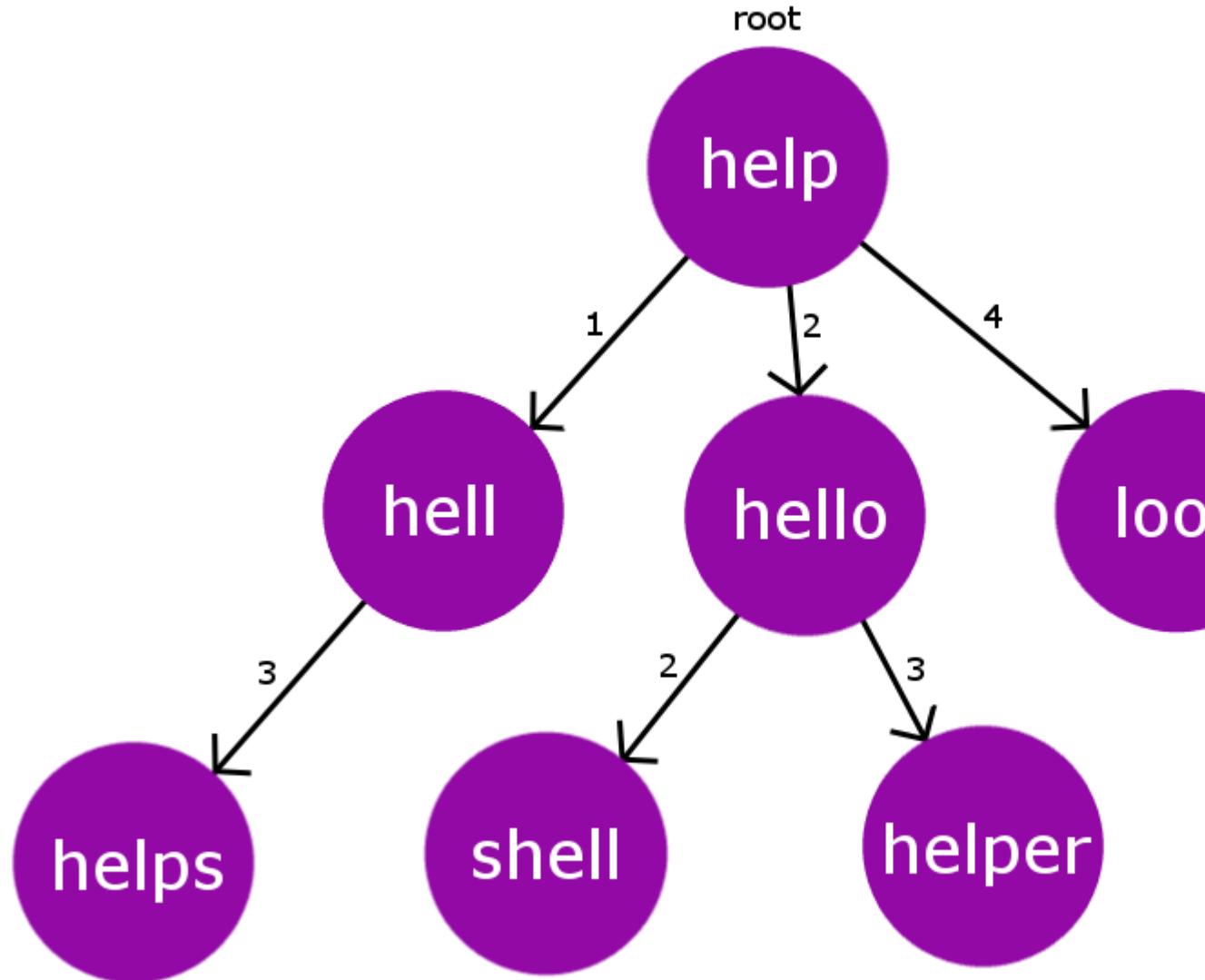


So, till now we have understood how we will build our BK Tree. Now, the question arises that how to find the closest correct word for our misspelled word? First of all, we need to set a tolerance value. This **tolerance value** is simply the maximum edit distance from our misspelled word to the correct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

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

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

Consider the below constructed BK Tree.



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

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

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

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

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

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

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

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

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

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

// defines the tolerance value
#define TOL 2

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

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

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

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

```

};

// stores the root Node
Node RT;

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

// index for current Node of tree
int ptr;

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

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

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

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

```

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

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

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

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

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

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

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

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

```

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

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

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

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

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

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

    return 0;
}

```

Output:

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

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

References

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

Source

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

Chapter 12

Barabasi Albert Graph (for Scale Free Models)

Barabasi Albert Graph (for Scale Free Models) - GeeksforGeeks

The current article would deal with the concepts surrounding the complex networks using the python library Networkx. It is a Python language software package for the creation, manipulation, and study of the structure, dynamics, and function of complex networks. With NetworkX you can load and store networks in standard and nonstandard data formats, generate many types of random and classic networks, analyze network structure, build network models, design new network algorithms, draw networks, and much more.

The current article would deal with the algorithm for generating random scale free networks for using preferential attachment model. The reason of interest behind this model dates back to the 1990s when Albert Lazlo Barabasi and Reka Albert came out with the path breaking research describing the model followed by the scale free networks around the world. They suggested that several natural and human-made systems, including the Internet, the World Wide Web, citation networks, and some social networks are thought to be approximately scale-free networks.

A scale-free network is a network whose degree distribution follows a power law, at least asymptotically. That is, the fraction $P(k)$ of nodes in the network having k connections to other nodes goes for large values of k as

$$P(k) \propto k^{-\gamma}$$

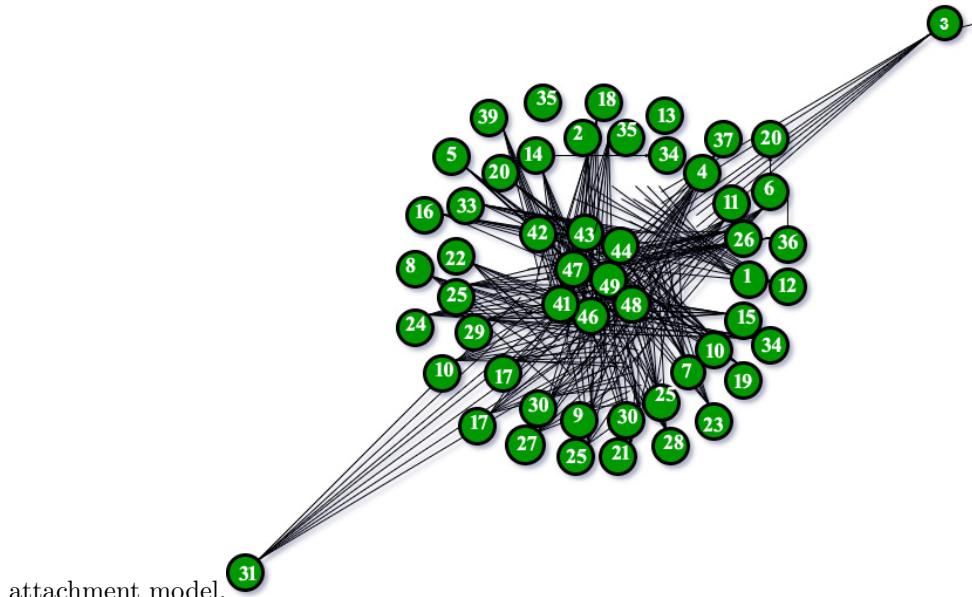
Where γ is a parameter whose value is typically in the range $2 < \gamma < 3$, although occasionally it may lie outside these bounds and c is a proportionality constant.

The Barabási-Albert model is one of several proposed models that generate scale-free networks. It incorporates two important general concepts: growth and preferential attachment. Both growth and preferential attachment exist widely in real networks. Growth means that the number of nodes in the network increases over time.

Preferential attachment means that the more connected a node is, the more likely it is to receive new links. Nodes with higher degree have stronger ability to grab links added to the

network. Intuitively, the preferential attachment can be understood if we think in terms of social networks connecting people. Here a link from A to B means that person A "knows" or "is acquainted with" person B. Heavily linked nodes represent well-known people with lots of relations. When a newcomer enters the community, s/he is more likely to become acquainted with one of those more visible people rather than with a relative unknown. The BA model was proposed by assuming that in the World Wide Web, new pages link preferentially to hubs, i.e. very well-known sites such as Google, rather than to pages that hardly anyone knows. If someone selects a new page to link to by randomly choosing an existing link, the probability of selecting a particular page would be proportional to its degree.

Following image will describe the BA Model graph with 50 nodes following the preferential



attachment model.

The above graph completely satisfies the logic of the rich getting richer and the poor getting poorer.

Code:

The following code is a part of the function which we will eventually implement using the networkx library.

```
def barabasi_albert_graph(n, m, seed=None):
    """Returns a random graph according to the Barabási-Albert preferential
    Attachment model.

    A graph of ``n`` nodes is grown by attaching new nodes each with ``m``.
    Edges that are preferentially attached to existing nodes with high degree.

    Parameters
    -----
    n : int
        Number of nodes
```

```

m : int
    Number of edges to attach from a new node to existing nodes
seed : int, optional
    Seed for random number generator (default=None).

Returns
-----
G : Graph

Raises
-----
NetworkXError
    If ``m`` does not satisfy ``1 <= m < n``.

if m < 1 or m >=n:
    raise nx.NetworkXError("Barabási-Albert network must have m >= 1"
                           " and m < n, m = %d, n = %d" % (m, n))
if seed is not None:
    random.seed(seed)

# Add m initial nodes (m0 in barabasi-speak)
G=empty_graph(m)
G.name="barabasi_albert_graph(%s,%s)"%(n,m)
# Target nodes for new edges
targets=list(range(m))
# List of existing nodes, with nodes repeated once for each adjacent edge
repeated_nodes=[]
# Start adding the other n-m nodes. The first node is m.
source=m
while source<n:
    # Add edges to m nodes from the source.
    G.add_edges_from(zip(*m,targets))
    # Add one node to the list for each new edge just created.
    repeated_nodes.extend(targets)
    # And the new node "source" has m edges to add to the list.
    repeated_nodes.extend(*m)
    # Now choose m unique nodes from the existing nodes
    # Pick uniformly from repeated_nodes (preferential attachment)
    targets = _random_subset(repeated_nodes,m)
    source += 1
return G

```

The above code is a part of the networkx library which is used to handle the random graphs efficiently in python. One will have to install it before running the following code.

```

>>> import networkx as nx
>>> G= nx.barabasi_albert_graph(50,40)

```

```
>>> nx.draw(G, with_labels=True)
```

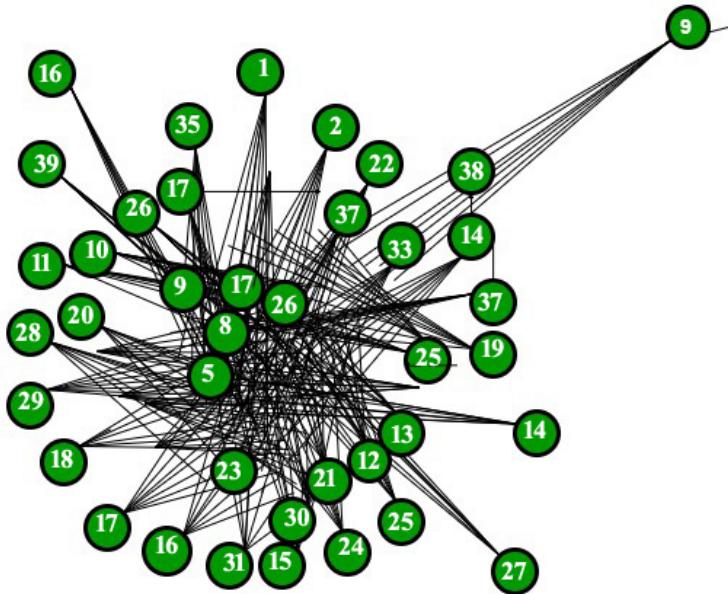
To display the above graph, I used the matplotlib library. We need to install it before the execution of the below codes.

```
>>> import matplotlib.pyplot as plt  
>>> plt.show()
```

So the final code seemed like:

```
>>> import networkx as nx  
>>> import matplotlib.pyplot as plt  
  
>>> G= nx.barabasi_albert_graph(40,15)  
>>> nx.draw(G, with_labels=True)  
>>> plt.show()
```

Output:



BA model for 40 nodes

Thus I would further like to describe more about the networkx library and its modules basically focusing on the centrality measure of a network (especially the scale free models).

References

You can read more about the same at

https://en.wikipedia.org/wiki/Barab%C3%A1si%E2%80%93Albert_model

<http://networkx.readthedocs.io/en/networkx-1.10/index.html>

Source

<https://www.geeksforgeeks.org/barabasi-albert-graph-scale-free-models/>

Chapter 13

Binary Indexed Tree : Range Update and Range Queries

Binary Indexed Tree : Range Update and Range Queries - GeeksforGeeks

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

1. **update(l, r, val)** : Add ‘val’ to all the elements in the array from [l, r].
2. **getRangeSum(l, r)** : Find sum of all elements in array from [l, r].

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

Example:

```
Input : n = 5    // {0, 0, 0, 0, 0}
Queries: update : l = 0, r = 4, val = 2
         update : l = 3, r = 4, val = 3
         getRangeSum : l = 2, r = 4
```

Output: Sum of elements of range [2, 4] is 12

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

In the [previous post](#), we discussed range update and point query solutions using BIT.

rangeUpdate(l, r, val) : We add ‘val’ to element at index ‘l’. We subtract ‘val’ from element at index ‘r+1’.

`getElement(index)` [or `getSum()`]: We return sum of elements from 0 to index which can be quickly obtained using BIT.

We can compute `rangeSum()` using `getSum()` queries.

$$\text{rangeSum}(l, r) = \text{getSum}(r) - \text{getSum}(l-1)$$

A **Simple Solution** is to use solutions discussed in [previous post](#). Range update query is same. Range sum query can be achieved by doing get query for all elements in range.

An **Efficient Solution** is to make sure that both queries can be done in $O(\log n)$ time. We get range sum using prefix sums. How to make sure that update is done in a way so that prefix sum can be done quickly? Consider a situation where prefix sum $[0, k]$ (where $0 \leq k < n$) is needed after range update on range $[l, r]$. Three cases arises as k can possibly lie in 3 regions.

Case 1: $0 < k < l$

The update query won't affect sum query.

Case 2: $l \leq k \leq r$

Consider an example:

```
Add 2 to range [2, 4], the resultant array would be:  
0 0 2 2 2  
If k = 3  
Sum from [0, k] = 4
```

How to get this result?

Simply add the val from l^{th} index to k^{th} index. Sum is incremented by “ $\text{val}^*(k) - \text{val}^*(l-1)$ ” after update query.

Case 3: $k > r$

For this case, we need to add “val” from l^{th} index to r^{th} index. Sum is incremented by “ $\text{val}^*r - \text{val}^*(l-1)$ ” due to update query.

Observations :

Case 1: is simple as sum would remain same as it was before update.

Case 2: Sum was incremented by $\text{val}^*k - \text{val}^*(l-1)$. We can find “val”, it is similar to finding the i^{th} element in [range update and point query article](#). So we maintain one BIT for Range Update and Point Queries, this BIT will be helpful in finding the value at k^{th} index. Now val^*k is computed, how to handle extra term $\text{val}^*(l-1)$?

In order to handle this extra term, we maintain another BIT (BIT2). Update $\text{val}^*(l-1)$ at l^{th} index, so when `getSum` query is performed on BIT2 will give result as $\text{val}^*(l-1)$.

Case 3 : The sum in case 3 was incremented by “ $\text{val}^*r - \text{val}^*(l-1)$ ”, the value of this term can be obtained using BIT2. Instead of adding, we subtract “ $\text{val}^*(l-1) - \text{val}^*r$ ” as we can get this value from BIT2 by adding $\text{val}^*(l-1)$ as we did in case 2 and subtracting val^*r in every update operation.

```
Update Query  
Update(BITree1, l, val)
```

```

Update(BITree1, r+1, -val)
UpdateBIT2(BITree2, 1, val*(l-1))
UpdateBIT2(BITree2, r+1, -val*r)

Range Sum
getSum(BITree1, k) *k) - getSum(BITree2, k)

```

C++ Implementation of above idea

```

// C++ program to demonstrate Range Update
// and Range Queries using BIT
#include <iostream>
using namespace std;

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

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

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

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

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

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

```

```

BITree[index] += val;

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

}

// Returns the sum of array from [0, x]
int sum(int x, int BITTree1[], int BITTree2[])
{
    return (getSum(BITTree1, x) * x) - getSum(BITTree2, x);
}

void updateRange(int BITTree1[], int BITTree2[], int n,
                 int val, int l, int r)
{
    // Update Both the Binary Index Trees
    // As discussed in the article

    // Update BIT1
    updateBIT(BITTree1,n,l,val);
    updateBIT(BITTree1,n,r+1,-val);

    // Update BIT2
    updateBIT(BITTree2,n,l,val*(l-1));
    updateBIT(BITTree2,n,r+1,-val*r);
}

int rangeSum(int l, int r, int BITTree1[], int BITTree2[])
{
    // Find sum from [0,r] then subtract sum
    // from [0,l-1] in order to find sum from
    // [l,r]
    return sum(r, BITTree1, BITTree2) -
           sum(l-1, BITTree1, BITTree2);
}

int *constructBITree(int n)
{
    // Create and initialize BITree[] as 0
    int *BITree = new int[n+1];
    for (int i=1; i<=n; i++)
        BITree[i] = 0;

    return BITree;
}

```

```
// Driver Program to test above function
int main()
{
    int n = 5;

    // Construct two BIT
    int *BITTree1, *BITTree2;

    // BIT1 to get element at any index
    // in the array
    BITTree1 = constructBITree(n);

    // BIT 2 maintains the extra term
    // which needs to be subtracted
    BITTree2 = constructBITree(n);

    // Add 5 to all the elements from [0,4]
    int l = 0 , r = 4 , val = 5;
    updateRange(BITTree1,BITTree2,n,val,l,r);

    // Add 2 to all the elements from [2,4]
    l = 2 , r = 4 , val = 10;
    updateRange(BITTree1,BITTree2,n,val,l,r);

    // Find sum of all the elements from
    // [1,4]
    l = 1 , r = 4;
    cout << "Sum of elements from [" << l
        << "," << r << "] is ";
    cout << rangeSum(l,r,BITTree1,BITTree2) << "\n";

    return 0;
}
```

Output:

```
Sum of elements from [1,4] is 50
```

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

Source

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

Chapter 14

Binary Indexed Tree : Range Updates and Point Queries

Binary Indexed Tree : Range Updates and Point Queries - GeeksforGeeks

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

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

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

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

Example:

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

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

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

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

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

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

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

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

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

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

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

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

C++

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

Output:

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

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

Method 3 (Using Binary Indexed Tree)

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

Below is C++ implementation.

C++

```

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

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

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

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

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

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

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

return BITree;
}

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

```

```

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

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

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

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

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

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

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

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

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

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

```

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

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

return 0;
}
```

Java

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

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

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

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

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

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

```
        }
    }

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

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

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

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

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

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

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

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

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

```

        index -= index & (-index);
    }

    // Return the sum
    return sum;
}

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

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

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

    constructBITree(arr,n);

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

    int index = 4;

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

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

    // Find the element
}

```

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

Output:

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

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

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

Improved By : [p_unit](#)

Source

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

Chapter 15

Binary Indexed Tree or Fenwick Tree

Binary Indexed Tree or Fenwick Tree - GeeksforGeeks

Let us consider the following problem to understand Binary Indexed Tree.

We have an array $\text{arr}[0 \dots n-1]$. We should be able to

1 Find the sum of first i elements.

2 Change value of a specified element of the array $\text{arr}[i] = x$ where $0 \leq i \leq n-1$.

A **simple solution** is to run a loop from 0 to $i-1$ and calculate sum of elements. To update a value, simply do $\text{arr}[i] = x$. The first operation takes $O(n)$ time and second operation takes $O(1)$ time. Another simple solution is to create another array and store sum from start to i at the i 'th index in this array. Sum of a given range can now be calculated in $O(1)$ time, but update operation takes $O(n)$ time now. This works well if the number of query operations are large and very few updates.

Can we perform both the operations in $O(\log n)$ time once given the array?

One Efficient Solution is to use [Segment Tree](#) that does both operations in $O(\log n)$ time.

Using Binary Indexed Tree, we can do both tasks in $O(\log n)$ time. The advantages of Binary Indexed Tree over Segment Tree, requires less space and very easy to implement..

Representation

Binary Indexed Tree is represented as an array. Let the array be $\text{BITree}[]$. Each node of Binary Indexed Tree stores sum of some elements of given array. Size of Binary Indexed Tree is equal to n where n is size of input array. In the below code, we have used size as $n+1$ for ease of implementation.

Construction

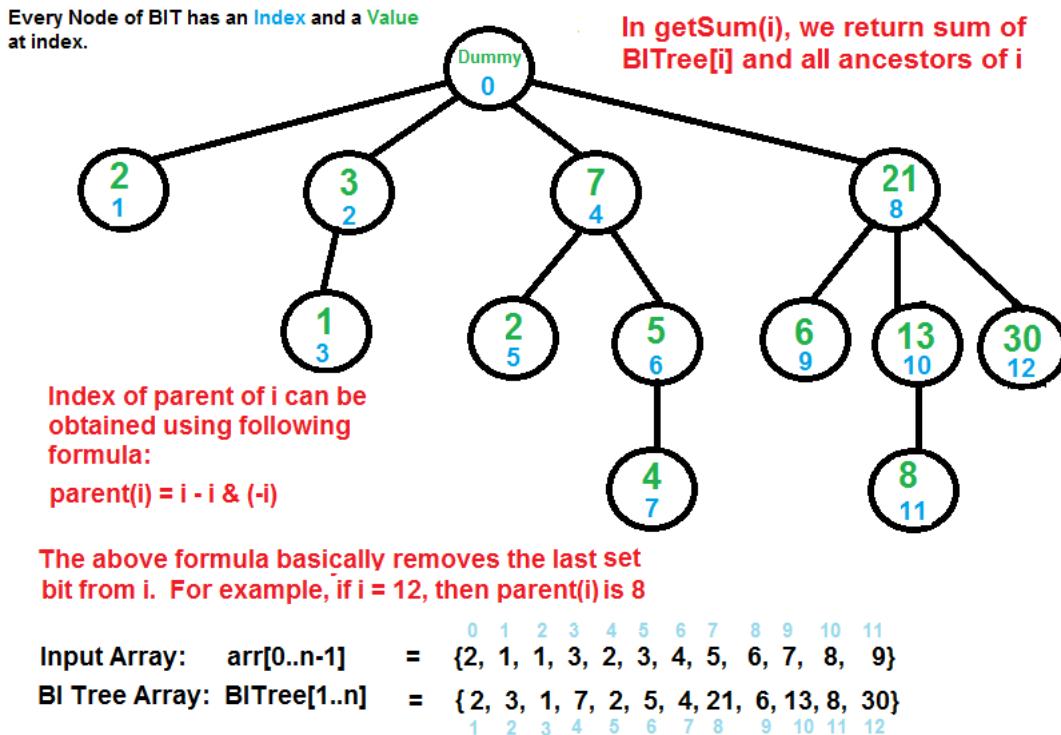
We construct the Binary Indexed Tree by first initializing all values in $\text{BITree}[]$ as 0. Then we call `update()` operation for all indexes to store actual sums, `update` is discussed below.

Operations

```

getSum(index): Returns sum of arr[0..index]
// Returns sum of arr[0..index] using BITree[0..n]. It assumes that
// BITree[] is constructed for given array arr[0..n-1]
1) Initialize sum as 0 and index as index+1.
2) Do following while index is greater than 0.
...a) Add BITree[index] to sum
...b) Go to parent of BITree[index]. Parent can be obtained by removing
    the last set bit from index, i.e., index = index - (index & (-index))
3) Return sum.

```



View of Binary Indexed Tree to understand getSum() operation

The above diagram demonstrates working of getSum(). Following are some important observations.

Node at index 0 is a dummy node.

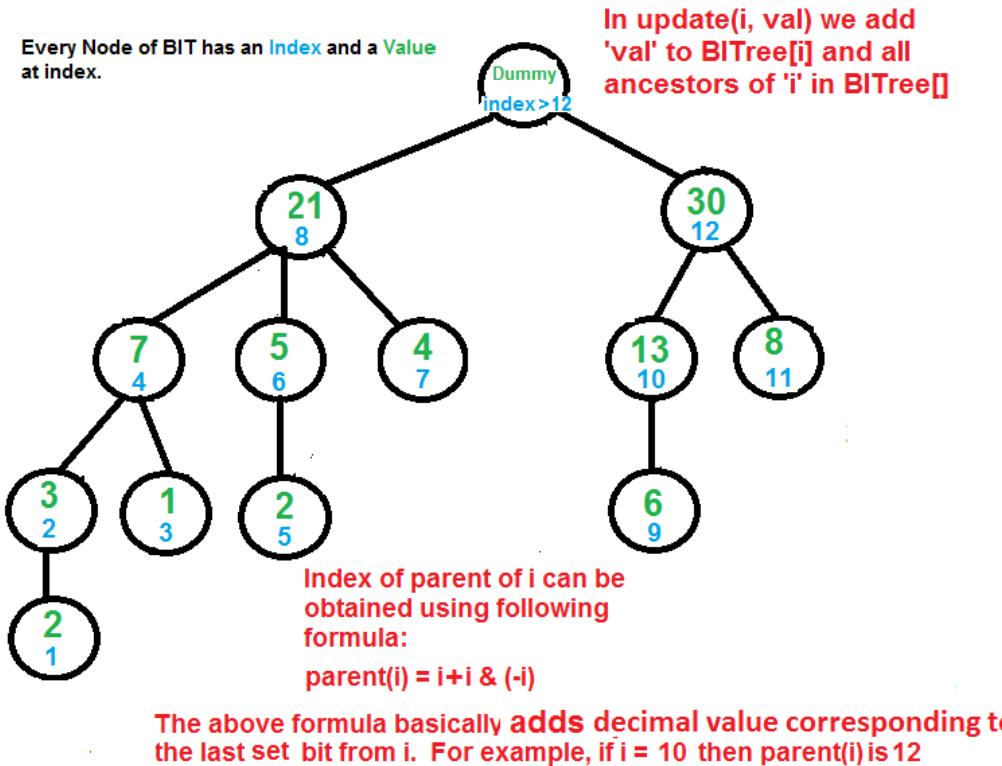
A node at index y is parent of a node at index x, iff y can be obtained by removing last set bit from binary representation of x.

A child x of a node y stores sum of elements from y(inclusive y) and of x(exclusive x).

```

update(index, val): Updates BIT for operation arr[index] += val
// Note that arr[] is not changed here. It changes
// only BI Tree for the already made change in arr[].
1) Initialize index as index+1.
2) Do following while index is smaller than or equal to n.
...a) Add value to BITree[index]
...b) Go to parent of BITree[index]. Parent can be obtained by removing
    the last set bit from index, i.e., index = index + (index & (-index))

```



Contents of arr[] and BITree[] are same as above diagram for getSum()

View of Binary Indexed Tree to understand update() operation

The update process needs to make sure that all BITree nodes that have arr[i] as part of the section they cover must be updated. We get all such nodes of BITree by repeatedly adding the decimal number corresponding to the last set bit.

How does Binary Indexed Tree work?

The idea is based on the fact that all positive integers can be represented as sum of powers of 2. For example 19 can be represented as $16 + 2 + 1$. Every node of BI Tree stores

sum of n elements where n is a power of 2. For example, in the above first diagram for getSum(), sum of first 12 elements can be obtained by sum of last 4 elements (from 9 to 12) plus sum of 8 elements (from 1 to 8). The number of set bits in binary representation of a number n is O(Logn). Therefore, we traverse at-most O(Logn) nodes in both getSum() and update() operations. Time complexity of construction is O(nLogn) as it calls update() for all n elements.

Implementation:

Following are the implementations of Binary Indexed Tree.

C++

```
// C++ code to demonstrate operations of Binary Index Tree
#include <iostream>

using namespace std;

/*      n --> No. of elements present in input array.
BITree[0..n] --> Array that represents Binary Indexed Tree.
arr[0..n-1] --> Input array for which prefix sum is evaluated. */

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

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

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

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

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

```
index = index + 1;

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

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

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

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

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

return BITree;
}

// Driver program to test above functions
int main()
{
    int freq[] = {2, 1, 1, 3, 2, 3, 4, 5, 6, 7, 8, 9};
    int n = sizeof(freq)/sizeof(freq[0]);
    int *BITree = constructBITree(freq, n);
    cout << "Sum of elements in arr[0..5] is "
        << getSum(BITree, 5);

    // Let us test the update operation
    freq[3] += 6;
    updateBIT(BITree, n, 3, 6); //Update BIT for above change in arr[]

    cout << "\nSum of elements in arr[0..5] after update is "
```

```
    << getSum(BITree, 5);

    return 0;
}
```

Java

```
// Java program to demonstrate lazy
// propagation in segment tree
import java.util.*;
import java.lang.*;
import java.io.*;

class BinaryIndexedTree
{
    // Max tree size
    final static int MAX = 1000;

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

    /* n --> No. of elements present in input array.
    BITree[0..n] --> Array that represents Binary
    Indexed Tree.
    arr[0..n-1] --> Input array for which prefix sum
    is evaluated. */

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

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

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

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

```
        }
        return sum;
    }

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

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

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

/* Function to construct fenwick tree
from given array.*/
void constructBITree(int arr[], int n)
{
    // Initialize BITree[] as 0
    for(int i=1; i<=n; i++)
        BITree[i] = 0;

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

// Main function
public static void main(String args[])
{
    int freq[] = {2, 1, 1, 3, 2, 3,
                 4, 5, 6, 7, 8, 9};
    int n = freq.length;
    BinaryIndexedTree tree = new BinaryIndexedTree();
```

```
// Build fenwick tree from given array
tree.constructBITree(freq, n);

System.out.println("Sum of elements in arr[0..5]"+
                   " is "+ tree.getSum(5));

// Let us test the update operation
freq[3] += 6;

// Update BIT for above change in arr[]
updateBIT(n, 3, 6);

// Find sum after the value is updated
System.out.println("Sum of elements in arr[0..5]"+
                   " after update is " + tree.getSum(5));
}

}

// This code is contributed by Ranjan Binwani
```

Python

```
# Python implementation of Binary Indexed Tree

# Returns sum of arr[0..index]. This function assumes
# that the array is preprocessed and partial sums of
# array elements are stored in BITree[].
def getsum(BITTree,i):
    s = 0 #initialize result

    # index in BITree[] is 1 more than the index in arr[]
    i = i+1

    # Traverse ancestors of BITree[index]
    while i > 0:

        # Add current element of BITree to sum
        s += BITTree[i]

        # Move index to parent node in getSum View
        i -= i & (-i)

    return s

# Updates a node in Binary Index Tree (BITree) at given index
# in BITree. The given value 'val' is added to BITree[i] and
# all of its ancestors in tree.
def updatebit(BITTree , n , i ,v):
```

```
# index in BITree[] is 1 more than the index in arr[]
i += 1

# Traverse all ancestors and add 'val'
while i <= n:

    # Add 'val' to current node of BI Tree
    BITTree[i] += v

    # Update index to that of parent in update View
    i += i & (-i)

# Constructs and returns a Binary Indexed Tree for given
# array of size n.
def construct(arr, n):

    # Create and initialize BITree[] as 0
    BITTree = [0]*(n+1)

    # Store the actual values in BITree[] using update()
    for i in range(n):
        updatebit(BITTree, n, i, arr[i])

    # Uncomment below lines to see contents of BITree[]
    #for i in range(1,n+1):
    #    print BITTree[i],
    return BITTree

# Driver code to test above methods
freq = [2, 1, 1, 3, 2, 3, 4, 5, 6, 7, 8, 9]
BITTree = construct(freq,len(freq))
print("Sum of elements in arr[0..5] is " + str(getsum(BITTree,5)))
freq[3] += 6
updatebit(BITTree, len(freq), 3, 6)
print("Sum of elements in arr[0..5] "+
      " after update is " + str(getsum(BITTree,5)))

# This code is contributed by Raju Varshney
```

Output:

```
Sum of elements in arr[0..5] is 12
Sum of elements in arr[0..5] after update is 18
```

Can we extend the Binary Indexed Tree for range Sum in Logn time?

This is simple to answer. The rangeSum(l, r) can be obtained as getSum(r) – getSum(l-1).

Applications:

Used to implement the arithmetic coding algorithm. Development of operations it supports were primarily motivated by use in that case. See [this](#) for more details.

Example Problems:

[Count inversions in an array Set 3 \(Using BIT\)](#)

[Two Dimensional Binary Indexed Tree or Fenwick Tree](#)

[Counting Triangles in a Rectangular space using BIT](#)

References:

http://en.wikipedia.org/wiki/Fenwick_tree

<http://community.topcoder.com/tc?module=Static&d1=tutorials&d2=binaryIndexedTrees>

Improved By : [Sai Krishna Chowrigari, BRAGHUNATHAN](#)

Source

<https://www.geeksforgeeks.org/binary-indexed-tree-or-fenwick-tree-2/>

Chapter 16

Binomial Heap

Binomial Heap - GeeksforGeeks

The main application of [Binary Heaps](#) is to implement priority queue. Binomial Heap is an extension of [Binary Heap](#) that provides faster union or merge operation together with other operations provided by Binary Heap.

A Binomial Heap is a collection of Binomial Trees

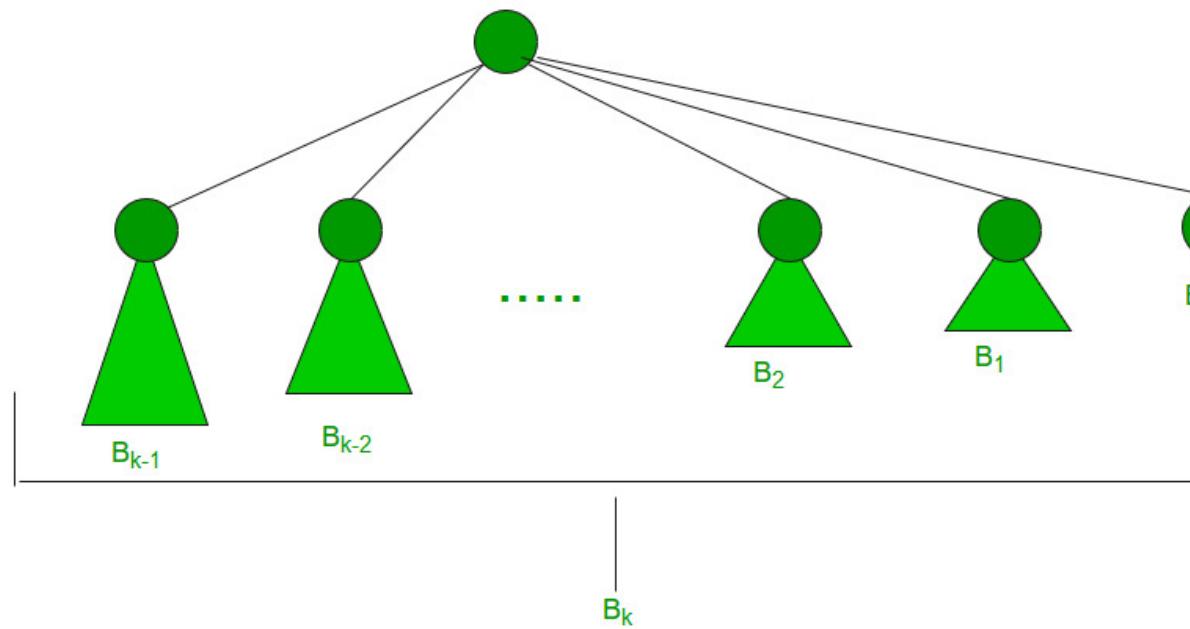
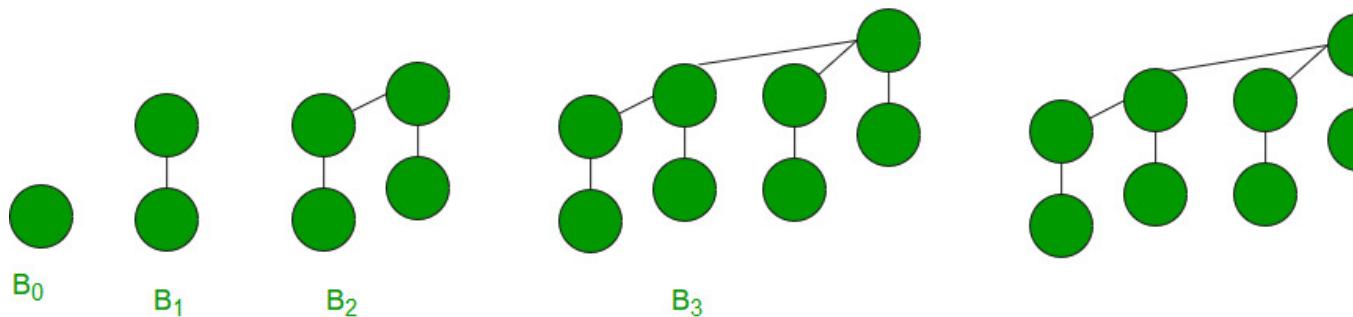
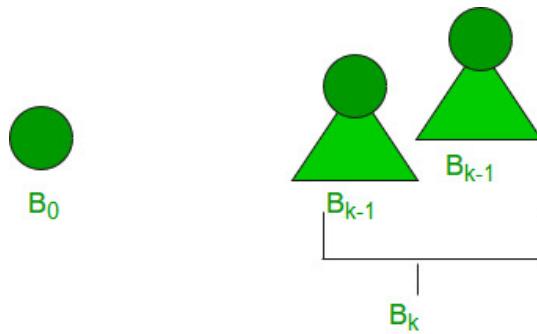
What is a Binomial Tree?

A Binomial Tree of order 0 has 1 node. A Binomial Tree of order k can be constructed by taking two binomial trees of order k-1 and making one as leftmost child or other.

A Binomial Tree of order k has following properties.

- a) It has exactly 2^k nodes.
- b) It has depth as k.
- c) There are exactly ${}^k C_i$ nodes at depth i for $i = 0, 1, \dots, k$.
- d) The root has degree k and children of root are themselves Binomial Trees with order k-1, k-2,.. 0 from left to right.

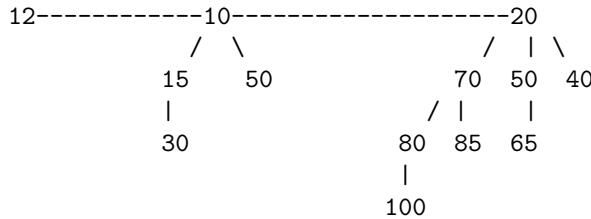
The following diagram is referred from 2nd Edition of [CLRS book](#).



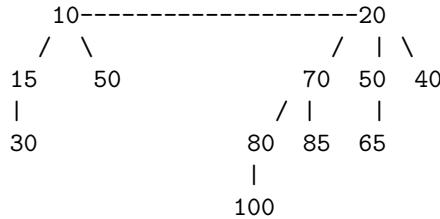
Binomial Heap:

A Binomial Heap is a set of Binomial Trees where each Binomial Tree follows Min Heap property. And there can be at most one Binomial Tree of any degree.

Examples Binomial Heap:



A Binomial Heap with 13 nodes. It is a collection of 3 Binomial Trees of orders 0, 2 and 3 from left to right.



A Binomial Heap with 12 nodes. It is a collection of 2 Binomial Trees of orders 2 and 3 from left to right.

Binary Representation of a number and Binomial Heaps

A Binomial Heap with n nodes has the number of Binomial Trees equal to the number of set bits in the Binary representation of n . For example let n be 13, there 3 set bits in the binary representation of n (00001101), hence 3 Binomial Trees. We can also relate the degree of these Binomial Trees with positions of set bits. With this relation, we can conclude that there are $O(\log n)$ Binomial Trees in a Binomial Heap with ' n ' nodes.

Operations of Binomial Heap:

The main operation in Binomial Heap is union(), all other operations mainly use this operation. The union() operation is to combine two Binomial Heaps into one. Let us first discuss other operations, we will discuss union later.

- 1) insert(H, k): Inserts a key 'k' to Binomial Heap 'H'. This operation first creates a Binomial Heap with single key 'k', then calls union on H and the new Binomial heap.
- 2) getMin(H): A simple way to getMin() is to traverse the list of root of Binomial Trees and return the minimum key. This implementation requires $O(\log n)$ time. It can be optimized to $O(1)$ by maintaining a pointer to minimum key root.
- 3) extractMin(H): This operation also uses union(). We first call getMin() to find the minimum key Binomial Tree, then we remove the node and create a new Binomial Heap by connecting all subtrees of the removed minimum node. Finally, we call union() on H and the newly created Binomial Heap. This operation requires $O(\log n)$ time.

4) delete(H): Like Binary Heap, delete operation first reduces the key to minus infinite, then calls extractMin().

5) decreaseKey(H): decreaseKey() is also similar to Binary Heap. We compare the decreases key with it parent and if parent's key is more, we swap keys and recur for the parent. We stop when we either reach a node whose parent has a smaller key or we hit the root node. Time complexity of decreaseKey() is O(Logn).

Union operation in Binomial Heap:

Given two Binomial Heaps H1 and H2, union(H1, H2) creates a single Binomial Heap.

1) The first step is to simply merge the two Heaps in non-decreasing order of degrees. In the following diagram, figure(b) shows the result after merging.

2) After the simple merge, we need to make sure that there is at most one Binomial Tree of any order. To do this, we need to combine Binomial Trees of the same order. We traverse the list of merged roots, we keep track of three-pointers, prev, x and next-x. There can be following 4 cases when we traverse the list of roots.

— Case 1: Orders of x and next-x are not same, we simply move ahead.

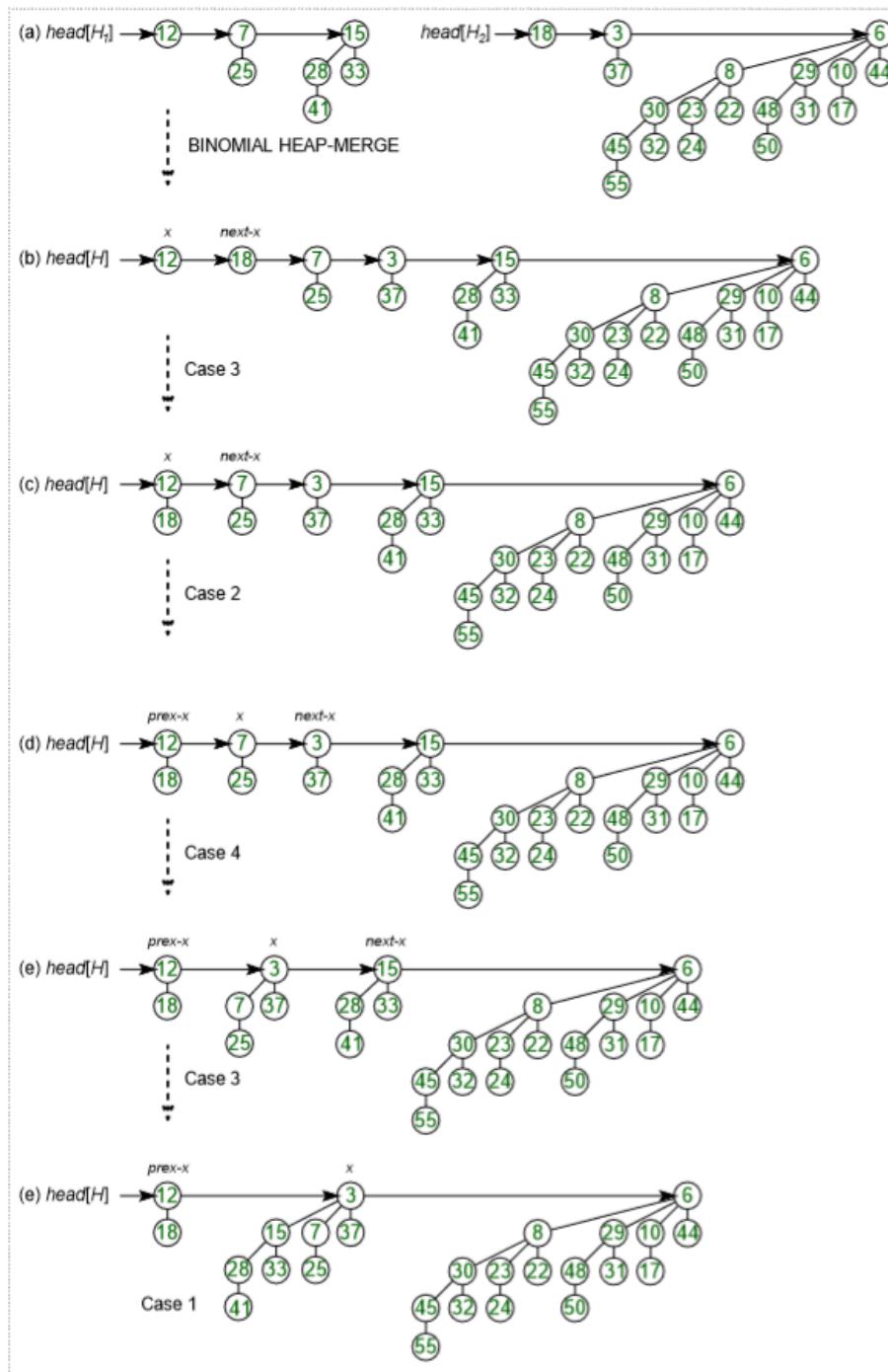
In following 3 cases orders of x and next-x are same.

— Case 2: If the order of next-next-x is also same, move ahead.

— Case 3: If the key of x is smaller than or equal to the key of next-x, then make next-x as a child of x by linking it with x.

— Case 4: If the key of x is greater, then make x as the child of next.

The following diagram is taken from 2nd Edition of [CLRS book](#).



How to represent Binomial Heap?

A Binomial Heap is a set of Binomial Trees. A Binomial Tree must be represented in a way that allows sequential access to all siblings, starting from the leftmost sibling (We need this

in and extractMin() and delete()). The idea is to represent Binomial Trees as the leftmost child and right-sibling representation, i.e., every node stores two pointers, one to the leftmost child and other to the right sibling.

Sources:

[Introduction to Algorithms](#) by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L.

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

Source

<https://www.geeksforgeeks.org/binomial-heap-2/>

Chapter 17

Boggle Set 2 (Using Trie)

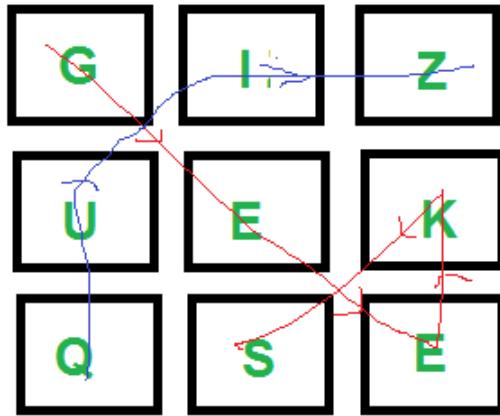
Boggle Set 2 (Using Trie) - GeeksforGeeks

Given a dictionary, a method to do lookup in dictionary and a M x N board where every cell has one character. Find all possible words that can be formed by a sequence of adjacent characters. Note that we can move to any of 8 adjacent characters, but a word should not have multiple instances of same cell.

Example:

```
Input: dictionary[] = {"GEEKS", "FOR", "QUIZ", "GO"};
       boggle[][]   = {{'G','I','Z'},
                      {'U','E','K'},
                      {'Q','S','E'}};
       isWord(str): returns true if str is present in dictionary
                     else false.
```

```
Output: Following words of the dictionary are present
        GEEKS
        QUIZ
```



We have discussed a Graph DFS based solution in below post.

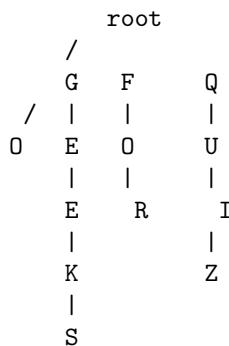
Boggle (Find all possible words in a board of characters) Set 1

Here we discuss a [Trie](#) based solution which is better then DFS based solution.

Given Dictionary `dictionary[] = {"GEEKS", "FOR", "QUIZ", "GO"}`

1. Create an Empty trie and insert all words of given dictionary into trie

After insertion, Trie looks like (leaf nodes are in RED)



2. After that we have pick only those character in `boggle[][]` which are child of root of Trie
Let for above we pick 'G' `boggle[0][0]` , 'Q' `boggle[2][0]` (they both are present in boggle matrix)

3. search a word in a trie which start with character that we pick in step 2

- 1) Create bool visited boolean matrix (`Visited[M][N] = false`)
- 2) Call `SearchWord()` for every cell (i, j) which has one of the first characters of dictionary words. In above example, we have 'G' and 'Q' as first characters.

`SearchWord(Trie *root, i, j, visited[] [N])`

```

if root->leaf == true
    print word

if we have seen this element first time then make it visited.
visited[i][j] = true
do
    traverse all child of current root
    k goes (0 to 26 ) [there are only 26 Alphabet]
    add current char and search for next character

    find next character which is adjacent to boggle[i][j]
    they are 8 adjacent cells of boggle[i][j] (i+1, j+1),
    (i+1, j) (i-1, j) and so on.

    make it unvisited visited[i][j] = false

```

Below is the implementation of above idea
C++

```

// C++ program for Boggle game
#include<bits/stdc++.h>
using namespace std;

// Converts key current character into index
// use only 'A' through 'Z'
#define char_int(c) ((int)c - (int)'A')

// Alphabet size
#define SIZE (26)

#define M 3
#define N 3

// trie Node
struct TrieNode
{
    TrieNode *Child[SIZE];

    // isLeaf is true if the node represents
    // end of a word
    bool leaf;
};

// Returns new trie node (initialized to NULLs)
TrieNode *getNode()
{
    TrieNode *newNode = new TrieNode;
    newNode->leaf = false;
}

```

```

for (int i =0 ; i< SIZE ; i++)
    newNode->Child[i] = NULL;
return newNode;
}

// If not present, inserts a key into the trie
// If the key is a prefix of trie node, just
// marks leaf node
void insert(TrieNode *root, char *Key)
{
    int n = strlen(Key);
    TrieNode * pChild = root;

    for (int i=0; i<n; i++)
    {
        int index = char_int(Key[i]);

        if (pChild->Child[index] == NULL)
            pChild->Child[index] = getNode();

        pChild = pChild->Child[index];
    }

    // make last node as leaf node
    pChild->leaf = true;
}

// function to check that current location
// (i and j) is in matrix range
bool isSafe(int i, int j, bool visited[M][N])
{
    return (i >=0 && i < M && j >=0 &&
            j < N && !visited[i][j]);
}

// A recursive function to print all words present on boggle
void searchWord(TrieNode *root, char boggle[M][N], int i,
                int j, bool visited[][][N], string str)
{
    // if we found word in trie / dictionary
    if (root->leaf == true)
        cout << str << endl ;

    // If both I and j in range and we visited
    // that element of matrix first time
    if (isSafe(i, j, visited))
    {
        // make it visited

```

```

visited[i][j] = true;

// traverse all childs of current root
for (int K =0; K < SIZE; K++)
{
    if (root->Child[K] != NULL)
    {
        // current character
        char ch = (char)K + (char)'A' ;

        // Recursively search reaming character of word
        // in trie for all 8 adjacent cells of boggle[i][j]
        if (isSafe(i+1,j+1,visited) && boggle[i+1][j+1] == ch)
            searchWord(root->Child[K],boggle,i+1,j+1,visited,str+ch);
        if (isSafe(i, j+1,visited) && boggle[i][j+1] == ch)
            searchWord(root->Child[K],boggle,i, j+1,visited,str+ch);
        if (isSafe(i-1,j+1,visited) && boggle[i-1][j+1] == ch)
            searchWord(root->Child[K],boggle,i-1, j+1,visited,str+ch);
        if (isSafe(i+1,j, visited) && boggle[i+1][j] == ch)
            searchWord(root->Child[K],boggle,i+1, j,visited,str+ch);
        if (isSafe(i+1,j-1,visited) && boggle[i+1][j-1] == ch)
            searchWord(root->Child[K],boggle,i+1, j-1,visited,str+ch);
        if (isSafe(i,j-1,visited)&& boggle[i][j-1] == ch)
            searchWord(root->Child[K],boggle,i, j-1,visited,str+ch);
        if (isSafe(i-1,j-1,visited) && boggle[i-1][j-1] == ch)
            searchWord(root->Child[K],boggle,i-1, j-1,visited,str+ch);
        if (isSafe(i-1, j,visited) && boggle[i-1][j] == ch)
            searchWord(root->Child[K],boggle,i-1, j, visited,str+ch);
    }
}

// make current element unvisited
visited[i][j] = false;
}

// Prints all words present in dictionary.
void findWords(char boggle[M][N], TrieNode *root)
{
    // Mark all characters as not visited
    bool visited[M][N];
    memset(visited,false,sizeof(visited));

    TrieNode *pChild = root ;

    string str = "";

    // traverse all matrix elements
}

```

```

for (int i = 0 ; i < M; i++)
{
    for (int j = 0 ; j < N ; j++)
    {
        // we start searching for word in dictionary
        // if we found a character which is child
        // of Trie root
        if (pChild->Child[char_int(boggle[i][j])] )
        {
            str = str+boggle[i][j];
            searchWord(pChild->Child[char_int(boggle[i][j])],
                       boggle, i, j, visited, str);
            str = "";
        }
    }
}

//Driver program to test above function
int main()
{
    // Let the given dictionary be following
    char *dictionary[] = {"GEEKS", "FOR", "QUIZ", "GEE"};

    // root Node of trie
    TrieNode *root = getNode();

    // insert all words of dictionary into trie
    int n = sizeof(dictionary)/sizeof(dictionary[0]);
    for (int i=0; i<n; i++)
        insert(root, dictionary[i]);

    char boggle[M][N] = {{'G','I','Z'},
                        {'U','E','K'},
                        {'Q','S','E'}
    };

    findWords(boggle, root);

    return 0;
}

```

Java

```

// Java program for Boggle game
public class Boggle {

    // Alphabet size

```

```
static final int SIZE = 26;

static final int M = 3;
static final int N = 3;

// trie Node
static class TrieNode
{
    TrieNode[] Child = new TrieNode[SIZE];

    // isLeaf is true if the node represents
    // end of a word
    boolean leaf;

    //constructor
    public TrieNode() {
        leaf = false;
        for (int i = 0 ; i < SIZE ; i++)
            Child[i] = null;
    }
}

// If not present, inserts a key into the trie
// If the key is a prefix of trie node, just
// marks leaf node
static void insert(TrieNode root, String Key)
{
    int n = Key.length();
    TrieNode pChild = root;

    for (int i=0; i<n; i++)
    {
        int index = Key.charAt(i) - 'A';

        if (pChild.Child[index] == null)
            pChild.Child[index] = new TrieNode();

        pChild = pChild.Child[index];
    }

    // make last node as leaf node
    pChild.leaf = true;
}

// function to check that current location
// (i and j) is in matrix range
static boolean isSafe(int i, int j, boolean visited[][])
{
```

```

        return (i >=0 && i < M && j >=0 &&
               j < N && !visited[i][j]);
    }

    // A recursive function to print all words present on boggle
    static void searchWord(TrieNode root, char boggle[][][], int i,
                           int j, boolean visited[][][], String str)
    {
        // if we found word in trie / dictionary
        if (root.leaf == true)
            System.out.println(str);

        // If both I and j in range and we visited
        // that element of matrix first time
        if (isSafe(i, j, visited))
        {
            // make it visited
            visited[i][j] = true;

            // traverse all child of current root
            for (int K =0; K < SIZE; K++)
            {
                if (root.Child[K] != null)
                {
                    // current character
                    char ch = (char) (K + 'A') ;

                    // Recursively search remaining character of word
                    // in trie for all 8 adjacent cells of
                    // boggle[i][j]
                    if (isSafe(i+1,j+1,visited) && boggle[i+1][j+1]
                        == ch)
                        searchWord(root.Child[K],boggle,i+1,j+1,
                                  visited,str+ch);
                    if (isSafe(i, j+1,visited) && boggle[i][j+1]
                        == ch)
                        searchWord(root.Child[K],boggle,i, j+1,
                                  visited,str+ch);
                    if (isSafe(i-1,j+1,visited) && boggle[i-1][j+1]
                        == ch)
                        searchWord(root.Child[K],boggle,i-1, j+1,
                                  visited,str+ch);
                    if (isSafe(i+1,j, visited) && boggle[i+1][j]
                        == ch)
                        searchWord(root.Child[K],boggle,i+1, j,
                                  visited,str+ch);
                    if (isSafe(i+1,j-1,visited) && boggle[i+1][j-1]
                        == ch)
                        searchWord(root.Child[K],boggle,i+1, j-1,
                                  visited,str+ch);
                }
            }
        }
    }
}

```

```

        searchWord(root.Child[K], boggle, i+1, j-1,
                   visited,str+ch);
        if (isSafe(i, j-1,visited)&& boggle[i][j-1]
            == ch)
            searchWord(root.Child[K], boggle, i, j-1,
                       visited,str+ch);
        if (isSafe(i-1,j-1,visited) && boggle[i-1][j-1]
            == ch)
            searchWord(root.Child[K], boggle, i-1, j-1,
                       visited,str+ch);
        if (isSafe(i-1, j,visited) && boggle[i-1][j]
            == ch)
            searchWord(root.Child[K], boggle, i-1, j,
                       visited,str+ch);
    }
}

// make current element unvisited
visited[i][j] = false;
}
}

// Prints all words present in dictionary.
static void findWords(char boggle[][] , TrieNode root)
{
    // Mark all characters as not visited
    boolean[][] visited = new boolean[M][N];
    TrieNode pChild = root ;

    String str = "";

    // traverse all matrix elements
    for (int i = 0 ; i < M; i++)
    {
        for (int j = 0 ; j < N ; j++)
        {
            // we start searching for word in dictionary
            // if we found a character which is child
            // of Trie root
            if (pChild.Child[(boggle[i][j]) - 'A'] != null)
            {
                str = str+boggle[i][j];
                searchWord(pChild.Child[(boggle[i][j]) - 'A'],
                           boggle, i, j, visited, str);
                str = "";
            }
        }
    }
}

```

```
}

// Driver program to test above function
public static void main(String args[])
{
    // Let the given dictionary be following
    String dictionary[] = {"GEEKS", "FOR", "QUIZ", "GEE"};

    // root Node of trie
    TrieNode root = new TrieNode();

    // insert all words of dictionary into trie
    int n = dictionary.length;
    for (int i=0; i<n; i++)
        insert(root, dictionary[i]);

    char boggle[][] = {{'G','I','Z'},
                      {'U','E','K'},
                      {'Q','S','E'}
                     };
    findWords(boggle, root);
}

// This code is contributed by Sumit Ghosh
```

Output:

GEE, GEEKS, QUIZ

Source

<https://www.geeksforgeeks.org/boggle-set-2-using-trie/>

Chapter 18

Burrows – Wheeler Data Transform Algorithm

Burrows - Wheeler Data Transform Algorithm - GeeksforGeeks

What is the Burrows – Wheeler Transform?

The BWT is a data transformation algorithm that restructures data in such a way that the transformed message is more compressible. Technically, it is a lexicographical reversible permutation of the characters of a string. It is first of the three steps to be performed in succession while implementing Burrows – Wheeler Data Compression algorithm that forms the basis of the Unix compression utility bzip2.

Why BWT? The main idea behind it.

The most important application of BWT is found in biological sciences where genomes(long strings written in A, C, T, G alphabets) don't have many runs but they do have many repeats.

The idea of the BWT is to build an array whose rows are all cyclic shifts of the input string in dictionary order, and return the last column of the array that tends to have long runs of identical characters. The benefit of this is that once the characters have been clustered together, they effectively have an ordering, which can make our string more compressible for other algorithms like run length encoding and Huffman Coding.

The remarkable thing about BWT is that this particular transform is reversible with minimal data overhead.

Steps involved in BWT algorithm

Let's take the word “banana\$” as an example.

Step 1: Form all cyclic rotations of the given text.

\$ b	banana\$
a a	\$banana
Cyclic rotations ----->	a\$banan
n n	na\$bana
	ana\$ban

a	nana\$ba anana\$b
---	----------------------

Step 2: The next step is to sort the rotations lexicographically. The ‘\$’ sign is viewed as first letter lexicographically, even before ‘a’.

banana\$ \$banana a\$banan na\$bana ana\$ban nana\$ba anana\$b	Sorting -----> alphabetically	\$banana a\$banan ana\$ban anana\$b banana\$ na\$bana nana\$ba
--	-------------------------------------	--

Step 3: The last column is what we output as BWT.

BWT(banana\$) = annb\$aa

Examples:

```
Input : banana$ // Input text
Output : annb$aa // Burrows - Wheeler Transform
```

```
Input : abracadabra$
Output : ard$rcaaaabb
```

Why last column is considered BWT?

1. The last column has better symbol clustering than any other columns.
2. If we only have BWT of our string, we can recover the rest of the cyclic rotations entirely. The rest of the columns don't possess this characteristic which is highly important while computing inverse of BWT.

Why ‘\$’ sign is embedded in the text?

We can compute BWT even if our text is not concatenated with any EOF character (‘\$’ here). The implication of ‘\$’ sign comes while computing the inverse of BWT.

Way of implementation

1. Let's instantiate “banana\$” as our **input_text** and instantiate character array **bwt_arr** for our output.
2. Let's get all the suffixes of “banana\$” and compute it's **suffix_arr** to store index of each suffix.

```

0 banana$          6 $
1 anana$          5 a$
2 nana$      Sorting   3 ana$
3 ana$      -----> 1 anana$
4 na$      alphabetically 0 banana$
5 a$          4 na$
6 $          2 nana$

```

3. Iterating over the **suffix_arr**, let's now add to our output array **bwt_arr**, the last character of each rotation.

4. The last character of each rotation of **input_text** starting at the position denoted by the current value in the suffix array can be calculated with **input_text[(suffix_arr[i] – 1 + n) % n]**, where **n** is the number of elements in the **suffix_arr**.

```

bwt_arr[0] = input_text[(suffix_arr[0] - 1 + 7) % 7]
            = input_text[5]
            = a
bwt_arr[1] = input_text[(suffix_arr[1] - 1 + 7) % 7]
            = input_text[4]
            = n

```

Following is the code for way of implementation explained above

```

// C program to find Burrows Wheeler transform of
// a given text
#include<stdio.h>
#include<string.h>
#include<stdlib.h>

// Structure to store data of a rotation
struct rotation
{
    int index;
    char *suffix;
};

// Compares the rotations and
// sorts the rotations alphabetically
int cmpfunc (const void *x, const void *y)
{
    struct rotation *rx = (struct rotation *)x;
    struct rotation *ry = (struct rotation *)y;
    return strcmp(rx -> suffix, ry -> suffix);
}

```

```

// Takes text to be transformed and its length as
// arguments and returns the corresponding suffix array
int *computeSuffixArray(char *input_text, int len_text)
{
    // Array of structures to store rotations and
    // their indexes
    struct rotation suff[len_text];

    // Structure is needed to maintain old indexes of
    // rotations after sorting them
    for(int i = 0; i < len_text; i++)
    {
        suff[i].index = i;
        suff[i].suffix = (input_text+i);
    }

    // Sorts rotations using comparison function defined above
    qsort(suff, len_text, sizeof(struct rotation), cmpfunc);

    // Stores the indexes of sorted rotations
    int *suffix_arr = (int *) malloc (len_text * sizeof(int));
    for (int i = 0; i < len_text; i++)
        suffix_arr[i] = suff[i].index;

    // Returns the computed suffix array
    return suffix_arr;
}

// Takes suffix array and its size as arguments and returns
// the Burrows – Wheeler Transform of given text
char *findLastChar(char *input_text, int *suffix_arr, int n)
{
    // Iterates over the suffix array to find
    // the last char of each cyclic rotation
    char *bwt_arr = (char *) malloc (n * sizeof(char));
    int i;
    for (i = 0; i < n; i++)
    {
        // Computes the last char which is given by
        // input_text[(suffix_arr[i] + n - 1) % n]
        int j = suffix_arr[i] - 1;
        if (j < 0)
            j = j + n;

        bwt_arr[i] = input_text[j];
    }

    bwt_arr[i] = '\0';
}

```

```

// Returns the computed Burrows – Wheeler Transform
return bwt_arr;
}

// Driver program to test functions above
int main()
{
    char input_text[] = "banana$";
    int len_text = strlen(input_text);

    // Computes the suffix array of our text
    int *suffix_arr = computeSuffixArray(input_text, len_text);

    // Adds to the output array the last char of each rotation
    char *bwt_arr = findLastChar(input_text, suffix_arr, len_text);

    printf("Input text : %s\n", input_text);
    printf("Burrows – Wheeler Transform : %s\n", bwt_arr);
    return 0;
}

```

Output:

```

Input text : banana$
Burrows – Wheeler Transform : annb$aa

```

Time Complexity: $O(n^2 \log n)$. This is because of the method used above to build suffix array which has $O(n^2 \log n)$ time complexity, due to $O(n)$ time for string comparisons in $O(n \log n)$ sorting algorithm.

Exercise: 1. Compute suffix array in $O(n \log n)$ time and then implement BWT.
2. Implement Inverse of Burrows – Wheeler Transform.

Source

<https://www.geeksforgeeks.org/burrows-wheeler-data-transform-algorithm/>

Chapter 19

C++ Program to implement Symbol Table

C++ Program to implement Symbol Table - GeeksforGeeks

Prerequisite: [Symbol Table](#)

A *Symbol table* is a data structure used by the compiler, where each identifier in program's source code is stored along with information associated with it relating to its declaration. It stores identifier as well as it's associated attributes like scope, type, line-number of occurrence, etc.

Symbol table can be implemented using various data structures like:

- LinkedList
- Hash Table
- Tree

A common data structure used to implement a symbol table is HashTable.

Operations of Symbol table – The basic operations defined on a symbol table include:

Operation	Function
allocate	to allocate a new empty symbol table
free	to remove all entries and free storage of symbol table
lookup	to search for a name and return pointer to its entry
insert	to insert a name in a symbol table and return a pointer to its entry
set_attribute	to associate an attribute with a given entry
get_attribute	to get an attribute associated with a given entry

Consider the following C++ function:

```
// Define a global function
int add(int a, int b)
{
    int sum = 0;
    sum = a + b;
    return sum;
}
```

Symbol Table for above code:

Name
Type
Scope
add
function
global
a
int
function parameter
b
int
function parameter
sum
int

local

Below is the C++ implementation of Symbol Table using the concept of [Hashing with separate chaining](#):

```
// C++ program to implement Symbol Table
#include <iostream>
using namespace std;

const int MAX = 100;

class Node {

    string identifier, scope, type;
    int lineNo;
    Node* next;

public:
    Node()
    {
        next = NULL;
    }

    Node(string key, string value, string type, int lineNo)
    {
        this->identifier = key;
        this->scope = value;
        this->type = type;
        this->lineNo = lineNo;
        next = NULL;
    }

    void print()
    {
        cout << "Identifier's Name:" << identifier
            << "\nType:" << type
            << "\nScope: " << scope
            << "\nLine Number: " << lineNo << endl;
    }
    friend class SymbolTable;
};

class SymbolTable {
    Node* head[MAX];

public:
    SymbolTable()
    {
        for (int i = 0; i < MAX; i++)
    }
```

```
        head[i] = NULL;
    }

    int hashf(string id); // hash function
    bool insert(string id, string scope,
                string Type, int lineno);

    string find(string id);

    bool deleteRecord(string id);

    bool modify(string id, string scope,
                string Type, int lineno);
};

// Function to modify an identifier
bool SymbolTable::modify(string id, string s,
                         string t, int l)
{
    int index = hashf(id);
    Node* start = head[index];

    if (start == NULL)
        return "-1";

    while (start != NULL) {
        if (start->identifier == id) {
            start->scope = s;
            start->type = t;
            start->lineNo = l;
            return true;
        }
        start = start->next;
    }

    return false; // id not found
}

// Function to delete an identifier
bool SymbolTable::deleteRecord(string id)
{
    int index = hashf(id);
    Node* tmp = head[index];
    Node* par = head[index];

    // no identifier is present at that index
    if (tmp == NULL) {
        return false;
```

```
}

// only one identifier is present
if (tmp->identifier == id && tmp->next == NULL) {
    tmp->next = NULL;
    delete tmp;
    return true;
}

while (tmp->identifier != id && tmp->next != NULL) {
    par = tmp;
    tmp = tmp->next;
}
if (tmp->identifier == id && tmp->next != NULL) {
    par->next = tmp->next;
    tmp->next = NULL;
    delete tmp;
    return true;
}

// delete at the end
else {
    par->next = NULL;
    tmp->next = NULL;
    delete tmp;
    return true;
}
return false;
}

// Function to find an identifier
string SymbolTable::find(string id)
{
    int index = hashf(id);
    Node* start = head[index];

    if (start == NULL)
        return "-1";

    while (start != NULL) {

        if (start->identifier == id) {
            start->print();
            return start->scope;
        }

        start = start->next;
    }
}
```

```
        return "-1"; // not found
    }

// Function to insert an identifier
bool SymbolTable::insert(string id, string scope,
                         string Type, int lineno)
{
    int index = hashf(id);
    Node* p = new Node(id, scope, Type, lineno);

    if (head[index] == NULL) {
        head[index] = p;
        cout << "\n"
            << id << " inserted";

        return true;
    }

    else {
        Node* start = head[index];
        while (start->next != NULL)
            start = start->next;

        start->next = p;
        cout << "\n"
            << id << " inserted";

        return true;
    }
}

return false;
}

int SymbolTable::hashf(string id)
{
    int asciiSum = 0;

    for (int i = 0; i < id.length(); i++) {
        asciiSum = asciiSum + id[i];
    }

    return (asciiSum % 100);
}

// Driver code
int main()
{
    SymbolTable st;
```

```
string check;
cout << "**** SYMBOL_TABLE ****\n";

// insert 'if'
if (st.insert("if", "local", "keyword", 4))
    cout << " -successfully";
else
    cout << "\nFailed to insert.\n";

// insert 'number'
if (st.insert("number", "global", "variable", 2))
    cout << " -successfully\n\n";
else
    cout << "\nFailed to insert\n";

// find 'if'
check = st.find("if");
if (check != "-1")
    cout << "Identifier Is present\n";
else
    cout << "\nIdentifier Not Present\n";

// delete 'if'
if (st.deleteRecord("if"))
    cout << "if Identifier is deleted\n";
else
    cout << "\nFailed to delete\n";

// modify 'number'
if (st.modify("number", "global", "variable", 3))
    cout << "\nNumber Identifier updated\n";

// find and print 'number'
check = st.find("number");
if (check != "-1")
    cout << "Identifier Is present\n";
else
    cout << "\nIdentifier Not Present";

return 0;
}
```

Output:

```
**** SYMBOL_TABLE ****
```

```
if inserted -successfully
```

```
number inserted -successfully
```

```
Identifier's Name:if
Type:keyword
Scope: local
Line Number: 4
Identifier Is present
```

```
if Identifier is deleted
```

```
number Identifier updated
```

```
Identifier's Name:number
Type:variable
Scope: global
Line Number: 3
Identifier Is present
```

Source

<https://www.geeksforgeeks.org/cpp-program-to-implement-symbol-table/>

Chapter 20

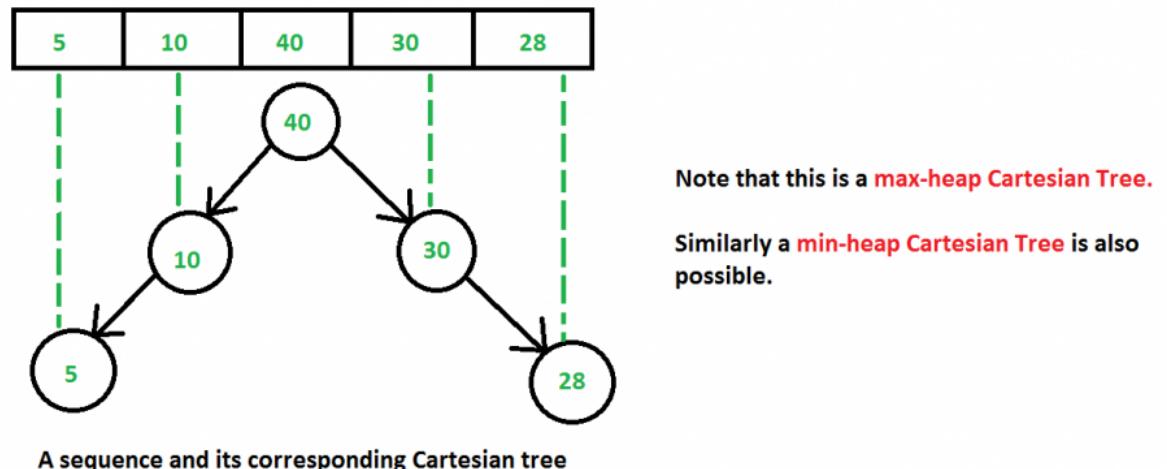
Cartesian Tree

Cartesian Tree - GeeksforGeeks

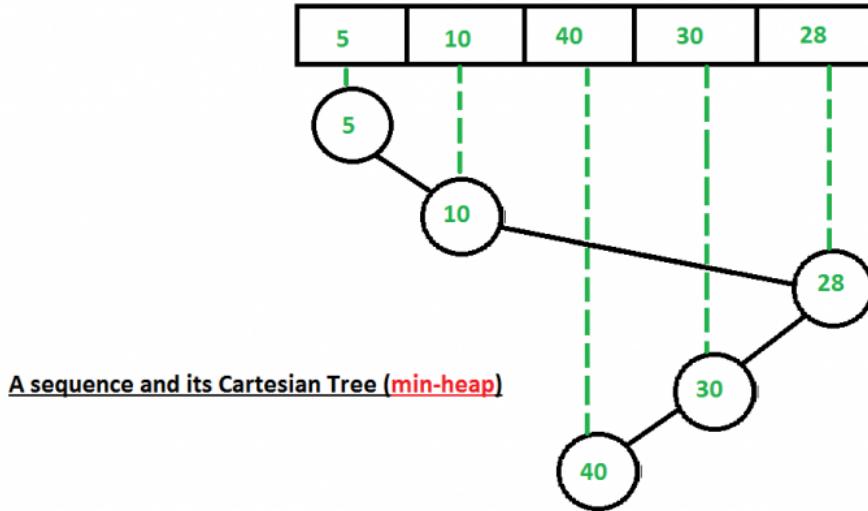
A Cartesian tree is a tree data structure created from a set of data that obeys the following structural invariants:

1. The tree obeys the min (or max) heap property – each node is less (or greater) than its children.
2. An inorder traversal of the nodes yields the values in the same order in which they appear in the initial sequence.

Suppose we have an input array- {5,10,40,30,28}. Then the max-heap Cartesian Tree would be.



A min-heap Cartesian Tree of the above input array will be-



Note:

1. Cartesian Tree is not a height-balanced tree.
2. Cartesian tree of a sequence of distinct numbers is always unique.

Cartesian tree of a sequence of distinct numbers is always unique.

We will prove this using induction. As a base case, empty tree is always unique. For the inductive case, assume that for all trees containing $n' < n$ elements, there is a unique Cartesian tree for each sequence of n' nodes. Now take any sequence of n elements. Because a Cartesian tree is a min-heap, the smallest element of the sequence must be the root of the Cartesian tree. Because an inorder traversal of the elements must yield the input sequence, we know that all nodes to the left of the min element must be in its left subtree and similarly for the nodes to the right. Since the left and right subtree are both Cartesian trees with at most $n-1$ elements in them (since the min element is at the root), by the induction hypothesis there is a unique Cartesian tree that could be the left or right subtree. Since all our decisions were forced, we end up with a unique tree, completing the induction.

How to construct Cartesian Tree?

A $O(n^2)$ solution for construction of Cartesian Tree is discussed [here](#) (Note that the above program [here](#) constructs the “special binary tree” (which is nothing but a Cartesian tree)

A $O(n)$ Algorithm :

It's possible to build a Cartesian tree from a sequence of data in linear time. Beginning with the empty tree,

Scan the given sequence from left to right adding new nodes as follows:

1. Position the node as the right child of the rightmost node.

2. Scan upward from the node's parent up to the root of the tree until a node is found whose value is greater than the current value.
3. If such a node is found, set its right child to be the new node, and set the new node's left child to be the previous right child.
4. If no such node is found, set the new child to be the root, and set the new node's left child to be the previous tree.

```

// A O(n) C++ program to construct cartesian tree
// from a given array
#include<bits/stdc++.h>

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

/* This function is here just to test buildTree() */
void printInorder (Node* node)
{
    if (node == NULL)
        return;
    printInorder (node->left);
    cout << node->data << " ";
    printInorder (node->right);
}

// Recursively construct subtree under given root using
// leftChil[] and rightchild
Node * buildCartesianTreeUtil (int root, int arr[],
                               int parent[], int leftchild[], int rightchild[])
{
    if (root == -1)
        return NULL;

    // Create a new node with root's data
    Node *temp = new Node;
    temp->data = arr[root] ;

    // Recursively construct left and right subtrees
    temp->left = buildCartesianTreeUtil( leftchild[root],
                                         arr, parent, leftchild, rightchild );
    temp->right = buildCartesianTreeUtil( rightchild[root],
                                         arr, parent, leftchild, rightchild );
}

```

```

        return temp ;
    }

// A function to create the Cartesian Tree in O(N) time
Node * buildCartesianTree (int arr[], int n)
{
    // Arrays to hold the index of parent, left-child,
    // right-child of each number in the input array
    int parent[n],leftchild[n],rightchild[n];

    // Initialize all array values as -1
    memset(parent, -1, sizeof(parent));
    memset(leftchild, -1, sizeof(leftchild));
    memset(rightchild, -1, sizeof(rightchild));

    // 'root' and 'last' stores the index of the root and the
    // last processed of the Cartesian Tree.
    // Initially we take root of the Cartesian Tree as the
    // first element of the input array. This can change
    // according to the algorithm
    int root = 0, last;

    // Starting from the second element of the input array
    // to the last on scan across the elements, adding them
    // one at a time.
    for (int i=1; i<=n-1; i++)
    {
        last = i-1;
        rightchild[i] = -1;

        // Scan upward from the node's parent up to
        // the root of the tree until a node is found
        // whose value is greater than the current one
        // This is the same as Step 2 mentioned in the
        // algorithm
        while (arr[last] <= arr[i] && last != root)
            last = parent[last];

        // arr[i] is the largest element yet; make it
        // new root
        if (arr[last] <= arr[i])
        {
            parent[root] = i;
            leftchild[i] = root;
            root = i;
        }
    }

    // Just insert it
}

```

```

    else if (rightchild[last] == -1)
    {
        rightchild[last] = i;
        parent[i] = last;
        leftchild[i] = -1;
    }

    // Reconfigure links
    else
    {
        parent[rightchild[last]] = i;
        leftchild[i] = rightchild[last];
        rightchild[last] = i;
        parent[i] = last;
    }
}

// Since the root of the Cartesian Tree has no
// parent, so we assign it -1
parent[root] = -1;

return (buildCartesianTreeUtil (root, arr, parent,
                                leftchild, rightchild));
}

/* Driver program to test above functions */
int main()
{
    /* Assume that inorder traversal of following tree
       is given
          40
         /   \
        10   30
       /     \
      5      28 */
    int arr[] = {5, 10, 40, 30, 28};
    int n = sizeof(arr)/sizeof(arr[0]);

    Node *root = buildCartesianTree(arr, n);

    /* Let us test the built tree by printing Inorder
       traversal */
    printf("Inorder traversal of the constructed tree : \n");
    printInorder(root);

    return(0);
}

```

}

Output:

```
Inorder traversal of the constructed tree :  
5 10 40 30 28
```

Time Complexity :

At first look, the code seems to be taking $O(n^2)$ time as there are two loops in `buildCartesianTree()`. But actually, it takes linear time.

The inner while loop represents the process in which we scan the tree upwards in the search of finding a suitable place to insert the new element. A keen observation can show that in total, the whole while loop (i.e- over all values from $i = 1$ to $i < n$) takes $O(n)$ time and not every time. Hence, the overall time complexity is $O(n)$.

Auxiliary Space:

We declare a structure for every node as well as three extra arrays- `leftchild[]`, `rightchild[]`, `parent[]` to hold the indices of left-child, right-child, parent of each value in the input array. Hence the overall $O(4*n) = O(n)$ extra space.

Application of Cartesian Tree

- [Cartesian Tree Sorting](#)
- A range minimum query on a sequence is equivalent to a lowest common ancestor query on the sequence's Cartesian tree. Hence, RMQ may be reduced to LCA using the sequence's Cartesian tree.
- [Treap, a balanced binary search tree structure](#), is a Cartesian tree of (key,priority) pairs; it is heap-ordered according to the priority values, and an inorder traversal gives the keys in sorted order.
- [Suffix tree](#) of a string may be constructed from the suffix array and the longest common prefix array. The first step is to compute the Cartesian tree of the longest common prefix array.

References:

http://wcipeg.com/wiki/Cartesian_tree

Source

<https://www.geeksforgeeks.org/cartesian-tree/>

Chapter 21

Cartesian Tree Sorting

Cartesian Tree Sorting - GeeksforGeeks

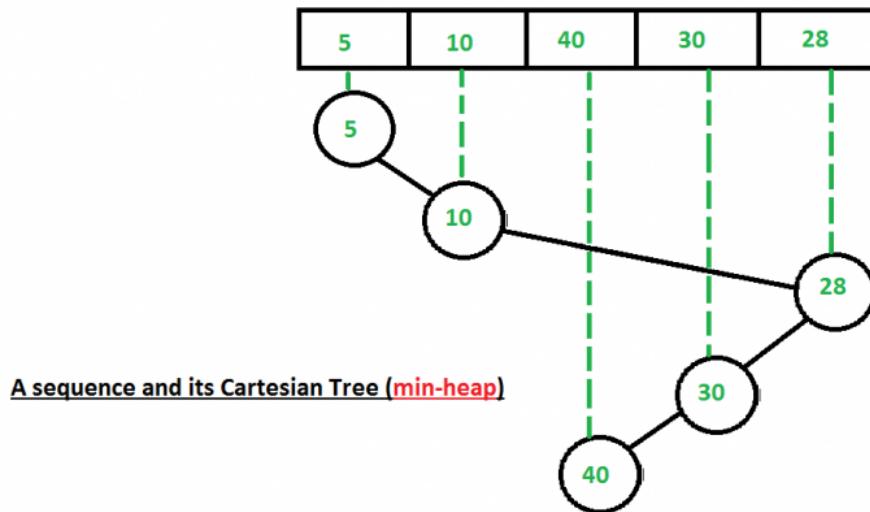
Prerequisites : [Cartesian Tree](#)

Cartesian Sort is an Adaptive Sorting as it sorts the data faster if data is partially sorted. In fact, there are very few sorting algorithms that make use of this fact.

For example consider the array {5, 10, 40, 30, 28}. The input data is partially sorted too as only one swap between “40” and “28” results in a completely sorted order. See how Cartesian Tree Sort will take advantage of this fact below.

Below are steps used for sorting.

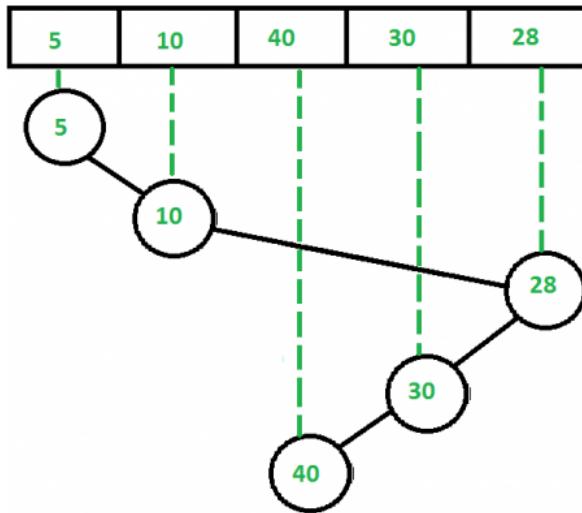
Step 1 : Build a (min-heap) [Cartesian Tree](#) from the given input sequence.



Step 2 : Starting from the root of the built Cartesian Tree, we push the nodes in a priority queue.

Then we pop the node at the top of the priority queue and push the children of the popped node in a pre-order manner.

1. Pop the node at the top of the priority queue and add it to a list.
2. Push left child of the popped node first (if present).
3. Push right child of the popped node next (if present).

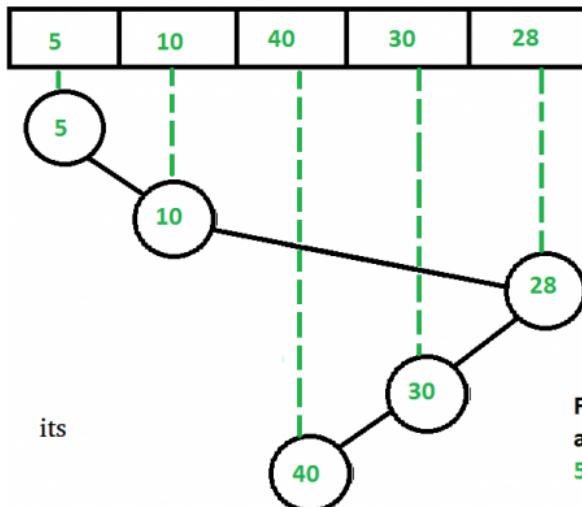
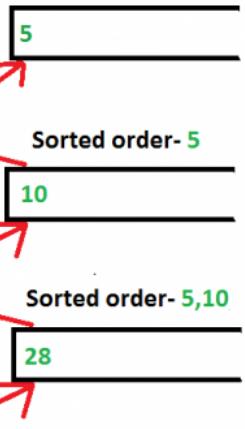


Inner Operations of Priority Queue

5 is pushed initially as it is the root node of the Cartesian Tree

5 is popped out and its left and right child are pushed. Note that 5 has only right child- 10

10 is popped out and its left and right child are pushed. Note that 10 has only right child- 28

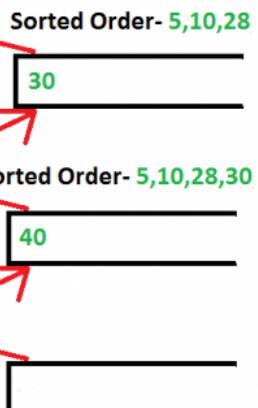


Inner Operations of Priority Queue

28 is popped out and its left and right child are pushed. Note that 28 has only left child-30

30 is popped out and its left and right child are pushed. Note that 30 has only left child-40

Finally, 40 is also popped out and the final sorted order is -



How to build (min-heap) Cartesian Tree?

Building min-heap is similar to building a (max-heap) Cartesian Tree (discussed in [previous post](#)), except the fact that now we scan upward from the node's parent up to the root of the tree until a node is found whose value is smaller (and not larger as in the case of a

max-heap Cartesian Tree) than the current one and then accordingly reconfigure links to build the min-heap Cartesian tree.

Why not to use only priority queue?

One might wonder that using priority queue would anyway result in a sorted data if we simply insert the numbers of the input array one by one in the priority queue (i.e- without constructing the Cartesian tree).

But the time taken differs a lot.

Suppose we take the input array – {5, 10, 40, 30, 28}

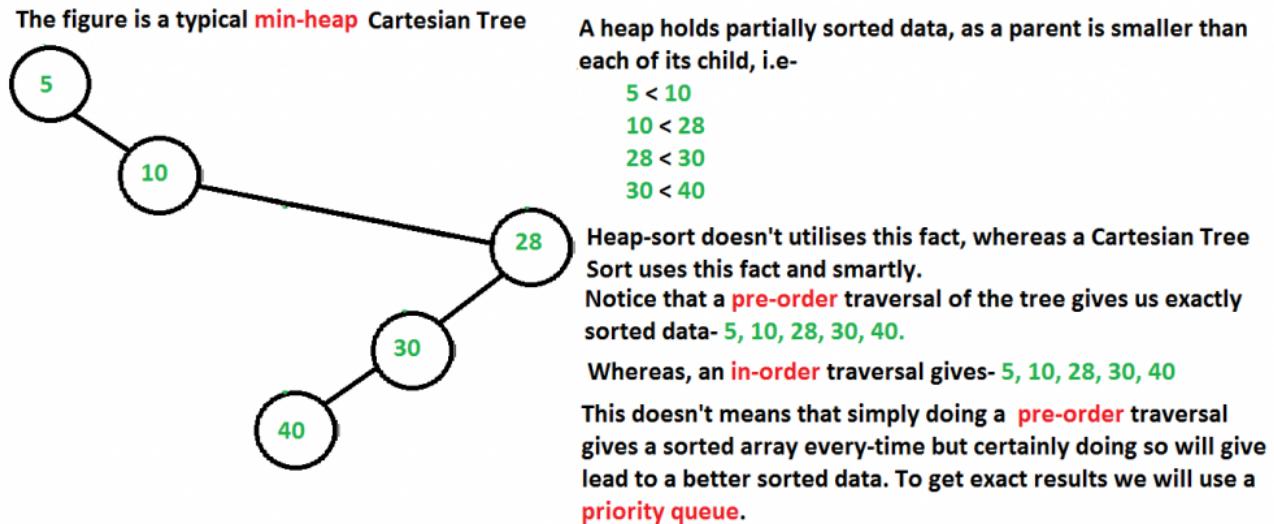
If we simply insert the input array numbers one by one (without using a Cartesian tree), then we may have to waste a lot of operations in adjusting the queue order everytime we insert the numbers (just like a typical heap performs those operations when a new number is inserted, as priority queue is nothing but a heap).

Whereas, here we can see that using a Cartesian tree took only 5 operations (see the above two figures in which we are continuously pushing and popping the nodes of Cartesian tree), which is linear as there are 5 numbers in the input array also. So we see that the best case of Cartesian Tree sort is $O(n)$, a thing where heap-sort will take much more number of operations, because it doesn't make advantage of the fact that the input data is partially sorted.

Why pre-order traversal?

The answer to this is that since Cartesian Tree is basically a heap- data structure and hence follows all the properties of a heap. Thus the root node is always smaller than both of its children. Hence, we use a pre-order fashion popping-and-pushing as in this, the root node is always pushed earlier than its children inside the priority queue and since the root node is always less than both its child, so we don't have to do extra operations inside the priority queue.

Refer to the below figure for better understanding-



```
// A C++ program to implement Cartesian Tree sort
```

```

// Note that in this program we will build a min-heap
// Cartesian Tree and not max-heap.
#include<bits/stdc++.h>
using namespace std;

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

// Creating a shortcut for int, Node* pair type
typedef pair<int, Node*> iNPair;

// This function sorts by pushing and popping the
// Cartesian Tree nodes in a pre-order like fashion
void pQBAsedTraversal(Node* root)
{
    // We will use a priority queue to sort the
    // partially-sorted data efficiently.
    // Unlike Heap, Cartesian tree makes use of
    // the fact that the data is partially sorted
    priority_queue <iNPair, vector<iNPair>, greater<iNPair>> pQueue;
    pQueue.push (make_pair (root->data,root));

    // Resembles a pre-order traverse as first data
    // is printed then the left and then right child.
    while (! pQueue.empty())
    {
        iNPair popped_pair = pQueue.top();
        printf("%d ",popped_pair.first);

        pQueue.pop();

        if (popped_pair.second->left != NULL)
            pQueue.push (make_pair(popped_pair.second->left->data,
                                  popped_pair.second->left));

        if (popped_pair.second->right != NULL)
            pQueue.push (make_pair(popped_pair.second->right->data,
                                  popped_pair.second->right));
    }

    return;
}

```

```

Node *buildCartesianTreeUtil(int root, int arr[],
                            int parent[], int leftchild[], int rightchild[])
{
    if (root == -1)
        return NULL;

    Node *temp = new Node;

    temp->data = arr[root];
    temp->left = buildCartesianTreeUtil(leftchild[root],
                                         arr, parent, leftchild, rightchild);

    temp->right = buildCartesianTreeUtil(rightchild[root],
                                         arr, parent, leftchild, rightchild);

    return temp ;
}

// A function to create the Cartesian Tree in O(N) time
Node *buildCartesianTree(int arr[], int n)
{
    // Arrays to hold the index of parent, left-child,
    // right-child of each number in the input array
    int parent[n],leftchild[n],rightchild[n];

    // Initialize all array values as -1
    memset(parent, -1, sizeof(parent));
    memset(leftchild, -1, sizeof(leftchild));
    memset(rightchild, -1, sizeof(rightchild));

    // 'root' and 'last' stores the index of the root and the
    // last processed of the Cartesian Tree.
    // Initially we take root of the Cartesian Tree as the
    // first element of the input array. This can change
    // according to the algorithm
    int root = 0, last;

    // Starting from the second element of the input array
    // to the last on scan across the elements, adding them
    // one at a time.
    for (int i=1; i<=n-1; i++)
    {
        last = i-1;
        rightchild[i] = -1;

        // Scan upward from the node's parent up to
        // the root of the tree until a node is found

```

```

// whose value is smaller than the current one
// This is the same as Step 2 mentioned in the
// algorithm
while (arr[last] >= arr[i] && last != root)
    last = parent[last];

// arr[i] is the smallest element yet; make it
// new root
if (arr[last] >= arr[i])
{
    parent[root] = i;
    leftchild[i] = root;
    root = i;
}

// Just insert it
else if (rightchild[last] == -1)
{
    rightchild[last] = i;
    parent[i] = last;
    leftchild[i] = -1;
}

// Reconfigure links
else
{
    parent[rightchild[last]] = i;
    leftchild[i] = rightchild[last];
    rightchild[last] = i;
    parent[i] = last;
}
}

// Since the root of the Cartesian Tree has no
// parent, so we assign it -1
parent[root] = -1;

return (buildCartesianTreeUtil (root, arr, parent,
                               leftchild, rightchild));
}

// Sorts an input array
int printSortedArr(int arr[], int n)
{
    // Build a cartesian tree
    Node *root = buildCartesianTree(arr, n);
}

```

```

printf("The sorted array is-\n");

// Do pr-order traversal and insert
// in priority queue
pQBasedTraversal(root);
}

/* Driver program to test above functions */
int main()
{
    /* Given input array- {5,10,40,30,28},
       it's corresponding unique Cartesian Tree
       is-

      5
      \
      10
      \
      28
      /
      30
      /
      40
    */
}

int arr[] = {5, 10, 40, 30, 28};
int n = sizeof(arr)/sizeof(arr[0]);

printSortedArr(arr, n);

return(0);
}

```

Output :

```

The sorted array is-
5 10 28 30 40

```

Time Complexity : $O(n)$ best-case behaviour (when the input data is partially sorted),
 $O(n \log n)$ worst-case behavior (when the input data is not partially sorted)

Auxiliary Space : We use a priority queue and a Cartesian tree data structure. Now, at any moment of time the size of the priority queue doesn't exceeds the size of the input array, as we are constantly pushing and popping the nodes. Hence we are using $O(n)$ auxiliary space.

References :

https://en.wikipedia.org/wiki/Adaptive_sort

<http://11011110.livejournal.com/283412.html>
<http://gradbot.blogspot.in/2010/06/cartesian-tree-sort.html>
<http://www.keithschwarz.com/interesting/code/?dir=cartesian-tree-sort>
https://en.wikipedia.org/wiki/Cartesian_tree#Application_in_sorting

Source

<https://www.geeksforgeeks.org/cartesian-tree-sorting/>

Chapter 22

Centroid Decomposition of Tree

Centroid Decomposition of Tree - GeeksforGeeks

Background :

What is centroid of Tree?

Centroid of a Tree is a node which if removed from the tree would split it into a ‘forest’, such that any tree in the forest would have at most half the number of vertices in the original tree.

Suppose there are n nodes in the tree. ‘Subtree size’ for a node is the size of the tree rooted at the node.

Let $S(v)$ be size of subtree rooted at node v

$S(v) = 1 + \sum S(u)$

Here u is a child to v (adjacent and at a depth one greater than the depth of v).

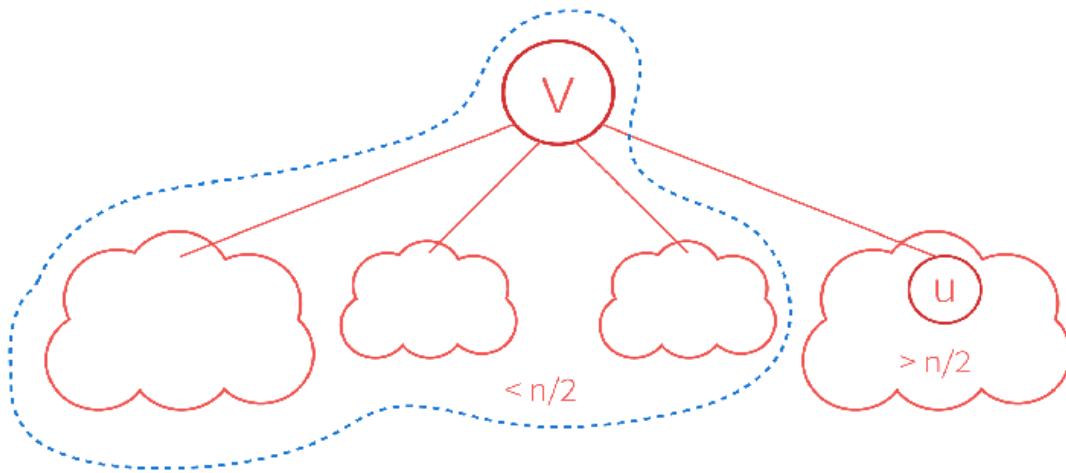
Centroid is a node v such that,

$\max(n - S(v), S(u_1), S(u_2), \dots, S(u_m)) \leq n/2$

where u_i is i 'th child to v .

Finding the centroid

Let T be an undirected tree with n nodes. Choose any arbitrary node v in the tree. If v satisfies the mathematical definition for the centroid, we have our centroid. Else, we know that our mathematical inequality did not hold, and from this we conclude that there exists some u adjacent to v such that $S(u) > n/2$. We make that u our new v and recurse.



We never revisit a node because when we decided to move away from it to a node with subtree size greater than $n/2$, we sort of declared that it now belongs to the component with nodes less than $n/2$, and we shall never find our centroid there.

In any case we are moving towards the centroid. Also, there are finitely many vertices in the tree. The process must stop, and it will, at the desired vertex.

Algorithm :

1. Select arbitrary node v
2. Start a DFS from v , and setup subtree sizes
3. Re-position to node v (or start at any arbitrary v that belongs to the tree)
4. Check mathematical condition of centroid for v
 - (a) If condition passed, return current node as centroid
 - (b) Else move to adjacent node with ‘greatest’ subtree size, and back to step 4

Theorem: Given a tree with n nodes, the centroid always exists.

Proof: Clear from our approach to the problem that we can always find a centroid using above steps.

Time Complexity

1. Select arbitrary node v : $O(1)$
2. DFS: $O(n)$
3. Reposition to v : $O(1)$
4. Find centroid: $O(n)$

Centroid Decomposition :

Finding the centroid for a tree is a part of what we are trying to achieve here. We need to think how can we organize the tree into a structure that decreases the complexity for answering certain ‘type’ of queries.

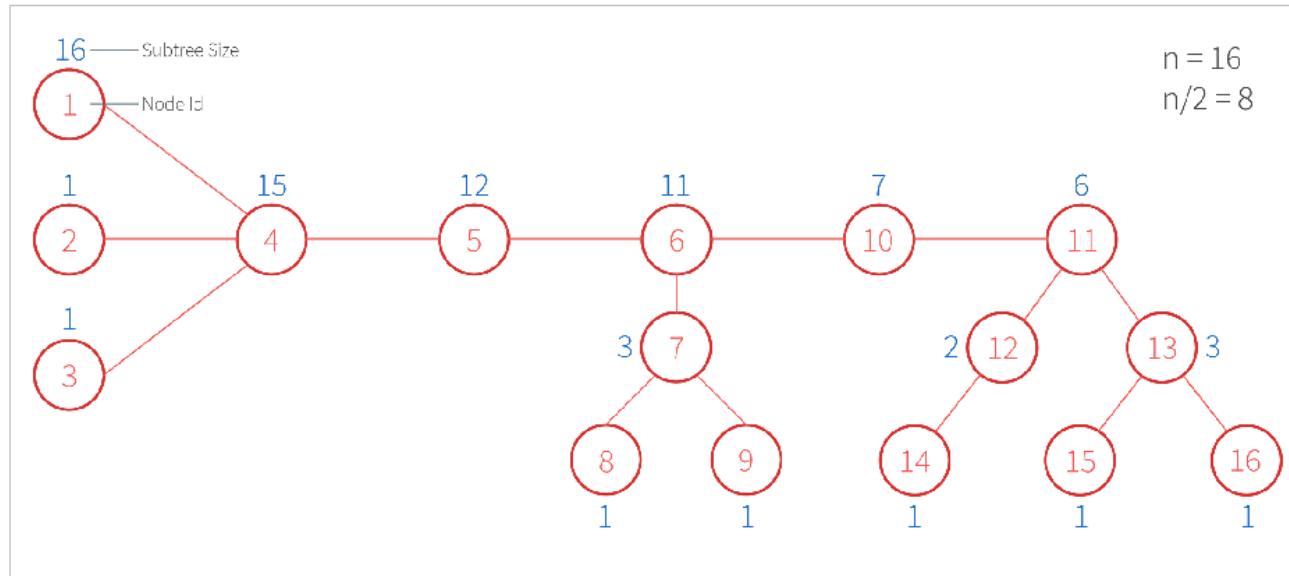
Algorithm

1. Make the centroid as the root of a new tree (which we will call as the ‘centroid tree’)
2. Recursively decompose the trees in the resulting forest
3. Make the centroids of these trees as children of the centroid which last split them.

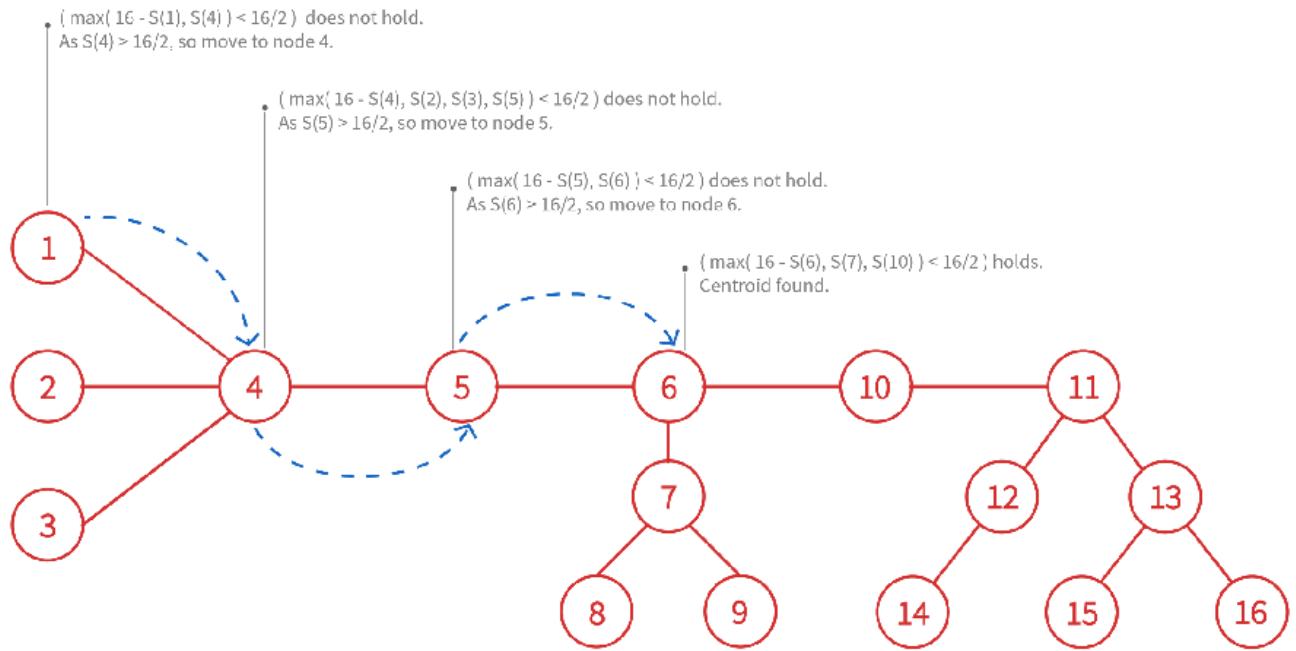
The centroid tree has depth $O(\lg n)$, and can be constructed in $O(n \lg n)$, as we can find the centroid in $O(n)$.

Illustrative Example

Let us consider a tree with 16 nodes. The figure has subtree sizes already set up using a DFS from node 1.

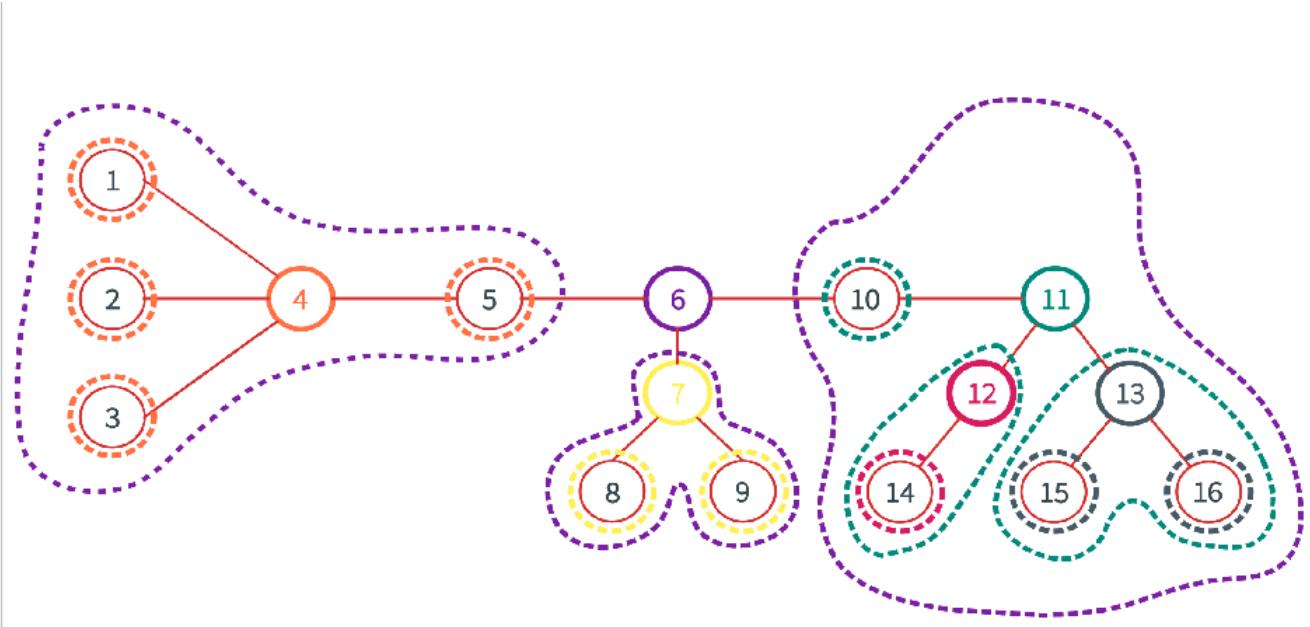


We start at node 1 and see if condition for centroid holds. Remember $S(v)$ is subtree size for v .

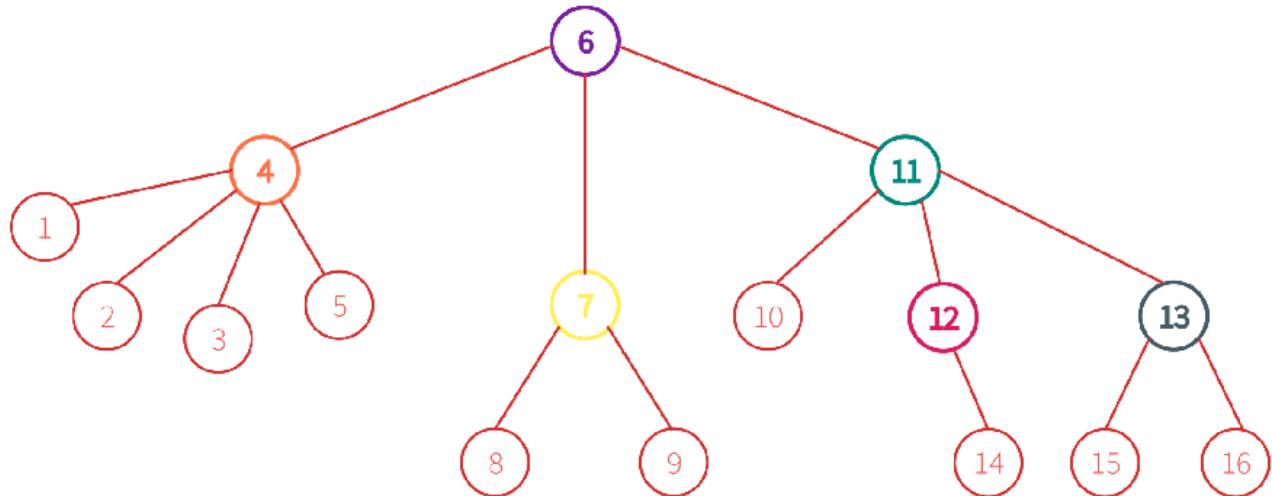


We make node 6 as the root of our centroid, and recurse on the 3 trees of the forest centroid split the original tree into.

NOTE: In the figure, subtrees generated by a centroid have been surrounded by a dotted line of the same color as the color of centroid.



We make the subsequently found centroids as the children to centroid that split them last, and obtain our centroid tree.



NOTE: The trees containing only a single element have the same element as their centroid. We haven't used color differentiation for such trees, and the leaf nodes represent them.

```

// C++ program for centroid decomposition of Tree
#include <bits/stdc++.h>
using namespace std;

#define MAXN 1025

vector<int> tree[MAXN];
vector<int> centroidTree[MAXN];
bool centroidMarked[MAXN];

/* method to add edge between two nodes of the undirected tree */
void addEdge(int u, int v)
{
    tree[u].push_back(v);
    tree[v].push_back(u);
}

/* method to setup subtree sizes and nodes in current tree */
void DFS(int src, bool visited[], int subtree_size[], int* n)
{
    /* mark node visited */
    visited[src] = true;

    /* increase count of nodes visited */
    *n += 1;

    for (int i = 0; i < tree[src].size(); i++)
        if (!visited[tree[src][i]])
            DFS(tree[src][i], visited, subtree_size, n);
}

```

```

/* initialize subtree size for current node*/
subtree_size[src] = 1;

vector<int>::iterator it;

/* recur on non-visited and non-centroid neighbours */
for (it = tree[src].begin(); it!=tree[src].end(); it++)
    if (!visited[*it] && !centroidMarked[*it])
    {
        DFS(*it, visited, subtree_size, n);
        subtree_size[src]+=subtree_size[*it];
    }
}

int getCentroid(int src, bool visited[], int subtree_size[], int n)
{
    /* assume the current node to be centroid */
    bool is_centroid = true;

    /* mark it as visited */
    visited[src] = true;

    /* track heaviest child of node, to use in case node is
       not centroid */
    int heaviest_child = 0;

    vector<int>::iterator it;

    /* iterate over all adjacent nodes which are children
       (not visited) and not marked as centroid to some
       subtree */
    for (it = tree[src].begin(); it!=tree[src].end(); it++)
        if (!visited[*it] && !centroidMarked[*it])
        {
            /* If any adjacent node has more than n/2 nodes,
               * current node cannot be centroid */
            if (subtree_size[*it]>n/2)
                is_centroid=false;

            /* update heaviest child */
            if (heaviest_child==0 ||
                subtree_size[*it]>subtree_size[heaviest_child])
                heaviest_child = *it;
        }

    /* if current node is a centroid */
    if (is_centroid && n-subtree_size[src]<=n/2)

```

```

        return src;

    /* else recur on heaviest child */
    return getCentroid(heaviest_child, visited, subtree_size, n);
}

/* function to get the centroid of tree rooted at src.
 * tree may be the original one or may belong to the forest */
int getCentroid(int src)
{
    bool visited[MAXN];

    int subtree_size[MAXN];

    /* initialize auxiliary arrays */
    memset(visited, false, sizeof visited);
    memset(subtree_size, 0, sizeof subtree_size);

    /* variable to hold number of nodes in the current tree */
    int n = 0;

    /* DFS to set up subtree sizes and nodes in current tree */
    DFS(src, visited, subtree_size, &n);

    for (int i=1; i<MAXN; i++)
        visited[i] = false;

    int centroid = getCentroid(src, visited, subtree_size, n);

    centroidMarked[centroid]=true;

    return centroid;
}

/* function to generate centroid tree of tree rooted at src */
int decomposeTree(int root)
{
    //printf("decomposeTree(%d)\n", root);

    /* get sentorid for current tree */
    int cend_tree = getCentroid(root);

    printf("%d ", cend_tree);

    vector<int>::iterator it;

    /* for every node adjacent to the found centroid
     * and not already marked as centroid */

```

```

for (it=tree[cend_tree].begin(); it!=tree[cend_tree].end(); it++)
{
    if (!centroidMarked[*it])
    {
        /* decompose subtree rooted at adjacent node */
        int cend_subtree = decomposeTree(*it);

        /* add edge between tree centroid and centroid of subtree */
        centroidTree[cend_tree].push_back(cend_subtree);
        centroidTree[cend_subtree].push_back(cend_tree);
    }
}

/* return centroid of tree */
return cend_tree;
}

// driver function
int main()
{
    /* number of nodes in the tree */
    int n = 16;

    /* arguments in order: node u, node v
     * sequencing starts from 1 */
    addEdge(1, 4);
    addEdge(2, 4);
    addEdge(3, 4);
    addEdge(4, 5);
    addEdge(5, 6);
    addEdge(6, 7);
    addEdge(7, 8);
    addEdge(7, 9);
    addEdge(6, 10);
    addEdge(10, 11);
    addEdge(11, 12);
    addEdge(11, 13);
    addEdge(12, 14);
    addEdge(13, 15);
    addEdge(13, 16);

    /* generates centroid tree */
    decomposeTree(1);

    return 0;
}

```

Output :

6 4 1 2 3 5 7 8 9 11 10 12 14 13 15 16

Application:

Consider below example problem

Given a weighted tree with N nodes, find the minimum number of edges in a path of length K, or return -1 if such a path does not exist.

1 <= N <= 200000
1 <= length(i;j) <= 1000000 (integer weights)
1 <= K <= 1000000

Brute force solution: For every node, perform DFS to find distance and number of edges to every other node

Time complexity: $O(N^2)$ Obviously inefficient because $N = 200000$

We can solve above problem in $O(N \log N)$ time **using Centroid Decomposition**.

1. Perform centroid decomposition to get a "tree of subtrees"
2. Start at the root of the decomposition, solve the problem for each subtree as follows
 - (a) Solve the problem for each "child tree" of the current subtree.
 - (b) Perform DFS from the centroid on the current subtree to compute the minimum edge count for paths that include the centroid
 - i. Two cases: centroid at the end or in the middle of path

Time complexity of centroid decomposition based solution is $O(n \log n)$

Reference :

<http://www.ugrad.cs.ubc.ca/~cs490/2014W2/pdf/jason.pdf>

Source

<https://www.geeksforgeeks.org/centroid-decomposition-of-tree/>

Chapter 23

Check if the given string of words can be formed from words present in the dictionary

Check if the given string of words can be formed from words present in the dictionary - GeeksforGeeks

Given a string array of M words and a dictionary of N words. The task is to check if the given string of words can be formed from words present in the dictionary.

Examples:

dict[] = { find, a, geeks, all, for, on, geeks, answers, inter }

Input: str[] = { "find", "all", "answers", "on", "geeks", "for", "geeks" };

Output: YES

all words of str[] are present in the dictionary so the output is YES

Input: str = {"find", "a", "geek"}

Output: NO

In str[], "find" and "a" were present in the dictionary but "geek" is not present in the dictionary so the output is NO

A **naive Approach** will be to match all words of the input sentence separately with each of the words in the dictionary and maintain a count of the number of occurrence of all words in the dictionary. So if the number of words in dictionary be n and no of words in the sentence be m this algorithm will take $O(M*N)$ time.

A **better approach** will be to use the modified version of the advanced data structure [Trie](#) the time complexity can be reduced to $O(M * t)$ where t is the length of longest word in the dictionary which is lesser than n. So here a modification has been done to the trie node such that the *isEnd* variable is now an integer storing the count of occurrence of the word ending on this node. Also, the search function has been modified to find a word in the trie

and once found decrease the count of isEnd of that node so that for multiple occurrences of a word in a sentence each is matched with a separate occurrence of that word in the dictionary.

Below is the illustration of the above approach:

```
// C++ program to check if a sentence
// can be formed from a given set of words.
#include <bits/stdc++.h>
using namespace std;
const int ALPHABET_SIZE = 26;

// here isEnd is an integer that will store
// count of words ending at that node
struct trieNode {
    trieNode* t[ALPHABET_SIZE];
    int isEnd;
};

// utility function to create a new node
trieNode* getNode()
{
    trieNode* temp = new (trieNode);

    // Initialize new node with null
    for (int i = 0; i < ALPHABET_SIZE; i++)
        temp->t[i] = NULL;
    temp->isEnd = 0;
    return temp;
}

// Function to insert new words in trie
void insert(trieNode* root, string key)
{
    trieNode* trail;
    trail = root;

    // Iterate for the length of a word
    for (int i = 0; i < key.length(); i++) {

        // If the next key does not contains the character
        if (trail->t[key[i] - 'a'] == NULL) {
            trieNode* temp;
            temp = getNode();
            trail->t[key[i] - 'a'] = temp;
        }
        trail = trail->t[key[i] - 'a'];
    }
}
```

```
// isEnd is increment so not only the word but its count is also stored
(trail->isEnd)++;
}
// Search function to find a word of a sentence
bool search_mod(trieNode* root, string word)
{
    trieNode* trail;
    trail = root;

    // Iterate for the complete length of the word
    for (int i = 0; i < word.length(); i++) {

        // If the character is not present then word
        // is also not present
        if (trail->t[word[i] - 'a'] == NULL)
            return false;

        // If present move to next character in Trie
        trail = trail->t[word[i] - 'a'];
    }

    // If word found then decrement count of the word
    if ((trail->isEnd) > 0 && trail != NULL) {
        // if the word is found decrement isEnd showing one
        // occurrence of this word is already taken so
        (trail->isEnd)--;
        return true;
    }
    else
        return false;
}
// Function to check if string can be
// formed from the sentence
void checkPossibility(string sentence[], int m, trieNode* root)
{
    int flag = 1;

    // Iterate for all words in the string
    for (int i = 0; i < m; i++) {

        if (search_mod(root, sentence[i]) == false) {

            // if a word is not found in a string then the
            // sentence cannot be made from this dictionary of words
            cout << "NO";

            return;
        }
    }
}
```

```
}

// If possible
cout << "YES";
}

// Function to insert all the words of dict in the Trie
void insertToTrie(string dictionary[], int n,
                   trieNode* root)
{

    for (int i = 0; i < n; i++)
        insert(root, dictionary[i]);
}

// Driver Code
int main()
{
    trieNode* root;
    root = getNode();

    // Dictionary of words
    string dictionary[] = { "find", "a", "geeks",
                           "all", "for", "on",
                           "geeks", "answers", "inter" };
    int N = sizeof(dictionary) / sizeof(dictionary[0]);

    // Calling Function to insert words of dictionary to tree
    insertToTrie(dictionary, N, root);

    // String to be checked
    string sentence[] = { "find", "all", "answers", "on",
                          "geeks", "for", "geeks" };

    int M = sizeof(sentence) / sizeof(sentence[0]);

    // Function call to check possibility
    checkPossibility(sentence, M, root);

    return 0;
}
```

Output:

YES

An **efficient approach** will be to use[map](#). Keep the count of words in the map, iterate in

the string and check if the word is present in the map. If present, then decrease the count of the word in the map. If it is not present, then it is not possible to make the given string from the given dictionary of words.

Below is the implementation of above approach :

```
// C++ program to check if a sentence
// can be formed from a given set of words.
#include <bits/stdc++.h>
using namespace std;

// Function to check if the word
// is in the dictionary or not
bool match_words(string dictionary[], string sentence[],
                  int n, int m)
{
    // map to store all words in
    // dictionary with their count
    unordered_map<string, int> mp;

    // adding all words in map
    for (int i = 0; i < n; i++) {
        mp[dictionary[i]]++;
    }

    // search in map for all
    // words in the sentence
    for (int i = 0; i < m; i++) {
        if (mp[sentence[i]])
            mp[sentence[i]] -= 1;
        else
            return false;
    }

    // all words of sentence are present
    return true;
}

// Driver Code
int main()
{
    string dictionary[] = { "find", "a", "geeks",
                           "all", "for", "on",
                           "geeks", "answers", "inter" };

    int n = sizeof(dictionary) / sizeof(dictionary[0]);

    string sentence[] = { "find", "all", "answers", "on",
                          "geeks", "for", "geeks" };
}
```

```
int m = sizeof(sentence) / sizeof(sentence[0]);  
  
// Calling function to check if words are  
// present in the dictionary or not  
if (match_words(dictionary, sentence, n, m))  
    cout << "YES";  
else  
    cout << "NO";  
  
return 0;  
}
```

Output:

YES

Time Complexity: O(M)

Source

<https://www.geeksforgeeks.org/check-if-the-given-string-of-words-can-be-formed-from-words-present-in-the-diction>

Chapter 24

Coalesced hashing

Coalesced hashing - GeeksforGeeks

[Coalesced hashing](#) is a collision avoidance technique when there is a fixed sized data. It is a combination of both [Separate chaining](#) and [Open addressing](#). It uses the concept of Open Addressing(linear probing) to find first empty place for colliding element from the bottom of the hash table and the concept of Separate Chaining to link the colliding elements to each other through pointers. The hash function used is $h=(key)\%(total\ number\ of\ keys)$. Inside the hash table, each node has three fields:

- $h(key)$: The value of hash function for a key.
- Data: The key itself.
- Next: The link to the next colliding elements.

The basic operations of Coalesced hashing are:

1. **INSERT(key):** The insert Operation inserts the key according to the hash value of that key if that hash value in the table is empty otherwise the key is inserted in first empty place from the bottom of the hash table and the address of this empty place is mapped in NEXT field of the previous pointing node of the chain.(Explained in example below).
2. **DELETE(Key):** The key if present is deleted.Also if the node to be deleted contains the address of another node in hash table then this address is mapped in the NEXT field of the node pointing to the node which is to be deleted
3. **SEARCH(key):** Returns **True** if key is present, otherwise return **False**.

The best case complexity of all these operations is $O(1)$ and the worst case complexity is $O(n)$ where n is the total number of keys.It is better than separate chaining because it inserts the colliding element in the memory of hash table only instead of creating a new linked list as in separate chaining.

Illustration:

Example:

```
n = 10  
Input : {20, 35, 16, 40, 45, 25, 32, 37, 22, 55}
```

Hash function

```
h(key) = key%10
```

Steps:

1. Initially empty hash table is created with all NEXT field initialised with NULL and h(key) values ranging from 0-9.
2. Let's start with inserting 20, as h(20)=0 and 0 index is empty so we insert 20 at 0 index.
3. Next element to be inserted is 35, h(35)=5 and 5th index empty so we insert 35 there.
4. Next we have 16, h(16)=6 which is empty so 16 is inserted at 6 index value.
5. Now we have to insert 40, h(40)=0 which is already occupied so we search for the first empty block from the bottom and insert it there i.e 9 index value. Also the address of this newly inserted node(from address we mean index value of a node) i.e(9)is initialised in the next field of 0th index value node.
6. To insert 45, h(45)=5 which is occupied so again we search for the empty block from the bottom i.e 8 index value and map the address of this newly inserted node i.e(8) to the Next field of 5th index value node i.e in the next field of key=35.
7. Next to insert 25, h(25)=5 is occupied so search for the first empty block from bottom i.e 7th index value and insert 25 there. Now it is important to note that the address of this new node cant be mapped on 5th index value node which is already pointing to some other node. To insert the address of new node we have to follow the link chain from the 5th index node until we get NULL in next field and map the address of new node to next field of that node i.e from 5th index node we go to 8th index node which contains NULL in next field so we insert address of new node i.e(7) in next field of 8th index node.
8. To insert 32, h(32)=2, which is empty so insert 32 at 2nd index value.
9. To insert 37, h(37)=7 which is occupied so search for the first free block from bottom which is 4th index value. So insert 37 at 4th index value and copy the address of this node in next field of 7th index value node.
10. To insert 22, h(22)=2 which is occupied so insert it at 3rd index value and map the address of this node in next field of 2nd index value node.

11. Finally, to insert 55 h(55)=5 which is occupied and the only empty space is 1st index value so insert 55 there. Now again to map the address of this new node we have to follow the chain starting from 5th index value node until we get NULL in next field i.e from 5th index->8th index->7th index->4th index which contains NULL in Next field, and we insert the address of newly inserted node at 4th index value node.

Hash value	Data	Next
0	20	9
1	55	NULL
2	32	3
3	22	NULL
4	37	1
5	35	8
6	16	NULL
7	25	4
8	45	7
9	40	NULL

Deletion process is simple, for example:

Case 1: To delete key=37, first search for 37. If it is present then simply delete the data value and if the node contains any address in next field and the node to be deleted i.e 37 is itself pointed by some other node(i.e key=25) then copy that address in the next field of 37 to the next field of node pointing to 37(i.e key=25) and initialize the NEXT field of key=37 as NULL again and erase the key=37.

Hash value	Data	Next
0	20	9
1	55	NULL
2	32	3
3	22	NULL
4	—	NULL
5	35	8
6	16	NULL
7	25	1
8	45	7
9	40	NULL

Case 2: If key to be deleted is 35 which is not pointed by any other node then we have to pull the chain attached to the node to be deleted i.e 35 one step back and mark last value of chain to NULL again.

Hash value	Data	Next
0	20	9
1	—	NULL

Hash value	Data	Next
2	32	3
3	22	NULL
4	—	NULL
5	45	8
6	16	NULL
7	55	NULL
8	25	7
9	40	NULL

Source

<https://www.geeksforgeeks.org/coalesced-hashing/>

Chapter 25

Comparisons involved in Modified Quicksort Using Merge Sort Tree

Comparisons involved in Modified Quicksort Using Merge Sort Tree - GeeksforGeeks

In [QuickSort](#), ideal situation is when median is always chosen as pivot as this results in minimum time. In this article, [Merge Sort Tree](#) is used to find median for different ranges in QuickSort and number of comparisons are analyzed.

Examples:

```
Input : arr = {4, 3, 5, 1, 2}
Output : 11
Explanation
We have to make 11 comparisons when we
apply quick sort to the array.
```

If we carefully analyze the quick sort algorithm then every time the array is given to the quick sort function, the array always consists of a permutation of the numbers in some range L to R. Initially, it's [1 to N], then its [1 to pivot – 1] and [pivot + 1 to N] and so on. Also it's not easy to observe that the relative ordering of numbers in every possible array does not change. Now in order to get the pivot element we just need to get the middle number i.e the $(r - 1 + 2)/2^{\text{th}}$ number among the array.

To do this efficiently we can use a [Persistent Segment Tree](#), a [Fenwick Tree](#), or a [Merge Sort Tree](#). This Article focuses on the Merge Sort Tree Implementation.

In the Modified Quick Sort Algorithm where we chose the pivot element of the array as the median of the array. Now, determining the median requires us to find the middle element considered, after sorting the array which is in itself a $O(n * \log(n))$ operation where n is the size of the array.

Let's say we have a range L to R then the median of this range is calculated as:

$$\begin{aligned}\text{Median of } A[L; R] &= \text{Middle element of sorted}(A[L; R]) \\ &= (R - L + 1)/2^{\text{th}} \text{ element of} \\ &\quad \text{sorted}(A[L; R])\end{aligned}$$

Let's consider we have P partitions during the quick sort algorithm which means we have to find the pivot by sorting the array range from L to R where L and R are the starting and ending points of each partition. This is costly.

But, we have a permutation of numbers from L to R in every partition, so we can just find the $\text{ceil}((R - L + 1)/2)^{\text{th}}$ smallest number in this range as we know when we would have sorted this partition then it would always have been this element that would have ended up as being the median element as a result also the pivot. Now the elements less than pivot go to the left subtree and the ones greater than it go in the right subtree also maintaining their order.

We repeat this procedure for all partitions P and find the comparisons involved in each partition. Since in the Current Partition all the elements from L to R of that partition are compared to the pivot, we have $(R - L + 1)$ comparisons in the current partition. We also need to consider, by recursively calculating, the total comparisons in the left and right subtrees formed too. Thus we conclude,

$$\begin{aligned}\text{Total Comparisons} &= \text{Comparisons in Current Partition} + \\ &\quad \text{Comparisons in Left partition} + \\ &\quad \text{Comparisons in right partition}\end{aligned}$$

We discussed above the approach to be used to find the pivot element efficiently here the [Kth order statistics using merge sort tree](#) can be used to find the same as discussed.

```
// CPP program to implement number of comparisons
// in modified quick sort
#include <bits/stdc++.h>
using namespace std;

const int MAX = 1000;

// Constructs a segment tree and stores tree[]
void buildTree(int treeIndex, int l, int r, int arr[],
               vector<int> tree[])
{
    /* l => start of range,
       r => ending of a range
       treeIndex => index in the Segment Tree/Merge
       Sort Tree */

```

```

/* leaf node */
if (l == r) {
    tree[treeIndex].push_back(arr[l - 1]);
    return;
}

int mid = (l + r) / 2;

/* building left subtree */
buildTree(2 * treeIndex, l, mid, arr, tree);

/* building left subtree */
buildTree(2 * treeIndex + 1, mid + 1, r, arr, tree);

/* merging left and right child in sorted order */
merge(tree[2 * treeIndex].begin(),
      tree[2 * treeIndex].end(),
      tree[2 * treeIndex + 1].begin(),
      tree[2 * treeIndex + 1].end(),
      back_inserter(tree[treeIndex]));
}

// Returns the Kth smallest number in query range
int queryRec(int segmentStart, int segmentEnd,
             int queryStart, int queryEnd, int treeIndex,
             int K, vector<int> tree[])
{
    /* segmentStart => start of a Segment,
       segmentEnd   => ending of a Segment,
       queryStart   => start of a query range,
       queryEnd     => ending of a query range,
       treeIndex     => index in the Segment
                           Tree/Merge Sort Tree,
       K   => kth smallest number to find */
    if (segmentStart == segmentEnd)
        return tree[treeIndex][0];

    int mid = (segmentStart + segmentEnd) / 2;

    // finds the last index in the segment
    // which is <= queryEnd
    int last_in_query_range =
        (upper_bound(tree[2 * treeIndex].begin(),
                    tree[2 * treeIndex].end(), queryEnd)
         - tree[2 * treeIndex].begin());

    // finds the first index in the segment
    // which is >= queryStart
}

```

```

int first_in_query_range =
    (lower_bound(tree[2 * treeIndex].begin(),
        tree[2 * treeIndex].end(), queryStart)
     - tree[2 * treeIndex].begin());

int M = last_in_query_range - first_in_query_range;

if (M >= K) {

    // Kth smallest is in left subtree,
    // so recursively call left subtree for Kth
    // smallest number
    return queryRec(segmentStart, mid, queryStart,
                    queryEnd, 2 * treeIndex, K, tree);
}

else {

    // Kth smallest is in right subtree,
    // so recursively call right subtree for the
    // (K-M)th smallest number
    return queryRec(mid + 1, segmentEnd, queryStart,
                    queryEnd, 2 * treeIndex + 1, K - M, tree);
}

// A wrapper over query()
int query(int queryStart, int queryEnd, int K, int n, int arr[],
          vector<int> tree[])
{

    return queryRec(1, n, queryStart, queryEnd,
                    1, K, tree);
}

/* Calculates total Comparisons Involved in Quick Sort
   Has the following parameters:

   start => starting index of array
   end   => ending index of array
   n     => size of array
   tree  => Merge Sort Tree */

int quickSortComparisons(int start, int end, int n, int arr[],
                         vector<int> tree[])
{
    /* Base Case */
    if (start >= end)

```

```
    return 0;

    // Compute the middle point of range and the pivot
    int middlePoint = (end - start + 2) / 2;
    int pivot = query(start, end, middlePoint, n, arr, tree);

    /* Total Comparisons = (Comparisons in Left part +
                           Comparisons of right +
                           Comparisons in parent) */

    // count comparisons in parent array
    int comparisons_in_parent = (end - start + 1);

    // count comparisons involved in left partition
    int comparisons_in_left_part =
        quickSortComparisons(start, pivot - 1, n, arr, tree);

    // count comparisons involved in right partition
    int comparisons_in_right_part =
        quickSortComparisons(pivot + 1, end, n, arr, tree);

    // Return Total Comparisons
    return comparisons_in_left_part +
           comparisons_in_parent +
           comparisons_in_right_part;
}

// Driver code
int main()
{
    int arr[] = { 4, 3, 5, 1, 2 };

    int n = sizeof(arr) / sizeof(arr[0]);

    // Construct segment tree in tree[]
    vector<int> tree[MAX];
    buildTree(1, 1, n, arr, tree);

    cout << "Number of Comparisons = "
        << quickSortComparisons(1, n, n, arr, tree);;

    return 0;
}
```

Output:

```
Number of Comparisons = 11
```

Complexity is $O(\log^2(n))$ per query for computing pivot

Source

<https://www.geeksforgeeks.org/comparisons-involved-modified-quicksort-using-merge-sort-tree/>

Chapter 26

Convert an Array to a Circular Doubly Linked List

Convert an Array to a Circular Doubly Linked List - GeeksforGeeks

Prerequisite: [Doubly Linked list](#), [Circular Linked List](#), [Circular Doubly Linked List](#)

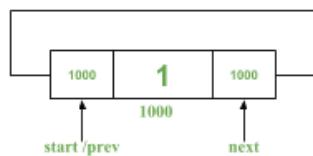
Given an array of N elements. The task is to write a program to convert the array into a [circular doubly linked list](#).

Array:

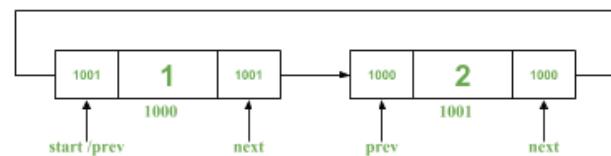
1	2	3
---	---	---

Linked List to be created:

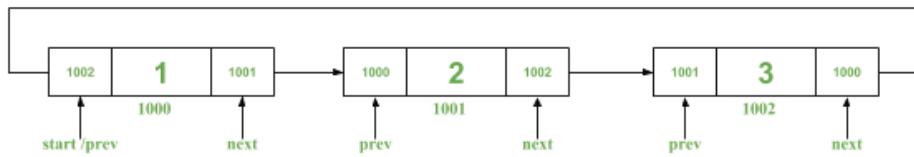
Iteration 1:



Iteration 2:



Iteration 3:



The idea is to start traversing the array and for every array element create a new list node and assign the *prev* and *next* pointers of this node accordingly.

- Create a pointer *start* to point to the starting of the list which will initially point to NULL(Empty list).
- For the first element of the array, create a new node and put that node's *prev* and *next* pointers to point to start to maintain the circular fashion of the list.
- For the rest of the array elements, insert those elements to the end of the created circular doubly linked list.

Below is the implementation of above idea:

```
// CPP program to convert array to
// circular doubly linked list

#include<iostream>
using namespace std;

// Doubly linked list node
struct node
{
    int data;
    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;
    }
}
```

```
        return 1;
    }
}

// Function to convert array into list
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 to be converted  
    int arr[] = {1,2,3,4,5};  
    int n = sizeof(arr) / sizeof(arr[0]);  
  
    // Start Pointer  
    struct node *start = NULL;  
  
    // Create the List  
    createList(arr, n, &start);  
  
    // Display the list  
    displayList(start);  
  
    return 0;  
}
```

Output:

The list is: 1 2 3 4 5

Source

<https://www.geeksforgeeks.org/convert-array-to-circular-doubly-linked-list/>

Chapter 27

Count Inversions of size three in a given array

Count Inversions of size three in a given array - GeeksforGeeks

Given an array arr[] of size n. Three elements arr[i], arr[j] and arr[k] form an inversion of size 3 if $a[i] > a[j] > a[k]$ and $i < j < k$. Find total number of inversions of size 3.

Example :

Input: {8, 4, 2, 1}

Output: 4

The four inversions are (8,4,2), (8,4,1), (4,2,1) and (8,2,1).

Input: {9, 6, 4, 5, 8}

Output: 2

The two inversions are {9, 6, 4} and {9, 6, 5}

We have already discussed inversion count of size two by [merge sort](#), [Self Balancing BST](#) and [BIT](#).

Simple approach :- Loop for all possible value of i, j and k and check for the condition $a[i] > a[j] > a[k]$ and $i < j < k$.

C++

```
// A Simple C++ O(n^3) program to count inversions of size 3
#include<bits/stdc++.h>
using namespace std;

// Returns counts of inversions of size three
int getInvCount(int arr[], int n)
{
```

```
int invcount = 0; // Initialize result

for (int i=0; i<n-2; i++)
{
    for (int j=i+1; j<n-1; j++)
    {
        if (arr[i]>arr[j])
        {
            for (int k=j+1; k<n; k++)
            {
                if (arr[j]>arr[k])
                    invcount++;
            }
        }
    }
}
return invcount;
}

// Driver program to test above function
int main()
{
    int arr[] = {8, 4, 2, 1};
    int n = sizeof(arr)/sizeof(arr[0]);
    cout << "Inversion Count : " << getInvCount(arr, n);
    return 0;
}
```

Java

```
// A simple Java implementation to count inversion of size 3
class Inversion{

    // returns count of inversion of size 3
    int getInvCount(int arr[], int n)
    {
        int invcount = 0; // initialize result

        for(int i=0 ; i< n-2; i++)
        {
            for(int j=i+1; j<n-1; j++)
            {
                if(arr[i] > arr[j])
                {
                    for(int k=j+1; k<n; k++)
                    {
                        if(arr[j] > arr[k])
                            invcount++;
                }
            }
        }
    }
}
```

```
        }
    }
}
return invcount;
}

// driver program to test above function
public static void main(String args[])
{
    Inversion inversion = new Inversion();
    int arr[] = new int[] {8, 4, 2, 1};
    int n = arr.length;
    System.out.print("Inversion count : " +
                      inversion.getInvCount(arr, n));
}
}

// This code is contributed by Mayank Jaiswal
```

Python

```
# A simple python O(n^3) program
# to count inversions of size 3

# Returns counts of inversions
# of size threee
def getInvCount(arr):
    n = len(arr)
    invcount = 0 #Initialize result
    for i in range(0,n-1):
        for j in range(i+1 , n):
            if arr[i] > arr[j]:
                for k in range(j+1 , n):
                    if arr[j] > arr[k]:
                        invcount += 1
    return invcount

# Driver program to test above function
arr = [8 , 4, 2 , 1]
print "Inversion Count : %d" %(getInvCount(arr))

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

C#

```
// A simple C# implementation to
// count inversion of size 3
```

```
using System;
class GFG {

    // returns count of inversion of size 3
    static int getInvCount(int []arr, int n)
    {

        // initialize result
        int invcount = 0;

        for(int i = 0 ; i < n - 2; i++)
        {
            for(int j = i + 1; j < n - 1; j++)
            {
                if(arr[i] > arr[j])
                {
                    for(int k = j + 1; k < n; k++)
                    {
                        if(arr[j] > arr[k])
                            invcount++;
                    }
                }
            }
        }
        return invcount;
    }

    // Driver Code
    public static void Main()
    {
        int []arr = new int[] {8, 4, 2, 1};
        int n = arr.Length;
        Console.WriteLine("Inversion count : " +
                           getInvCount(arr, n));
    }
}

// This code is contributed by anuj_67.
```

PHP

```
<?php
// A O(n^2) PHP program to
// count inversions of size 3

// Returns count of
// inversions of size 3
```

```
function getInvCount($arr, $n)
{
    // Initialize result
    $invcount = 0;

    for ($i = 1; $i < $n - 1; $i++)
    {
        // Count all smaller elements
        // on right of arr[i]
        $small = 0;
        for($j = $i + 1; $j < $n; $j++)
            if ($arr[$i] > $arr[$j])
                $small++;

        // Count all greater elements
        // on left of arr[i]
        $great = 0;
        for($j = $i - 1; $j >= 0; $j--)
            if ($arr[$i] < $arr[$j])
                $great++;

        // Update inversion count by
        // adding all inversions
        // that have arr[i] as
        // middle of three elements
        $invcount += $great * $small;
    }

    return $invcount;
}

// Driver Code
$arr = array(8, 4, 2, 1);
$n = sizeof($arr);
echo "Inversion Count : "
    , getInvCount($arr, $n);

// This code is contributed m_kit
?>
```

Output:

```
Inversion Count : 4
```

Time complexity of this approach is : $O(n^3)$

Better Approach :

We can reduce the complexity if we consider every element $arr[i]$ as middle element of inversion, find all the numbers greater than $a[i]$ whose index is less than i , find all the numbers which are smaller than $a[i]$ and index is more than i . We multiply the number of elements greater than $a[i]$ to the number of elements smaller than $a[i]$ and add it to the result.

Below is the implementation of the idea.

C++

```
// A O(n^2) C++ program to count inversions of size 3
#include<bits/stdc++.h>
using namespace std;

// Returns count of inversions of size 3
int getInvCount(int arr[], int n)
{
    int invcount = 0; // Initialize result

    for (int i=1; i<n-1; i++)
    {
        // Count all smaller elements on right of arr[i]
        int small = 0;
        for (int j=i+1; j<n; j++)
            if (arr[i] > arr[j])
                small++;

        // Count all greater elements on left of arr[i]
        int great = 0;
        for (int j=i-1; j>=0; j--)
            if (arr[i] < arr[j])
                great++;

        // Update inversion count by adding all inversions
        // that have arr[i] as middle of three elements
        invcount += great*small;
    }

    return invcount;
}

// Driver program to test above function
int main()
{
    int arr[] = {8, 4, 2, 1};
    int n = sizeof(arr)/sizeof(arr[0]);
    cout << "Inversion Count : " << getInvCount(arr, n);
}
```

```
    return 0;
}
```

Java

```
// A O(n^2) Java program to count inversions of size 3

class Inversion {

    // returns count of inversion of size 3
    int getInvCount(int arr[], int n)
    {
        int invcount = 0; // initialize result

        for (int i=0 ; i< n-1; i++)
        {
            // count all smaller elements on right of arr[i]
            int small=0;
            for (int j=i+1; j<n; j++)
                if (arr[i] > arr[j])
                    small++;

            // count all greater elements on left of arr[i]
            int great = 0;
            for (int j=i-1; j>=0; j--)
                if (arr[i] < arr[j])
                    great++;

            // update inversion count by adding all inversions
            // that have arr[i] as middle of three elements
            invcount += great*small;
        }
        return invcount;
    }

    // driver program to test above function
    public static void main(String args[])
    {
        Inversion inversion = new Inversion();
        int arr[] = new int[] {8, 4, 2, 1};
        int n = arr.length;
        System.out.print("Inversion count : " +
                        inversion.getInvCount(arr, n));
    }
}

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

Python3

```
# A O(n^2) Python3 program to
# count inversions of size 3

# Returns count of inversions
# of size 3
def getInvCount(arr, n):

    # Initialize result
    invcount = 0

    for i in range(1,n-1):

        # Count all smaller elements
        # on right of arr[i]
        small = 0
        for j in range(i+1 ,n):
            if (arr[i] > arr[j]):
                small+=1

        # Count all greater elements
        # on left of arr[i]
        great = 0;
        for j in range(i-1,-1,-1):
            if (arr[i] < arr[j]):
                great+=1

        # Update inversion count by
        # adding all inversions that
        # have arr[i] as middle of
        # three elements
        invcount += great * small

    return invcount

# Driver program to test above function
arr = [8, 4, 2, 1]
n = len(arr)
print("Inversion Count : ",getInvCount(arr, n))

# This code is Contributed by Smitha Dinesh Semwal

C#
// A O(n^2) Java program to count inversions
// of size 3
using System;

public class Inversion {
```

```
// returns count of inversion of size 3
static int getInvCount(int []arr, int n)
{
    int invcount = 0; // initialize result

    for (int i = 0 ; i < n-1; i++)
    {

        // count all smaller elements on
        // right of arr[i]
        int small = 0;
        for (int j = i+1; j < n; j++)
            if (arr[i] > arr[j])
                small++;

        // count all greater elements on
        // left of arr[i]
        int great = 0;
        for (int j = i-1; j >= 0; j--)
            if (arr[i] < arr[j])
                great++;

        // update inversion count by
        // adding all inversions that
        // have arr[i] as middle of
        // three elements
        invcount += great * small;
    }

    return invcount;
}

// driver program to test above function
public static void Main()
{

    int []arr = new int[] {8, 4, 2, 1};
    int n = arr.Length;
    Console.WriteLine("Inversion count : "
                      + getInvCount(arr, n));
}
}

// This code has been contributed by anuj_67.
```

PHP

```
<?php
// A O(n^2) PHP program to count
// inversions of size 3

// Returns count of
// inversions of size 3
function getInvCount($arr, $n)
{
    // Initialize result
    $invcount = 0;

    for ($i = 1; $i < $n - 1; $i++)
    {
        // Count all smaller elements
        // on right of arr[i]
        $small = 0;
        for ($j = $i + 1; $j < $n; $j++)
            if ($arr[$i] > $arr[$j])
                $small++;

        // Count all greater elements
        // on left of arr[i]
        $great = 0;
        for ($j = $i - 1; $j >= 0; $j--)
            if ($arr[$i] < $arr[$j])
                $great++;

        // Update inversion count by
        // adding all inversions that
        // have arr[i] as middle of
        // three elements
        $invcount += $great * $small;
    }

    return $invcount;
}

// Driver Code
$arr = array (8, 4, 2, 1);
$n = sizeof($arr);
echo "Inversion Count : " ,
     getInvCount($arr, $n);

// This code is contributed by m_kit
?>
```

Output :

Inversion Count : 4

Time Complexity of this approach : $O(n^2)$

Binary Indexed Tree Approach :

Like inversions of size 2, we can use Binary indexed tree to find inversions of size 3. It is strongly recommended to refer below article first.

[Count inversions of size two Using BIT](#)

The idea is similar to above method. We count the number of greater elements and smaller elements for all the elements and then multiply greater[] to smaller[] and add it to the result.

Solution :

1. To find out the number of smaller elements for an index we iterate from n-1 to 0. For every element $a[i]$ we calculate the `getSum()` function for $(a[i]-1)$ which gives the number of elements till $a[i]-1$.
2. To find out the number of greater elements for an index we iterate from 0 to n-1. For every element $a[i]$ we calculate the sum of numbers till $a[i]$ (sum smaller or equal to $a[i]$) by `getSum()` and subtract it from i (as i is the total number of element till that point) so that we can get number of elements greater than $a[i]$.

Source

<https://www.geeksforgeeks.org/count-inversions-of-size-three-in-a-give-array/>

Chapter 28

Count and Toggle Queries on a Binary Array

Count and Toggle Queries on a Binary Array - GeeksforGeeks

Given a size n in which initially all elements are 0. The task is to perform multiple multiple queries of following two types. The queries can appear in any order.

1. **toggle(start, end)** : Toggle (0 into 1 or 1 into 0) the values from range ‘start’ to ‘end’.
2. **count(start, end)** : Count the number of 1’s within given range from ‘start’ to ‘end’.

```
Input : n = 5      // we have n = 5 blocks
        toggle 1 2 // change 1 into 0 or 0 into 1
        Toggle 2 4
        Count 2 3 // count all 1's within the range
        Toggle 2 4
        Count 1 4 // count all 1's within the range
Output : Total number of 1's in range 2 to 3 is = 1
         Total number of 1's in range 1 to 4 is = 2
```

A simple solution for this problem is to traverse the complete range for “Toggle” query and when you get “Count” query then count all the 1’s for given range. But the time complexity for this approach will be $O(q*n)$ where q=total number of queries.

An efficient solution for this problem is to use [Segment Tree](#) with [Lazy Propagation](#). Here we collect the updates until we get a query for “Count”. When we get the query for “Count”, we make all the previously collected Toggle updates in array and then count number of 1’s with in the given range.

```

// C++ program to implement toggle and count
// queries on a binary array.
#include<bits/stdc++.h>
using namespace std;
const int MAX = 100000;

// segment tree to store count of 1's within range
int tree[MAX] = {0};

// bool type tree to collect the updates for toggling
// the values of 1 and 0 in given range
bool lazy[MAX] = {false};

// function for collecting updates of toggling
// node --> index of current node in segment tree
// st --> starting index of current node
// en --> ending index of current node
// us --> starting index of range update query
// ue --> ending index of range update query
void toggle(int node, int st, int en, int us, int ue)
{
    // If lazy value is non-zero for current node of segment
    // tree, then there are some pending updates. So we need
    // to make sure that the pending updates are done before
    // making new updates. Because this value may be used by
    // parent after recursive calls (See last line of this
    // function)
    if (lazy[node])
    {
        // Make pending updates using value stored in lazy nodes
        lazy[node] = false;
        tree[node] = en - st + 1 - tree[node];

        // checking if it is not leaf node because if
        // it is leaf node then we cannot go further
        if (st < en)
        {
            // We can postpone updating children we don't
            // need their new values now.
            // Since we are not yet updating children of 'node',
            // we need to set lazy flags for the children
            lazy[node<<1] = !lazy[node<<1];
            lazy[1+(node<<1)] = !lazy[1+(node<<1)];
        }
    }

    // out of range
    if (st>en || us > en || ue < st)

```

```

        return ;

    // Current segment is fully in range
    if (us<=st && en<=ue)
    {
        // Add the difference to current node
        tree[node] = en-st+1 - tree[node];

        // same logic for checking leaf node or not
        if (st < en)
        {
            // This is where we store values in lazy nodes,
            // rather than updating the segment tree itself
            // Since we don't need these updated values now
            // we postpone updates by storing values in lazy[]
            lazy[node<<1] = !lazy[node<<1];
            lazy[1+(node<<1)] = !lazy[1+(node<<1)];
        }
        return;
    }

    // If not completely in rang, but overlaps, recur for
    // children,
    int mid = (st+en)/2;
    toggle((node<<1), st, mid, us, ue);
    toggle((node<<1)+1, mid+1,en, us, ue);

    // And use the result of children calls to update this node
    if (st < en)
        tree[node] = tree[node<<1] + tree[(node<<1)+1];
    }

/* node --> Index of current node in the segment tree.
   Initially 0 is passed as root is always at'
   index 0
   st & en --> Starting and ending indexes of the
   segment represented by current node,
   i.e., tree[node]
   qs & qe --> Starting and ending indexes of query
   range */

// function to count number of 1's within given range
int countQuery(int node, int st, int en, int qs, int qe)
{
    // current node is out of range
    if (st>en || qs > en || qe < st)
        return 0;

    // If lazy flag is set for current node of segment tree,

```

```

// then there are some pending updates. So we need to
// make sure that the pending updates are done before
// processing the sub sum query
if (lazy[node])
{
    // Make pending updates to this node. Note that this
    // node represents sum of elements in arr[st..en] and
    // all these elements must be increased by lazy[node]
    lazy[node] = false;
    tree[node] = en-st+1-tree[node];

    // checking if it is not leaf node because if
    // it is leaf node then we cannot go further
    if (st<en)
    {
        // Since we are not yet updating children os si,
        // we need to set lazy values for the children
        lazy[(node<<1)] = !lazy[node<<1];
        lazy[(node<<1)+1] = !lazy[(node<<1)+1];
    }
}

// At this point we are sure that pending lazy updates
// are done for current node. So we can return value
// If this segment lies in range
if (qs<=st && en<=qe)
    return tree[node];

// If a part of this segment overlaps with the given range
int mid = (st+en)/2;
return countQuery((node<<1), st, mid, qs, qe) +
       countQuery((node<<1)+1, mid+1, en, qs, qe);
}

// Driver program to run the case
int main()
{
    int n = 5;
    toggle(1, 0, n-1, 1, 2); // Toggle 1 2
    toggle(1, 0, n-1, 2, 4); // Toggle 2 4

    cout << countQuery(1, 0, n-1, 2, 3) << endl; // Count 2 3

    toggle(1, 0, n-1, 2, 4); // Toggle 2 4

    cout << countQuery(1, 0, n-1, 1, 4) << endl; // Count 1 4

    return 0;
}

```

}

Output:

1
2

Source

<https://www.geeksforgeeks.org/count-toggle-queries-binary-array/>

Chapter 29

Count elements which divide all numbers in range L-R

Count elements which divide all numbers in range L-R - GeeksforGeeks

Given N numbers and Q queries, each query consists of L and R. Task is to write a program which prints the count of numbers which divides all numbers in the given range L-R.

Examples :

```
Input : a = {3, 4, 2, 2, 4, 6}
        Q = 2
        L = 1 R = 4
        L = 2 R = 6

Output : 0
        2
```

Explanation : The range 1-4 has {3, 4, 2, 2} which does not have any number that divides all the numbers in this range.

The range 2-6 has {4, 2, 2, 4, 6} which has 2 numbers {2, 2} which divides all numbers in the given range.

```
Input: a = {1, 2, 3, 5}
      Q = 2
      L = 1 R = 4
      L = 2 R = 4
Output: 1
      0
```

Naive approach : Iterate from range L-R for every query and check if the given element

at index-i divide all the numbers in the range. We keep a count for of all the elements which divides all the numbers. The complexity of every query at worst case will be $O(n^2)$.

Below is the implementation of Naive Approach :

```
// CPP program to Count elements which
// divides all numbers in range L-R
#include <bits/stdc++.h>
using namespace std;

// function to count element
// Time complexity O(n^2) worst case
int answerQuery(int a[], int n,
                 int l, int r)
{
    // answer for query
    int count = 0;

    // 0 based index
    l = l - 1;

    // iterate for all elements
    for (int i = l; i < r; i++)
    {
        int element = a[i];
        int divisors = 0;

        // check if the element divides
        // all numbers in range
        for (int j = l; j < r; j++)
        {
            // no of elements
            if (a[j] % a[i] == 0)
                divisors++;
            else
                break;
        }

        // if all elements are divisible by a[i]
        if (divisors == (r - l))
            count++;
    }

    // answer for every query
    return count;
}

// Driver Code
int main()
```

```
{  
    int a[] = { 1, 2, 3, 5 };  
    int n = sizeof(a) / sizeof(a[0]);  
  
    int l = 1, r = 4;  
    cout << answerQuery(a, n, l, r) << endl;  
  
    l = 2, r = 4;  
    cout << answerQuery(a, n, l, r) << endl;  
    return 0;  
}
```

Output:

```
1  
0
```

Efficient approach : Use [Segment Trees](#) to solve this problem. If an element divides all the numbers in a given range, then the element is the minimum number in that range and it is the gcd of all elements in the given range L-R. So the count of the number of minimums in range L-R, given that minimum is equal to the gcd of that range will be our answer to every query. The problem boils down to finding the [GCD](#), [MINIMUM](#) and [countMINIMUM](#) for every range using Segment trees. On every node of the tree, three values are stored.

On querying for a given range, if the gcd and minimum of the given range are equal, [count-MINIMUM](#) is returned as the answer. If they are unequal, 0 is returned as the answer.

Below is the implementation of efficient approach :

```
// CPP program to Count elements  
// which divides all numbers in  
// range L-R efficient approach  
#include <bits/stdc++.h>  
using namespace std;  
  
#define N 100005  
  
// predefines the tree with nodes  
// storing gcd, min and count  
struct node  
{  
    int gcd;  
    int min;  
    int cnt;  
} tree[5 * N];
```

```
// function to construct the tree
void buildtree(int low, int high,
               int pos, int a[])
{
    // base condition
    if (low == high)
    {
        // initially always gcd and min
        // are same at leaf node
        tree[pos].min = tree[pos].gcd = a[low];
        tree[pos].cnt = 1;

        return;
    }

    int mid = (low + high) >> 1;

    // left-subtree
    buildtree(low, mid, 2 * pos + 1, a);

    // right-subtree
    buildtree(mid + 1, high, 2 * pos + 2, a);

    // finds gcd of left and right subtree
    tree[pos].gcd = __gcd(tree[2 * pos + 1].gcd,
                          tree[2 * pos + 2].gcd);

    // left subtree has the minimum element
    if (tree[2 * pos + 1].min < tree[2 * pos + 2].min)
    {
        tree[pos].min = tree[2 * pos + 1].min;
        tree[pos].cnt = tree[2 * pos + 1].cnt;
    }

    // right subtree has the minimum element
    else
    if (tree[2 * pos + 1].min > tree[2 * pos + 2].min)
    {
        tree[pos].min = tree[2 * pos + 2].min;
        tree[pos].cnt = tree[2 * pos + 2].cnt;
    }

    // both subtree has the same minimum element
    else
    {
        tree[pos].min = tree[2 * pos + 1].min;
        tree[pos].cnt = tree[2 * pos + 1].cnt +
```

```

        tree[2 * pos + 2].cnt;
    }
}

// function that answers every query
node query(int s, int e, int low, int high, int pos)
{
    node dummy;

    // out of range
    if (e < low or s > high)
    {
        dummy.gcd = dummy.min = dummy.cnt = 0;
        return dummy;
    }

    // in range
    if (s >= low and e <= high)
    {
        node dummy;
        dummy.gcd = tree[pos].gcd;
        dummy.min = tree[pos].min;
        if (dummy.gcd != dummy.min)
            dummy.cnt = 0;
        else
            dummy.cnt = tree[pos].cnt;

        return dummy;
    }

    int mid = (s + e) >> 1;

    // left-subtree
    node ans1 = query(s, mid, low,
                      high, 2 * pos + 1);

    // right-subtree
    node ans2 = query(mid + 1, e, low,
                      high, 2 * pos + 2);

    node ans;

    // when both left subtree and
    // right subtree is in range
    if (ans1.gcd and ans2.gcd)
    {
        // merge two trees
        ans.gcd = __gcd(ans1.gcd, ans2.gcd);
    }
}

```

```
ans.min = min(ans1.min, ans2.min);

// when gcd is not equal to min
if (ans.gcd != ans.min)
    ans.cnt = 0;
else
{
    // add count when min is
    // same of both subtree
    if (ans1.min == ans2.min)
        ans.cnt = ans2.cnt + ans1.cnt;

    // store the minimal's count
    else
        if (ans1.min < ans2.min)
            ans.cnt = ans1.cnt;
        else
            ans.cnt = ans2.cnt;
}

return ans;
}

// only left subtree is in range
else if (ans1.gcd)
    return ans1;

// only right subtree is in range
else if (ans2.gcd)
    return ans2;
}

// function to answer query in range l-r
int answerQuery(int a[], int n, int l, int r)
{
    // calls the function which returns
    // a node this function returns the
    // count which will be the answer
    return query(0, n - 1, l - 1, r - 1, 0).cnt;
}

// Driver Code
int main()
{
    int a[] = { 3, 4, 2, 2, 4, 6 };

    int n = sizeof(a) / sizeof(a[0]);
    buildtree(0, n - 1, 0, a);
```

```
int l = 1, r = 4;

// answers 1-st query
cout << answerQuery(a, n, l, r) << endl;

l = 2, r = 6;
// answers 2nd query
cout << answerQuery(a, n, l, r) << endl;
return 0;
}
```

Output:

```
0
2
```

Time Complexity: Time Complexity for tree construction is **O(n logn)** since tree construction takes O(n) and finding out gcd takes O(log n). The time taken for every query in worst case will be **O(log n * log n)** since the inbuilt function `__gcd` takes **O(log n)**

Source

<https://www.geeksforgeeks.org/count-elements-which-divide-all-numbers-in-range-l-r/>

Chapter 30

Count greater nodes in AVL tree

Count greater nodes in AVL tree - GeeksforGeeks

In this article we will see that how to calculate number of elements which are greater than given value in [AVL tree](#).

Examples:

```
Input : x = 5
        Root of below AVL tree
          9
         / \
        1   10
       / \   \
      0   5   11
     /   / \
    -1  2   6
Output : 4
```

Explanation: there are 4 values which are greater than 5 in AVL tree which are 6, 9, 10 and 11.

Prerequisites :

- [Insertion in AVL tree](#)
- [Deletion in AVL tree](#)

1. We maintain an extra field ‘desc’ for storing the number of descendant nodes for every node. Like for above example node having value 5 has a desc field value equal to 2.

2. for calculating the number of nodes which are greater than given value we simply traverse the tree. While traversing three cases can occur-

I Case- x(given value) is greater than the value of current node. So, we go to the right child of the current node.

II Case- x is lesser than the value of current node. we increase the current count by number of successors of the right child of the current node and then again add two to the current count(one for the current node and one for the right child.). In this step first, we make sure that right child exists or not. Then we move to left child of current node.

III Case- x is equal to the value of current node. In this case we add the value of **desc** field of right child of current node to current count and then add one to it (for counting right child). Also in this case we see that right child exists or not.

Calculating values of desc field

1. **Insertion** – When we insert a node we increment one to child field of every predecessor of the new node. In the leftRotate and rightRotate functions we make appropriate changes in the value of child fields of nodes.
2. **Deletion** – When we delete a node then we decrement one from every predecessor node of deleted node. Again, In the leftRotate and rightRotate functions we make appropriate changes in the value of child fields of nodes.

```
// C program to find number of elements
// greater than a given value in AVL
#include <stdio.h>
#include <stdlib.h>
struct Node {
    int key;
    struct Node* left, *right;
    int height;
    int desc;
};

int height(struct Node* N)
{
    if (N == NULL)
        return 0;
    return N->height;
}

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

struct Node* newNode(int key)
```

```
{
    struct Node* node = (struct Node*)
        malloc(sizeof(struct Node));
    node->key = key;
    node->left = NULL;
    node->right = NULL;
    node->height = 1; // initially added at leaf
    node->desc = 0;
    return (node);
}

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

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

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

    // calculate the number of children of x and y
    // which are changed due to rotation.
    int val = (T2 != NULL) ? T2->desc : -1;
    y->desc = y->desc - (x->desc + 1) + (val + 1);
    x->desc = x->desc - (val + 1) + (y->desc + 1);

    return x;
}

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

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

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

```

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

// calculate the number of children of x and y
// which are changed due to rotation.
int val = (T2 != NULL) ? T2->desc : -1;
x->desc = x->desc - (y->desc + 1) + (val + 1);
y->desc = y->desc - (val + 1) + (x->desc + 1);

return y;
}

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

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

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

    else if (key > node->key) {
        node->right = insert(node->right, key);
        node->desc++;
    }

    else // Equal keys not allowed
        return node;

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

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

    // If node becomes unbalanced, 4 cases arise
}

```

```

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

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

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

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

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

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

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

    return current;
}

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

    if (root == NULL)

```

```
    return root;

    // If the key to be deleted is smaller than the
    // root's key, then it lies in left subtree
    if (key < root->key) {
        root->left = deleteNode(root->left, key);
        root->desc = root->desc - 1;
    }

    // If the key to be deleted is greater than the
    // root's key, then it lies in right subtree
    else if (key > root->key) {
        root->right = deleteNode(root->right, key);
        root->desc = root->desc - 1;
    }

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

            struct Node* temp = root->left ?
                                root->left : root->right;

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

            }
            else // One child case
            {
                *root = *temp; // Copy the contents of
                               // the non-empty child
                free(temp);
            }
        } else {
            // node with two children: Get the inorder
            // successor (smallest in the right subtree)
            struct Node* temp = minValueNode(root->right);

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

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

```

        root->desc = root->desc - 1;
    }
}

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

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

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

// If this node becomes unbalanced, 4 cases arise

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

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

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

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

return root;
}

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

```

```
        preOrder(root->right);
    }
}

// Returns count of
int CountGreater(struct Node* root, int x)
{
    int res = 0;

    // Search for x. While searching, keep
    // updating res if x is greater than
    // current node.
    while (root != NULL) {

        int desc = (root->right != NULL) ?
                    root->right->desc : -1;

        if (root->key > x) {
            res = res + desc + 1 + 1;
            root = root->left;
        } else if (root->key < x)
            root = root->right;
        else {
            res = res + desc + 1;
            break;
        }
    }
    return res;
}

/* Driver program to test above function*/
int main()
{
    struct Node* root = NULL;
    root = insert(root, 9);
    root = insert(root, 5);
    root = insert(root, 10);
    root = insert(root, 0);
    root = insert(root, 6);
    root = insert(root, 11);
    root = insert(root, -1);
    root = insert(root, 1);
    root = insert(root, 2);

    /* The constructed AVL Tree would be
         9
        /   \
       1    10
```

```
      / \      \
      0   5      11
      /   / \
     -1   2   6    */
printf("Preorder traversal of the constructed AVL "
      "tree is \n");
preOrder(root);
printf("\nNumber of elements greater than 9 are %d",
      CountGreater(root, 9));

root = deleteNode(root, 10);

/* The AVL Tree after deletion of 10
      1
      / \
      0   9
      /   / \
     -1   5   11
      / \
     2   6 */

printf("\nPreorder traversal after deletion of 10 \n");
preOrder(root);
printf("\nNumber of elements greater than 9 are %d",
      CountGreater(root, 9));
return 0;
}
```

Output:

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

Time Complexity: Time complexity of CountGreater function is $O(\log(n))$ where n is number of nodes in avl tree, as we are basically searching for the given number in avl which takes $O(\log(n))$ time.

Source

<https://www.geeksforgeeks.org/count-greater-nodes-in-avl-tree/>

Chapter 31

Count inversion pairs in a matrix

Count inversion pairs in a matrix - GeeksforGeeks

Given a matrix \mathbf{A} of size $N \times N$, we need to find the number of inversion pairs in it. Inversion count in a matrix is defined as the number of pairs satisfying the following conditions :

- $x_1 < x_2$
- $y_1 < y_2$
- $A[x_2][y_2] < A[x_1][y_1]$

Constraints :

- $1 \leq A_{i,j} \leq 10^9$
- $1 \leq N \leq 10^3$

Examples:

For simplicity, let's take a 2×2 matrix :

```
A = {{7, 5},  
     {3, 1}};
```

The inversion pairs are : (7, 5), (3, 1), (7, 3), (5, 1) and (7, 1)

Output : 5

To solve this problem, we need to know the following things :

1. Finding number of inversion pairs in a 1D array using Binary Indexed Tree (BIT)
<https://www.geeksforgeeks.org/count-inversions-array-set-3-using-bit>

2. 2D BIT

<https://www.geeksforgeeks.org/two-dimensional-binary-indexed-tree-or-fenwick-tree>

Since, we need to find number of inversion pairs in a matrix, first thing we need to do is to store the elements of the matrix in another array, say v and sort the array v so that we can compare the elements of the unsorted matrix with v and find the number of inversion pairs using BIT. But it is given that the values of the elements are very large (10^9), so we cannot use the values of the elements in the matrix as indices in the BIT. Thus, we need to use the position of the elements as indexes in the 2D BIT.

We are going to use the tuple $(-A[i][j], i, j)$ for each element of the matrix and store it in an array, say ' v '. Then we need to sort v according to the value of $-A[i][j]$ in ascending order, so that the largest element of the matrix will be stored at index 0 and the smallest one at the last index of v . Now, the problem is reduced to finding inversion pairs in a 1D array, the only exception is that we are going to use a 2D BIT.

Note that here we are using negative values of $A[i][j]$, simply because we are going to traverse v from left to right, i.e., from the largest number in the matrix to the smallest one(because that's what we do when finding inversion pairs in a 1D array using BIT). One can also use positive values and traverse v from right to left fashion, final result will remain same.

Algorithm :

1. Initialize $\text{inv_pair_cnt} = 0$, which will store the number of inversion pairs.
2. Store the tuple $(-A[i][j], i, j)$ in an array, say v , where $A[i][j]$ is the element of the matrix A at position (i, j) .
3. Sort the array v according to the first element of the tuple, i.e., according to the value of $-A[i][j]$.
4. Traverse the array v and do the following :
 - Initialize an array, say 'pairs' to store the position (i, j) of the tuples of v .
 - while the current tuple of v and all its adjacent tuples whose first value, i.e., $-A[i][j]$ is same, do
 - Push the current tuple's position pair (i, j) into 'pairs'.
 - Add to inv_pair_cnt , the number of elements which are less than the current element(i.e., $A[i][j]$) and lie on the right side in the sorted array v , by calling the query operation of BIT and passing i and j as arguments.
 - For each position pair (i, j) stored in the array 'pairs', update the position (i, j) in the 2D BIT by 1.
5. Finally, inv_pair_cnt will contain the number of inversion pairs.

Here is the C++ implementation :

```
// C++ program to count the number of inversion
// pairs in a 2D matrix
#include <bits/stdc++.h>
using namespace std;
```

```

// for simplicity, we are taking N as 4
#define N 4

// Function to update a 2D BIT. It updates the
// value of bit[l][r] by adding val to bit[l][r]
void update(int l, int r, int val, int bit[][][N + 1])
{
    for (int i = l; i <= N; i += i & -i)
        for (int j = r; j <= N; j += j & -j)
            bit[i][j] += val;
}

// function to find cumulative sum upto
// index (l, r) in the 2D BIT
long long query(int l, int r, int bit[][][N + 1])
{
    long long ret = 0;
    for (int i = l; i > 0; i -= i & -i)
        for (int j = r; j > 0; j -= j & -j)
            ret += bit[i][j];
}

return ret;
}

// function to count and return the number
// of inversion pairs in the matrix
long long countInversionPairs(int mat[][][N])
{
    // the 2D bit array and initialize it with 0.
    int bit[N+1][N+1] = {0};

    // v will store the tuple (-mat[i][j], i, j)
    vector<pair<int, pair<int, int>> v;

    // store the tuples in the vector v
    for (int i = 0; i < N; ++i)
        for (int j = 0; j < N; ++j)

            // Note that we are not using the pair
            // (0, 0) because BIT update and query
            // operations are not done on index 0
            v.push_back(make_pair(-mat[i][j],
                                  make_pair(i+1, j+1)));

    // sort the vector v according to the
    // first element of the tuple, i.e., -mat[i][j]
    sort(v.begin(), v.end());
}

```

```

// inv_pair_cnt will store the number of
// inversion pairs
long long inv_pair_cnt = 0;

// traverse all the tuples of vector v
int i = 0;
while (i < v.size())
{
    int curr = i;

    // 'pairs' will store the position of each element,
    // i.e., the pair (i, j) of each tuple of the vector v
    vector<pair<int, int> > pairs;

    // consider the current tuple in v and all its
    // adjacent tuples whose first value, i.e., the
    // value of -mat[i][j] is same
    while (curr < v.size() &&
           (v[curr].first == v[i].first))
    {
        // push the position of the current element in 'pairs'
        pairs.push_back(make_pair(v[curr].second.first,
                                   v[curr].second.second));

        // add the number of elements which are
        // less than the current element and lie on the right
        // side in the vector v
        inv_pair_cnt += query(v[curr].second.first,
                              v[curr].second.second, bit);

        curr++;
    }

    vector<pair<int, int> >::iterator it;

    // traverse the 'pairs' vector
    for (it = pairs.begin(); it != pairs.end(); ++it)
    {
        int x = it->first;
        int y = it->second;

        // update the positon (x, y) by 1
        update(x, y, 1, bit);
    }

    i = curr;
}

```

```
    return inv_pair_cnt;
}

// Driver program
int main()
{
    int mat[N][N] = { { 4, 7, 2, 9 },
                      { 6, 4, 1, 7 },
                      { 5, 3, 8, 1 },
                      { 3, 2, 5, 6 } };

    long long inv_pair_cnt = countInversionPairs(mat);

    cout << "The number of inversion pairs are : "
         << inv_pair_cnt << endl;

    return 0;
}
```

Output:

```
The number of inversion pairs are : 43
```

Time Complexity : $O(\log(N \times N))$, where N is the size of the matrix
Space Complexity : $O(N \times N)$

Source

<https://www.geeksforgeeks.org/count-inversion-pairs-matrix/>

Chapter 32

Count inversions in an array Set 3 (Using BIT)

Count inversions in an array Set 3 (Using BIT) - GeeksforGeeks

Inversion Count for an array indicates – how far (or close) the array is from being sorted. If array is already sorted then inversion count is 0. If array is sorted in reverse order that inversion count is the maximum.

Two elements $a[i]$ and $a[j]$ form an inversion if $a[i] > a[j]$ and $i < j$. For simplicity, we may assume that all elements are unique.

Example:

Input: $\text{arr}[] = \{8, 4, 2, 1\}$

Output: 6

Given array has six inversions (8,4), (4,2), (8,2), (8,1), (4,1), (2,1).

We have already discussed below methods to solve inversion count

1) [Naive and Modified Merge Sort](#)

2) [Using AVL Tree](#)

Background:

BIT basically supports two operations for an array $\text{arr}[]$ of size n:

1. Sum of elements till $\text{arr}[i]$ in $O(\log n)$ time.
2. Update an array element in $O(\log n)$ time.

BIT is implemented using an array and works in form of trees. Note that there are two ways of looking at BIT as a tree.

1. The sum operation where parent of index x is " $x - (x \& -x)$ ".
2. The update operation where parent of index x is " $x + (x \& -x)$ ".

We recommend you to refer [Binary Indexed Tree \(BIT\)](#) before further reading this post.

Basic Approach using BIT of size $\Theta(\text{maxElement})$:

The idea is to iterate the array from $n-1$ to 0 . When we are at i^{th} index, we check how many numbers less than $\text{arr}[i]$ are present in BIT and add it to the result. To get the count of smaller elements, [getSum\(\) of BIT](#) is used. In his basic idea, BIT is represented as an array of size equal to maximum element plus one. So that elements can be used as an index. After that we add current element to to the $\text{BIT}[]$ by doing an update operation that updates count of current element from 0 to 1 , and therefore updates ancestors of current element in BIT (See [update\(\) in BIT](#) for details).

Below is C++ implementation of basic idea that uses BIT.

```
// C++ program to count inversions using Binary Indexed Tree
#include<bits/stdc++.h>
using namespace std;

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

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

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

// Updates a node in Binary Index Tree (BITree) at given index
// in BITree. The given value 'val' is added to BITree[i] and
// all of its ancestors in tree.
void updateBIT(int BITree[], int n, int index, int val)
{
    // Traverse all ancestors and add 'val'
    while (index <= n)
    {
        // Add 'val' to current node of BI Tree
        BITree[index] += val;

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

```
BITree[index] += val;

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

// Returns inversion count arr[0..n-1]
int getInvCount(int arr[], int n)
{
    int invcount = 0; // Initialize result

    // Find maximum element in arr[]
    int maxElement = 0;
    for (int i=0; i<n; i++)
        if (maxElement < arr[i])
            maxElement = arr[i];

    // Create a BIT with size equal to maxElement+1 (Extra
    // one is used so that elements can be directly be
    // used as index)
    int BIT[maxElement+1];
    for (int i=1; i<=maxElement; i++)
        BIT[i] = 0;

    // Traverse all elements from right.
    for (int i=n-1; i>=0; i--)
    {
        // Get count of elements smaller than arr[i]
        invcount += getSum(BIT, arr[i]-1);

        // Add current element to BIT
        updateBIT(BIT, maxElement, arr[i], 1);
    }
}

return invcount;
}

// Driver program
int main()
{
    int arr[] = {8, 4, 2, 1};
    int n = sizeof(arr)/sizeof(int);
    cout << "Number of inversions are : " << getInvCount(arr,n);
    return 0;
}
```

Output:

Number of inversions are : 6

Time Complexity :- The update function and getSum function runs for $O(\log(\text{maximumelement}))$ and we are iterating over n elements. So overall time complexity is : $O(n\log(\text{maximumelement}))$.

Auxiliary space : $O(\text{maxElement})$

Better Approach using BIT of size $\Theta(n)$:

The problem with the previous approach is that it doesn't work for negative numbers as index cannot be negative. Also by updating the value till maximum element we waste time and space as it is quite possible that we may never use intermediate value. For example, lots of space and time is wasted for an array like {1, 100000}.

The idea is to convert given array to an array with values from 1 to n and relative order of smaller and greater elements remains

Example :-

arr[] = {7, -90, 100, 1}

It gets converted to,

arr[] = {3, 1, 4, 2}
as $-90 < 1 < 7 < 100$.

We only have to make $\text{BIT}[]$ of number of elements instead of maximum element.

Changing element will not have any change in the answer as the greater elements remain greater and at same position.

```
// C++ program to count inversions using Binary Indexed Tree
#include<bits/stdc++.h>
using namespace std;

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

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

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

```
}  
  
// Updates a node in Binary Index Tree (BITree) at given index  
// in BITree. The given value 'val' is added to BITree[i] and  
// all of its ancestors in tree.  
void updateBIT(int BITree[], int n, int index, int val)  
{  
    // Traverse all ancestors and add 'val'  
    while (index <= n)  
    {  
        // Add 'val' to current node of BI Tree  
        BITree[index] += val;  
  
        // Update index to that of parent in update View  
        index += index & (-index);  
    }  
}  
  
// Converts an array to an array with values from 1 to n  
// and relative order of smaller and greater elements remains  
// same. For example, {7, -90, 100, 1} is converted to  
// {3, 1, 4, 2 }  
void convert(int arr[], int n)  
{  
    // Create a copy of arr[] in temp and sort the temp array  
    // in increasing order  
    int temp[n];  
    for (int i=0; i<n; i++)  
        temp[i] = arr[i];  
    sort(temp, temp+n);  
  
    // Traverse all array elements  
    for (int i=0; i<n; i++)  
    {  
        // lower_bound() Returns pointer to the first element  
        // greater than or equal to arr[i]  
        arr[i] = lower_bound(temp, temp+n, arr[i]) - temp + 1;  
    }  
}  
  
// Returns inversion count arr[0..n-1]  
int getInvCount(int arr[], int n)  
{  
    int invcount = 0; // Initialize result  
  
    // Convert arr[] to an array with values from 1 to n and  
    // relative order of smaller and greater elements remains  
    // same. For example, {7, -90, 100, 1} is converted to
```

```
// {3, 1, 4 ,2 }
convert(arr, n);

// Create a BIT with size equal to maxElement+1 (Extra
// one is used so that elements can be directly be
// used as index)
int BIT[n+1];
for (int i=1; i<=n; i++)
    BIT[i] = 0;

// Traverse all elements from right.
for (int i=n-1; i>=0; i--)
{
    // Get count of elements smaller than arr[i]
    invcount += getSum(BIT, arr[i]-1);

    // Add current element to BIT
    updateBIT(BIT, n, arr[i], 1);
}

return invcount;
}

// Driver program
int main()
{
    int arr[] = {8, 4, 2, 1};
    int n = sizeof(arr)/sizeof(int);
    cout << "Number of inversions are : " << getInvCount(arr,n);
    return 0;
}
```

Output:

```
Number of inversions are : 6
```

Time Complexity :- The update function and getSum function runs for $O(\log(n))$ and we are iterating over n elements. So overall time complexity is : $O(n\log n)$.

Auxiliary space : $O(n)$

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

Source

<https://www.geeksforgeeks.org/count-inversions-array-set-3-using-bit/>

Chapter 33

Count inversions of size k in a given array

Count inversions of size k in a given array - GeeksforGeeks

Given an array of n distinct integers and an integer k. Find out the number of sub-sequences of a such that, and . In other words output the total number of inversions of length k.

Examples:

Input : a[] = {9, 3, 6, 2, 1}, k = 3

Output : 7

The seven inversions are {9, 3, 2}, {9, 3, 1}, {9, 6, 2}, {9, 6, 1}, {9, 2, 1}, {3, 2, 1} and {6, 2, 1}.

Input : a[] = {5, 6, 4, 9, 2, 7, 1}, k = 4

Output : 2

The two inversions are {5, 4, 2, 1}, {6, 4, 2, 1}.

We have already discussed counting inversion of length three [here](#). This post will generalise the approach using [Binary Indexed Tree](#) and [Counting inversions using BIT](#). More on Binary Indexed Trees can be found from the actual paper that Peter M. Fenwick published, [here](#).

1. To begin with, we first convert the given array to a permutation of elements (Note that this is always possible since the elements are distinct).
2. The approach is to maintain a set of k Fenwick Trees. Let it be denoted by

where $\text{BIT}[t][x]$, keeps track of the number of l – length sub-sequences that start with x .

3. We iterate from the end of the converted array to the beginning. For every converted array element x , we update the Fenwick tree set, as: For each $1 \leq i \leq k$, each sub-sequence of length l that start with a number less than x is also a part of sequences of length $i + l$.

The final result is found by sum of occurrences of $\text{BIT}[t][x]$.

The implementation is discussed below:

C++

```
// C++ program to count inversions of size k using
// Binary Indexed Tree
#include <bits/stdc++.h>
using namespace std;

// It is beneficial to declare the 2D BIT globally
// since passing it into functions will create
// additional overhead
const int K = 51;
const int N = 100005;
int BIT[K][N] = { 0 };

// update function. "t" denotes the t'th Binary
// indexed tree
void updateBIT(int t, int i, int val, int n)
{
    // Traversing the t'th BIT
    while (i <= n) {
        BIT[t][i] = BIT[t][i] + val;
        i = i + (i & (-i));
    }
}

// function to get the sum.
// "t" denotes the t'th Binary indexed tree
int getSum(int t, int i)
{
    int res = 0;

    // Traversing the t'th BIT
    while (i > 0) {
        res = res + BIT[t][i];
        i = i - (i & (-i));
    }
}
```

```
    return res;
}

// Converts an array to an array with values from 1 to n
// and relative order of smaller and greater elements
// remains same. For example, {7, -90, 100, 1} is
// converted to {3, 1, 4, 2 }
void convert(int arr[], int n)
{
    // Create a copy of arr[] in temp and sort
    // the temp array in increasing order
    int temp[n];
    for (int i = 0; i < n; i++)
        temp[i] = arr[i];
    sort(temp, temp + n);

    // Traverse all array elements
    for (int i = 0; i < n; i++) {

        // lower_bound() Returns pointer to the
        // first element greater than or equal
        // to arr[i]
        arr[i] = lower_bound(temp, temp + n,
                            arr[i]) - temp + 1;
    }
}

// Returns count of inversions of size three
int getInvCount(int arr[], int n, int k)
{
    // Convert arr[] to an array with values from
    // 1 to n and relative order of smaller and
    // greater elements remains same. For example,
    // {7, -90, 100, 1} is converted to {3, 1, 4, 2 }
    convert(arr, n);

    // iterating over the converted array in
    // reverse order.
    for (int i = n - 1; i >= 0; i--) {
        int x = arr[i];

        // update the BIT for l = 1
        updateBIT(1, x, 1, n);

        // update BIT for all other BITS
        for (int l = 1; l < k; l++) {
            updateBIT(l + 1, x, getSum(l, x - 1), n);
        }
    }
}
```

```
}

// final result
return getSum(k, n);
}

// Driver program to test above function
int main()
{
    int arr[] = { 5, 6, 4, 9, 3, 7, 2, 1 };
    int n = sizeof(arr) / sizeof(arr[0]);
    int k = 4;
    cout << "Inversion Count : " << getInvCount(arr, n, k);
    return 0;
}
```

Java

```
// Java program to count
// inversions of size k using
// Binary Indexed Tree
import java.io.*;
import java.util.Arrays;
import java.util.ArrayList;
import java.lang.*;
import java.util.Collections;

class GFG
{

// It is beneficial to declare
// the 2D BIT globally since
// passing it into functions
// will create additional overhead
static int K = 51;
static int N = 100005;
static int BIT[][] = new int[K][N];

// update function. "t" denotes
// the t'th Binary indexed tree
static void updateBIT(int t, int i,
                      int val, int n)
{

    // Traversing the t'th BIT
    while (i <= n)
    {
        BIT[t][i] = BIT[t][i] + val;
```

```

        i = i + (i & (-i));
    }
}

// function to get the sum.
// "t" denotes the t'th
// Binary indexed tree
static int getSum(int t, int i)
{
    int res = 0;

    // Traversing the t'th BIT
    while (i > 0)
    {
        res = res + BIT[t][i];
        i = i - (i & (-i));
    }
    return res;
}

// Converts an array to an
// array with values from
// 1 to n and relative order
// of smaller and greater
// elements remains same.
// For example, {7, -90, 100, 1}
// is converted to {3, 1, 4, 2 }
static void convert(int arr[], int n)
{
    // Create a copy of arr[] in
    // temp and sort the temp
    // array in increasing order
    int temp[] = new int[n];
    for (int i = 0; i < n; i++)
        temp[i] = arr[i];
    Arrays.sort(temp);

    // Traverse all array elements
    for (int i = 0; i < n; i++)
    {

        // lower_bound() Returns
        // pointer to the first
        // element greater than
        // or equal to arr[i]
        arr[i] = Arrays.binarySearch(temp,
                                     arr[i]) + 1;
    }
}

```

```
}

// Returns count of inversions
// of size three
static int getInvCount(int arr[],
                      int n, int k)
{

    // Convert arr[] to an array
    // with values from 1 to n and
    // relative order of smaller
    // and greater elements remains
    // same. For example, {7, -90, 100, 1}
    // is converted to {3, 1, 4, 2 }
    convert(arr, n);

    // iterating over the converted
    // array in reverse order.
    for (int i = n - 1; i >= 0; i--)
    {
        int x = arr[i];

        // update the BIT for l = 1
        updateBIT(1, x, 1, n);

        // update BIT for all other BITS
        for (int l = 1; l < k; l++)
        {
            updateBIT(l + 1, x,
                      getSum(l, x - 1), n);
        }
    }

    // final result
    return getSum(k, n);
}

// Driver Code
public static void main(String[] args)
{

    int arr[] = { 5, 6, 4, 9,
                 3, 7, 2, 1 };
    int n = arr.length;
    int k = 4;
    System.out.println("Inversion Count : " +
                       getInvCount(arr, n, k));
}
```

}

Output:

Inversion Count : 11

Time Complexity $\mathcal{O}(n \log n)$

Auxiliary Space $\mathcal{O}(n)$

It should be noted that this is not the only approach to solve the problem of finding k-inversions. Obviously, any problem solvable by BIT is also solvable by Segment Tree. Besides, we can use Merge-Sort based algorithm, and C++ policy based data structure too. Also, at the expense of higher time complexity, Dynamic Programming approach can also be used.

Source

<https://www.geeksforgeeks.org/count-inversions-of-size-k-in-a-given-array/>

Chapter 34

Count number of smallest elements in given range

Count number of smallest elements in given range - GeeksforGeeks

Given an array of N numbers and Q queries, each query consists of L and R. We need to write a program that prints the number of occurrence of the smallest element in the range L-R.

Examples:

```
Input: a[] = {1, 1, 2, 4, 3, 3}
      Q = 2
      L = 1 R = 4
      L = 3 R = 6
```

```
Output: 2
      1
```

Explanation: The smallest element in range 1-4 is 1 which occurs 2 times. The smallest element in the range 3-6 is 2 which occurs once.

```
Input : a[] = {1, 2, 3, 3, 1}
      Q = 2
      L = 1 R = 5
      L = 3 R = 4
Output : 2
      2
```

A **normal approach** will be to iterate from L-R and find out the smallest element in the range. Iterate again in the range L-R and count the number of times the smallest element occurs in the range L-R. In the worst case, the complexity will be O(N) if L=1 and R=N.

An **efficient approach** will be to use [Segment Trees](#) to solve the above problem. At each node of the segment tree, smallest element and count of smallest element is stored. At leaf nodes, the array element is stored in the minimum and count stores 1. For all other nodes except the leaf nodes, we merge the right and left nodes following the given conditions:

1. **min(left_subtree) < min(right_subtree):**
node.min=min(left_subtree), node.count = left_subtree.count
2. **min(left_subtree) > min(right_subtree):**
node.min=min(right_subtree), node.count=right_subtree.count
3. **min(left_subtree) = min(right_subtree):**
node.min=min(left_subtree) or min(right_subtree), node.count=left_subtree.count + right_subtree.count

Given below is the implementation of the above approach:

```
// CPP program to Count number of occurrence of
// smallest element in range L-R
#include <bits/stdc++.h>
using namespace std;

#define N 100005

// predefines the tree with nodes
// storing min and count
struct node {
    int min;
    int cnt;
} tree[5 * N];

// function to construct the tree
void buildtree(int low, int high, int pos, int a[])
{
    // base condition
    if (low == high) {

        // leaf node has a single element
        tree[pos].min = a[low];
        tree[pos].cnt = 1;
        return;
    }

    int mid = (low + high) >> 1;
    // left-subtree
    buildtree(low, mid, 2 * pos + 1, a);

    // right-subtree
    buildtree(mid + 1, high, 2 * pos + 2, a);
}
```

```
// left subtree has the minimum element
if (tree[2 * pos + 1].min < tree[2 * pos + 2].min) {
    tree[pos].min = tree[2 * pos + 1].min;
    tree[pos].cnt = tree[2 * pos + 1].cnt;
}

// right subtree has the minimum element
else if (tree[2 * pos + 1].min > tree[2 * pos + 2].min) {
    tree[pos].min = tree[2 * pos + 2].min;
    tree[pos].cnt = tree[2 * pos + 2].cnt;
}

// both subtree has the same minimum element
else {
    tree[pos].min = tree[2 * pos + 1].min;
    tree[pos].cnt = tree[2 * pos + 1].cnt + tree[2 * pos + 2].cnt;
}

// function that answers every query
node query(int s, int e, int low, int high, int pos)
{
    node dummy;
    // out of range
    if (e < low or s > high) {
        dummy.min = dummy.cnt = INT_MAX;
        return dummy;
    }

    // in range
    if (s >= low and e <= high) {
        return tree[pos];
    }

    int mid = (s + e) >> 1;

    // left-subtree
    node ans1 = query(s, mid, low, high, 2 * pos + 1);

    // right-subtree
    node ans2 = query(mid + 1, e, low, high, 2 * pos + 2);

    node ans;
    ans.min = min(ans1.min, ans2.min);

    // add count when min is same of both subtree
    if (ans1.min == ans2.min)
```

```
ans.cnt = ans2.cnt + ans1.cnt;

// store the minimal's count
else if (ans1.min < ans2.min)
    ans.cnt = ans1.cnt;

else
    ans.cnt = ans2.cnt;

return ans;
}

// function to answer query in range l-r
int answerQuery(int a[], int n, int l, int r)
{
    // calls the function which returns a node
    // this function returns the count which
    // will be the answer
    return query(0, n - 1, l - 1, r - 1, 0).cnt;
}

// Driver Code
int main()
{
    int a[] = { 1, 1, 2, 4, 3, 3 };

    int n = sizeof(a) / sizeof(a[0]);
    buildtree(0, n - 1, 0, a);
    int l = 1, r = 4;

    // answers 1-st query
    cout << answerQuery(a, n, l, r) << endl;

    l = 2, r = 6;
    // answers 2nd query
    cout << answerQuery(a, n, l, r) << endl;
    return 0;
}
```

Output:

```
2
1
```

Time Complexity: $O(n)$ for the construction of tree. $O(\log n)$ for every query.

Source

<https://www.geeksforgeeks.org/count-number-of-smallest-elements-in-given-range/>

Chapter 35

Count of distinct substrings of a string using Suffix Array

Count of distinct substrings of a string using Suffix Array - GeeksforGeeks

Given a string of length n of lowercase alphabet characters, we need to count total number of distinct substrings of this string.

Examples:

```
Input : str = "ababa"
Output : 10
Total number of distinct substring are 10, which are,
 "", "a", "b", "ab", "ba", "aba", "bab", "abab", "baba"
and "ababa"
```

We have discussed a Suffix Trie based solution in below post :

[Count of distinct substrings of a string using Suffix Trie](#)

We can solve this problem using [suffix array](#) and longest common prefix concept. A suffix array is a sorted array of all suffixes of a given string.

For string “ababa” suffixes are : “ababa”, “baba”, “aba”, “ba”, “a”. After taking these suffixes in sorted form we get our suffix array as [4, 2, 0, 3, 1]

Then we calculate lcp array using [kasai's algorithm](#). For string “ababa”, lcp array is [1, 3, 0, 2, 0]

After constructing both arrays, we calculate total number of distinct substring by keeping this fact in mind : If we look through the prefixes of each suffix of a string, we cover all substrings of that string.

We will explain the procedure for above example,

```
String = "ababa"
```

```
Suffixes in sorted order : "a", "aba", "ababa",
                           "ba", "baba"
Initializing distinct substring count by length
of first suffix,
Count = length("a") = 1
Substrings taken in consideration : "a"

Now we consider each consecutive pair of suffix,
lcp("a", "aba") = "a".
All characters that are not part of the longest
common prefix contribute to a distinct substring.
In the above case, they are 'b' and 'a'. So they
should be added to Count.
Count += length("aba") - lcp("a", "aba")
Count = 3
Substrings taken in consideration : "aba", "ab"

Similarly for next pair also,
Count += length("ababa") - lcp("aba", "ababa")
Count = 5
Substrings taken in consideration : "ababa", "abab"

Count += length("ba") - lcp("ababa", "ba")
Count = 7
Substrings taken in consideration : "ba", "b"

Count += length("baba") - lcp("ba", "baba")
Count = 9
Substrings taken in consideration : "baba", "bab"

We finally add 1 for empty string.
count = 10
```

Above idea is implemented in below code.

```
// C++ code to count total distinct substrings
// of a string
#include <bits/stdc++.h>
using namespace std;

// Structure to store information of a suffix
struct suffix
{
    int index; // To store original index
    int rank[2]; // To store ranks and next
                 // rank pair
};
```

```

// A comparison function used by sort() to compare
// two suffixes. Compares two pairs, returns 1 if
// first pair is smaller
int cmp(struct suffix a, struct suffix b)
{
    return (a.rank[0] == b.rank[0])?
           (a.rank[1] < b.rank[1] ?1: 0):
           (a.rank[0] < b.rank[0] ?1: 0);
}

// This is the main function that takes a string
// 'txt' of size n as an argument, builds and return
// the suffix array for the given string
vector<int> buildSuffixArray(string txt, int n)
{
    // A structure to store suffixes and their indexes
    struct suffix suffixes[n];

    // Store suffixes and their indexes in an array
    // of structures. The structure is needed to sort
    // the suffixes alphabetically and maintain their
    // old indexes while sorting
    for (int i = 0; i < n; i++)
    {
        suffixes[i].index = i;
        suffixes[i].rank[0] = txt[i] - 'a';
        suffixes[i].rank[1] = ((i+1) < n)?
                             (txt[i + 1] - 'a'): -1;
    }

    // Sort the suffixes using the comparison function
    // defined above.
    sort(suffixes, suffixes+n, cmp);

    // At this point, all suffixes are sorted according
    // to first 2 characters. Let us sort suffixes
    // according to first 4 characters, then first
    // 8 and so on
    int ind[n]; // This array is needed to get the
                // index in suffixes[] from original
                // index. This mapping is needed to get
                // next suffix.
    for (int k = 4; k < 2*n; k = k*2)
    {
        // Assigning rank and index values to first suffix
        int rank = 0;
        int prev_rank = suffixes[0].rank[0];
        suffixes[0].rank[0] = rank;

```

```

ind[suffixes[0].index] = 0;

// Assigning rank to suffixes
for (int i = 1; i < n; i++)
{
    // If first rank and next ranks are same as
    // that of previous suffix in array, assign
    // the same new rank to this suffix
    if (suffixes[i].rank[0] == prev_rank &&
        suffixes[i].rank[1] == suffixes[i-1].rank[1])
    {
        prev_rank = suffixes[i].rank[0];
        suffixes[i].rank[0] = rank;
    }

    else // Otherwise increment rank and assign
    {
        prev_rank = suffixes[i].rank[0];
        suffixes[i].rank[0] = ++rank;
    }
    ind[suffixes[i].index] = i;
}

// Assign next rank to every suffix
for (int i = 0; i < n; i++)
{
    int nextindex = suffixes[i].index + k/2;
    suffixes[i].rank[1] = (nextindex < n)?
        suffixes[ind[nextindex]].rank[0]: -1;
}

// Sort the suffixes according to first k characters
sort(suffixes, suffixes+n, cmp);
}

// Store indexes of all sorted suffixes in the suffix
// array
vector<int>suffixArr;
for (int i = 0; i < n; i++)
    suffixArr.push_back(suffixes[i].index);

// Return the suffix array
return suffixArr;
}

/* To construct and return LCP */
vector<int> kasai(string txt, vector<int> suffixArr)
{

```

```

int n = suffixArr.size();

// To store LCP array
vector<int> lcp(n, 0);

// An auxiliary array to store inverse of suffix array
// elements. For example if suffixArr[0] is 5, the
// invSuff[5] would store 0. This is used to get next
// suffix string from suffix array.
vector<int> invSuff(n, 0);

// Fill values in invSuff[]
for (int i=0; i < n; i++)
    invSuff[suffixArr[i]] = i;

// Initialize length of previous LCP
int k = 0;

// Process all suffixes one by one starting from
// first suffix in txt[]
for (int i=0; i<n; i++)
{
    /* If the current suffix is at n-1, then we don't
       have next substring to consider. So lcp is not
       defined for this substring, we put zero. */
    if (invSuff[i] == n-1)
    {
        k = 0;
        continue;
    }

    /* j contains index of the next substring to
       be considered to compare with the present
       substring, i.e., next string in suffix array */
    int j = suffixArr[invSuff[i]+1];

    // Directly start matching from k'th index as
    // at-least k-1 characters will match
    while (i+k<n && j+k<n && txt[i+k]==txt[j+k])
        k++;

    lcp[invSuff[i]] = k; // lcp for the present suffix.

    // Deleting the starting character from the string.
    if (k>0)
        k--;
}

```

```
// return the constructed lcp array
return lcp;
}

// method to return count of total distinct substring
int countDistinctSubstring(string txt)
{
    int n = txt.length();
    // calculating suffix array and lcp array
    vector<int> suffixArr = buildSuffixArray(txt, n);
    vector<int> lcp = kasai(txt, suffixArr);

    // n - suffixArr[i] will be the length of suffix
    // at ith position in suffix array initializing
    // count with length of first suffix of sorted
    // suffixes
    int result = n - suffixArr[0];

    for (int i = 1; i < lcp.size(); i++)

        // subtract lcp from the length of suffix
        result += (n - suffixArr[i]) - lcp[i - 1];

    result++; // For empty string
    return result;
}

// Driver code to test above methods
int main()
{
    string txt = "ababa";
    cout << countDistinctSubstring(txt);
    return 0;
}
```

Output:

10

Source

<https://www.geeksforgeeks.org/count-distinct-substrings-string-using-suffix-array/>

Chapter 36

Count of distinct substrings of a string using Suffix Trie

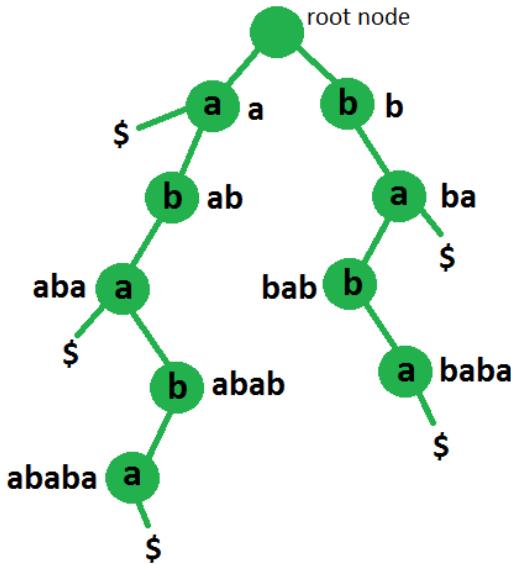
Count of distinct substrings of a string using Suffix Trie - GeeksforGeeks

Given a string of length n of lowercase alphabet characters, we need to count total number of distinct substrings of this string.

Examples:

```
Input : str = "ababa"
Output : 10
Total number of distinct substring are 10, which are,
 "", "a", "b", "ab", "ba", "aba", "bab", "abab", "baba"
and "ababa"
```

The idea is create a [Trie of all suffixes](#) of given string. Once the Trie is constricted, our answer is total number of nodes in the constructed Trie. For example below diagram represent Trie of all suffixes for “ababa”. Total number of nodes is 10 which is our answer.



Trie for string ababa with corresponding substring for each node

How does this work?

- Each root to node path of a **Trie** represents a prefix of words present in Trie. Here we words are suffixes. So each node represents a prefix of suffixes.
- Every substring of a string “str” is a prefix of a suffix of “str”.

Below is implementation based on above idea.

C++

```
// A C++ program to find the count of distinct substring
// of a string using trie data structure
#include <bits/stdc++.h>
#define MAX_CHAR 26
using namespace std;

// A Suffix Trie (A Trie of all suffixes) Node
class SuffixTreeNode
{
public:
    SuffixTreeNode *children[MAX_CHAR];
    SuffixTreeNode() // Constructor
    {
        // Initialize all child pointers as NULL
        for (int i = 0; i < MAX_CHAR; i++)
    }
}
```

```

        children[i] = NULL;
    }

    // A recursive function to insert a suffix of the s
    // in subtree rooted with this node
    void insertSuffix(string suffix);
};

// A Trie of all suffixes
class SuffixTrie
{
    SuffixTreeNode *root;
    int _countNodesInTrie(SuffixTreeNode *);
public:
    // Constructor (Builds a trie of suffies of the given text)
    SuffixTrie(string s)
    {
        root = new SuffixTreeNode();

        // Consider all suffixes of given string and insert
        // them into the Suffix Trie using recursive function
        // insertSuffix() in SuffixTreeNode class
        for (int i = 0; i < s.length(); i++)
            root->insertSuffix(s.substr(i));
    }

    // method to count total nodes in suffix trie
    int countNodesInTrie() { return _countNodesInTrie(root); }
};

// A recursive function to insert a suffix of the s in
// subtree rooted with this node
void SuffixTreeNode::insertSuffix(string s)
{
    // If string has more characters
    if (s.length() > 0)
    {
        // Find the first character and convert it
        // into 0-25 range.
        char cIndex = s.at(0) - 'a';

        // If there is no edge for this character,
        // add a new edge
        if (children[cIndex] == NULL)
            children[cIndex] = new SuffixTreeNode();

        // Recur for next suffix
        children[cIndex]->insertSuffix(s.substr(1));
    }
}

```

```
        }
    }

// A recursive function to count nodes in trie
int SuffixTrie::_countNodesInTrie(SuffixTreeNode* node)
{
    // If all characters of pattern have been processed,
    if (node == NULL)
        return 0;

    int count = 0;
    for (int i = 0; i < MAX_CHAR; i++)
    {
        // if children is not NULL then find count
        // of all nodes in this subtrie
        if (node->children[i] != NULL)
            count += _countNodesInTrie(node->children[i]);
    }

    // return count of nodes of subtrie and plus
    // 1 because of node's own count
    return (1 + count);
}

// Returns count of distinct substrings of str
int countDistinctSubstring(string str)
{
    // Construct a Trie of all suffixes
    SuffixTrie sTrie(str);

    // Return count of nodes in Trie of Suffixes
    return sTrie.countNodesInTrie();
}

// Driver program to test above function
int main()
{
    string str = "ababa";
    cout << "Count of distinct substrings is "
         << countDistinctSubstring(str);
    return 0;
}
```

Java

```
// A Java program to find the count of distinct substring
// of a string using trie data structure
public class Suffix
```

```
{
    // A Suffix Trie (A Trie of all suffixes) Node
    static class SuffixTreeNode
    {
        static final int MAX_CHAR = 26;
        SuffixTreeNode[] children = new SuffixTreeNode[MAX_CHAR];

        SuffixTreeNode() // Constructor
        {
            // Initialize all child pointers as NULL
            for (int i = 0; i < MAX_CHAR; i++)
                children[i] = null;
        }

        // A recursive function to insert a suffix of the s in
        // subtree rooted with this node
        void insertSuffix(String s)
        {
            // If string has more characters
            if (s.length() > 0)
            {
                // Find the first character and convert it
                // into 0-25 range.
                char cIndex = (char) (s.charAt(0) - 'a');

                // If there is no edge for this character,
                // add a new edge
                if (children[cIndex] == null)
                    children[cIndex] = new SuffixTreeNode();

                // Recur for next suffix
                children[cIndex].insertSuffix(s.substring(1));
            }
        }
    }

    // A Trie of all suffixes
    static class Suffix_trie
    {
        static final int MAX_CHAR = 26;
        SuffixTreeNode root;

        // Constructor (Builds a trie of suffies of the given text)
        Suffix_trie(String s) {
            root = new SuffixTreeNode();

            // Consider all suffixes of given string and insert
            // them into the Suffix Trie using recursive function
    }
}
```

```

// insertSuffix() in SuffixTreeNode class
for (int i = 0; i < s.length(); i++)
    root.insertSuffix(s.substring(i));
}

// A recursive function to count nodes in trie
int _countNodesInTrie(SuffixTreeNode node)
{
    // If all characters of pattern have been processed,
    if (node == null)
        return 0;

    int count = 0;
    for (int i = 0; i < MAX_CHAR; i++) {

        // if children is not NULL then find count
        // of all nodes in this subtree
        if (node.children[i] != null)
            count += _countNodesInTrie(node.children[i]);
    }

    // return count of nodes of subtree and plus
    // 1 because of node's own count
    return (1 + count);
}

// method to count total nodes in suffix trie
int countNodesInTrie()
{
    return _countNodesInTrie(root);
}

}

// Returns count of distinct substrings of str
static int countDistinctSubstring(String str)
{
    // Construct a Trie of all suffixes
    Suffix_trie sTrie = new Suffix_trie(str);

    // Return count of nodes in Trie of Suffixes
    return sTrie.countNodesInTrie();
}

// Driver program to test above function
public static void main(String args[])
{
    String str = "ababa";
}

```

```
System.out.println("Count of distinct substrings is "
+ countDistinctSubstring(str));

}

// This code is contributed by Sumit Ghosh
```

Output:

Count of distinct substrings is 10

We will soon be discussing [Suffix Array](#) and [Suffix Tree](#) based approaches for this problem.

Source

<https://www.geeksforgeeks.org/count-distinct-substrings-string-using-suffix-trie/>

Chapter 37

Counting Triangles in a Rectangular space using BIT

Counting Triangles in a Rectangular space using BIT - GeeksforGeeks

Pre-requisite : [BIT\(Binary Indexed Tree or Fenwick Tree\)](#), [2D BIT](#)

Given a 2D plane, respond to Q queries, each of the following type:

1. **Insert x y** – Insert a point (x, y) coordinate.
2. **Triangle x1 y1 x2 y2** – Print the number of triangles that can be formed, by joining the points inside the rectangle, described by two points (x1, y1) and (x2, y2), (x1, y1) is the lower left corner while (x2, y2) is the upper right corner. We will represent it as $\{(x_1, y_1), (x_2, y_2)\}$. (Include the the triangles with zero area also in your answer)

Example:

In the red rectangle there are 6 points inserted, when each of them is joined with a line with every other point, in all 20 triangles will be formed.

Let's say we somehow have a mechanism to find the number of points inside a given rectangle, let number of points be 'm' in an instance.

Number of triangles that can be formed with it are mC_3 (every 3 points when joined will make a triangle); if we had to count only degenerate triangles, we would have to subtract the number of triangles with zero area or formed from points on the same line.

Now we sneak into the mechanism to find the number of points in a rectangle; let's assume we have a mechanism to find number of points inside a rectangle .

Then number of points inside the rectangle $\{(x_1, y_1), (x_2, y_2)\}$ are,

$$\begin{aligned} P(x_2, y_2) &= \\ P(x_1 - 1, y_2) &- \\ P(x_2, y_1 - 1) &+ \\ P(x_1 - 1, y_1 - 1) \end{aligned}$$

Where,

$P(x, y)$ is number of triangles in rectangle from $(1, 1)$ to (x, y) , i.e., $\{(1, 1), (x, y)\}$

Now once we find a way to find $P(x, y)$, we are done.

If all the insert queries were made first, then all the triangle queries, it would have been an easy job, we could maintain a 2D table to store the points and $\text{table}[x][y]$ would contain the total number of points in the rectangle $\{(1, 1), (x, y)\}$.

It can be created using the following DP:

$$\text{table}[x][y] = \text{table}[x][y - 1] + \text{table}[x - 1][y] - \text{table}[x - 1][y - 1]$$

Or we can use a 2D BIT to insert a point and evaluate $P(x, y)$. For visualization of a 2D BIT, refer [Top Coder](#). The image shows a $16 * 8$ 2D BIT, and the state after an insertion at $(5, 3)$, the blue nodes are the ones that are updated.

The horizontal BITs correspond to 3, and store 1 at index 3, 1 at index 4, and 1 at index 8; while the vertical representation corresponds to which horizontal BITs will receive that update, the ones that correspond to 5, so 5th BIT from bottom gets an update, 6th BIT, then 8th BIT, and then 16th BIT.

Let's consider update in a 1D BIT

```
update(BIT, x)
    while ( x < maxn )
        BIT[x] += val
        x += x & -x
```

Update for 2D BIT goes like:

```
update2DBIT(x, y)

// update BIT at index x (from bottom, in the image)
while ( x < maxn )
    update(BIT[x], y)
    x += x & -x
```

```

// A C++ program implementing the above queries
#include<bits/stdc++.h>
#define maxn 2005
using namespace std;

// 2D Binary Indexed Tree. Note: global variable
// will have initially all elements zero
int bit[maxn][maxn];

// function to add a point at (x, y)
void update(int x, int y)
{
    int y1;
    while (x < maxn)
    {
        // x is the xth BIT that will be updated
        // while y is the indices where an update
        // will be made in xth BIT
        y1 = y;
        while (y1 < maxn)
        {
            bit[x][y1]++;
            y1 += (y1 & -y1);
        }

        // next BIT that should be updated
        x += x & -x;
    }
}

// Function to return number of points in the
// rectangle (1, 1), (x, y)
int query(int x, int y)
{
    int res = 0, y1;
    while (x > 0)
    {
        // xth BIT's yth node must be added to the result
        y1 = y;
        while (y1 > 0)
        {
            res += bit[x][y1];
            y1 -= y1 & -y1;
        }

        // next BIT that will contribute to the result
        x -= x & -x;
    }
}

```

```
        return res;
    }

    // (x1, y1) is the lower left and (x2, y2) is the
    // upper right corner of the rectangle
    int pointsInRectangle(int x1, int y1, int x2, int y2)
    {
        // Returns number of points in the rectangle
        // (x1, y1), (x2, y2) as described in text above
        return query(x2, y2) - query(x1 - 1, y2) -W
               query(x2, y1 - 1) + query(x1 - 1, y1 - 1);
    }

    // Returns count of triangles with n points, i.e.,
    // it returns nC3
    int findTriangles(int n)
    {
        // returns pts choose 3
        return (n * (n - 1) * (n - 2)) / 6;
    }

    //driver code
    int main()
    {
        //inserting points
        update(2, 2);
        update(3, 5);
        update(4, 2);
        update(4, 5);
        update(5, 4);

        cout << "No. of triangles in the rectangle (1, 1)"
            " (6, 6) are: "
            << findTriangles(pointsInRectangle(1, 1, 6, 6));

        update(3, 3);

        cout << "\nNo. of triangles in the rectangle (1, 1)"
            " (6, 6) are: "
            << findTriangles( pointsInRectangle(1, 1, 6, 6));

        return 0;
    }
```

Output:

No of triangles in the rectangle (1, 1) (6, 6) are: 10
No of triangles in the rectangle (1, 1) (6, 6) are: 20

Time Complexity : For each query of either type: $O(\log(x) * \log(y))$ Or infact: $O(\text{number of ones in binary representation of } x * \text{number of ones in binary representation of } y)$

Total time complexity: $O(Q * \log(\max X) * \log(\max Y))$

Source

<https://www.geeksforgeeks.org/counting-triangles-in-a-rectangular-space-using-2d-bit/>

Chapter 38

Counting k-mers via Suffix Array

Counting k-mers via Suffix Array - GeeksforGeeks

Pre-requisite: [Suffix Array](#).

What are k-mers?

The term [k-mer](#) typically refers to all the possible substrings of length k that are contained in a string. Counting all the k-mers in DNA/RNA sequencing reads is the preliminary step of many bioinformatics applications.

What is a Suffix Array?

A suffix array is a sorted array of all suffixes of a string. It is a data structure used, among others, in full text indices, data compression algorithms. More information can be found [here](#).

Problem: We are given a string str and an integer k. We have to find all pairs (substr, i) such that substr is a length – k substring of str that occurs exactly i times.

Steps involved in the approach:

Let's take the word “banana\$” as an example.

Step 1: Compute the suffix array of the given text.

6	\$
5	a\$
3	ana\$
1	anana\$
0	banana\$
4	na\$
2	nana\$

Step 2: Iterate through the suffix array keeping “curr_count”.

1. If the length of current suffix is less than k, then skip the iteration. That is, if **k = 2**,

then iteration would be skipped when current suffix is \$.

2. If the current suffix begins with the same length – k substring as the previous suffix, then increment curr_count. For example, during fourth iteration current suffix “anana\$” starts with same substring of length k “an” as previous suffix “ana\$” started with. So, we will increment curr_count in this case.
3. If condition 2 is not satisfied, then if length of previous suffix is equal to k, then that it is a valid pair and we will output it along with its current count, otherwise, we will skip that iteration.

		curr_count	Valid Pair
6	\$	1	
5	a\$	1	
3	ana\$	1	(a\$, 1)
1	anana\$	1	
0	banana\$	2	(an, 2)
4	na\$	1	(ba, 1)
2	nana\$	1	(na, 2)

Examples:

```
Input : banana$ // Input text
Output : (a$, 1) // k- mers
          (an, 2)
          (ba, 1)
          (na, 2)
```

```
Input : geeksforgeeks
Output : (ee, 2)
          (ek, 2)
          (fo, 1)
          (ge, 2)
          (ks, 2)
          (or, 1)
          (sf, 1)
```

The following is the C code for approach explained above:

```
// C program to solve K-mer counting problem
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// Structure to store data of a rotation
struct rotation {
    int index;
```

```
char* suffix;
};

// Compares the rotations and
// sorts the rotations alphabetically
int cmpfunc(const void* x, const void* y)
{
    struct rotation* rx = (struct rotation*)x;
    struct rotation* ry = (struct rotation*)y;
    return strcmp(rx->suffix, ry->suffix);
}

// Takes input_text and its length as arguments
// and returns the corresponding suffix array
char** computeSuffixArray(char* input_text,
                           int len_text)
{
    int i;

    // Array of structures to store rotations
    // and their indexes
    struct rotation suff[len_text];

    // Structure is needed to maintain old
    // indexes of rotations after sorting them
    for (i = 0; i < len_text; i++) {
        suff[i].index = i;
        suff[i].suffix = (input_text + i);
    }

    // Sorts rotations using comparison function
    // defined above
    qsort(suff, len_text, sizeof(struct rotation), cmpfunc);

    // Stores the suffixes of sorted rotations
    char** suffix_arr =
        (char**)malloc(len_text * sizeof(char*));

    for (i = 0; i < len_text; i++) {
        suffix_arr[i] =
            (char*)malloc((len_text + 1) * sizeof(char));
        strcpy(suffix_arr[i], suff[i].suffix);
    }

    // Returns the computed suffix array
    return suffix_arr;
}
```

```

// Takes suffix array, its size and valid length as
// arguments and outputs the valid pairs of k - mers
void findValidPairs(char** suffix_arr, int n, int k)
{
    int curr_count = 1, i;
    char* prev_suff = (char*)malloc(n * sizeof(char));

    // Iterates over the suffix array,
    // keeping a current count
    for (i = 0; i < n; i++) {

        // Skipping the current suffix
        // if it has length < valid length
        if (strlen(suffix_arr[i]) < k) {
            strcpy(prev_suff, suffix_arr[i]);
            continue;
        }

        // Incrementing the curr_count if first
        // k chars of prev_suff and current suffix
        // are same
        if (!(memcmp(prev_suff, suffix_arr[i], k))) {
            curr_count++;
        }
        else {

            // Pair is valid when i!=0 (as there is
            // no prev_suff for i = 0) and when strlen
            // of prev_suff is k
            if (i != 0 && strlen(prev_suff) == k) {
                printf("(%s, %d)\n", prev_suff, curr_count);
                curr_count = 1;
            }
            else {
                memcpy(prev_suff, suffix_arr[i], k);
                prev_suff[k] = '\0';
                continue;
            }
        }

        // Modifying prev_suff[i] to current suffix
        memcpy(prev_suff, suffix_arr[i], k);
        prev_suff[k] = '\0';
    }

    // Printing the last valid pair
    printf("(%s, %d)\n", prev_suff, curr_count);
}

```

```
// Driver program to test functions above
int main()
{
    char input_text[] = "geeksforgeeks";
    int k = 2;
    int len_text = strlen(input_text);

    // Computes the suffix array of our text
    printf("Input Text: %s\n", input_text);
    char** suffix_arr =
        computeSuffixArray(input_text, len_text);

    // Finds and outputs all valid pairs
    printf("k-mers: \n");
    findValidPairs(suffix_arr, len_text, k);

    return 0;
}
```

Output:

```
Input Text: banana$
k-mers:
(a$, 1)
(an, 2)
(ba, 1)
(na, 2)
```

Time Complexity: $O(s * \text{len_text} * \log(\text{len_text}))$, assuming s is the length of the longest suffix.

Sources:

1. [Suffix Array Wikipedia](#)
2. [Suffix Array CMU](#)

Source

<https://www.geeksforgeeks.org/counting-k-mers-via-suffix-array/>

Chapter 39

Counting the number of words in a Trie

Counting the number of words in a Trie - GeeksforGeeks

A [Trie](#) is used to store dictionary words so that they can be searched efficiently and prefix search can be done. The task is to write a function to count the number of words.

Example :

```
Input :      root
            /   \   \
          t     a     b
          |     |     |
          h     n     y
          |     |   \
          e     s     y   e
          /   |   |
          i     r     w
          |   |   |
          r     e     e
                      |
                      r

Output : 8
Explanation : Words formed in the Trie :
"the", "a", "there", "answer", "any", "by",
"bye", "their".
```

In Trie structure, we have a field to store end of word marker, we call it `isLeaf` in below implementation. To count words, we need to simply traverse the Trie and count all nodes where `isLeaf` is set.

C++

```
// C++ implementation to count words in a trie
#include <bits/stdc++.h>
using namespace std;

#define ARRAY_SIZE(a) sizeof(a)/sizeof(a[0])

// Alphabet size (# of symbols)
#define ALPHABET_SIZE (26)

// Converts key current character into index
// use only 'a' through 'z' and lower case
#define CHAR_TO_INDEX(c) ((int)c - (int)'a')

// Trie node
struct TrieNode
{
    struct TrieNode *children[ALPHABET_SIZE];

    // isLeaf is true if the node represents
    // end of a word
    bool isLeaf;
};

// Returns new trie node (initialized to NULLs)
struct TrieNode *getNode(void)
{
    struct TrieNode *pNode = new TrieNode;
    pNode->isLeaf = false;

    for (int i = 0; i < ALPHABET_SIZE; i++)
        pNode->children[i] = NULL;

    return pNode;
}

// If not present, inserts key into trie
// If the key is prefix of trie node, just
// marks leaf node
void insert(struct TrieNode *root, const char *key)
{
    int length = strlen(key);

    struct TrieNode *pCrawl = root;

    for (int level = 0; level < length; level++)
    {
        int index = CHAR_TO_INDEX(key[level]);
        if (!pCrawl->children[index])
```

```
pCrawl->children[index] = getNode();  
  
    pCrawl = pCrawl->children[index];  
}  
  
// mark last node as leaf  
pCrawl->isLeaf = true;  
}  
  
// Function to count number of words  
int wordCount(struct TrieNode *root)  
{  
    int result = 0;  
  
    // Leaf denotes end of a word  
    if (root -> isLeaf)  
        result++;  
  
    for (int i = 0; i < ALPHABET_SIZE; i++)  
        if (root -> children[i])  
            result += wordCount(root -> children[i]);  
  
    return result;  
}  
  
// Driver  
int main()  
{  
    // Input keys (use only 'a' through 'z'  
    // and lower case)  
    char keys[][] [8] = {"the", "a", "there", "answer",  
                        "any", "by", "bye", "their"};  
  
    struct TrieNode *root = getNode();  
  
    // Construct Trie  
    for (int i = 0; i < ARRAY_SIZE(keys); i++)  
        insert(root, keys[i]);  
  
    cout << wordCount(root);  
  
    return 0;  
}
```

Java

```
// Java implementation to count words in a trie
```

```
public class Words_in_trie {

    // Alphabet size (# of symbols)
    static final int ALPHABET_SIZE = 26;

    // Trie node
    static class TrieNode
    {
        TrieNode[] children = new TrieNode[ALPHABET_SIZE];
        // isLeaf is true if the node represents
        // end of a word
        boolean isLeaf;

        TrieNode(){
            isLeaf = false;
            for (int i = 0; i < ALPHABET_SIZE; i++)
                children[i] = null;
        }
    };
    static TrieNode root;

    // If not present, inserts key into trie
    // If the key is prefix of trie node, just
    // marks leaf node
    static void insert(String key)
    {
        int length = key.length();

        TrieNode pCrawl = root;

        for (int level = 0; level < length; level++)
        {
            int index = key.charAt(level) - 'a';
            if (pCrawl.children[index] == null)
                pCrawl.children[index] = new TrieNode();

            pCrawl = pCrawl.children[index];
        }

        // mark last node as leaf
        pCrawl.isLeaf = true;
    }

    // Function to count number of words
    static int wordCount(TrieNode root)
    {
        int result = 0;
```

```
// Leaf denotes end of a word
if (root.isLeaf)
    result++;

for (int i = 0; i < ALPHABET_SIZE; i++)
    if (root.children[i] != null )
        result += wordCount(root.children[i]);

return result;
}

// Driver Program
public static void main(String args[])
{
    // Input keys (use only 'a' through 'z'
    // and lower case)
    String keys[] = {"the", "a", "there", "answer",
                     "any", "by", "bye", "their"};

    root = new TrieNode();

    // Construct Trie
    for (int i = 0; i < keys.length; i++)
        insert(keys[i]);

    System.out.println(wordCount(root));
}
}

// This code is contributed by Sumit Ghosh
```

Output:

8

Source

<https://www.geeksforgeeks.org/counting-number-words-trie/>

Chapter 40

Data Structure for Dictionary and Spell Checker?

Data Structure for Dictionary and Spell Checker? - GeeksforGeeks

Which data structure can be used for efficiently building a word dictionary and Spell Checker?

The answer depends upon the functionalists required in Spell Checker and availability of memory. For example following are few possibilities.

Hashing is one simple option for this. We can put all words in a hash table. Refer [this paper](#) which compares hashing with self-balancing Binary Search Trees and Skip List, and shows that hashing performs better.

Hashing doesn't support operations like prefix search. Prefix search is something where a user types a prefix and your dictionary shows all words starting with that prefix. Hashing also doesn't support efficient printing of all words in dictionary in alphabetical order and nearest neighbor search.

If we want both operations, look up and prefix search, **Trie** is suited. With Trie, we can support all operations like insert, search, delete in $O(n)$ time where n is length of the word to be processed. Another advantage of Trie is, we can print all words in alphabetical order which is not possible with hashing.

The disadvantage of Trie is, it requires lots of space. If space is concern, then **Ternary Search Tree** can be preferred. In Ternary Search Tree, time complexity of search operation is $O(h)$ where h is height of the tree. Ternary Search Trees also supports other operations supported by Trie like prefix search, alphabetical order printing and nearest neighbor search.

If we want to support suggestions, like google shows “*did you mean ...*”, then we need to find the closest word in dictionary. The closest word can be defined as the word that can be obtained with minimum number of character transformations (add, delete, replace). A Naive way is to take the given word and generate all words which are 1 distance (1 edit or 1 delete or 1 replace) away and one by one look them in dictionary. If nothing found, then look for all words which are 2 distant and so on. There are many complex algorithms for

this. As per[the wiki page](#), The most successful algorithm to date is Andrew Golding and Dan Roth's Window-based spelling correction algorithm.

See [this](#)for a simple spell checker implementation.

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

Source

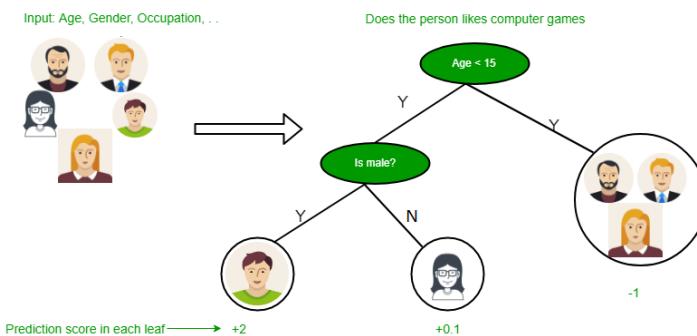
<https://www.geeksforgeeks.org/data-structure-dictionary-spell-checker/>

Chapter 41

Decision Tree Introduction with example

Decision Tree Introduction with example - GeeksforGeeks

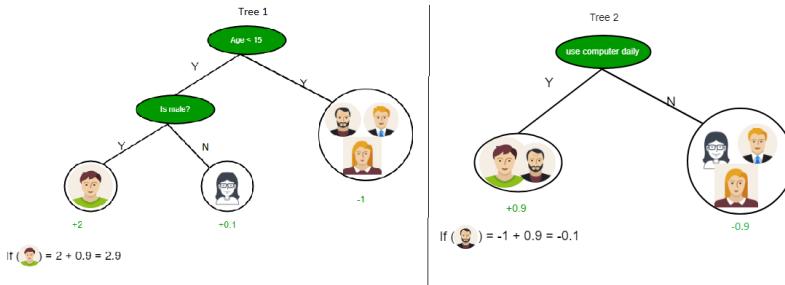
- Decision tree algorithm falls under the category of the supervised learning. They can be used to solve both regression and classification problems.
- Decision tree uses the tree representation to solve the problem in which each leaf node corresponds to a class label and attributes are represented on the internal node of the tree.
- We can represent any boolean function on discrete attributes using the decision tree.



Below are some assumptions that we made while using decision tree:

- At the beginning, we consider the whole training set as the root.
- Feature values are preferred to be categorical. If the values are continuous then they are discretized prior to building the model.
- On the basis of attribute values records are distributed recursively.

- We use statistical methods for ordering attributes as root or the internal node.



As you can see from the above image that Decision Tree works on the Sum of Product form which is also known as *Disjunctive Normal Form*. In the above image we are predicting the use of computer in daily life of the people.

In Decision Tree the major challenge is to identification of the attribute for the root node in each level. This process is known as attribute selection. We have two popular attribute selection measures:

1. Information Gain
2. Gini Index

1. Information Gain

When we use a node in a decision tree to partition the training instances into smaller subsets the entropy changes. Information gain is a measure of this change in entropy.

Definition: Suppose S is a set of instances, A is an attribute, S_v is the subset of S with $A = v$, and $\text{Values}(A)$ is the set of all possible values of A , then

$$\text{Information}(S, A) = \text{Entropy}(S) - \frac{\sum_{v \in \text{Values}(A)} |S_v|}{|S|} \cdot \text{Entropy}(S_v)$$

Entropy

Entropy is the measure of uncertainty of a random variable, it characterizes the impurity of an arbitrary collection of examples. The higher the entropy more the information content.

Definition: Suppose S is a set of instances, A is an attribute, S_v is the subset of S with $A = v$, and $\text{Values}(A)$ is the set of all possible values of A , then

$$\text{Entropy}(S) = -\sum_{v \in \text{Values}(A)} \frac{|S_v|}{|S|} \cdot \text{Entropy}(S_v)$$

Example:

For the set $X = \{a, a, a, b, b, b, b, b\}$

Total instances: 8

Instances of b : 5

Instances of a : 3

$$\begin{aligned} &= -[0.375 * (-1.415) + 0.625 * (-0.678)] \\ &= -(-0.53 - 0.424) \\ &= 0.954 \end{aligned}$$

Building Decision Tree using Information Gain

The essentials:

- Start with all training instances associated with the root node
- Use info gain to choose which attribute to label each node with
- *Note:* No root-to-leaf path should contain the same discrete attribute twice
- Recursively construct each subtree on the subset of training instances that would be classified down that path in the tree.
- If all positive or all negative training instances remain, label that node “yes” or “no” accordingly
- If no attributes remain, label with a majority vote of training instances left at that node
- If no instances remain, label with a majority vote of the parent’s training instances

Example:

Now, lets draw a Decision Tree for the following data using Information gain.

Training set: 3 features and 2 classes

X

Y

Z

C

1

1

1

I

1

1

0

I

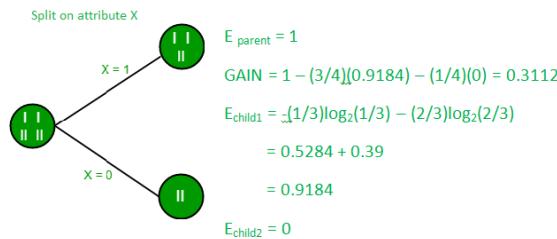
0

0

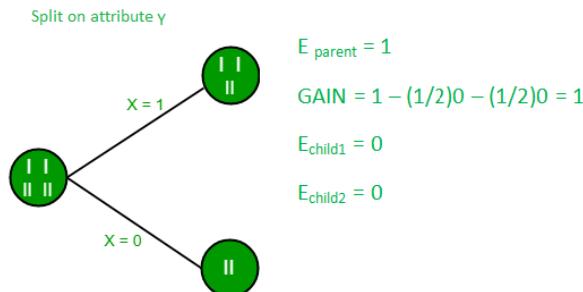
1
II
1
0
0
II

Here, we have 3 features and 2 output classes.

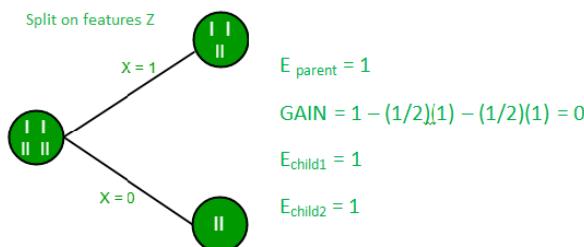
To build a decision tree using Information gain. We will take each of the feature and calculate the information for each feature.



Split on feature X



Split on feature Y

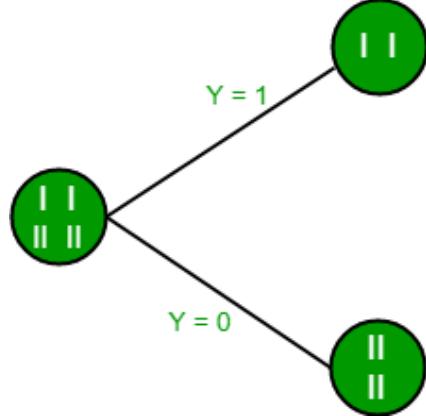


Split on feature Z

From the above images we can see that the information gain is maximum when we make a split on feature Y. So, for the root node best suited feature is feature Y. Now we can see

that while splitting the dataset by feature Y, the child contains pure subset of the target variable. So we don't need to further split the dataset.

The final tree for the above dataset would look like this:



2. Gini Index

- Gini Index is a metric to measure how often a randomly chosen element would be incorrectly identified.
- It means an attribute with lower Gini index should be preferred.
- Sklearn supports “Gini” criteria for Gini Index and by default, it takes “gini” value.
- The Formula for the calculation of the Gini Index is given below.

A	B	C	D
≥ 5	≥ 3.0	≥ 4.2	≥ 1.4
< 5	< 3.0	< 4.2	< 1.4

Calculating Gini Index for Var A:

Value ≥ 5 : 12

Attribute A ≥ 5 & class = positive: $\frac{3}{12}$

Attribute A ≥ 5 & class = negative: $\frac{1}{12}$

Gini(5, 7) = $1 - \left[\frac{3}{12} + \frac{1}{12} \right] = 0.4667$

Value < 5 : 4

Attribute A < 5 & class = positive: $\frac{3}{4}$

Attribute A < 5 & class = negative: $\frac{1}{4}$

$$\text{Gini}(3, 1) = 1 - \left[\frac{5}{16} + \frac{11}{16} \right] = 0.375$$

Gini(3, 1) = $1 - \left[\frac{5}{16} + \frac{11}{16} \right]$
By adding weight and sum each of the gini indices:

$$\text{Gini}(T_{\text{Attr} A}, A) = \left(\frac{5}{16} \right) * (0.486) + \left(\frac{11}{16} \right) * (0.375) = 0.45825$$

Calculating Gini Index for Var B:

Value ≥ 3 : 12

$$\text{Attribute B } \geq 3 \text{ & class = positive: } \frac{8}{12}$$

$$\text{Attribute B } \geq 5 \text{ & class = negative: } \frac{4}{12}$$

$$\text{Gini}(5, 7) = 1 - \left[\frac{8}{12} + \frac{4}{12} \right] = 0.44444$$

Value < 3 : 4

$$\text{Attribute A } < 3 \text{ & class = positive: } \frac{4}{4}$$

$$\text{Attribute A } < 3 \text{ & class = negative: } \frac{4}{4}$$

$$\text{Gini}(3, 1) = 1 - \left[\frac{4}{4} + \frac{4}{4} \right] = 0$$

By adding weight and sum each of the gini indices:

$$\text{Gini}(T_{\text{Attr} B}, B) = \left(\frac{8}{12} \right) * (0.44444) + \left(\frac{4}{12} \right) * (0) = 0.3345$$

Using the same approach we can calculate the Gini index for C and D attributes.

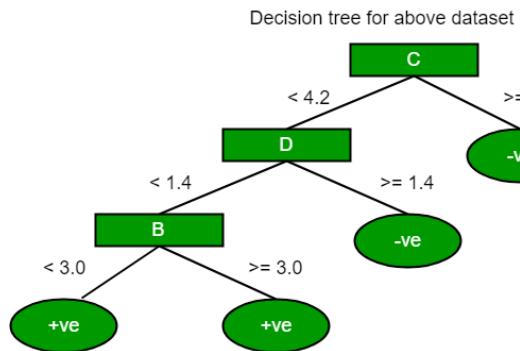
	Positive	Negative
For $A \geq 5.0$	5	7
< 5	3	1
Gini Index of A	= 0.45825	

	Positive	Negative
For $B \geq 3.0$	8	4
< 3.0	0	4
Gini Index of B	= 0.3345	

	Positive	Negative
For $C \geq 4.2$	0	6
< 4.2	8	2

Gini Index of C= 0.2

	Positive	Negative
For D >= 1.4	0	5
< 1.4	8	3
Gini Index of D=	0.273	



Reference: [dataaspirant](#)

Source

<https://www.geeksforgeeks.org/decision-tree-introduction-example/>

Chapter 42

Decision Trees – Fake (Counterfeit) Coin Puzzle (12 Coin Puzzle)

Decision Trees - Fake (Counterfeit) Coin Puzzle (12 Coin Puzzle) - GeeksforGeeks

Let us solve the classic “fake coin” puzzle using decision trees. There are the two different variants of the puzzle given below. I am providing description of both the puzzles below, try to solve on your own, assume $N = 8$.

Easy: Given a two pan fair balance and N identically looking coins, out of which only one coin is **lighter** (or **heavier**). To figure out the odd coin, how many minimum number of weighing are required in the worst case?

Difficult: Given a two pan fair balance and N identically looking coins out of which only one coin **may be** defective. How can we trace which coin, if any, is odd one and also determine whether it is lighter or heavier in minimum number of trials in the worst case?

Let us start with relatively simple examples. After reading every problem try to solve on your own.

Problem 1: (Easy)

Given 5 coins out of which one coin is **lighter**. In the worst case, how many minimum number of weighing are required to figure out the odd coin?

Name the coins as 1, 2, 3, 4 and 5. We know that one coin is lighter. Considering best out come of balance, we can group the coins in two different ways, [(1, 2), (3, 4) and (5)], or [(12), (34) and (5)]. We can easily rule out groups like [(123) and (45)], as we will get obvious answer. Any other combination will fall into one of these two groups, like [(2)(45) and (13)], etc.

Consider the first group, pairs (1, 2) and (3, 4). We can check (1, 2), if they are equal we go ahead with (3, 4). We need two weighing in worst case. The same analogy can be applied when the coin is heavier.

With the second group, weigh (12) and (34). If they balance (5) is defective one, otherwise pick the lighter pair, and we need one more weighing to find odd one.

Both the combinations need two weighing in case of 5 coins with prior information of one coin is lighter.

Analysis: In general, if we know that the coin is heavy or light, we can trace the coin in $\log_3(N)$ trials (rounded to next integer). If we represent the outcome of balance as ternary tree, every leaf represent an outcome. Since any coin among N coins can be defective, we need to get a 3-ary tree having minimum of N leaves. A 3-ary tree at k-th level will have 3^k leaves and hence we need $3^k \geq N$.

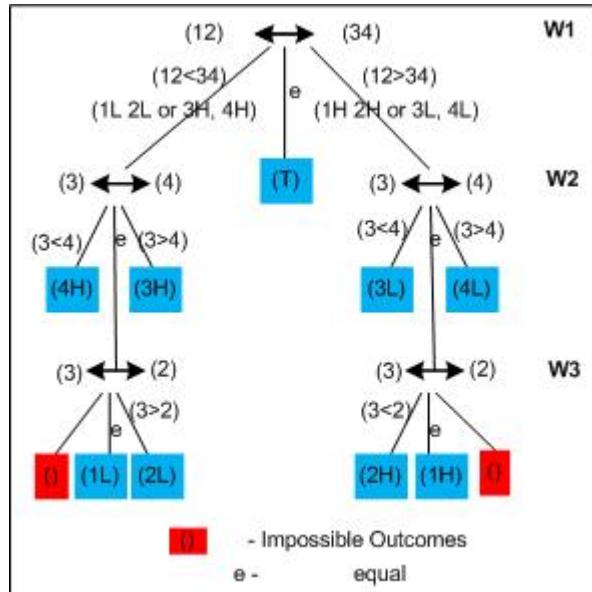
In other-words, in k trials we can examine upto 3^k coins, if we know whether the defective coin is heavier or lighter. Given that a coin is heavier, verify that 3 trials are sufficient to find the odd coin among 12 coins, because $3^2 < 12 < 3^3$.

Problem 2: (Difficult)

We are given 4 coins, out of which only one coin **may be** defective. We don't know, whether all coins are genuine or any defective one is present. How many number of weighing are required in worst case to figure out the odd coin, if present? We also need to tell whether it is heavier or lighter.

From the above analysis we may think that $k = 2$ trials are sufficient, since a two level 3-ary tree yields 9 leaves which is greater than $N = 4$ (read the problem once again). Note that it is impossible to solve above 4 coins problem in two weighing. The decision tree confirms the fact (try to draw).

We can group the coins in two different ways, [(12, 34)] or [(1, 2) and (3, 4)]. Let us consider the combination (12, 34), the corresponding decision tree is given below. Blue leaves are valid outcomes, and red leaves are impossible cases. We arrived at impossible cases due to the assumptions made earlier on the path.



The outcome can be $(12) < (34)$ i.e. we go on to left subtree or $(12) > (34)$ i.e. we go on to right subtree.

The left subtree is possible in two ways,

- A) Either 1 or 2 can be lighter OR
- B) Either 3 or 4 can be heavier.

Further on the left subtree, as second trial, we weigh $(1, 2)$ or $(3, 4)$. Let us consider $(3, 4)$ as the analogy for $(1, 2)$ is similar. The outcome of second trial can be three ways

- A) $(3) < (4)$ yielding 4 as defective heavier coin, OR
- B) $(3) > (4)$ yielding 3 as defective heavier coin OR
- C) $(3) = (4)$, yielding ambiguity. Here we need one more weighing to check a genuine coin against 1 or 2. In the figure I took $(3, 2)$ where 3 is confirmed as genuine. We can get $(3) > (2)$ in which 2 is lighter, or $(3) = (2)$ in which 1 is lighter. Note that it is impossible to get $(3) < (2)$, it contradicts our assumption leaned to left side.

Similarly we can analyze the right subtree. We need two more weighings on right subtree as well.

Overall we need 3 weighings to trace the odd coin. Note that we are unable to utilize two outcomes of 3-ary trees. Also, the tree is not full tree, middle branch terminated after first weighing. Infact, we can get 27 leaves of 3 level full 3-ary tree, but only we got 11 leaves including impossible cases.

Analysis: Given N coins, all may be genuine or only one coin is defective. We need a decision tree with atleast $(2N + 1)$ leaves correspond to the outputs. Because there can be N leaves to be lighter, or N leaves to be heavier or one genuine case, on total $(2N + 1)$ leaves.

As explained earlier ternary tree at level k, can have utmost 3^k leaves and we need a tree with leaves of $3^k > (2N + 1)$.

In other words, we need atleast $k > \log_3(2N + 1)$ weighing to find the defective one.

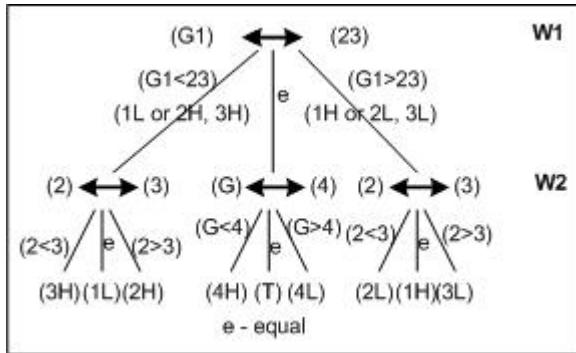
Observe the above figure that not all the branches are generating leaves, i.e. we are missing valid outputs under some branches that leading to more number of trials. When possible, we should group the coins in such a way that every branch is going to yield valid output (in simple terms generate full 3-ary tree). Problem 4 describes this approach of 12 coins.

Problem 3: (Special case of two pan balance)

We are given 5 coins, a group of 4 coins out of which one coin is defective (we don't know whether it is heavier or lighter), and one coin is genuine. How many weighing are required in worst case to figure out the odd coin whether it is heavier or lighter?

Label the coins as 1, 2, 3, 4 and G (genuine). We now have some information on coin purity. We need to make use that in the groupings.

We can best group them as $[(G1, 23) \text{ and } (4)]$. Any other group can't generate full 3-ary tree, try yourself. The following diagram explains the procedure.



The middle case $(G1) = (23)$ is self explanatory, i.e. 1, 2, 3 are genuine and 4th coin can be figured out lighter or heavier in one more trial.

The left side of tree corresponds to the case $(G1) < (23)$. This is possible in two ways, either 1 should be lighter or either of (2, 3) should be heavier. The former instance is obvious when next weighing (2, 3) is balanced, yielding 1 as lighter. The later instance could be $(2) < (3)$ yielding 3 as heavier or $(2) > (3)$ yielding 2 as heavier. The leaf nodes on left branch are named to reflect these outcomes.

The right side of tree corresponds to the case $(G1) > (23)$. This is possible in two ways, either 1 is heavier or either of (2, 3) should be lighter. The former instance is obvious when the next weighing (2, 3) is balanced, yielding 1 as heavier. The later case could be $(2) < (3)$ yielding 2 as lighter coin, or $(2) > (3)$ yielding 3 as lighter.

In the above problem, under any possibility we need only two weighing. We are able to use all outcomes of two level full 3-ary tree. We started with $(N + 1) = 5$ coins where $N = 4$, we end up with $(2N + 1) = 9$ leaves. *Infact we should have 11 outcomes since we stared with 5 coins, where are other 2 outcomes? These two outcomes can be declared at the root of tree itself (prior to first weighing), can you figure these two outcomes?*

If we observe the figure, after the first weighing the problem reduced to “we know three coins, either one can be lighter (heavier) or one among other two can be heavier (lighter)”. This can be solved in one weighing (read Problem 1).

Analysis: Given $(N + 1)$ coins, one is genuine and the rest N can be genuine or only one coin is defective. The required decision tree should result in minimum of $(2N + 1)$ leaves. Since the total possible outcomes are $(2(N + 1) + 1)$, number of weighing (trials) are given by the height of ternary tree, $k \geq \log_3[2(N + 1) + 1]$. Note the equality sign.

Rearranging k and N, we can weigh maximum of $N \leq (3^k - 3)/2$ coins in k trials.

Problem 4: (The classic 12 coin puzzle)

You are given two pan fair balance. You have 12 identically looking coins out of which one coin may be lighter or heavier. How can you find odd coin, if any, in minimum trials, also determine whether defective coin is lighter or heavier, in the worst case?

How do you want to group them? Bi-set or tri-set? Clearly we can discard the option of dividing into two equal groups. It can't lead to best tree. *From the above two examples, we can ensure that the decision tree can be used in optimal way if we can reveal atleast one genuine coin.* Remember to group coins such that the first weighing reveals atleast one genuine coin.

Let us name the coins as 1, 2, ... 8, A, B, C and D. We can combine the coins into 3 groups, namely (1234), (5678) and (ABCD). Weigh (1234) and (5678). You are encouraged to draw decision tree while reading the procedure. The outcome can be three ways,

1. $(1234) = (5678)$, both groups are equal. Defective coin may be in (ABCD) group.
2. $(1234) < (5678)$, i.e. first group is less in weight than second group.
3. $(1234) > (5678)$, i.e. first group is more in weight than second group.

The output (1) can be solved in two more weighing as special case of two pan balance given in Problem 3. We know that groups (1234) and (5678) are genuine and defective coin may be in (ABCD). Pick one genuine coin from any of weighed groups, and proceed with (ABCD) as explained in Problem 3.

Outcomes (2) and (3) are special. In both the cases, we know that (ABCD) is genuine. And also, we know a set of coins being lighter and a set of coins being heavier. We need to shuffle the weighed two groups in such a way that we end up with smaller height decision tree.

Consider the second outcome where $(1234) < (5678)$. It is possible when any coin among (1, 2, 3, 4) is lighter or any coin among (5, 6, 7, 8) is heavier. We revealed lighter or heavier possibility after first weighing. If we proceed as in Problem 1, we will not generate best decision tree. Let us shuffle coins as (1235) and (4BCD) as new groups (there are different shuffles possible, they also lead to minimum weighing). If we weigh these two groups again the outcome can be three ways, i) $(1235) < (4BCD)$ yielding one among 1, 2, 3 is lighter which is similar to Problem 1 explained above, we need one more weighing, ii) $(1235) = (4BCD)$ yielding one among 6, 7, 8 is heavier which is similar to Problem 1 explained above, we need one more weighing iii) $(1235) > (4BCD)$ yielding either 5 as heavier coin or 4 as lighter coin, at the expense of one more weighing.

Similar way we can also solve the right subtree (third outcome where $(1234) > (5678)$) in two more weighing.

We are able to solve the 12 coin puzzle in 3 weighing in the worst case.

Few Interesting Puzzles:

1. Solve Problem 4 with $N = 8$ and $N = 13$, How many minimum trials are required in each case?
2. Given a function $\text{int weigh}(\text{A}\[], \text{B}\[])$ where A and B are arrays (need not be equal size). The function returns -1, 0 or 1. It returns 0 if sum of all elements in A and B are equal, -1 if $A < B$ and 1 if $A > B$. Given an array of 12 elements, all elements are equal except one. The odd element can be as that of others, smaller or greater than others. Write a program to find the odd element (if any) using $\text{weigh}()$ minimum number of times.

3. You might have seen 3-pan balance in science labs during school days. Given a 3-pan balance (4 outcomes) and N coins, how many minimum trials are needed to figure out odd coin?

References:

Similar problem was provided in one of the exercises of the book “Introduction to Algorithms by Levitin”. Specifically read section 5.5 and section 11.2 including exercises.

--- by [Venki](#). 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/decision-trees-fake-coin-puzzle/>

Chapter 43

Design a data structure that supports insert, delete, search and getRandom in constant time

Design a data structure that supports insert, delete, search and getRandom in constant time
- GeeksforGeeks

Design a data structure that supports following operations in $\Theta(1)$ time.

`insert(x)`: Inserts an item x to the data structure if not already present.

`remove(x)`: Removes an item x from the data structure if present.

`search(x)`: Searches an item x in the data structure.

`getRandom()`: Returns a random element from current set of elements

We can use [hashing](#) to support first 3 operations in $\Theta(1)$ time. How to do the 4th operation?
The idea is to use a resizable array (ArrayList in Java, vector in C) together with hashing.
[Resizable arrays support insert in \$\Theta\(1\)\$ amortized time complexity](#). To implement `getRandom()`, we can simply pick a random number from 0 to size-1 (size is number of current elements) and return the element at that index. The hash map stores array values as keys and array indexes as values.

Following are detailed operations.

insert(x)

- 1) Check if x is already present by doing a hash map lookup.
- 2) If not present, then insert it at the end of the array.
- 3) Add in hash table also, x is added as key and last array index as index.

remove(x)

- 1) Check if x is present by doing a hash map lookup.

- 2) If present, then find its index and remove it from hash map.
- 3) Swap the last element with this element in array and remove the last element.
Swapping is done because the last element can be removed in O(1) time.
- 4) Update index of last element in hash map.

getRandom()

- 1) Generate a random number from 0 to last index.
- 2) Return the array element at the randomly generated index.

search(x)

Do a lookup for x in hash map.

Below is implementation of the data structure.

Java

```
/* Java program to design a data structure that support folloiwng operations
in Theta(n) time
a) Insert
b) Delete
c) Search
d) getRandom */
import java.util.*;

// class to represent the required data structure
class MyDS
{
    ArrayList<Integer> arr; // A resizable array

    // A hash where keys are array elements and vlaues are
    // indexes in arr[]
    HashMap<Integer, Integer> hash;

    // Constructor (creates arr[] and hash)
    public MyDS()
    {
        arr = new ArrayList<Integer>();
        hash = new HashMap<Integer, Integer>();
    }

    // A Theta(1) function to add an element to MyDS
    // data structure
    void add(int x)
    {
        // If ekement is already present, then noting to do
        if (hash.get(x) != null)
            return;

        // Else put element at the end of arr[]
        int s = arr.size();
```

```
arr.add(x);

// And put in hash also
hash.put(x, s);
}

// A Theta(1) function to remove an element from MyDS
// data structure
void remove(int x)
{
    // Check if element is present
    Integer index = hash.get(x);
    if (index == null)
        return;

    // If present, then remove element from hash
    hash.remove(x);

    // Swap element with last element so that remove from
    // arr[] can be done in O(1) time
    int size = arr.size();
    Integer last = arr.get(size-1);
    Collections.swap(arr, index, size-1);

    // Remove last element (This is O(1))
    arr.remove(size-1);

    // Update hash table for new index of last element
    hash.put(last, index);
}

// Returns a random element from MyDS
int getRandom()
{
    // Find a random index from 0 to size - 1
    Random rand = new Random(); // Choose a different seed
    int index = rand.nextInt(arr.size());

    // Return element at randomly picked index
    return arr.get(index);
}

// Returns index of element if element is present, otherwise null
Integer search(int x)
{
    return hash.get(x);
}
```

```
// Driver class
class Main
{
    public static void main (String[] args)
    {
        MyDS ds = new MyDS();
        ds.add(10);
        ds.add(20);
        ds.add(30);
        ds.add(40);
        System.out.println(ds.search(30));
        ds.remove(20);
        ds.add(50);
        System.out.println(ds.search(50));
        System.out.println(ds.getRandom());
    }
}
```

C++

```
/* C++ program to design a DS that supports following operations
in Theta(n) time
a) Insert
b) Delete
c) Search
d) getRandom */

#include<bits/stdc++.h>
using namespace std;

// class to represent the required data structure
class myStructure
{
    // A resizable array
    vector <int> arr;

    // A hash where keys are array elements and values are
    // indexes in arr[]
    map <int, int> Map;

public:
    // A Theta(1) function to add an element to MyDS
    // data structure
    void add(int x)
    {
        // If element is already present, then nothing to do
        if(Map.find(x) != Map.end())
```

```
return;

// Else put element at the end of arr[]
int index = arr.size();
arr.push_back(x);

// and hashmap also
Map.insert(std::pair<int,int>(x, index));
}

// function to remove a number to DS in O(1)
void remove(int x)
{
    // element not found then return
    if(Map.find(x) == Map.end())
        return;

    // remove element from map
    int index = Map.at(x);
    Map.erase(x);

    // swap with last element in arr
    // then remove element at back
    int last = arr.size() - 1;
    swap(arr[index], arr[last]);
    arr.pop_back();

    // Update hash table for new index of last element
    Map.at(arr[index]) = index;
}

// Returns index of element if element is present, otherwise null
int search(int x)
{
    if(Map.find(x) != Map.end())
        return Map.at(x);
    return -1;
}

// Returns a random element from myStructure
int getRandom()
{
    // Find a random index from 0 to size - 1
    srand (time(NULL));
    int random_index = rand() % arr.size();

    // Return element at randomly picked index
    return arr.at(random_index);
```

```
    }
};

// Driver main
int main()
{
    myStructure ds;
    ds.add(10);
    ds.add(20);
    ds.add(30);
    ds.add(40);
    cout << ds.search(30) << endl;
    ds.remove(20);
    ds.add(50);
    cout << ds.search(50) << endl;
    cout << ds.getRandom() << endl;
}

// This code is contributed by Aditi Sharma
```

Output:

```
2
3
40
```

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

Improved By : [SwapnilShukla1](#)

Source

<https://www.geeksforgeeks.org/design-a-data-structure-that-supports-insert-delete-search-and-getrandom-in-constant-time/>

Chapter 44

Design an efficient data structure for given operations

Design an efficient data structure for given operations - GeeksforGeeks

Design a Data Structure for the following operations. The data structure should be efficient enough to accommodate the operations according to their frequency.

- 1) `findMin()` : Returns the minimum item.
Frequency: Most frequent
- 2) `findMax()` : Returns the maximum item.
Frequency: Most frequent
- 3) `deleteMin()` : Delete the minimum item.
Frequency: Moderate frequent
- 4) `deleteMax()` : Delete the maximum item.
Frequency: Moderate frequent
- 5) `Insert()` : Inserts an item.
Frequency: Least frequent
- 6) `Delete()` : Deletes an item.
Frequency: Least frequent.

A **simple solution** is to maintain a sorted array where smallest element is at first position and largest element is at last. The time complexity of `findMin()`, `findMax()` and `deleteMax()` is $O(1)$. But time complexities of `deleteMin()`, `insert()` and `delete()` will be $O(n)$.

Can we do the most frequent two operations in $O(1)$ and other operations in $O(\log n)$ time?

The idea is to use two binary heaps (one max and one min heap). The main challenge is, while deleting an item, we need to delete from both min-heap and max-heap. So, we need some kind of mutual data structure. In the following design, we have used doubly linked list as a mutual data structure. The doubly linked list contains all input items and indexes of corresponding min and max heap nodes. The nodes of min and max heaps store addresses of nodes of doubly linked list. The root node of min heap stores the address of minimum item in doubly linked list. Similarly, root of max heap stores address of maximum item in doubly linked list. Following are the details of operations.

1) findMax(): We get the address of maximum value node from root of Max Heap. So this is a O(1) operation.

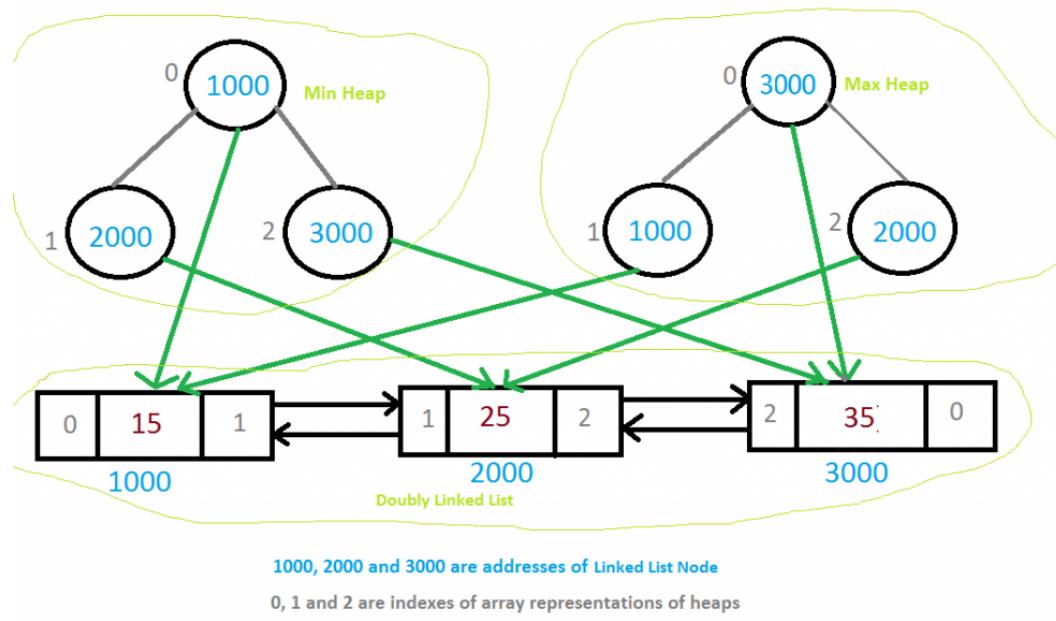
1) findMin(): We get the address of minimum value node from root of Min Heap. So this is a O(1) operation.

3) deleteMin(): We get the address of minimum value node from root of Min Heap. We use this address to find the node in doubly linked list. From the doubly linked list, we get node of Max Heap. We delete node from all three. We can delete a node from doubly linked list in O(1) time. delete() operations for max and min heaps take O(Logn) time.

4) deleteMax(): is similar to deleteMin()

5) Insert() We always insert at the beginning of linked list in O(1) time. Inserting the address in Max and Min Heaps take O(Logn) time. So overall complexity is O(Logn)

6) Delete() We first search the item in Linked List. Once the item is found in O(n) time, we delete it from linked list. Then using the indexes stored in linked list, we delete it from Min Heap and Max Heaps in O(Logn) time. *So overall complexity of this operation is O(n). The Delete operation can be optimized to O(Logn) by using a balanced binary search tree instead of doubly linked list as a mutual data structure. Use of balanced binary search will not effect time complexity of other operations as it will act as a mutual data structure like doubly Linked List.*



Following is C implementation of the above data structure.

```
// C program for efficient data structure
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

// A node of doubly linked list
struct LNode
{
    int data;
    int minHeapIndex;
    int maxHeapIndex;
    struct LNode *next, *prev;
};

// Structure for a doubly linked list
struct List
{
    struct LNode *head;
};

// Structure for min heap
struct MinHeap
{
```

```
int size;
int capacity;
struct LNode* *array;
};

// Structure for max heap
struct MaxHeap
{
    int size;
    int capacity;
    struct LNode* *array;
};

// The required data structure
struct MyDS
{
    struct MinHeap* minHeap;
    struct MaxHeap* maxHeap;
    struct List* list;
};

// Function to swap two integers
void swapData(int* a, int* b)
{ int t = *a; *a = *b; *b = t; }

// Function to swap two List nodes
void swapLNode(struct LNode** a, struct LNode** b)
{ struct LNode* t = *a; *a = *b; *b = t; }

// A utility function to create a new List node
struct LNode* newLNode(int data)
{
    struct LNode* node =
        (struct LNode*) malloc(sizeof(struct LNode));
    node->minHeapIndex = node->maxHeapIndex = -1;
    node->data = data;
    node->prev = node->next = NULL;
    return node;
}

// Utility function to create a max heap of given capacity
struct MaxHeap* createMaxHeap(int capacity)
{
    struct MaxHeap* maxHeap =
        (struct MaxHeap*) malloc(sizeof(struct MaxHeap));
    maxHeap->size = 0;
    maxHeap->capacity = capacity;
    maxHeap->array =
```

```
(struct LNode**) malloc(maxHeap->capacity * sizeof(struct LNode*));
return maxHeap;
}

// Utility function to create a min heap of given capacity
struct MinHeap* createMinHeap(int capacity)
{
    struct MinHeap* minHeap =
        (struct MinHeap*) malloc(sizeof(struct MinHeap));
    minHeap->size = 0;
    minHeap->capacity = capacity;
    minHeap->array =
        (struct LNode**) malloc(minHeap->capacity * sizeof(struct LNode*));
    return minHeap;
}

// Utility function to create a List
struct List* createList()
{
    struct List* list =
        (struct List*) malloc(sizeof(struct List));
    list->head = NULL;
    return list;
}

// Utility function to create the main data structure
// with given capacity
struct MyDS* createMyDS(int capacity)
{
    struct MyDS* myDS =
        (struct MyDS*) malloc(sizeof(struct MyDS));
    myDS->minHeap = createMinHeap(capacity);
    myDS->maxHeap = createMaxHeap(capacity);
    myDS->list = createList();
    return myDS;
}

// Some basic operations for heaps and List
int isEmpty(struct MaxHeap* heap)
{   return (heap->size == 0); }

int isEmpty(struct MinHeap* heap)
{   return heap->size == 0; }

int isFull(struct MaxHeap* heap)
{   return heap->size == heap->capacity; }

int isFull(struct MinHeap* heap)
```

```
{  return heap->size == heap->capacity; }

int isEmpty(struct List* list)
{  return !list->head;  }

int hasOnlyOneLNode(struct List* list)
{  return !list->head->next && !list->head->prev; }

// The standard minheapify function. The only thing it does extra
// is swapping indexes of heaps inside the List
void minHeapify(struct MinHeap* minHeap, int index)
{
    int smallest, left, right;
    smallest = index;
    left = 2 * index + 1;
    right = 2 * index + 2;

    if ( minHeap->array[left] &&
        left < minHeap->size &&
        minHeap->array[left]->data < minHeap->array[smallest]->data
    )
        smallest = left;

    if ( minHeap->array[right] &&
        right < minHeap->size &&
        minHeap->array[right]->data < minHeap->array[smallest]->data
    )
        smallest = right;

    if (smallest != index)
    {
        // First swap indexes inside the List using address
        // of List nodes
        swapData(&(minHeap->array[smallest]->minHeapIndex),
                 &(minHeap->array[index]->minHeapIndex));

        // Now swap pointers to List nodes
        swapLNode(&minHeap->array[smallest],
                  &minHeap->array[index]);

        // Fix the heap downward
        minHeapify(minHeap, smallest);
    }
}

// The standard maxHeapify function. The only thing it does extra
// is swapping indexes of heaps inside the List
```

```

void maxHeapify(struct MaxHeap* maxHeap, int index)
{
    int largest, left, right;
    largest = index;
    left = 2 * index + 1;
    right = 2 * index + 2;

    if ( maxHeap->array[left] &&
        left < maxHeap->size &&
        maxHeap->array[left]->data > maxHeap->array[largest]->data
    )
        largest = left;

    if ( maxHeap->array[right] &&
        right < maxHeap->size &&
        maxHeap->array[right]->data > maxHeap->array[largest]->data
    )
        largest = right;

    if (largest != index)
    {
        // First swap indexes inside the List using address
        // of List nodes
        swapData(&maxHeap->array[largest]->maxHeapIndex,
                 &maxHeap->array[index]->maxHeapIndex);

        // Now swap pointers to List nodes
        swapLNode(&maxHeap->array[largest],
                  &maxHeap->array[index]);

        // Fix the heap downward
        maxHeapify(maxHeap, largest);
    }
}

// Standard function to insert an item in Min Heap
void insertMinHeap(struct MinHeap* minHeap, struct LNode* temp)
{
    if (isMinHeapFull(minHeap))
        return;

    ++minHeap->size;
    int i = minHeap->size - 1;
    while (i && temp->data < minHeap->array[(i - 1) / 2]->data )
    {
        minHeap->array[i] = minHeap->array[(i - 1) / 2];
        minHeap->array[i]->minHeapIndex = i;
        i = (i - 1) / 2;
    }
}

```

```
}

minHeap->array[i] = temp;
minHeap->array[i]->minHeapIndex = i;
}

// Standard function to insert an item in Max Heap
void insertMaxHeap(struct MaxHeap* maxHeap, struct LNode* temp)
{
    if (isMaxHeapFull(maxHeap))
        return;

    ++maxHeap->size;
    int i = maxHeap->size - 1;
    while (i && temp->data > maxHeap->array[(i - 1) / 2]->data )
    {
        maxHeap->array[i] = maxHeap->array[(i - 1) / 2];
        maxHeap->array[i]->maxHeapIndex = i;
        i = (i - 1) / 2;
    }

    maxHeap->array[i] = temp;
    maxHeap->array[i]->maxHeapIndex = i;
}

// Function to find minimum value stored in the main data structure
int findMin(struct MyDS* myDS)
{
    if (isMinHeapEmpty(myDS->minHeap))
        return INT_MAX;

    return myDS->minHeap->array[0]->data;
}

// Function to find maximum value stored in the main data structure
int findMax(struct MyDS* myDS)
{
    if (isMaxHeapEmpty(myDS->maxHeap))
        return INT_MIN;

    return myDS->maxHeap->array[0]->data;
}

// A utility function to remove an item from linked list
void removeLNode(struct List* list, struct LNode** temp)
{
    if (hasOnlyOneLNode(list))
```

```

list->head = NULL;

else if (!(*temp)->prev) // first node
{
    list->head = (*temp)->next;
    (*temp)->next->prev = NULL;
}
// any other node including last
else
{
    (*temp)->prev->next = (*temp)->next;
    // last node
    if ((*temp)->next)
        (*temp)->next->prev = (*temp)->prev;
}
free(*temp);
*temp = NULL;
}

// Function to delete maximum value stored in the main data structure
void deleteMax(struct MyDS* myDS)
{
    MinHeap *minHeap = myDS->minHeap;
    MaxHeap *maxHeap = myDS->maxHeap;

    if (isMaxHeapEmpty(maxHeap))
        return;
    struct LNode* temp = maxHeap->array[0];

    // delete the maximum item from maxHeap
    maxHeap->array[0] =
        maxHeap->array[maxHeap->size - 1];
    --maxHeap->size;
    maxHeap->array[0]->maxHeapIndex = 0;
    maxHeapify(maxHeap, 0);

    // remove the item from minHeap
    minHeap->array[temp->minHeapIndex] = minHeap->array[minHeap->size - 1];
    --minHeap->size;
    minHeap->array[temp->minHeapIndex]->minHeapIndex = temp->minHeapIndex;
    minHeapify(minHeap, temp->minHeapIndex);

    // remove the node from List
    removeLNode(myDS->list, &temp);
}

// Function to delete minimum value stored in the main data structure
void deleteMin(struct MyDS* myDS)

```

```

{

    MinHeap *minHeap = myDS->minHeap;
    MaxHeap *maxHeap = myDS->maxHeap;

    if (isMinHeapEmpty(minHeap))
        return;
    struct LNode* temp = minHeap->array[0];

    // delete the minimum item from minHeap
    minHeap->array[0] = minHeap->array[minHeap->size - 1];
    --minHeap->size;
    minHeap->array[0]->minHeapIndex = 0;
    minHeapify(minHeap, 0);

    // remove the item from maxHeap
    maxHeap->array[temp->maxHeapIndex] = maxHeap->array[maxHeap->size - 1];
    --maxHeap->size;
    maxHeap->array[temp->maxHeapIndex]->maxHeapIndex = temp->maxHeapIndex;
    maxHeapify(maxHeap, temp->maxHeapIndex);

    // remove the node from List
    removeLNode(myDS->list, &temp);
}

// Function to enList an item to List
void insertAtHead(struct List* list, struct LNode* temp)
{
    if (isEmpty(list))
        list->head = temp;

    else
    {
        temp->next = list->head;
        list->head->prev = temp;
        list->head = temp;
    }
}

// Function to delete an item from List. The function also
// removes item from min and max heaps
void Delete(struct MyDS* myDS, int item)
{
    MinHeap *minHeap = myDS->minHeap;
    MaxHeap *maxHeap = myDS->maxHeap;

    if (isEmpty(myDS->list))
        return;
}

```

```
// search the node in List
struct LNode* temp = myDS->list->head;
while (temp && temp->data != item)
    temp = temp->next;

// if item not found
if (!temp || temp && temp->data != item)
    return;

// remove item from min heap
minHeap->array[temp->minHeapIndex] = minHeap->array[minHeap->size - 1];
--minHeap->size;
minHeap->array[temp->minHeapIndex]->minHeapIndex = temp->minHeapIndex;
minHeapify(minHeap, temp->minHeapIndex);

// remove item from max heap
maxHeap->array[temp->maxHeapIndex] = maxHeap->array[maxHeap->size - 1];
--maxHeap->size;
maxHeap->array[temp->maxHeapIndex]->maxHeapIndex = temp->maxHeapIndex;
maxHeapify(maxHeap, temp->maxHeapIndex);

// remove node from List
removeLNode(myDS->list, &temp);
}

// insert operation for main data structure
void Insert(struct MyDS* myDS, int data)
{
    struct LNode* temp = newLNode(data);

    // insert the item in List
    insertAtHead(myDS->list, temp);

    // insert the item in min heap
    insertMinHeap(myDS->minHeap, temp);

    // insert the item in max heap
    insertMaxHeap(myDS->maxHeap, temp);
}

// Driver program to test above functions
int main()
{
    struct MyDS *myDS = createMyDS(10);
    // Test Case #1
    /*Insert(myDS, 10);
    Insert(myDS, 2);
    Insert(myDS, 32);
```

```
Insert(myDS, 40);
Insert(myDS, 5); */

// Test Case #2
Insert(myDS, 10);
Insert(myDS, 20);
Insert(myDS, 30);
Insert(myDS, 40);
Insert(myDS, 50);

printf("Maximum = %d \n", findMax(myDS));
printf("Minimum = %d \n\n", findMin(myDS));

deleteMax(myDS); // 50 is deleted
printf("After deleteMax()\n");
printf("Maximum = %d \n", findMax(myDS));
printf("Minimum = %d \n\n", findMin(myDS));

deleteMin(myDS); // 10 is deleted
printf("After deleteMin()\n");
printf("Maximum = %d \n", findMax(myDS));
printf("Minimum = %d \n\n", findMin(myDS));

Delete(myDS, 40); // 40 is deleted
printf("After Delete()\n");
printf("Maximum = %d \n", findMax(myDS));
printf("Minimum = %d \n", findMin(myDS));

return 0;
}
```

Output:

```
Maximum = 50
Minimum = 10

After deleteMax()
Maximum = 40
Minimum = 10

After deleteMin()
Maximum = 40
Minimum = 20

After Delete()
Maximum = 30
Minimum = 20
```

This article is compiled by [Aashish Barnwal](#) and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

<https://www.geeksforgeeks.org/a-data-structure-question/>

Chapter 45

Difference Array Range update query in O(1)

Difference Array Range update query in O(1) - GeeksforGeeks

Consider an array A[] of integers and following two types of queries.

1. update(l, r, x) : Adds x to all values from A[l] to A[r] (both inclusive).
2. printArray() : Prints the current modified array.

Examples :

Input : A [] { 10, 5, 20, 40 }

 update(0, 1, 10)

 printArray()

 update(1, 3, 20)

 update(2, 2, 30)

 printArray()

Output : 20 15 20 40

 20 35 70 60

Explanation : The query update(0, 1, 10) adds 10 to A[0] and A[1]. After update, A[] becomes {20, 15, 20, 40}

Query update(1, 3, 20) adds 20 to A[1], A[2] and A[3]. After update, A[] becomes {20, 35, 40, 60}.

Query update(2, 2, 30) adds 30 to A[2]. After update, A[] becomes {20, 35, 70, 60}.

A **simple solution** is to do following :

1. update(l, r, x) : Run a loop from l to r and add x to all elements from A[l] to A[r]
2. printArray() : Simply print A[].

Time complexities of both of the above operations is O(n)

An **efficient solution** is to use difference array.

Difference array D[i] of a given array A[i] is defined as $D[i] = A[i] - A[i-1]$ (for $0 < i < N$) and $D[0] = A[0]$ considering 0 based indexing. Difference array can be used to perform range update queries "l r x" where l is left index, r is right index and x is value to be added and after all queries you can return original array from it. Where update range operations can be performed in O(1) complexity.

1. update(l, r, x) : Add x to D[l] and subtract it from D[r+1], i.e., we do $D[l] += x$, $D[r+1] -= x$
2. printArray() : Do $A[0] = D[0]$ and print it. For rest of the elements, do $A[i] = A[i-1] + D[i]$ and print them.

Time complexity of update here is improved to **O(1)**. Note that printArray() still takes O(n) time.

C++

```
// C++ code to demonstrate Difference Array
#include <bits/stdc++.h>
using namespace std;

// Creates a diff array D[] for A[] and returns
// it after filling initial values.
vector<int> initializeDiffArray(vector<int>& A)
{
    int n = A.size();

    // We use one extra space because
    // update(l, r, x) updates D[r+1]
    vector<int> D(n + 1);

    D[0] = A[0], D[n] = 0;
    for (int i = 1; i < n; i++)
        D[i] = A[i] - A[i - 1];
    return D;
}

// Does range update
void update(vector<int>& D, int l, int r, int x)
{
    D[l] += x;
    D[r + 1] -= x;
}
```

```
}

// Prints updated Array
int printArray(vector<int>& A, vector<int>& D)
{
    for (int i = 0; i < A.size(); i++) {
        if (i == 0)
            A[i] = D[i];

        // Note that A[0] or D[0] decides
        // values of rest of the elements.
        else
            A[i] = D[i] + A[i - 1];

        cout << A[i] << " ";
    }
    cout << endl;
}

// Driver Code
int main()
{
    // Array to be updated
    vector<int> A{ 10, 5, 20, 40 };

    // Create and fill difference Array
    vector<int> D = initializeDiffArray(A);

    // After below update(l, r, x), the
    // elements should become 20, 15, 20, 40
    update(D, 0, 1, 10);
    printArray(A, D);

    // After below updates, the
    // array should become 30, 35, 70, 60
    update(D, 1, 3, 20);
    update(D, 2, 2, 30);
    printArray(A, D);

    return 0;
}
```

Java

```
// Java code to demonstrate Difference Array
class GFG {

    // Creates a diff array D[] for A[] and returns
```

```
// it after filling initial values.
static void initializeDiffArray(int A[], int D[])
{
    int n = A.length;

    D[0] = A[0];
    D[n] = 0;
    for (int i = 1; i < n; i++)
        D[i] = A[i] - A[i - 1];
}

// Does range update
static void update(int D[], int l, int r, int x)
{
    D[l] += x;
    D[r + 1] -= x;
}

// Prints updated Array
static int printArray(int A[], int D[])
{
    for (int i = 0; i < A.length; i++) {

        if (i == 0)
            A[i] = D[i];

        // Note that A[0] or D[0] decides
        // values of rest of the elements.
        else
            A[i] = D[i] + A[i - 1];

        System.out.print(A[i] + " ");
    }

    System.out.println();
    return 0;
}

// Driver Code
public static void main(String[] args)
{
    // Array to be updated
    int A[] = { 10, 5, 20, 40 };
    int n = A.length;
    // Create and fill difference Array
    // We use one extra space because
```

```
// update(l, r, x) updates D[r+1]
int D[] = new int[n + 1];
initializeDiffArray(A, D);

// After below update(l, r, x), the
// elements should become 20, 15, 20, 40
update(D, 0, 1, 10);
printArray(A, D);

// After below updates, the
// array should become 30, 35, 70, 60
update(D, 1, 3, 20);
update(D, 2, 2, 30);

printArray(A, D);
}

}

// This code is contributed by Anant Agarwal.
```

Python3

```
# Python3 code to demonstrate Difference Array

# Creates a diff array D[] for A[] and returns
# it after filling initial values.
def initializeDiffArray( A):
    n = len(A)

    # We use one extra space because
    # update(l, r, x) updates D[r+1]
    D = [0 for i in range(0 , n + 1)]

    D[0] = A[0]; D[n] = 0

    for i in range(1, n ):
        D[i] = A[i] - A[i - 1]
    return D

# Does range update
def update(D, l, r, x):

    D[l] += x
    D[r + 1] -= x

# Prints updated Array
```

```
def printArray(A, D):  
  
    for i in range(0 , len(A)):  
        if (i == 0):  
            A[i] = D[i]  
  
        # Note that A[0] or D[0] decides  
        # values of rest of the elements.  
        else:  
            A[i] = D[i] + A[i - 1]  
  
    print(A[i], end = " ")  
  
    print ("")  
  
# Driver Code  
A = [ 10, 5, 20, 40 ]  
  
# Create and fill difference Array  
D = initializeDiffArray(A)  
  
# After below update(l, r, x), the  
# elements should become 20, 15, 20, 40  
update(D, 0, 1, 10)  
printArray(A, D)  
  
# After below updates, the  
# array should become 30, 35, 70, 60  
update(D, 1, 3, 20)  
update(D, 2, 2, 30)  
printArray(A, D)  
  
# This code is contributed by Gitanjali.
```

C#

```
// C# code to demonstrate Difference Array  
using System;  
  
class GFG {  
  
    // Creates a diff array D[] for A[] and returns  
    // it after filling initial values.  
    static void initializeDiffArray(int []A, int []D)  
    {  
  
        int n = A.Length;
```

```
D[0] = A[0];
D[n] = 0;
for (int i = 1; i < n; i++)
    D[i] = A[i] - A[i - 1];
}

// Does range update
static void update(int []D, int l, int r, int x)
{
    D[l] += x;
    D[r + 1] -= x;
}

// Prints updated Array
static int printArray(int []A, int []D)
{
    for (int i = 0; i < A.Length; i++) {

        if (i == 0)
            A[i] = D[i];

        // Note that A[0] or D[0] decides
        // values of rest of the elements.
        else
            A[i] = D[i] + A[i - 1];

        Console.Write(A[i] + " ");
    }

    Console.WriteLine();
}

return 0;
}

// Driver Code
public static void Main()
{
    // Array to be updated
    int []A = { 10, 5, 20, 40 };
    int n = A.Length;
    // Create and fill difference Array
    // We use one extra space because
    // update(l, r, x) updates D[r+1]
    int []D = new int[n + 1];
    initializeDiffArray(A, D);

    // After below update(l, r, x), the
```

```
// elements should become 20, 15, 20, 40
update(D, 0, 1, 10);
printArray(A, D);

// After below updates, the
// array should become 30, 35, 70, 60
update(D, 1, 3, 20);
update(D, 2, 2, 30);

printArray(A, D);
}

}

// This code is contributed by vt_m.
```

Output:

```
20 15 20 40
20 35 70 60
```

Source

<https://www.geeksforgeeks.org/difference-array-range-update-query-o1/>

Chapter 46

Difference between Inverted Index and Forward Index

Difference between Inverted Index and Forward Index - GeeksforGeeks

Inverted Index

1. It is a data structure that stores mapping from words to documents or set of documents i.e. directs you from word to document.
2. Steps to build Inverted index are:
 - Fetch the document and gather all the words.
 - Check for each word, if it is present then add reference of document to index else create new entry in index for that word.
 - Repeat above steps for all documents and sort the words.
3. Indexing is slow as it first checks that word is present or not.
4. Searching is very fast.
5. Example of Inverted index:

Word	Documents
hello	doc1
sky	doc1, doc3
coffee	doc2
hi	doc2
greetings	doc3

It does not store duplicate keywords in index.

6. Real life examples of Inverted index:

- Index at the back of the book.
- Reverse lookup

Forward Index:

1. It is a data structure that stores mapping from documents to words i.e. directs you from document to word.
2. Steps to build Forward index are:
 - Fetch the document and gather all the keywords.
 - Append all the keywords in the index entry for this document.
 - Repeat above steps for all documents
3. Indexing is quite fast as it only append keywords as it move forwards.
4. Searching is quite difficult as it has to look at every contents of index just to retrieve all pages related to word.
5. Example of forward index:

Document	Keywords
doc1	hello, sky, morning
doc2	tea, coffee, hi
doc3	greetings, sky

It stores duplicate keywords in index. Eg: word “sky” is stored multiple times.

6. Real life examples of Forward index:
 - Table of contents in book.
 - DNS lookup

Similarity between Forward index and Inverted Index:

- Both are used to search text in document or set of documents.

Source

<https://www.geeksforgeeks.org/difference-inverted-index-forward-index/>

Chapter 47

Disjoint Set Data Structures (Java Implementation)

Disjoint Set Data Structures (Java Implementation) - GeeksforGeeks

Consider a situation with a number of persons and following tasks to be performed on them.

1. Add a new friendship relation, i.e., a person x becomes friend of another person y.
2. Find whether individual x is a friend of individual y (direct or indirect friend)

Example:

We are given 10 individuals say,
a, b, c, d, e, f, g, h, i, j

Following are relationships to be added.
a <-> b
b <-> d
c <-> f
c <-> i
j <-> e
g <-> j

And given queries like whether a is a friend of d or not.

We basically need to create following 4 groups and maintain a quickly accessible connection among group items:
G1 = {a, b, d}

```
G2 = {c, f, i}
G3 = {e, g, j}
G4 = {h}
```

Problem : To find whether x and y belong to same group or not, i.e., to find if x and y are direct/indirect friends.

Solution : Partitioning the individuals into different sets according to the groups in which they fall. This method is known as disjoint set data structure which maintains collection of disjoint sets and each set is represented by its representative which is one of its members.

Approach:

- **How to Resolve sets ?** Initially all elements belong to different sets. After working on the given relations, we select a member as representative. There can be many ways to select a representative, a simple one is to select with the biggest index.
- **Check if 2 persons are in the same group ?** If representatives of two individuals are same, then they'll become friends.

Data Structures used:

Array : An array of integers, called parent[]. If we are dealing with n items, i'th element of the array represents the i'th item. More precisely, the i'th element of the array is the parent of the i'th item. These relationships create one, or more, virtual trees.

Tree : It is a disjoint set. If two elements are in the same tree, then they are in the same disjoint set. The root node (or the topmost node) of each tree is called the representative of the set. There is always a single unique representative of each set. A simple rule to identify representative is, if i is the representative of a set, then parent[i] = i. If i is not the representative of his set, then it can be found by traveling up the tree until we find the representative.

Operations :

Find : Can be implemented by recursively traversing the parent array until we hit a node who is parent of itself.

```
// Finds the representative of the set that
// i is an element of
public int find(int i)
{
    // If i is the parent of itself
    if (parent[i] == i)
    {
        // Then i is the representative of
        // this set
```

```

        return i;
    }
else
{
    // Else if i is not the parent of
    // itself, then i is not the
    // representative of his set. So we
    // recursively call Find on its parent
    return find(parent[i]);
}
}

```

Union: It takes, as input, two elements. And finds the representatives of their sets using the find operation, and finally puts either one of the trees (representing the set) under the root node of the other tree, effectively merging the trees and the sets.

```

// Unites the set that includes i and
// the set that includes j
public void union(int i, int j)
{
    // Find the representatives
    // (or the root nodes) for the set
    // that includes i

    int irep = this.Find(i),

    // And do the same for the set
    // that includes j
    int jrep = this.Find(j);

    // Make the parent of i's representative
    // be j's representative effectively
    // (moving all of i's set into j's set)
    this.Parent[irep] = jrep;
}

```

Improvements (Union by Rank and Path Compression)

The efficiency depends heavily on the height of the tree. We need to minimize the height of tree in order improve the efficiency. We can use Path Compression and Union by rank methods to do so.

Path Compression (Modifications to find()): It speeds up the data structure by compressing the height of the trees. It can be achieved by inserting a small caching mechanism into the Find operation. Take a look at the code for more details:

```

// Finds the representative of the set that i
// is an element of.
int find(int i)
{
    // If i is the parent of itself
    if (Parent[i] == i)
    {
        // Then i is the representative
        return i;
    }
    else
    {
        // Recursively find the representative.
        int result = find(Parent[i]);

        // We cache the result by moving i's node
        // directly under the representative of this
        // set
        Parent[i] = result;

        // And then we return the result
        return result;
    }
}

```

Union by Rank: First of all, we need a new array of integers called rank[]. Size of this array is same as the parent array. If i is a representative of a set, rank[i] is the height of the tree representing the set.

Now recall that, in the Union operation, it doesn't matter which of the two trees is moved under the other (see last two image examples above). Now what we want to do is minimize the height of the resulting tree. If we are uniting two trees (or sets), let's call them left and right, then it all depends on the rank of left and the rank of right.

- If the rank of left is less than the rank of right, then it's best to move left under right, because that won't change the rank of right (while moving right under left would increase the height). In the same way, if the rank of right is less than the rank of left, then we should move right under left.
- If the ranks are equal, it doesn't matter which tree goes under the other, but the rank of the result will always be one greater than the rank of the trees.

```

// Unites the set that includes i and the set
// that includes j
void union(int i, int j)
{
    // Find the representatives (or the root nodes)
    // for the set that includes i

```

```
int irep = this.find(i);

// And do the same for the set that includes j
int jrep = this.Find(j);

// Elements are in same set, no need to
// unite anything.
if (irep == jrep)
    return;

// Get the rank of i's tree
irank = Rank[irep], 

// Get the rank of j's tree
jrank = Rank[jrep];

// If i's rank is less than j's rank
if (irank < jrank)
{
    // Then move i under j
    this.parent[irep] = jrep;
}

// Else if j's rank is less than i's rank
else if (jrank < irank)
{
    // Then move j under i
    this.Parent[jrep] = irep;
}

// Else if their ranks are the same
else
{
    // Then move i under j (doesn't matter
    // which one goes where)
    this.Parent[irep] = jrep;

    // And increment the the result tree's
    // rank by 1
    Rank[jrep]++;
}
```

Java Implementation

```
// A Java program to implement Disjoint Set Data
// Structure.
```

```
import java.io.*;
import java.util.*;

class DisjointUnionSets
{
    int[] rank, parent;
    int n;

    // Constructor
    public DisjointUnionSets(int n)
    {
        rank = new int[n];
        parent = new int[n];
        this.n = n;
        makeSet();
    }

    // Creates n sets with single item in each
    void makeSet()
    {
        for (int i=0; i<n; i++)
        {
            // Initially, all elements are in
            // their own set.
            parent[i] = i;
        }
    }

    // Returns representative of x's set
    int find(int x)
    {
        // Finds the representative of the set
        // that x is an element of
        if (parent[x] !=x)
        {
            // if x is not the parent of itself
            // Then x is not the representative of
            // his set,
            parent[x] = find(parent[x]);

            // so we recursively call Find on its parent
            // and move i's node directly under the
            // representative of this set
        }
        return parent[x];
    }
}
```

```
// Unites the set that includes x and the set
// that includes x
void union(int x, int y)
{
    // Find representatives of two sets
    int xRoot = find(x), yRoot = find(y);

    // Elements are in the same set, no need
    // to unite anything.
    if (xRoot == yRoot)
        return;

    // If x's rank is less than y's rank
    if (rank[xRoot] < rank[yRoot])

        // Then move x under y so that depth
        // of tree remains less
        parent[xRoot] = yRoot;

    // Else if y's rank is less than x's rank
    else if (rank[yRoot] < rank[xRoot])

        // Then move y under x so that depth of
        // tree remains less
        parent[yRoot] = xRoot;

    else // if ranks are the same
    {
        // Then move y under x (doesn't matter
        // which one goes where)
        parent[yRoot] = xRoot;

        // And increment the the result tree's
        // rank by 1
        rank[xRoot] = rank[xRoot] + 1;
    }
}

// Driver code
public class Main
{
    public static void main(String[] args)
    {
        // Let there be 5 persons with ids as
        // 0, 1, 2, 3 and 4
        int n = 5;
        DisjointUnionSets dus =
```

```
new DisjointUnionSets(n);

// 0 is a friend of 2
dus.union(0, 2);

// 4 is a friend of 2
dus.union(4, 2);

// 3 is a friend of 1
dus.union(3, 1);

// Check if 4 is a friend of 0
if (dus.find(4) == dus.find(0))
    System.out.println("Yes");
else
    System.out.println("No");

// Check if 1 is a friend of 0
if (dus.find(1) == dus.find(0))
    System.out.println("Yes");
else
    System.out.println("No");
}
}
```

Output:

Yes
No

Related Articles:

[Union-Find Algorithm Set 1 \(Detect Cycle in a an Undirected Graph\)](#)
[Union-Find Algorithm Set 2 \(Union By Rank and Path Compression\)](#)

Try to solve [this](#) problem and check how much you learnt and do comment on the complexity of the given question.

Source

<https://www.geeksforgeeks.org/disjoint-set-data-structures-java-implementation/>

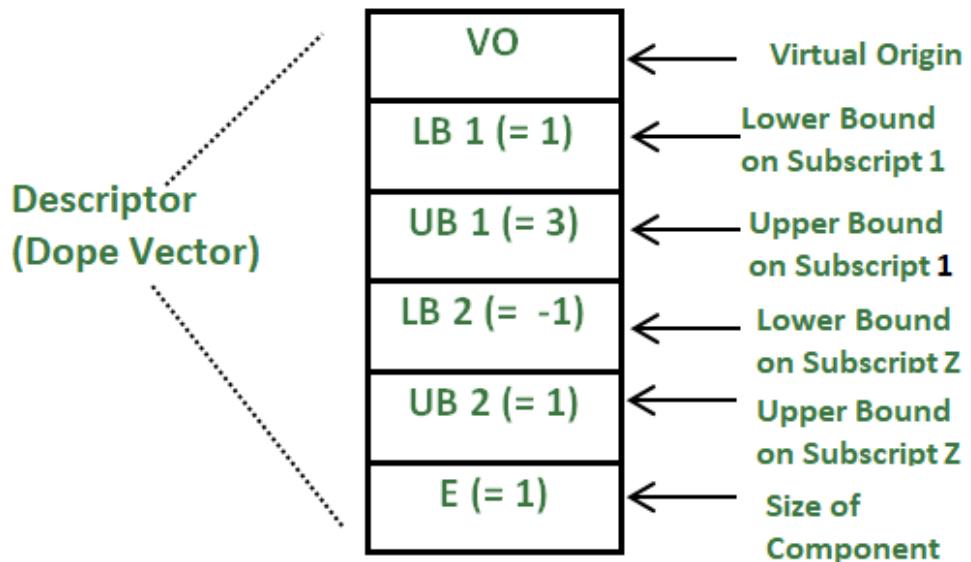
Chapter 48

Dope Vector

Dope Vector - GeeksforGeeks

Dope Vectors is a data structure that is used by compilers to store some metadata about the array like its total size, the size of one unit also called stride of the array, etc. These are used to describe arrays and other similar structures that store multiple values of one datatype as a complete block of memory. It can also describe structures that contain arrays and similar structures as its component. Dope vectors help compilers to access the arrays with ease. Different checks that are implemented by compiler like **Out of Bound check,datatype check,etc.** are all possible because of dope vector associated with the array.

The following details are stored in the dope vector for a particular array:



- **E** denotes the size of one single array element like $E = 1$ for character, $E = 4$ for integers, etc.
- **VO** denotes the virtual starting memory of the array inside the RAM.
- **LU1** denotes the starting lower bound of the array.
- **UB1** denotes the upper limit up to which the array is currently occupied.
- **UB2** denotes the upper limit up to which the array can be filled.

The details that a dope vector stores vary from one operating system to another, but mostly it contains the following information regarding an array:

- Rank or number of dimensions of array or
- The base address of the array.
- The type of elements stored in the array.
- The stride of the array.
- The extent of the array.

and many other details.

There are many problems that are solved by using the concept of dope vectors with the arrays at the cost of a small computation overhead (fetching data from dope vector) like:

- It is difficult to release the extra memory associated with the array without the use of dope vector. Suppose that we initially allocated 200KB of memory for the array. But when it was used it required only 150 Kb of memory. Releasing the extra memory becomes quite easy with the use of dope vector as it stores the extent of the memory currently occupied by the array.
- Without the dope vectors it is very difficult to determine the number of elements in the array, since it stores the information of the total size of the array and the length of the stride. It can be used to calculate the total number of elements in the array.

Source

<https://www.geeksforgeeks.org/dope-vector-gap-buffer-arrays/>

Chapter 49

Dynamic Connectivity Set 1 (Incremental)

Dynamic Connectivity Set 1 (Incremental) - GeeksforGeeks

Dynamic connectivity is a data structure that dynamically maintains the information about three connected components of graph. In simple words suppose there is a graph $G(V, E)$ in which no. of vertices V is constant but no. of edges E is variable. There are three ways in which we can change the number of edges

1. Incremental Connectivity : Edges are only added to the graph.
2. Decremental Connectivity : Edges are only deleted from the graph.
3. Fully Dynamic Connectivity : Edges can both be deleted and added to the graph.

In this article only **Incremental connectivity** is discussed. There are mainly two operations that need to be handled.

1. An edge is added to the graph.
2. Information about two nodes x and y whether they are in the same connected components or not.

Example:

```
Input : V = 7
        Number of operations = 11
        1 0 1
        2 0 1
        2 1 2
```

```

1 0 2
2 0 2
2 2 3
2 3 4
1 0 5
2 4 5
2 5 6
1 2 6

Note: 7 represents number of nodes,
      11 represents number of queries.
      There are two types of queries
      Type 1 : 1 x y in this if the node
                 x and y are connected print
                 Yes else No
      Type 2 : 2 x y in this add an edge
                 between node x and y

```

```

Output : No
         Yes
         No
         Yes

```

Explanation :

Initially there are no edges so node 0 and 1 will be disconnected so answer will be No
 Node 0 and 2 will be connected through node 1 so answer will be Yes similarly for other queries we can find whether two nodes are connected or not

To solve the problems of incremental connectivity [disjoint data structure](#) is used. Here each connected component represents a set and if the two nodes belong to the same set it means that they are connected.

C++ implementation is given below here we are using [union by rank and path compression](#)

```

// C++ implementation of incremental connectivity
#include<bits/stdc++.h>
using namespace std;

// Finding the root of node i
int root(int arr[], int i)
{
    while (arr[i] != i)
    {
        arr[i] = arr[arr[i]];
        i = arr[i];
    }
    return i;
}

```

```

// union of two nodes a and b
void weighted_union(int arr[], int rank[],
                     int a, int b)
{
    int root_a = root(arr, a);
    int root_b = root(arr, b);

    // union based on rank
    if (rank[root_a] < rank[root_b])
    {
        arr[root_a] = arr[root_b];
        rank[root_b] += rank[root_a];
    }
    else
    {
        arr[root_b] = arr[root_a];
        rank[root_a] += rank[root_b];
    }
}

// Returns true if two nodes have same root
bool areSame(int arr[], int a, int b)
{
    return (root(arr, a) == root(arr, b));
}

// Performing an operation according to query type
void query(int type, int x, int y, int arr[], int rank[])
{
    // type 1 query means checking if node x and y
    // are connected or not
    if (type == 1)
    {
        // If roots of x and y is same then yes
        // is the answer
        if (areSame(arr, x, y) == true)
            cout << "Yes" << endl;
        else
            cout << "No" << endl;
    }

    // type 2 query refers union of x and y
    else if (type == 2)
    {
        // If x and y have different roots then
        // union them
        if (areSame(arr, x, y) == false)

```

```
        weighted_union(arr, rank, x, y);
    }
}

// Driver function
int main()
{
    // No.of nodes
    int n = 7;

    // The following two arrays are used to
    // implement disjoint set data structure.
    // arr[] holds the parent nodes while rank
    // array holds the rank of subset
    int arr[n], rank[n];

    // initializing both array and rank
    for (int i=0; i<n; i++)
    {
        arr[i] = i;
        rank[i] = 1;
    }

    // number of queries
    int q = 11;
    query(1, 0, 1, arr, rank);
    query(2, 0, 1, arr, rank);
    query(2, 1, 2, arr, rank);
    query(1, 0, 2, arr, rank);
    query(2, 0, 2, arr, rank);
    query(2, 2, 3, arr, rank);
    query(2, 3, 4, arr, rank);
    query(1, 0, 5, arr, rank);
    query(2, 4, 5, arr, rank);
    query(2, 5, 6, arr, rank);
    query(1, 2, 6, arr, rank);
    return 0;
}
```

Output:

```
No
Yes
No
Yes
```

Time Complexity:

The amortized time complexity is $O(\alpha(n))$ per operation where α is [inverse acker-mann function](#) which is nearly constant.

Reference:

https://en.wikipedia.org/wiki/Dynamic_connectivity

Source

<https://www.geeksforgeeks.org/dynamic-connectivity-set-1-incremental/>

Chapter 50

Dynamic Disjoint Set Data Structure for large range values

Dynamic Disjoint Set Data Structure for large range values - GeeksforGeeks

Prerequisites:

- [Disjoint Set Data Structure](#)
- [Set](#)
- [Unordered_Map](#)

[Disjoint Set data structure](#) is used to keeps track of a set of elements partitioned into a number of disjoint (non-overlapping) subsets.

In this article, we will learn about constructing the same Data Structure dynamically. This data structure basically helps in situation where we cannot simply use arrays for creating disjoint sets because of large inputs of order 10^9 .

To illustrate this, consider the following problem. We need to find the total number of [connected components in the Graph](#) when the total Number of Vertices can be up to 10^9 .

Examples:

```
Input : Edges : { { 1, 2 },
                  { 2, 3 },
                  { 4, 5 } }

Output : 2
Explanation: {1, 2, 3} forms a component and
{4, 5} forms another component.
```

The idea to solve this problem is, we will maintain two hash tables (implemented using [unordered_maps](#) in C++). One for *parent* and other for *degree*. Parent[V] will give the parent of the component which the Vertex V is part of and Degree will give the number of vertices in that component.

Initially, both Parent and Degree will be empty. We will keep inserting vertices to the maps as sequentially.

See the code and the explanation simultaneously for a better understanding. Below are the methods used in the code to solve the above problem:

1. **getParent(V):** This method will give the parent of the vertex V. Here we recursively find the parent of the vertex V(see code), meanwhile we assign all the vertex in that component to have the same parent.(In disjoint set data structure all the vertex in the same component have the same parent.)
2. **Union():** When we add a edge and the two vertexes are of different components we call the Union() method to join both the component. Here the parent of the component formed after joining both the components will be the parent of the component among the two which had more number of vertexes before the union. The degree of the new component is updated accordingly.
3. **getTotalComponent():** Vertex in the same component will have the same parent. We use [unordered_set \(STL\)](#) to count the total number of components. As we have maintained the Data Structure as Dynamic so, there can be any vertex which has not been added to any of the components hence they are different component alone. So the total number of components will be given by,

Total no of Component = Total Vertices - Number of Vertices
in parent (Map) + Number of Component
formed from the Vertexes inserted
in the graph.

Below is the implementation of above idea:

```
// Dynamic Disjoint Set Data Structure
// Union-Find

#include <bits/stdc++.h>
using namespace std;

int N;
int Edges[3][2];

// Dynamic Disjoint Set Data Structure
struct DynamicDisjointSetDS {

    // We will add the vertex to the edge
```

```
// only when it is asked to i.e. maintain
// a dynamic DS.
unordered_map<int, int> parent, degree;

// Total number of Vertex in the Graph
int N;

// Constructor
DynamicDisjointSetDS(int n)
{
    N = n;
}

// Get Parent of vertex V
int getParent(int vertex)
{
    // If the vertex is already inserted
    // in the graph
    if (parent.find(vertex) != parent.end()) {

        if (parent[vertex] != vertex) {
            parent[vertex] =
                getParent(parent[vertex]);
            return parent[vertex];
        }
    }
}

// if the vertex is operated for the first
// time
else {

    // insert the vertex and assign its
    // parent to itself
    parent.insert(make_pair(vertex, vertex));

    // Degree of the vertex
    degree.insert(make_pair(vertex, 1));
}

return vertex;
}

// Union by Rank
void Union(int vertexA, int vertexB)
{
    // Parent of Vertex A
    int x = getParent(vertexA);
```

```

// Parent of Vertex B
int y = getParent(vertexB);

// if both have same parent
// Do Nothing
if (x == y)
    return;

// Merging the component
// Assigning the parent of smaller Component
// as the parent of the bigger Component.
if (degree[x] > degree[y]) {
    parent[y] = x;
    degree[x] = degree[x] + degree[y];
}
else {
    parent[x] = y;
    degree[y] = degree[y] + degree[x];
}
}

// Count total Component in the Graph
int GetTotalComponent()
{
    // To count the total Component formed
    // from the inserted vertex in the Graph
    unordered_set<int> total;

    // Iterate through the parent
    for (auto itr = parent.begin();
         itr != parent.end(); itr++) {

        // Add the parent of each Vertex
        // to the set
        total.insert(getParent(itr->first));
    }

    // Total Component = Total Vertices -
    // Number of Vertices in the parent +
    // Number of Components formed from
    // the Vertices inserted in the Graph
    return N - parent.size() + total.size();
}

// Solve
void Solve()

```

```
{  
  
    // Declaring the Dynamic Disjoint Set DS  
    DynamicDisjointSetDS dsu(N);  
  
    // Traversing through the Edges  
    for (int i = 0; i < 3; i++) {  
  
        // If the Vertexes in the Edges  
        // have same parent do nothing  
        if (dsu.getParent(Edges[i][0]) ==  
            dsu.getParent(Edges[i][1])) {  
            continue;  
        }  
  
        // else Do Union of both the Components.  
        else {  
            dsu.Union(Edges[i][0], Edges[i][1]);  
        }  
    }  
  
    // Get total Components  
    cout << dsu.GetTotalComponent();  
}  
  
// Driver Code  
int main()  
{  
    // Total Number of Vertexes  
    N = 5;  
  
    /* Edges  
     * 1 <--> 2  
     * 2 <--> 3  
     * 4 <--> 5      */  
  
    Edges[0][0] = 1;  
    Edges[0][1] = 2;  
    Edges[1][0] = 2;  
    Edges[1][1] = 3;  
    Edges[2][0] = 4;  
    Edges[2][1] = 3;  
  
    // Solve  
    Solve();  
  
    return 0;  
}
```

Output:

2

Note : If the number of vertexes are even larger we can implement the same code just by changing the data type from int to long long.

Source

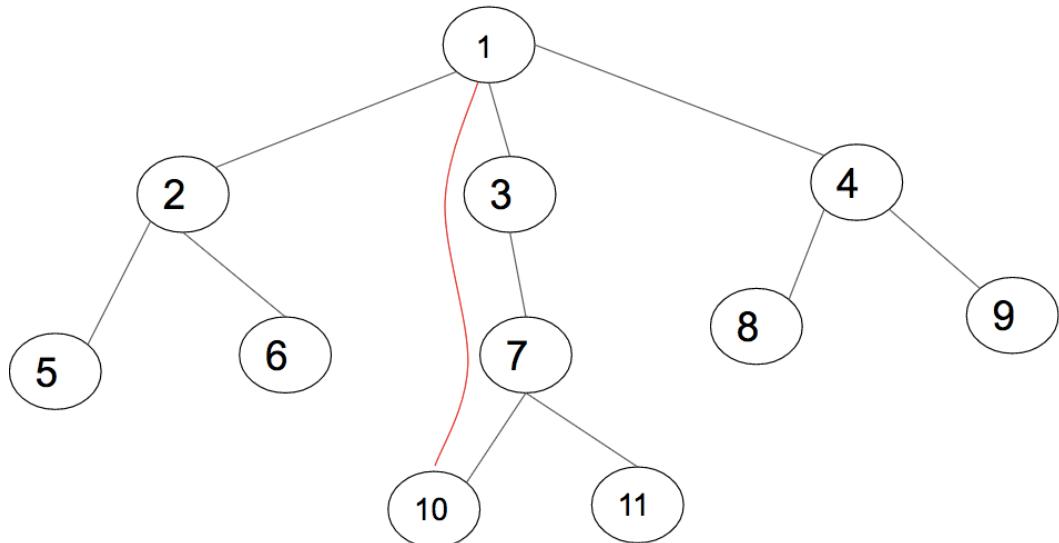
<https://www.geeksforgeeks.org/dynamic-disjoint-set-data-structure-for-large-range-values/>

Chapter 51

Dynamic Programming on Trees Set 2

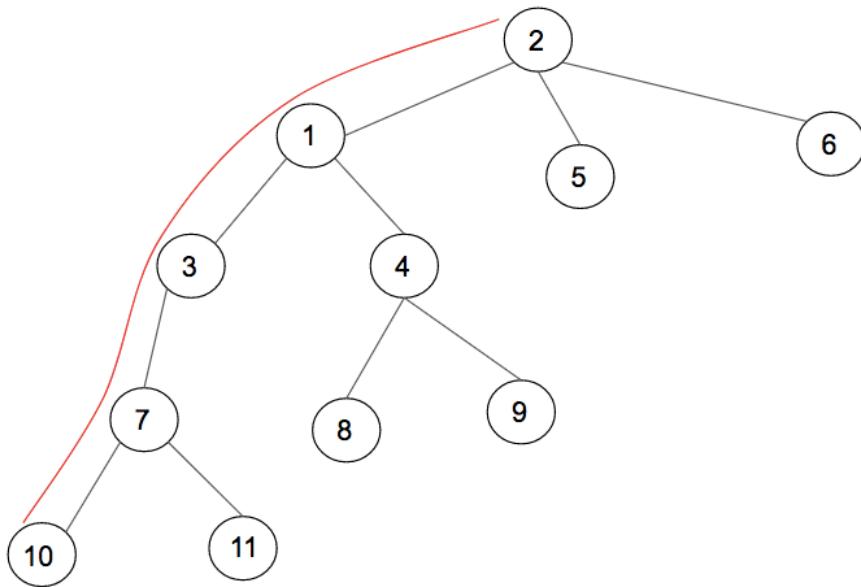
Dynamic Programming on Trees Set 2 - GeeksforGeeks

Given a tree with N nodes and N-1 edges, find out the **maximum height of tree** when any node in the tree is considered as the root of the tree.



Red line denotes the maximum height of tree when node-1 is the root

The above diagram represents a tree with **11 nodes** and **10 edges**, and the path which gives us the maximum height when node 1 is considered as root. The maximum height is 3.



Red line denotes the maximum height of the tree when node-2 is taken as root.

In the above diagram, when 2 is considered as root, then the longest path found is in RED color. A **naive approach** will be to traverse the tree using [DFS traversal](#) for every node and calculate the maximum height when the node is treated as the root of the tree. The time complexity for DFS traversal of a tree is $O(N)$. **The overall time complexity of DFS for all N nodes will be $O(N)*N$ i.e., $O(N^2)$.**

The above problem can be solved by using **Dynamic Programming on Trees**. To solve this problem, pre-calculate two things for every node. One will be the maximum height while traveling downwards via its branches to the leaves. While the other will be the maximum height when traveling upwards via its parent to any of the leaves.

Optimal Substructure :

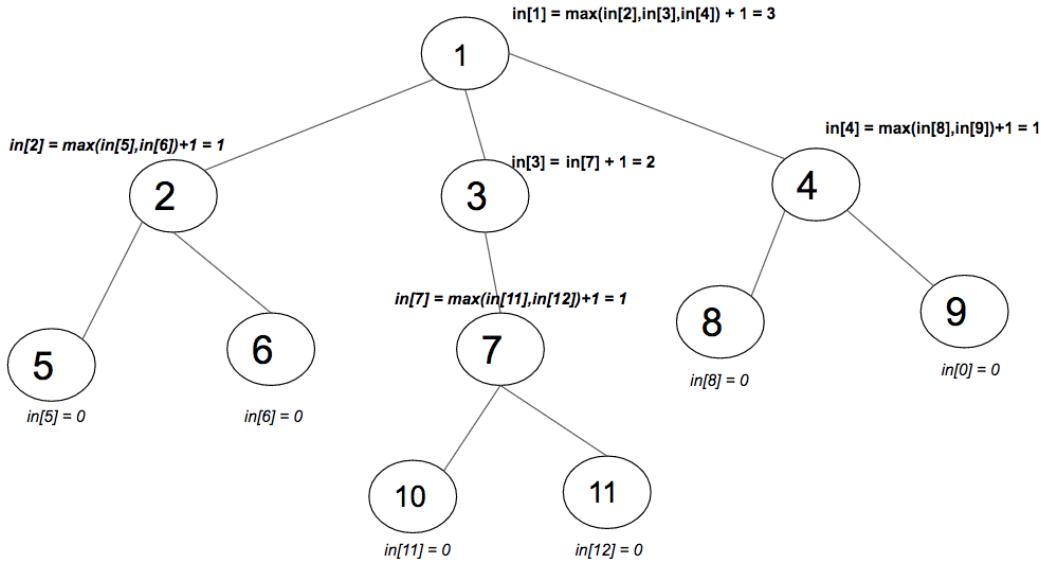
When node i is considered as root,

in[i] be the maximum height of tree when we travel downwards via its sub-trees and leaves.

Also, **out[i]** be the maximum height of the tree while traveling upwards via its parent.

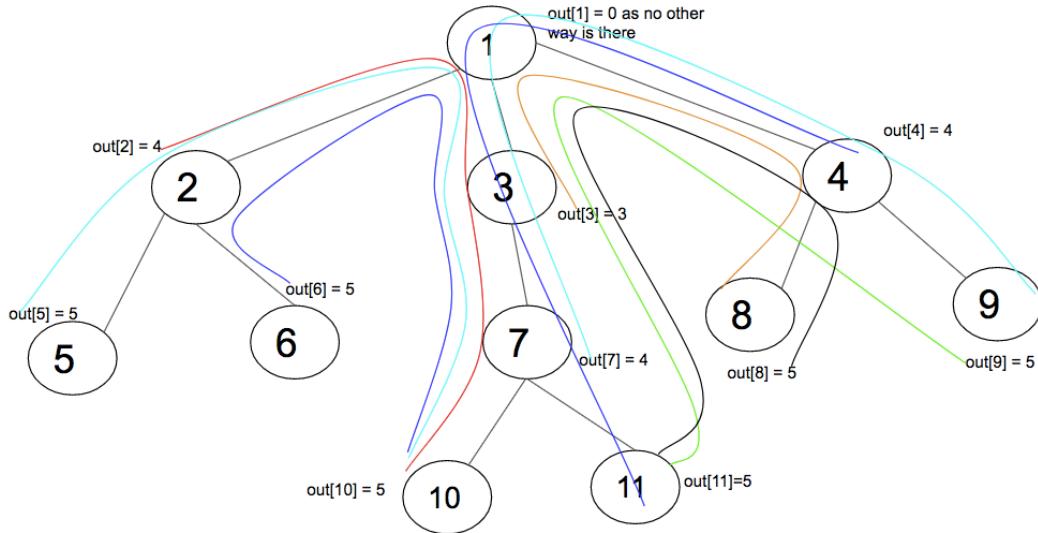
The maximum height of tree when node i is considered as root will be **max(in[i], out[i])**.

Calculation of in[i] :



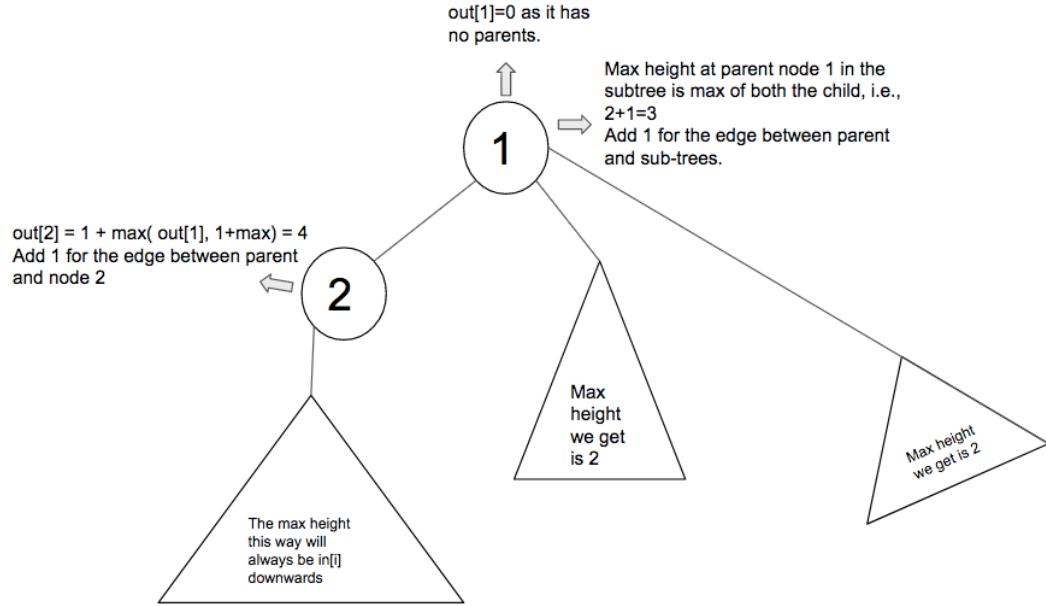
In the image above, values of $\text{in}[i]$ have been calculated for every node i . The maximum of every subtree is taken and added with 1 to the parent of that subtree. Add 1 for the edge between parent and subtree. Traverse the tree using DFS and calculate $\text{in}[i]$ as $\max(\text{in}[i], 1+\text{in}[\text{child}])$ for every node.

Calculation of out[i] :



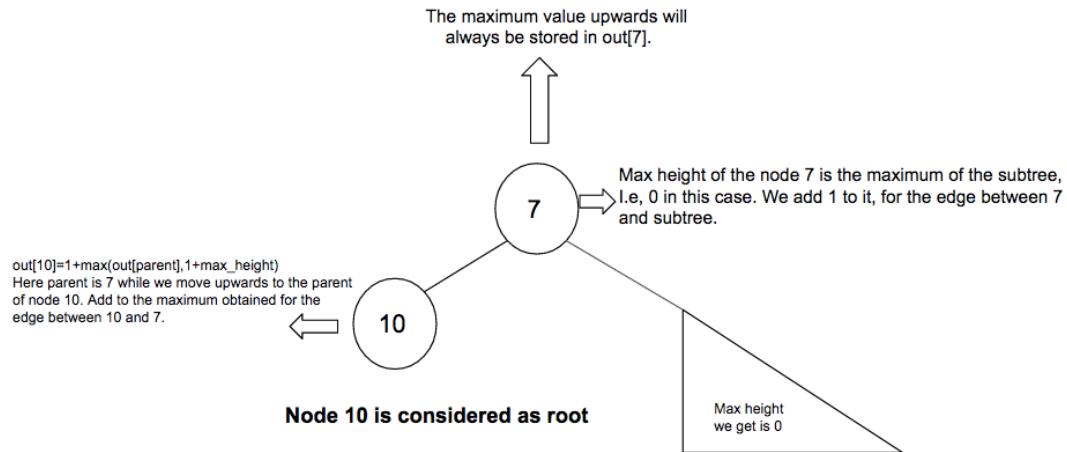
The above diagram shows all the $\text{out}[i]$ values and the path. For calculation of $\text{out}[i]$, move upwards to the parent of node i . From the parent of node i , there are two ways to move in, one will be in all the branches of the parent. The other direction is to move to the parent(call it parent2 to avoid confusion) of the parent(call it parent1) of node i . The maximum height upwards via parent2 is $\text{out}[\text{parent1}]$ itself. Generally, $\text{out}[\text{node } i]$ as $1+\max(\text{out}[i], 1+\max$

of all branches). Add 1 for the edges between node and parent.



Node 2 is considered as root

The above diagram explains the calculation of $out[i]$ when 2 is considered as the root of the tree. The branches of node 2 is not taken into count since the maximum height via that path has already been calculated and stored in $i[2]$. Moving up, in this case, the parent of 2 i.e., 1 has no parent. So, the branches except for the one which has the node are considered while calculating the maximum.



The above diagram explains the calculation of $out[10]$. The parent of node 10, i.e., 7 has a parent and a branch(precisely a child in this case). So the maximum height of both has been taken to count in such cases when parent and branches exist.

In case of multiple branches of a parent, take the longest of them to count(excluding the branch in which the node lies)

Calculating the maximum height of all the branches connected to parent :
 in[i] stores the maximum height while moving downwards. No need to store all the lengths of branches. Only the first and second maximum length among all the branches will give answer. Since, algorithm used is based on DFS, all the branches connected to parent will be considered, including the branch which has the node. If the first maximum path thus obtained is same as in[i], then maximum1 is the length of the branch in which node i lies. In this case, our longest path will be maximum2.

Recurrence relation of in[i] and out[i] :

$$\begin{aligned} \text{in}[i] &= \max(\text{in}[i], 1 + \text{in}[\text{child}]) \\ \text{out}[i] &= 1 + \max(\text{out}[\text{parent of } i], 1 + \text{longest path of all branches of parent of } i) \end{aligned}$$

Below is the implementation of the above idea :

C++

```
// CPP code to find the maximum path length
// considering any node as root
#include <bits/stdc++.h>
using namespace std;
const int MAX_NODES = 100;

int in[MAX_NODES];
int out[MAX_NODES];

// function to pre-calculate the array in[]
// which stores the maximum height when travelled
// via branches
void dfs1(vector<int> v[], int u, int parent)
{
    // initially every node has 0 height
    in[u] = 0;

    // traverse in the subtree of u
    for (int child : v[u]) {

        // if child is same as parent
        if (child == parent)
            continue;

        // dfs called
        dfs1(v, child, u);

        // recursively calculate the max height
        in[u] = max(in[u], 1 + in[child]);
    }
}

// calculate the out[] array
void dfs2(vector<int> v[], int u, int parent)
{
    // calculate the out[u]
    out[u] = 1 + max(out[parent], in[u]);

    // traverse in the subtree of u
    for (int child : v[u]) {
        // if child is same as parent
        if (child == parent)
            continue;

        // dfs called
        dfs2(v, child, u);
    }
}
```

```

        in[u] = max(in[u], 1 + in[child]);
    }
}

// function to pre-calculate the array ouut[]
// which stores the maximum height when traveled
// via parent
void dfs2(vector<int> v[], int u, int parent)
{
    // stores the longest and second
    // longest branches
    int mx1 = -1, mx2 = -1;

    // traverse in the subtress of u
    for (int child : v[u]) {
        if (child == parent)
            continue;

        // compare and store the longest
        // and second longest
        if (in[child] >= mx1) {
            mx2 = mx1;
            mx1 = in[child];
        }

        else if (in[child] > mx2)
            mx2 = in[child];
    }

    // traverse in the subtree of u
    for (int child : v[u]) {
        if (child == parent)
            continue;

        int longest = mx1;

        // if longest branch has the node, then
        // consider the second longest branch
        if (mx1 == in[child])
            longest = mx2;

        // recursively calculate out[i]
        out[child] = 1 + max(out[u], 1 + longest);

        // dfs fucntion call
        dfs2(v, child, u);
    }
}

```

```
// function to print all the maximum heights
// from every node
void printHeights(vector<int> v[], int n)
{
    // traversal to calculate in[] array
    dfs1(v, 1, 0);

    // traversal to calculate out[] array
    dfs2(v, 1, 0);

    // print all maximum heights
    for (int i = 1; i <= n; i++)
        cout << "The maximum height when node "
            << i << " is considered as root"
            << " is " << max(in[i], out[i])
            << "\n";
}

// Driver Code
int main()
{
    int n = 11;
    vector<int> v[n + 1];

    // initialize the tree given in the diagram
    v[1].push_back(2), v[2].push_back(1);
    v[1].push_back(3), v[3].push_back(1);
    v[1].push_back(4), v[4].push_back(1);
    v[2].push_back(5), v[5].push_back(2);
    v[2].push_back(6), v[6].push_back(2);
    v[3].push_back(7), v[7].push_back(3);
    v[7].push_back(10), v[10].push_back(7);
    v[7].push_back(11), v[11].push_back(7);
    v[4].push_back(8), v[8].push_back(4);
    v[4].push_back(9), v[9].push_back(4);

    // function to print the maximum height from every node
    printHeights(v, n);

    return 0;
}
```

Output :

```
The maximum height when node 1 is considered as root is 3
The maximum height when node 2 is considered as root is 4
```

The maximum height when node 3 is considered as root is 3
The maximum height when node 4 is considered as root is 4
The maximum height when node 5 is considered as root is 5
The maximum height when node 6 is considered as root is 5
The maximum height when node 7 is considered as root is 4
The maximum height when node 8 is considered as root is 5
The maximum height when node 9 is considered as root is 5
The maximum height when node 10 is considered as root is 5
The maximum height when node 11 is considered as root is 5

Time Complexity : $O(N)$

Auxiliary Space : $O(N)$

Source

<https://www.geeksforgeeks.org/dynamic-programming-trees-set-2/>

Chapter 52

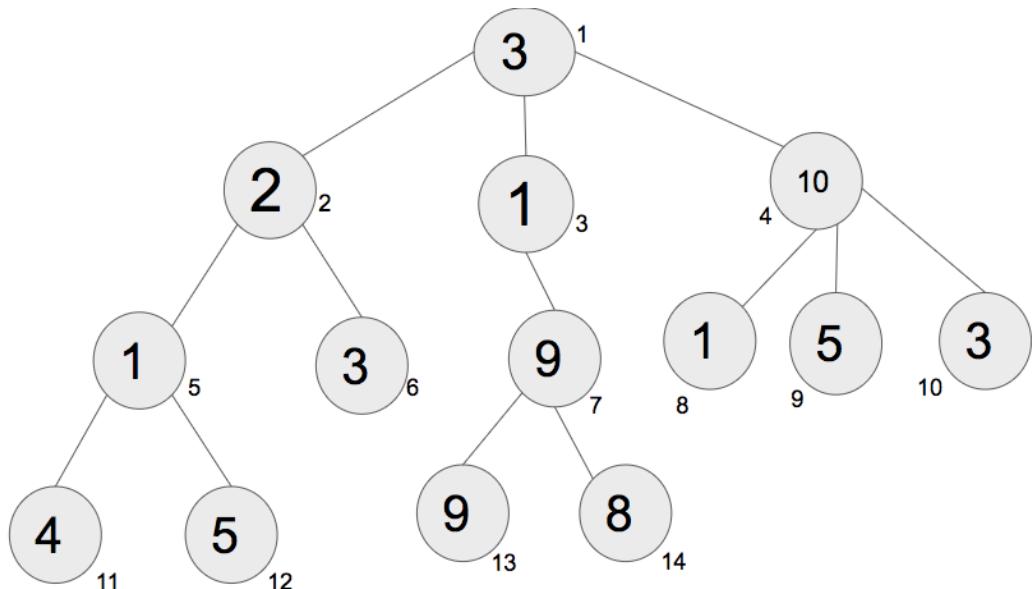
Dynamic Programming on Trees Set-1

Dynamic Programming on Trees Set-1 - GeeksforGeeks

Dynamic Programming(DP) is a technique to solve problems by breaking them down into overlapping sub-problems which follows the optimal substructure. There are various problems using DP like subset sum, knapsack, coin change etc. DP can also be applied on trees to solve some specific problems.

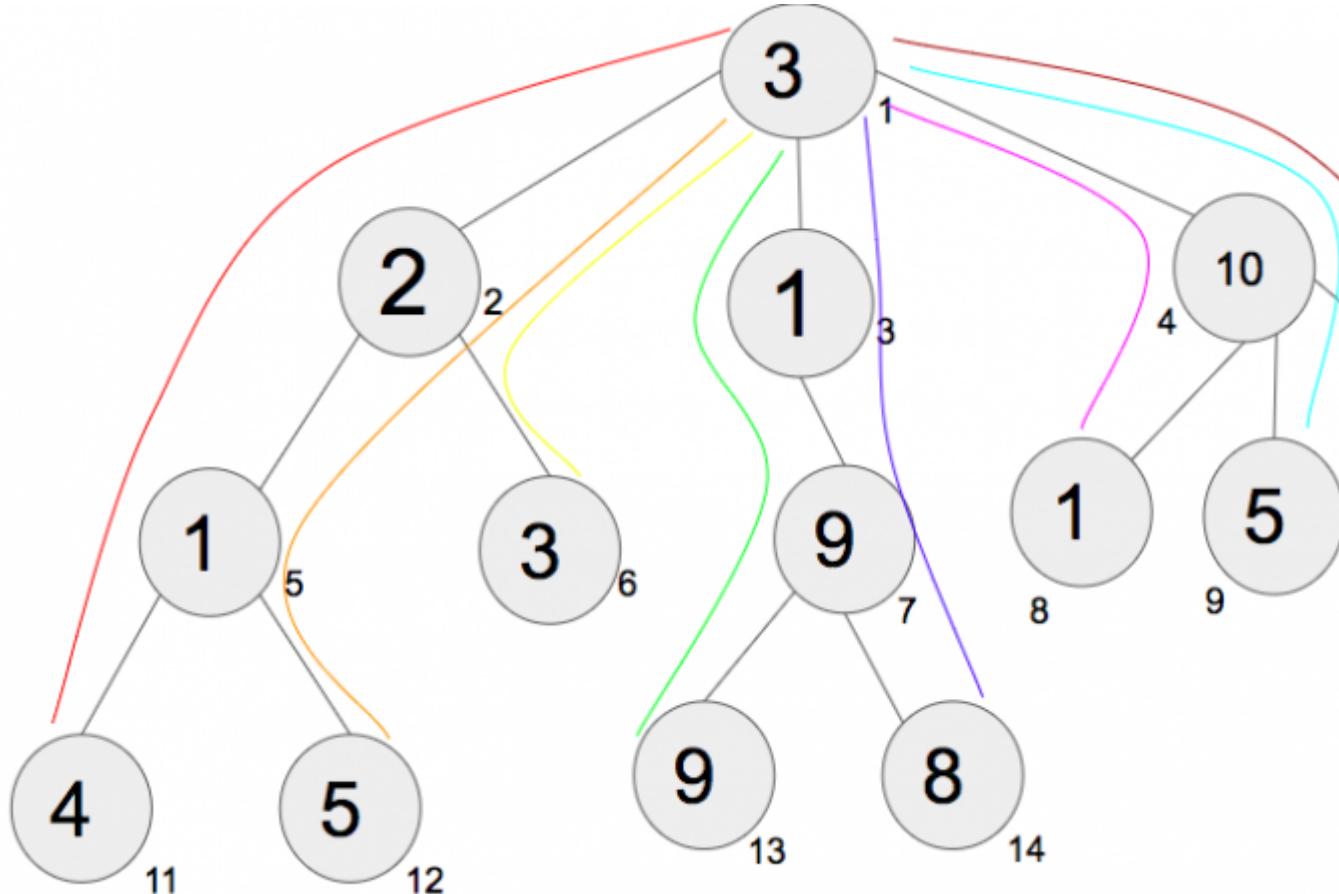
Pre-requisite: [DFS](#)

Given a tree with N nodes and N-1 edges, calculate the maximum sum of the node values from root to any of the leaves without re-visiting any node.



Given above is a diagram of a tree with $N=14$ nodes and $N-1=13$ edges. The values at node being **3, 2, 1, 10, 1, 3, 9, 1, 5, 3, 4, 5, 9** and **8** respectively for nodes 1, 2, 3, 4....14.

The diagram below shows all the paths from root to leaves :



All the paths are marked by different colors :

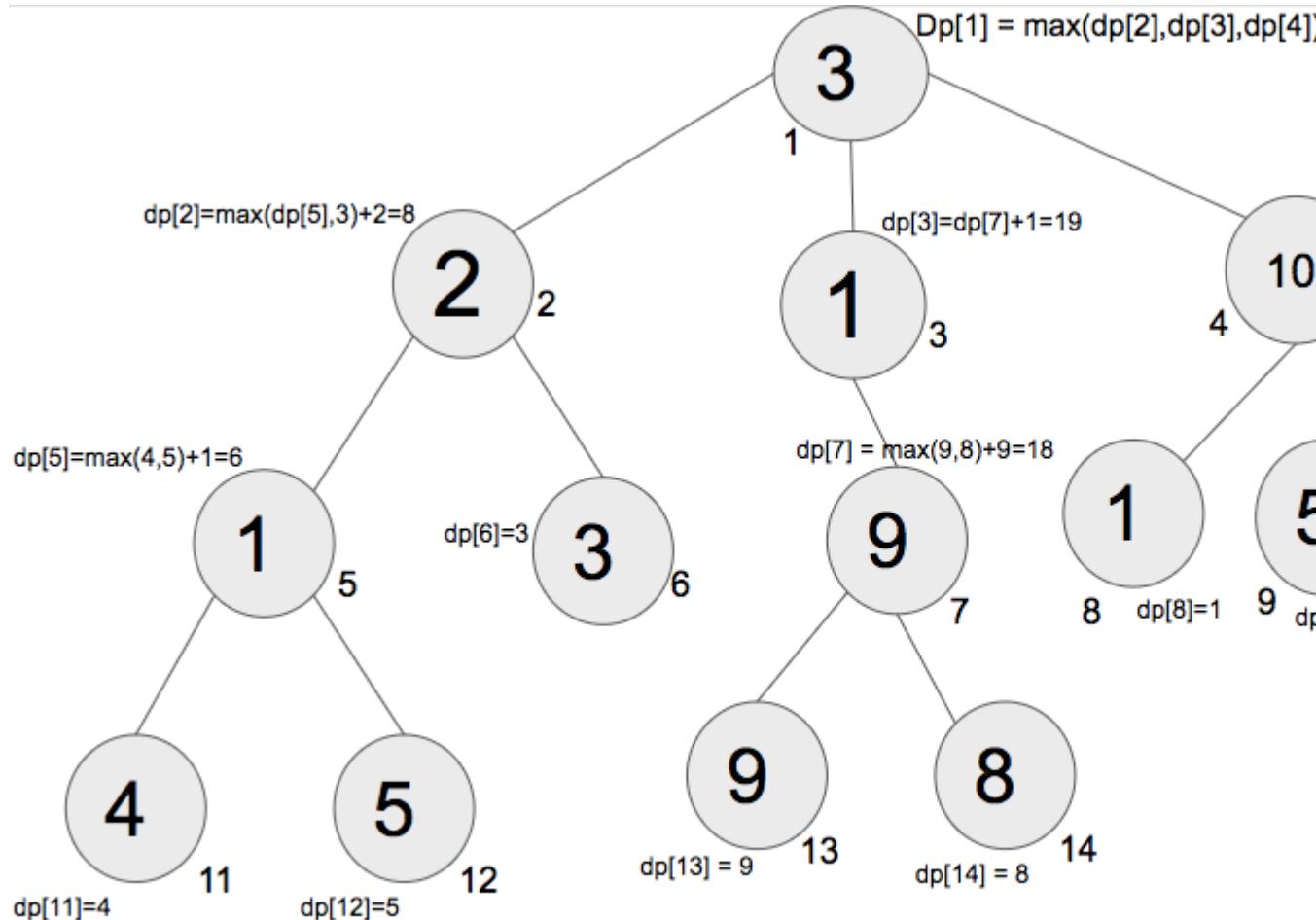
- Path 1(red, 3-2-1-4) : sum of all node values = 10
- Path 2(orange, 3-2-1-5) : sum of all node values = 11
- Path 3(yellow, 3-2-3) : sum of all node values = 8
- Path 4(green, 3-1-9-9) : sum of all node values = 22
- Path 5(violet, 3-1-9-8) : sum of all node values = 21
- Path 6(pink, 3-10-1) : sum of all node values = 14
- Path 7(blue, 3-10-5) : sum of all node values = 18
- Path 8(brown, 3-10-3) : sum of all node values = 16

The answer is 22, as Path 4 has the maximum sum of values of nodes in its path from a root to leaves.

The greedy approach fails in this case. Starting from the root and take 3 from the

first level, 10 from the next level and 5 from the third level greedily. Result is path-7 if after following greedy approach, hence do not apply greedy approach over here.

The problem can be solved using **Dynamic Programming on trees**. Start memoizing from the leaves and add the maximum of leaves to the root of every sub-tree. At the last step, there will be root and the sub-tree under it, adding the value at node and maximum of sub-tree will give us the maximum sum of the node values from root to any of the leaves.



The diagram above shows how to **start from the leaves and add the maximum of leaves of a sub-tree to its root**. Move upward and repeat the same procedure of storing the maximum of every sub-tree leaves and adding it to its root. In this example, the maximum of node 11 and 12 is taken to count and then added to node 5(**In this sub-tree, 5 is the root and 11, 12 are its leaves**). Similarly, the maximum of node 13 and 15 is taken to count and then added to node 7. Repeat the steps for every sub-tree till we reach the node.

Let DP_i be the maximum summation of node values in the path between i and any of its leaves moving downwards. **Traverse the tree using DFS traversal**. Store the maximum

of all the leaves of the sub-tree, and add it to the root of the sub-tree. At the end, DP_1 will have the maximum sum of the node values from root to any of the leaves without re-visiting any node.

Below is the implementation of the above idea :

CPP

```
// CPP code to find the maximum path sum
#include <bits/stdc++.h>
using namespace std;

int dp[100];

// function for dfs traversal and to store the
// maximum value in dp[] for every node till the leaves
void dfs(int a[], vector<int> v[], int u, int parent)
{
    // initially dp[u] is always a[u]
    dp[u] = a[u - 1];

    // stores the maximum value from nodes
    int maximum = 0;

    // traverse the tree
    for (int child : v[u]) {

        // if child is parent, then we continue
        // without recursing further
        if (child == parent)
            continue;

        // call dfs for further traversal
        dfs(a, v, child, u);

        // store the maximum of previous visited node
        // and present visited node
        maximum = max(maximum, dp[child]);
    }

    // add the maximum value returned to the parent node
    dp[u] += maximum;
}

// function that returns the maximum value
int maximumValue(int a[], vector<int> v[])
{
    dfs(a, v, 1, 0);
    return dp[1];
}
```

```
// Driver Code
int main()
{
    // number of nodes
    int n = 14;

    // adjacency list
    vector<int> v[n + 1];

    // create undirected edges
    // initialize the tree given in the diagram
    v[1].push_back(2), v[2].push_back(1);
    v[1].push_back(3), v[3].push_back(1);
    v[1].push_back(4), v[4].push_back(1);
    v[2].push_back(5), v[5].push_back(2);
    v[2].push_back(6), v[6].push_back(2);
    v[3].push_back(7), v[7].push_back(3);
    v[4].push_back(8), v[8].push_back(4);
    v[4].push_back(9), v[9].push_back(4);
    v[4].push_back(10), v[10].push_back(4);
    v[5].push_back(11), v[11].push_back(5);
    v[5].push_back(12), v[12].push_back(5);
    v[7].push_back(13), v[13].push_back(7);
    v[7].push_back(14), v[14].push_back(7);

    // values of node 1, 2, 3....14
    int a[] = { 3, 2, 1, 10, 1, 3, 9, 1, 5, 3, 4, 5, 9, 8 };

    // function call
    cout << maximumValue(a, v);

    return 0;
}
```

Output:

22

Time Complexity : $O(N)$, where N is the number of nodes.

Source

<https://www.geeksforgeeks.org/dynamic-programming-trees-set-1/>

Chapter 53

Efficiently design Insert, Delete and Median queries on a set

Efficiently design Insert, Delete and Median queries on a set - GeeksforGeeks

Given an empty set initially and a number of queries on it, each possibly of the following types:

1. **Insert** – Insert a new element ‘x’.
2. **Delete** – Delete an existing element ‘x’.
3. **Median** – Print the median element of the numbers currently in the set

Example:

```
Input : Insert 1
        Insert 4
        Insert 7
        Median
Output : The first three queries
         should insert 1, 4 and 7
         into an empty set. The
         fourth query should return
         4 (median of 1, 4, 7).
```

For expository purpose, we assume the following, but these assumptions are not the limitations of the method discussed here:

1. At any instance, all elements are distinct, that is, none of them occurs more than once.
2. The ‘Median’ query is made only when there are odd number of elements in the set. (We will need to make two queries on our segment tree, in case of even numbers)
3. The elements in the set range from 1 to $+10^6$.

Method 1 (Naive)

In naive implementation, we can do first two queries in $O(1)$, but the last query in $O(\max_elem)$, where \max_elem is the maximum element of all times (including deleted elements).

Let us assume an array **count[]** (of size $10^6 + 1$) to maintain the count of each element in the subset. Following are simple and self explanatory algorithms for the 3 queries:

Insert x query:

```
count[x]++;
if (x > max_elem)
    max_elem = x;
n++;
```

Delete x query:

```
if (count[x] > 0)
    count[x]--;
n--;
```

Median query:

```
sum = 0;
i = 0;
while( sum <= n / 2 )
{
    i++;
    sum += count[i];
}
median = i;
return median;
```

Illustration of array **count[]**, representing the set {1, 4, 7, 8, 9}, the median element is ‘7’:

0	1	2	3	4	5	6	7	8	9	10	11	12
0	1	0	0	1	0	0	1	1	1	0	0	0

The ‘Median’ query intends to find the $(n + 1)/2$ th ‘1’ in the array, in this case, 3rd ‘1’; now we do the same using a segment tree.

Method 2(Using Segment Tree)

We make a [segment tree](#) to store sum of intervals, where an interval $[a, b]$ represents the number of elements present in the set, currently, in the range $[a, b]$. For example, if we consider the above example, query(3, 7) returns 2, query(4, 4) returns 1, query(5, 5) returns 0.

Insert and delete queries are simple and both can be implemented using function update(int x, int diff) (adds ‘diff’ at index ‘x’)

Algorithm

```

// adds 'diff' at index 'x'
update(node, a, b, x, diff)

// If leaf node
If a == b and a == x
    segmentTree[node] += diff

// If non-leaf node and x lies in its range
If x is in [a, b]

    // Update children recursively
    update(2*node, a, (a + b)/2, x, diff)
    update(2*node + 1, (a + b)/2 + 1, b, x, diff)

    // Update node
    segmentTree[node] = segmentTree[2 * node] +
        segmentTree[2 * node + 1]

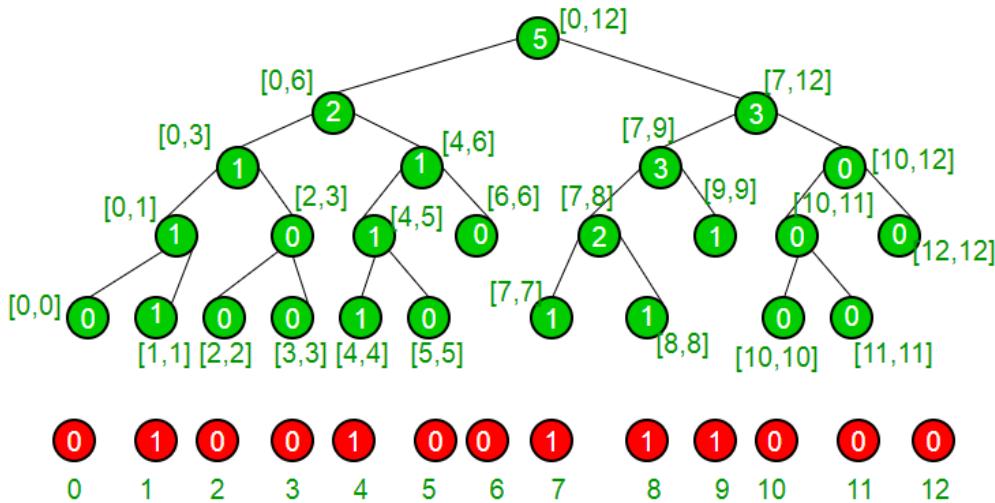
```

The above recursive function runs in $O(\log(\max_elem))$ (in this case \max_elem is 10^6) and used for both insertion and deletion with the following calls:

1. Insert ‘x’ is done using $\text{update}(1, 0, 10^6, x, 1)$. Note that root of tree is passed, start index is passed as 0 and end index as 10^6 so that all ranges that have x are updated.
2. Delete ‘x’ is done using $\text{update}(1, 0, 10^6, x, -1)$. Note that root of tree is passed, start index is passed as 0 and end index as 10^6 so that all ranges that have x are updated.

Now, the function to find the index with kth ‘1’, where ‘k’ in this case will always be $(n + 1) / 2$, this is going to work a lot like binary search, you can think of it as a recursive binary search function on a segment tree.

Let’s take an example to understand, our set currently has elements { 1, 4, 7, 8, 9 }, and hence is represented by the following segment tree.



If we are at a non-leaf node, we are sure that it has both children, we see if the left child has more or equal number of one's as 'k', if yes, we are sure our index lies in the left subtree, otherwise, if left subtree has less number of 1's than k, then we are sure that our index lies in the right subtree. We do this recursively to reach our index and from there, we return it.

Algorithm

```

1.findKth(node, a, b, k)
2. If a != b
3.   If segmentTree[ 2 * node ] >= k
4.     return findKth(2*node, a, (a + b)/2, k)
5.   else
6.     return findKth(2*node + 1, (a + b)/2 + 1,
                      b, k - segmentTree[ 2 * node ])
7.   else
8.     return a

```

The above recursive function runs in $O(\log(\max_elem))$.

```

// A C++ program to implement insert, delete and
// median queries using segment tree
#include<bits/stdc++.h>
#define maxn 3000005
#define max_elem 1000000
using namespace std;

// A global array to store segment tree.
// Note: Since it is global, all elements are 0.
int segmentTree[maxn];

```

```

// Update 'node' and its children in segment tree.
// Here 'node' is index in segmentTree[], 'a' and
// 'b' are starting and ending indexes of range stored
// in current node.
// 'diff' is the value to be added to value 'x'.
void update(int node, int a, int b, int x, int diff)
{
    // If current node is a leaf node
    if (a == b && a == x)
    {
        // add 'diff' and return
        segmentTree[node] += diff;
        return ;
    }

    // If current node is non-leaf and 'x' is in its
    // range
    if (x >= a && x <= b)
    {
        // update both sub-trees, left and right
        update(node*2, a, (a + b)/2, x, diff);
        update(node*2 + 1, (a + b)/2 + 1, b, x, diff);

        // Finally update current node
        segmentTree[node] = segmentTree[node*2] +
                           segmentTree[node*2 + 1];
    }
}

// Returns k'th node in segment tree
int findKth(int node, int a, int b, int k)
{
    // non-leaf node, will definitely have both
    // children; left and right
    if (a != b)
    {
        // If kth element lies in the left subtree
        if (segmentTree[node*2] >= k)
            return findKth(node*2, a, (a + b)/2, k);

        // If kth one lies in the right subtree
        return findKth(node*2 + 1, (a + b)/2 + 1,
                      b, k - segmentTree[node*2]);
    }

    // if at a leaf node, return the index it stores
    // information about
}

```

```
        return (segmentTree[node])? a : -1;
    }

// insert x in the set
void insert(int x)
{
    update(1, 0, max_elem, x, 1);
}

// delete x from the set
void delet(int x)
{
    update(1, 0, max_elem, x, -1);
}

// returns median element of the set with odd
// cardinality only
int median()
{
    int k = (segmentTree[1] + 1) / 2;
    return findKth(1, 0, max_elem, k);
}

// Driver code
int main()
{
    insert(1);
    insert(4);
    insert(7);
    cout << "Median for the set {1,4,7} = "
          << median() << endl;
    insert(8);
    insert(9);
    cout << "Median for the set {1,4,7,8,9} = "
          << median() << endl;
    delet(1);
    delet(8);
    cout << "Median for the set {4,7,9} = "
          << median() << endl;
    return 0;
}
```

Output:

```
Median for the set {1,4,7} = 4
Median for the set {1,4,7,8,9} = 7
Median for the set {4,7,9} = 7
```

Conclusion:

All three queries run in $O(\log(\max_elem))$, in this case $\max_elem = 10^6$, so $\log(\max_elem)$ is approximately equal to 20.

The segment tree uses $O(\max_elem)$ space.

If the delete query wasn't there, the problem could have also been done with a famous algorithm [here](#).

Source

<https://www.geeksforgeeks.org/efficiently-design-insert-delete-median-queries-set/>

Chapter 54

Euler Tour Subtree Sum using Segment Tree

Euler Tour Subtree Sum using Segment Tree - GeeksforGeeks

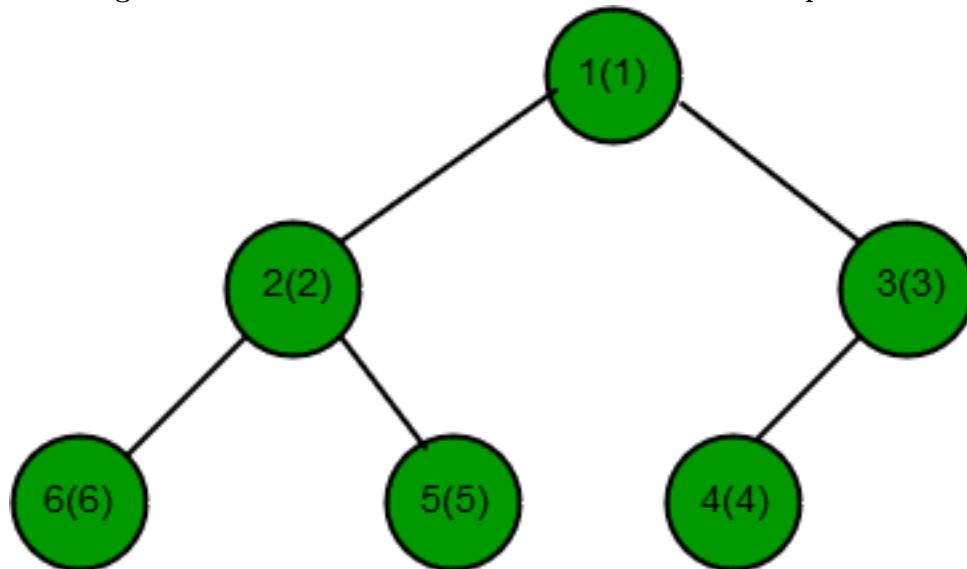
Euler tour tree (ETT) is a method for representing a rooted tree as a number sequence. When traversing the tree using [Depth for search\(DFS\)](#), insert each node in a vector twice, once while entered it and next after visiting all its children. This method is very useful for solving subtree problems and one such problem is **Subtree Sum**.

Prerequisite : [Segment Tree\(Sum of given range\)](#)

Naive Approach :

Consider a rooted tree with 6 vertices connected as given in the below diagram. Apply DFS for different queries.

The weight associated with each node is written inside the parenthesis.

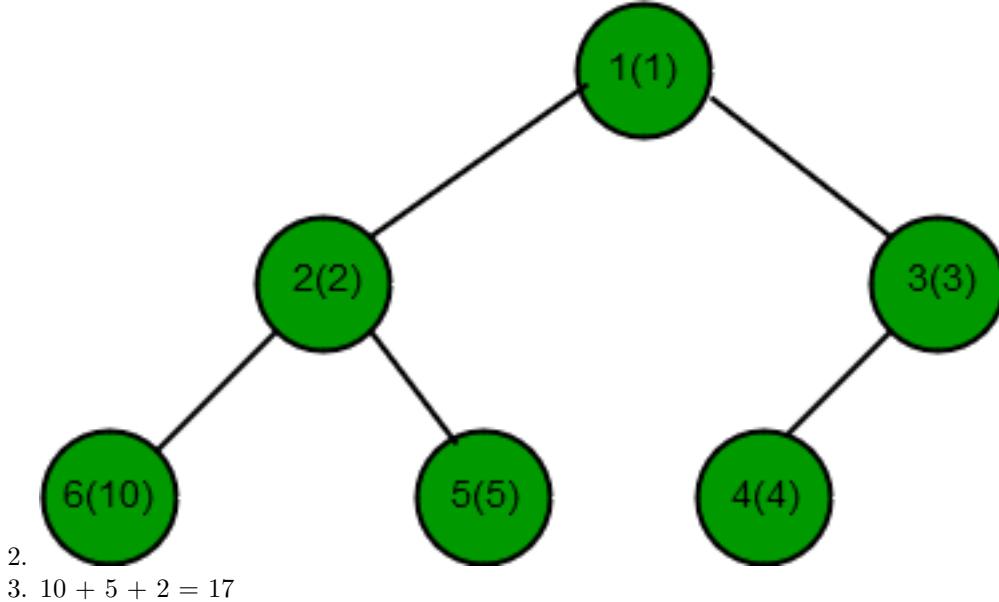


Queries :

1. Sum of all the subtrees of node 1.
2. Update the value of node 6 to 10.
3. Sum of all the subtrees of node 2.

Answers :

1. $6 + 5 + 4 + 3 + 2 + 1 = 21$



- 2.
3. $10 + 5 + 2 = 17$

Time Complexity Analysis :

Such queries can be performed using depth first search(dfs) in **O(n)** time complexity.

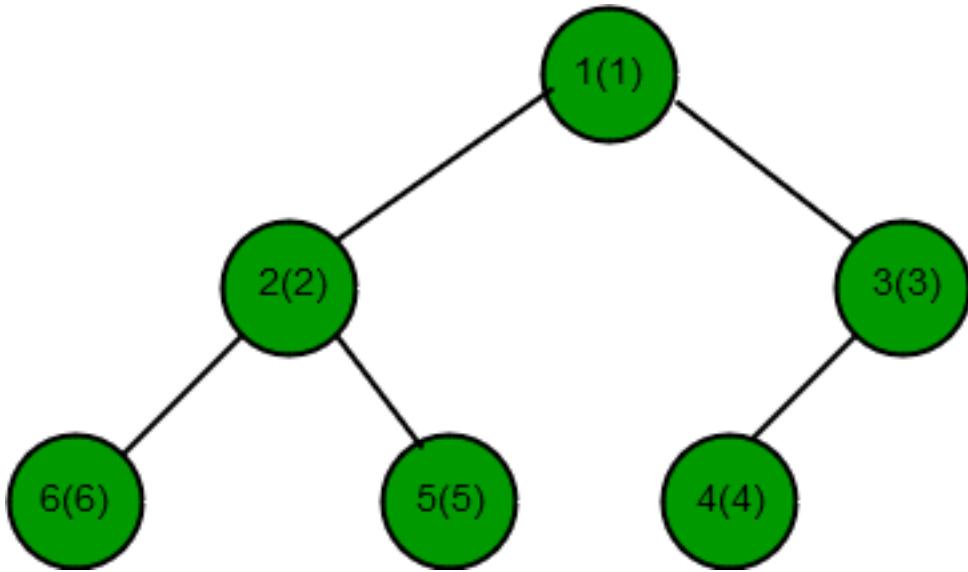
Efficient Approach :

The time complexity for such queries can be reduced to **O(log(n))** time by converting the rooted tree into segment tree using Euler tour technique. So, When the number of queries are q, the total complexity becomes **O(q*5log(n))**.

Euler Tour :

In Euler tour Technique, each vertex is added to the vector twice, while descending into it and while leaving it.

Let us understand with the help of previous example :



On performing depth first search(DFS) using euler tour technique on the given rooted tree, the vector so formed is :

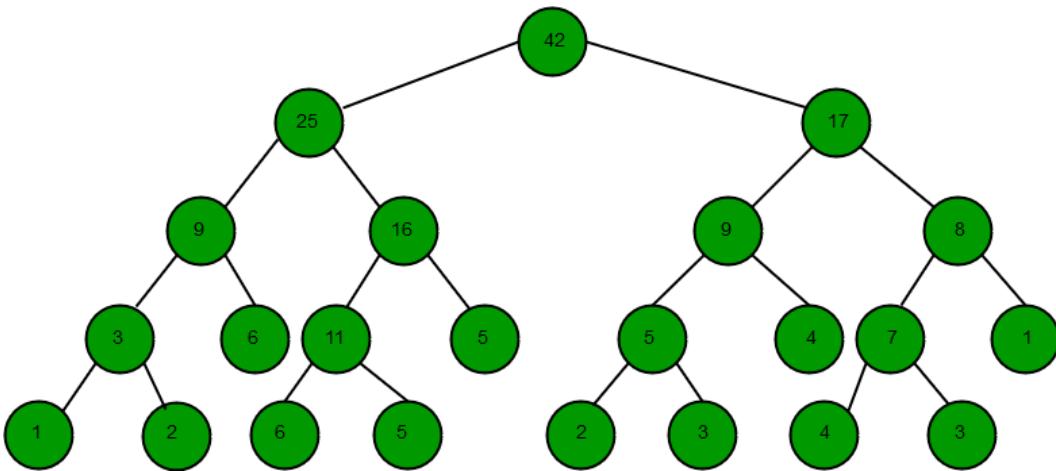
```
s[]={1, 2, 6, 6, 5, 5, 2, 3, 4, 4, 3, 1}
```

```
// DFS function to traverse the tree
int dfs(int root)
{
    s.push_back(root);
    if (v[root].size() == 0)
        return root;

    for (int i = 0; i < v[root].size(); i++) {
        int temp = dfs(v[root][i]);
        s.push_back(temp);
    }
    return root;
}
```

Now, use vector s[] to [Create Segment Tree](#).

Below is the representation of segment tree of vector s[].



For the output and update query, store the entry time and exit time(which serve as index range) for each node of the rooted tree.

```
s[]={1, 2, 6, 6, 5, 5, 2, 3, 4, 4, 3, 1}
```

Node	Entry time	Exit time
1	1	12
2	2	7
3	8	11
4	9	10
5	5	6
6	3	4

Query of type 1 :

Find the range sum on segment tree for output query where range is exit time and entry time of the rooted tree node. Deduce that the answer is always twice the expected answer because each node is added twice in segment tree. So reduce the answer by half.

Query of type 2 :

For update query, update the leaf node of segment tree at the entry time and exit time of the rooted tree node.

Below is the implementation of above approach :

```
// C++ program for implementation of
// Euler Tour | Subtree Sum.
#include <bits/stdc++.h>
using namespace std;

vector<int> v[1001];
vector<int> s;
int seg[1001] = { 0 };
```

```

// Value/Weight of each node of tree,
// value of 0th(no such node) node is 0.
int ar[] = { 0, 1, 2, 3, 4, 5, 6 };

int vertices = 6;
int edges = 5;

// A recursive function that constructs
// Segment Tree for array ar[] = { }.
// 'pos' is index of current node
// in segment tree seg[] .
int segment(int low, int high, int pos)
{
    if (high == low) {
        seg[pos] = ar[s[low]];
    }
    else {
        int mid = (low + high) / 2;
        segment(low, mid, 2 * pos);
        segment(mid + 1, high, 2 * pos + 1);
        seg[pos] = seg[2 * pos] + seg[2 * pos + 1];
    }
}

/* Return sum of elements in range
   from index l to r . It uses the
   seg[] array created using segment()
   function. 'pos' is index of current
   node in segment tree seg[] .
*/
int query(int node, int start,
          int end, int l, int r)
{
    if (r < start || end < l) {
        return 0;
    }

    if (l <= start && end <= r) {
        return seg[node];
    }

    int mid = (start + end) / 2;
    int p1 = query(2 * node, start,
                  mid, l, r);
    int p2 = query(2 * node + 1, mid + 1,
                  end, l, r);

    return (p1 + p2);
}

```

```

}

/* A recursive function to update the
   nodes which have the given index in
   their range. The following are
   parameters pos --> index of current
   node in segment tree seg[]. idx -->
   index of the element to be updated.
   This index is in input array.
   val --> Value to be change at node idx
*/
int update(int pos, int low, int high,
           int idx, int val)
{
    if (low == high) {
        seg[pos] = val;
    }
    else {
        int mid = (low + high) / 2;

        if (low <= idx && idx <= mid) {
            update(2 * pos, low, mid,
                   idx, val);
        }
        else {
            update(2 * pos + 1, mid + 1,
                   high, idx, val);
        }

        seg[pos] = seg[2 * pos] + seg[2 * pos + 1];
    }
}

/* A recursive function to form array
   ar[] from a directed tree .
*/
int dfs(int root)
{
    // pushing each node in vector s
    s.push_back(root);
    if (v[root].size() == 0)
        return root;

    for (int i = 0; i < v[root].size(); i++) {
        int temp = dfs(v[root][i]);
        s.push_back(temp);
    }
    return root;
}

```

```

}

// Driver program to test above functions
int main()
{
    // Edges between the nodes
    v[1].push_back(2);
    v[1].push_back(3);
    v[2].push_back(6);
    v[2].push_back(5);
    v[3].push_back(4);

    // Calling dfs function.
    int temp = dfs(1);
    s.push_back(temp);

    // Storing entry time and exit
    // time of each node
    vector<pair<int, int> > p;

    for (int i = 0; i <= vertices; i++)
        p.push_back(make_pair(0, 0));

    for (int i = 0; i < s.size(); i++) {
        if (p[s[i]].first == 0)
            p[s[i]].first = i + 1;
        else
            p[s[i]].second = i + 1;
    }

    // Build segment tree from array ar[].
    segment(0, s.size() - 1, 1);

    // query of type 1 return the
    // sum of subtree at node 1.
    int node = 1;
    int e = p[node].first;
    int f = p[node].second;

    int ans = query(1, 1, s.size(), e, f);

    // print the sum of subtree
    cout << "Subtree sum of node " << node << " is : " << (ans / 2) << endl;

    // query of type 2 return update
    // the subtree at node 6.
    int val = 10;
    node = 6;
}

```

```
e = p[node].first;
f = p[node].second;
update(1, 1, s.size(), e, val);
update(1, 1, s.size(), f, val);

// query of type 1 return the
// sum of subtree at node 2.
node = 2;

e = p[node].first;
f = p[node].second;

ans = query(1, 1, s.size(), e, f);

// print the sum of subtree
cout << "Subtree sum of node " << node << " is : " << (ans / 2) << endl;

return 0;
}
```

Output:

```
Subtree sum of node 1 is : 21
Subtree sum of node 2 is : 17
```

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

Source

<https://www.geeksforgeeks.org/euler-tour-subtree-sum-using-segment-tree/>

Chapter 55

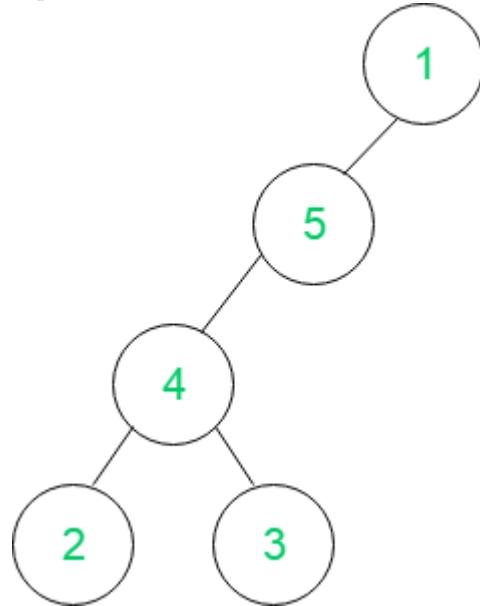
Euler tour of Binary Tree

Euler tour of Binary Tree - GeeksforGeeks

Given a binary tree where each node can have at most two child nodes, the task is to find the Euler tour of the binary tree. Euler tour is represented by a pointer to the topmost node in the tree. If the tree is empty, then value of root is NULL.

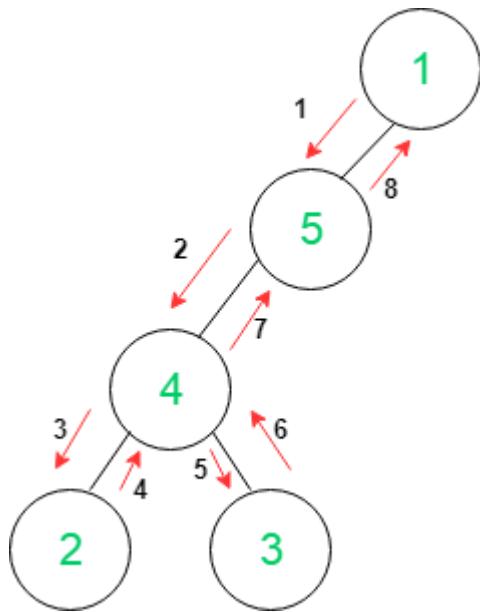
Examples:

Input :



Output: 1 5 4 2 4 3 4 5 1

Approach:



- (1) First, start with root node 1, **Euler[0]=1**
- (2) Go to left node i.e, node 5, **Euler[1]=5**
- (3) Go to left node i.e, node 4, **Euler[2]=4**
- (4) Go to left node i.e, node 2, **Euler[3]=2**
- (5) Go to left node i.e, NULL, go to parent node 4 **Euler[4]=4**
- (6) Go to right node i.e, node 3 **Euler[5]=3**
- (7) No child, go to parent, node 4 **Euler[6]=4**
- (8) All child discovered, go to parent node 5 **Euler[7]=5**
- (9) All child discovered, go to parent node 1 **Euler[8]=1**

[Euler tour of tree](#) has been already discussed where it can be applied to N-ary tree which is represented by adjacency list. If a Binary tree is represented by the classical structured way by links and nodes, then there need to first convert the tree into adjacency list representation and then we can find the Euler tour if we want to apply method discussed in the original post. But this increases the space complexity of the program. Here, In this post, a generalized space-optimized version is discussed which can be directly applied to binary trees represented by structure nodes.

This method :

- (1) Works without the use of Visited arrays.
- (2) Requires exactly $2*N-1$ vertices to store Euler tour.

```

// C++ program to find euler tour of binary tree
#include <bits/stdc++.h>
using namespace std;

/* A tree node structure */
struct Node {
    int data;
    struct Node* left;
    
```

```

    struct Node* right;
};

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

// Find Euler Tour
void eulerTree(struct Node* root, vector<int> &Euler)
{
    // store current node's data
    Euler.push_back(root->data);

    // If left node exists
    if (root->left)
    {
        // traverse left subtree
        eulerTree(root->left, Euler);

        // store parent node's data
        Euler.push_back(root->data);
    }

    // If right node exists
    if (root->right)
    {
        // traverse right subtree
        eulerTree(root->right, Euler);

        // store parent node's data
        Euler.push_back(root->data);
    }
}

// Function to print Euler Tour of tree
void printEulerTour(Node *root)
{
    // Stores Euler Tour
    vector<int> Euler;

    eulerTree(root, Euler);

    for (int i = 0; i < Euler.size(); i++)

```

```
        cout << Euler[i] << " ";
}

/* Driver function to test above functions */
int main()
{
    // Constructing tree given in the above figure
    Node* root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->left = newNode(6);
    root->right->right = newNode(7);
    root->right->left->right = newNode(8);

    // print Euler Tour
    printEulerTour(root);

    return 0;
}
```

Output:

```
1 2 4 2 5 2 1 3 6 8 6 3 7 3 1
```

Time Complexity: $O(2*N-1)$ where N is number of nodes in the tree.
Auxiliary Space : $O(2*N-1)$ where N is number of nodes in the tree.

Source

<https://www.geeksforgeeks.org/euler-tour-binary-tree/>

Chapter 56

Extended Mo's Algorithm with O(1) time complexity

Extended Mo's Algorithm with O(1) time complexity - GeeksforGeeks

Given an array of n elements and q range queries (range sum in this article) with no updates, task is to answer these queries with efficient time and space complexity. The time complexity of a range query after applying square root decomposition comes out to be $O(\sqrt{n})$. This square-root factor can be decreased to a constant linear factor by applying square root decomposition on the block of the array which was decomposed earlier.

Prerequisite: [Mo's Algorithm Prefix Array](#)

Approach :

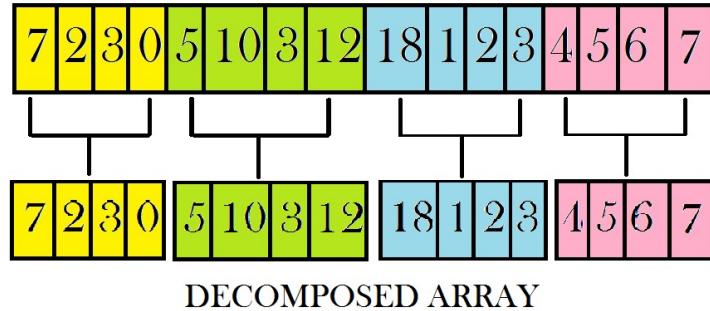
As we apply [square root decomposition](#) to the given array, querying a range-sum comes in $O(\sqrt{n})$ time.

Here, calculate the sum of blocks which are in between the blocks under consideration(cross blocks), which takes $O(\sqrt{n})$ iterations.

Initial Array :

7	2	3	0	5	10	3	12	18	1	2	3	4	5	6	7
---	---	---	---	---	----	---	----	----	---	---	---	---	---	---	---

Decomposition of array into blocks :



And the calculation time for the sum on the starting block and ending block both takes $O(\sqrt{n})$ iterations.

Which will leaves us per query time complexity of :

$$\begin{aligned}
 &= O(\sqrt{n}) + O(\sqrt{n}) + O(\sqrt{n}) \\
 &= 3 * O(\sqrt{n}) \\
 &\sim O(\sqrt{n})
 \end{aligned}$$

Here, we can reduce the runtime complexity of our query algorithm cleverly by calculating blockwise prefix sum and using it to calculate the sum accumulated in the blocks which lie between the blocks under consideration. Consider the code below :

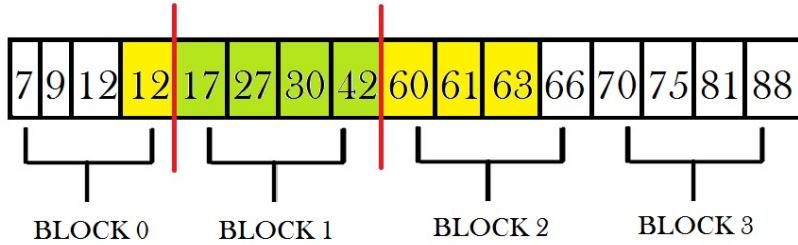
```
interblock_sum[x1][x2] = prefixData[x2 - 1] - prefixData[x1];
```

Time taken for calculation of above table is :

$$\begin{aligned}
 &= O(\sqrt{n}) * O(\sqrt{n}) \\
 &\sim O(n)
 \end{aligned}$$

NOTE : We haven't taken the sum of blocks x_1 & x_2 under consideration as they might be carrying partial data.

Prefix Array :



Suppose we want to query for the sum for range from 4 to 11, we consider the sum between block 0 and block 2 (excluding the data contained in block 0 and block 1), which can be calculated using the sum in the green coloured blocks represented in the above image.

$$\text{Sum between block 0 and block 2} = 42 - 12 = 30$$

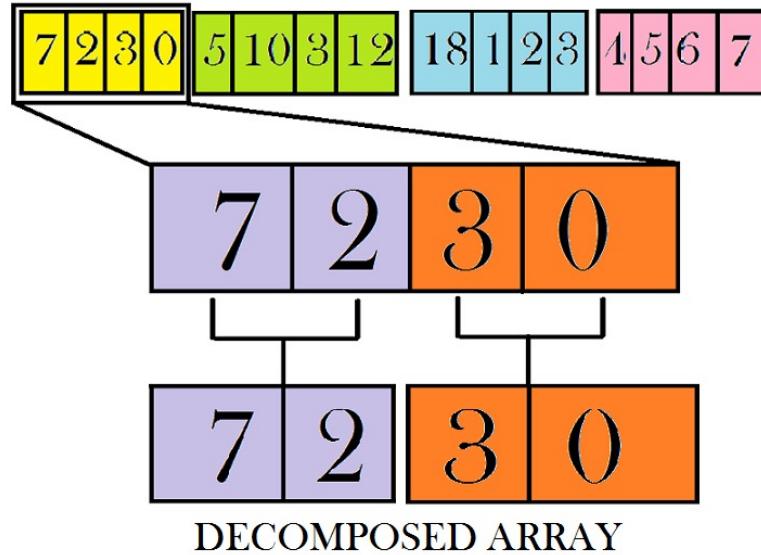
For calculation of rest of the sum present in the yellow blocks, consider the prefix array at the decomposition level-2 and repeat the process again.

Here, observe that we have reduced our time complexity per query significantly, though our runtime remains similar to our last approach :

Our new time complexity can be calculated as :

$$\begin{aligned} &= O(\sqrt{n}) + O(1) + O(\sqrt{n}) \\ &= 2 * O(\sqrt{n}) \\ &\sim O(\sqrt{n}) \end{aligned}$$

Square-root Decomposition at Level-2 :



Further, we apply square root decomposition again on every decomposed block retained from the previous decomposition. Now at this level, we have approximately $\sqrt{\sqrt{n}}$ sub-blocks in each block which were decomposed at last level. So, we need to run a range query on these blocks only two times, one time for starting block and one time for ending block.

Precalcuation Time taken for level 2 decomposition :

No of blocks at level 1 $\sim \sqrt{n}$

No of blocks at level 2 $\sim \sqrt{\sqrt{n}}$

Level-2 Decomposition Runtime of a level-1 decomposed block :

$$= O(\sqrt{n})$$

Overall runtime of level-2 decomposition over all blocks :

$$= O(\sqrt{n}) * O(\sqrt{n})$$

$$\sim O(n)$$

Now, we can query our level-2 decomposed blocks in $O(\sqrt{\sqrt{n}})$ time.

So, we have reduced our overall time complexity from $O(\sqrt{n})$ to $O(\sqrt{\sqrt{n}})$

Time complexity taken in querying edge blocks :

$$= O(\sqrt{\sqrt{n}}) + O(1) + O(\sqrt{\sqrt{n}})$$

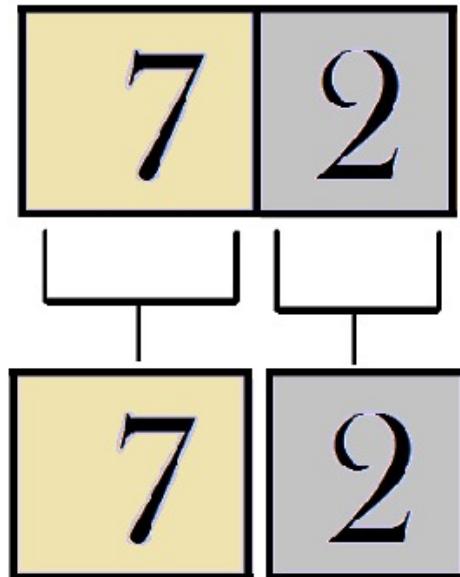
$$= 2 * O(\sqrt{\sqrt{n}})$$

$$\sim O(\sqrt{\sqrt{n}})$$

Total Time complexity can be calculated as :

$$\begin{aligned}
 &= O(\sqrt{\sqrt{n}}) + O(1) + O(\sqrt{\sqrt{n}}) \\
 &= 2 * O(\sqrt{\sqrt{n}}) \\
 &\sim O(\sqrt{\sqrt{n}})
 \end{aligned}$$

Square-root Decomposition at Level-3 :



DECOMPOSED ARRAY

Using this method we can decompose our array again and again recursively **d times** to reduce our time complexity to a factor of **constant linearity**.

$O(d * n^{1/(2^d)}) \sim O(k)$, as d increases this factor converges to a constant linear term

The code presented below is a representation of triple square root decomposition where $d = 3$:

$O(q * d * n^{1/(2^3)}) \sim O(q * k) \sim O(q)$

[where q represents number of range queries]

```
// CPP code for offline queries in
// approx constant time.
#include<bits/stdc++.h>
using namespace std;

int n1;

// Structure to store decomposed data
typedef struct
{
    vector<int> data;
    vector<vector<int>> rdata;
    int blocks;
    int blk_sz;
}sqrtD;

vector<vector<sqrtD>> Sq3;
vector<sqrtD> Sq2;
sqrtD Sq1;

// Square root Decomposition of
// a given array
sqrtD decompose(vector<int> arr)
{
    sqrtD sq;
    int n = arr.size();
    int blk_idx = -1;
    sq.blk_sz = sqrt(n);
    sq.data.resize((n/sq.blk_sz) + 1, 0);

    // Calculation of data in blocks
    for (int i = 0; i < n; i++)
    {
        if (i % sq.blk_sz == 0)
        {
            blk_idx++;
        }
        sq.data[blk_idx] += arr[i];
    }

    int blocks = blk_idx + 1;
    sq.blocks = blocks;

    // Calculation of prefix data
    int prefixData[blocks];
    prefixData[0] = sq.data[0];
    for(int i = 1; i < blocks; i++)
    {
```

```

        prefixData[i] =
            prefixData[i - 1] + sq.data[i];
    }

    sq.rdata.resize(blocks + 1,
                    vector<int>(blocks + 1));

    // Calculation of data between blocks
    for(int i = 0 ;i < blocks; i++)
    {
        for(int j = i + 1; j < blocks; j++)
        {
            sq.rdata[i][j] = sq.rdata[j][i] =
                prefixData[j - 1] - prefixData[i];
        }
    }

    return sq;
}

// Sqaure root Decompostion at level3
vector<vector<sqrtD>> tripleDecompose(sqrtD sq1,
                                             sqrtD sq2, vector<int> &arr)
{
    vector<vector<sqrtD>> sq(sq1.blocks,
                                vector<sqrtD>(sq1.blocks));

    int blk_idx1 = -1;

    for(int i = 0; i < sq1.blocks; i++)
    {
        int blk_ldx1 = blk_idx1 + 1;
        blk_idx1 = (i + 1) * sq1.blk_sz - 1;
        blk_idx1 = min(blk_idx1,n1 - 1);

        int blk_idx2 = blk_ldx1 - 1;

        for(int j = 0; j < sq2.blocks; ++j)
        {
            int blk_ldx2 = blk_idx2 + 1;
            blk_idx2 = blk_ldx1 + (j + 1) *
                sq2.blk_sz - 1;
            blk_idx2 = min(blk_idx2, blk_idx1);

            vector<int> ::iterator it1 =
                arr.begin() + blk_ldx2;
            vector<int> ::iterator it2 =
                arr.begin() + blk_idx2 + 1;
        }
    }
}

```

```

        vector<int> vec(it1, it2);
        sq[i][j] = decompose(vec);
    }
}
return sq;
}

// Sqaure root Decompostion at level2
vector<sqrtD> doubleDecompose(sqrtD sq1,
                                vector<int> &arr)
{
    vector<sqrtD> sq(sq1.blocks);
    int blk_idx = -1;
    for(int i = 0; i < sq1.blocks; i++)
    {
        int blk_ldx = blk_idx + 1;
        blk_idx = (i + 1) * sq1.blk_sz - 1;
        blk_idx = min(blk_idx, n1 - 1);
        vector<int> ::iterator it1 =
            arr.begin() + blk_ldx;
        vector<int> ::iterator it2 =
            arr.begin() + blk_idx + 1;
        vector<int> vec(it1, it2);
        sq[i] = decompose(vec);
    }

    return sq;
}

// Sqaure root Decompostion at level1
void singleDecompose(vector<int> &arr)
{
    sqrtD sq1 = decompose(arr);
    vector<sqrtD> sq2(sq1.blocks);
    sq2 = doubleDecompose(sq1, arr);

    vector<vector<sqrtD>> sq3(sq1.blocks,
                               vector<sqrtD>(sq2[0].blocks));

    sq3 = tripleDecompose(sq1, sq2[0], arr);

    // ASSIGNMENT TO GLOBAL VARIABLES
    Sq1 = sq1;
    Sq2.resize(sq1.blocks);
    Sq2 = sq2;
    Sq3.resize(sq1.blocks,
              vector<sqrtD>(sq2[0].blocks));
    Sq3 = sq3;
}

```

```

}

// Function for query at level 3
int queryLevel3(int start,int end, int main_blk,
                int sub_main_blk, vector<int> &arr)
{
    int blk_sz= Sq3[0][0].blk_sz;

    // Element Indexing at level2 decompostion
    int nstart = start - main_blk *
        Sq1.blk_sz - sub_main_blk * Sq2[0].blk_sz;
    int nend = end - main_blk *
        Sq1.blk_sz - sub_main_blk * Sq2[0].blk_sz;

    // Block indexing at level3 decompostion
    int st_blk = nstart / blk_sz;
    int en_blk = nend / blk_sz;

    int answer =
        Sq3[main_blk][sub_main_blk].rdata[st_blk][en_blk];

    // If start and end point dont lie in same block
    if(st_blk != en_blk)
    {
        int left = 0, en_idx = main_blk * Sq1.blk_sz +
                    sub_main_blk * Sq2[0].blk_sz +
                    (st_blk + 1) * blk_sz -1;

        for(int i = start; i <= en_idx; i++)
        {
            left += arr[i];
        }

        int right = 0, st_idx = main_blk * Sq1.blk_sz +
                        sub_main_blk * Sq2[0].blk_sz +
                        (en_blk) * blk_sz;

        for(int i = st_idx; i <= end; i++)
        {
            right += arr[i];
        }

        answer += left;
        answer += right;
    }
    else
    {
        for(int i = start; i <= end; i++)
    }
}

```

```

    {
        answer += arr[i];
    }
}

return answer;
}

// Function for splitting query to level two
int queryLevel2(int start, int end, int main_blk,
                vector<int> &arr)
{
    int blk_sz = Sq2[0].blk_sz;

    // Element Indexing at level1 decompostion
    int nstart = start - (main_blk * Sq1.blk_sz);
    int nend = end - (main_blk * Sq1.blk_sz);

    // Block indexing at level2 decompostion
    int st_blk = nstart / blk_sz;
    int en_blk = nend / blk_sz;

    // Interblock data level2 decompostion
    int answer = Sq2[main_blk].rdata[st_blk][en_blk];

    if(st_blk == en_blk)
    {
        answer += queryLevel3(start, end, main_blk,
                               st_blk, arr);
    }
    else
    {
        answer += queryLevel3(start, (main_blk *
                                      Sq1.blk_sz) + ((st_blk + 1) *
                                                      blk_sz) - 1, main_blk, st_blk, arr);

        answer += queryLevel3((main_blk * Sq1.blk_sz) +
                               (en_blk * blk_sz), end, main_blk, en_blk, arr);
    }
}

return answer;
}

// Function to return answer according to query
int Query(int start,int end,vector<int>& arr)
{
    int blk_sz = Sq1.blk_sz;
    int st_blk = start / blk_sz;
    int en_blk = end / blk_sz;
}

```

```

// Interblock data level1 decompostion
int answer = Sq1.rdata[st_blk][en_blk];

if(st_blk == en_blk)
{
    answer += queryLevel2(start, end, st_blk, arr);
}
else
{
    answer += queryLevel2(start, (st_blk + 1) *
                           blk_sz - 1, st_blk, arr);
    answer += queryLevel2(en_blk * blk_sz, end,
                           en_blk, arr);
}

// returning final answer
return answer;
}

// Driver code
int main()
{
    n1 = 16;

    vector<int> arr = {7, 2, 3, 0, 5, 10, 3, 12,
                       18, 1, 2, 3, 4, 5, 6, 7};

    singleDecompose(arr);

    int q = 5;
    pair<int, int> query[q] = {{6, 10}, {7, 12},
                                {4, 13}, {4, 11}, {12, 16}};

    for(int i = 0; i < q; i++)
    {
        int a = query[i].first, b = query[i].second;
        printf("%d\n", Query(a - 1, b - 1, arr));
    }

    return 0;
}

```

Output:

44
39
58

51

25

Time Complexity : $O(q * d * n^{1/(2^3)})$ $O(q * k)$ $O(q)$

Auxiliary Space : $O(k * n)$ $O(n)$

Note : This article is to only explain the method of decomposing the square root to further decomposition.

Improved By : [vaibhav2992](#)

Source

<https://www.geeksforgeeks.org/extended-mos-algorithm-o1-time-complexity/>

Chapter 57

Fibonacci Heap Set 1 (Introduction)

Fibonacci Heap Set 1 (Introduction) - GeeksforGeeks

Heaps are mainly used for implementing priority queue. We have discussed below heaps in previous posts.

[Binary Heap](#)

[Binomial Heap](#)

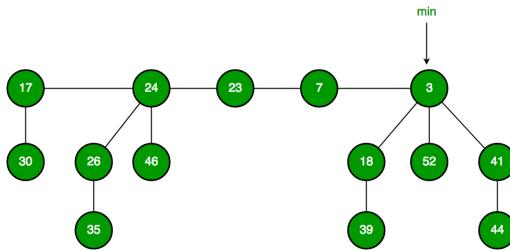
In terms of Time Complexity, Fibonacci Heap beats both Binary and Binomial Heaps.

Below are [amortized time complexities](#) of **Fibonacci Heap**.

- 1) Find Min: $\Theta(1)$ [Same as both Binary and Binomial]
- 2) Delete Min: $\Theta(\log n)$ [$\Theta(\log n)$ in both Binary and Binomial]
- 3) Insert: $\Theta(1)$ [$\Theta(\log n)$ in Binary and $\Theta(1)$ in Binomial]
- 4) Decrease-Key: $\Theta(1)$ [$\Theta(\log n)$ in both Binary and Binomial]
- 5) Merge: $\Theta(1)$ [$\Theta(m \log n)$ or $\Theta(m+n)$ in Binary and
 $\Theta(\log n)$ in Binomial]

Like [Binomial Heap](#), Fibonacci Heap is a collection of trees with min-heap or max-heap property. In Fibonacci Heap, trees can have any shape even all trees can be single nodes (This is unlike Binomial Heap where every tree has to be Binomial Tree).

Below is an example Fibonacci Heap taken from [here](#).



Fibonacci Heap maintains a pointer to minimum value (which is root of a tree). All tree roots are connected using circular doubly linked list, so all of them can be accessed using single ‘min’ pointer.

The main idea is to execute operations in “lazy” way. For example merge operation simply links two heaps, insert operation simply adds a new tree with single node. The operation extract minimum is the most complicated operation. It does delayed work of consolidating trees. This makes delete also complicated as delete first decreases key to minus infinite, then calls extract minimum.

Below are some interesting facts about Fibonacci Heap

1. The reduced time complexity of Decrease-Key has importance in Dijkstra and Prim algorithms. With Binary Heap, time complexity of these algorithms is $O(V\log V + E\log V)$. If Fibonacci Heap is used, then time complexity is improved to $O(V\log V + E)$
2. Although Fibonacci Heap looks promising time complexity wise, it has been found slow in practice as hidden constants are high (Source [Wiki](#)).
3. Fibonacci heap are mainly called so because Fibonacci numbers are used in the running time analysis. Also, every node in Fibonacci Heap has degree at most $O(\log n)$ and the size of a subtree rooted in a node of degree k is at least F_{k+2} , where F_k is the k th Fibonacci number.

We will soon be discussing Fibonacci Heap operations in detail.

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

Source

<https://www.geeksforgeeks.org/fibonacci-heap-set-1-introduction/>

Chapter 58

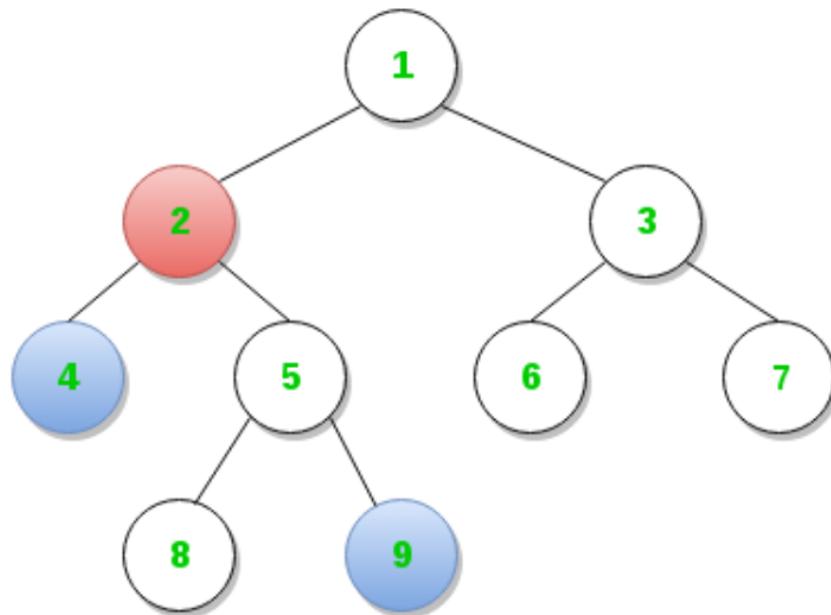
Find LCA in Binary Tree using RMQ

Find LCA in Binary Tree using RMQ - GeeksforGeeks

The article describes an approach to solving the problem of finding the LCA of two nodes in a tree by reducing it to a RMQ problem.

Lowest Common Ancestor (LCA) of two nodes u and v in a rooted tree T is defined as the node located farthest from the root that has both u and v as descendants.

For example, in below diagram, LCA of node 4 and node 9 is node 2.

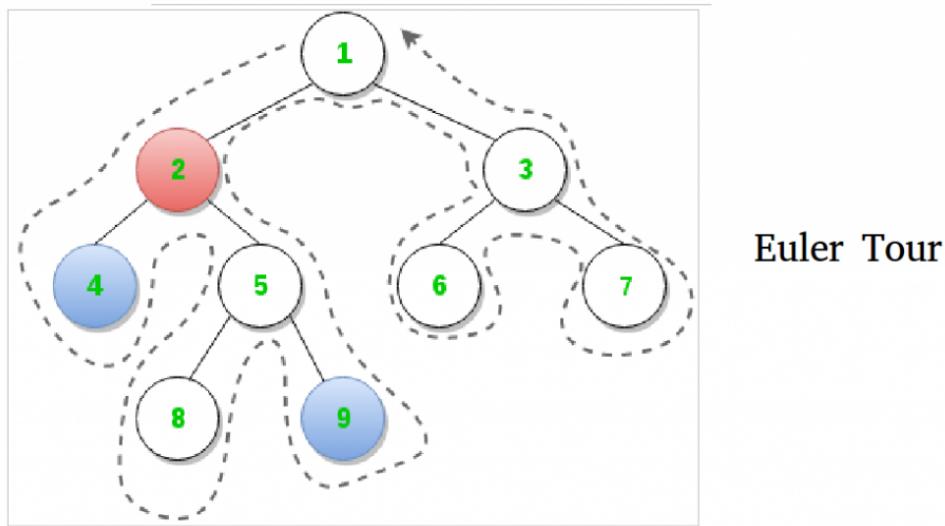


There can be many approaches to solve the LCA problem. The approaches differ in their time and space complexities. [Here](#) is a link to a couple of them (these do not involve reduction to RMQ).

Range Minimum Query (RMQ) is used on arrays to find the position of an element with the minimum value between two specified indices. Different approaches for solving RMQ have been discussed [here](#) and [here](#). In this article, Segment Tree based approach is discussed. With segment tree, preprocessing time is $O(n)$ and time to for range minimum query is $O(\log n)$. The extra space required is $O(n)$ to store the segment tree.

Reduction of LCA to RMQ:

The idea is to traverse the tree starting from root by an Euler tour (traversal without lifting pencil), which is a DFS-type traversal with preorder traversal characteristics.



An euler tour of the tree starting from node 1 will yield:

1	2	4	2	5	8	5	9	5	2	1	3	6	3	7	3	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

The corresponding levels for every node in Euler tour:

0	1	2	1	2	3	2	3	2	1	0	1	2	1	2	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Observation: The LCA of nodes 4 and 9 is node 2, which happens to be the node closest to the root amongst all those encountered between the visits of 4 and 9 during a DFS of T. This observation is the key to the reduction. Let's rephrase: Our node is the node at the smallest level and the only node at that level amongst all the nodes that occur between consecutive occurrences (any) of u and v in the Euler tour of T.

We require three arrays for implementation:

1. Nodes visited in order of Euler tour of T
2. Level of each node visited in Euler tour of T

3. Index of the **first** occurrence of a node in Euler tour of T (since any occurrence would be good, let's track the first one)

The first occurrences corresponding to every node in Euler tour of T:

Node	1	2	3	4	5	6	7	8	9
First Occurrence	0	1	11	2	4	12	14	5	7

Algorithm:

1. Do a Euler tour on the tree, and fill the euler, level and first occurrence arrays.
2. Using the first occurrence array, get the indices corresponding to the two nodes which will be the corners of the range in the level array that is fed to the RMQ algorithm for the minimum value.
3. Once the algorithm return the index of the minimum level in the range, we use it to determine the LCA using Euler tour array.

Below is the implementation of above algorithm.

C++

```
/* C++ Program to find LCA of u and v by reducing the problem to RMQ */
#include<bits/stdc++.h>
#define V 9           // number of nodes in input tree

int euler[2*V - 1];    // For Euler tour sequence
int level[2*V - 1];    // Level of nodes in tour sequence
int firstOccurrence[V+1]; // First occurrences of nodes in tour
int ind;                // Variable to fill-in euler and level arrays

// A Binary Tree node
struct Node
{
    int key;
    struct Node *left, *right;
};

// Utility function creates a new binary tree node with given key
Node * newNode(int k)
{
    Node *temp = new Node;
    temp->key = k;
    temp->left = temp->right = NULL;
    return temp;
}
```

```

// log base 2 of x
int Log2(int x)
{
    int ans = 0 ;
    while (x>>=1) ans++;
    return ans ;
}

/* A recursive function to get the minimum value in a given range
   of array indexes. The following are parameters for this function.

   st    --> Pointer to segment tree
   index --> Index of current node in the segment tree. Initially
              0 is passed as root is always at index 0
   ss & se  --> Starting and ending indexes of the segment represented
                  by current node, i.e., st[index]
   qs & qe  --> Starting and ending indexes of query range */
int RMQUtil(int index, int ss, int se, int qs, int qe, int *st)
{
    // If segment of this node is a part of given range, then return
    // the min of the segment
    if (qs <= ss && qe >= se)
        return st[index];

    // If segment of this node is outside the given range
    else if (se < qs || ss > qe)
        return -1;

    // If a part of this segment overlaps with the given range
    int mid = (ss + se)/2;

    int q1 = RMQUtil(2*index+1, ss, mid, qs, qe, st);
    int q2 = RMQUtil(2*index+2, mid+1, se, qs, qe, st);

    if (q1===-1) return q2;

    else if (q2===-1) return q1;

    return (level[q1] < level[q2]) ? q1 : q2;
}

// Return minimum of elements in range from index qs (query start) to
// qe (query end). It mainly uses RMQUtil()
int RMQ(int *st, int n, int qs, int qe)
{
    // Check for erroneous input values
    if (qs < 0 || qe > n-1 || qs > qe)

```

```

{
    printf("Invalid Input");
    return -1;
}

return RMQUtil(0, 0, n-1, qs, qe, st);
}

// A recursive function that constructs Segment Tree for array[ss..se].
// si is index of current node in segment tree st
void constructSTUtil(int si, int ss, int se, int arr[], int *st)
{
    // If there is one element in array, store it in current node of
    // segment tree and return
    if (ss == se) st[si] = ss;

    else
    {
        // If there are more than one elements, then recur for left and
        // right subtrees and store the minimum of two values in this node
        int mid = (ss + se)/2;
        constructSTUtil(si*2+1, ss, mid, arr, st);
        constructSTUtil(si*2+2, mid+1, se, arr, st);

        if (arr[st[2*si+1]] < arr[st[2*si+2]])
            st[si] = st[2*si+1];
        else
            st[si] = st[2*si+2];
    }
}

/* Function to construct segment tree from given array. This function
   allocates memory for segment tree and calls constructSTUtil() to
   fill the allocated memory */
int *constructST(int arr[], int n)
{
    // Allocate memory for segment tree

    // Height of segment tree
    int x = Log2(n)+1;

    // Maximum size of segment tree
    int max_size = 2*(1<<x) - 1; // 2*pow(2,x) -1

    int *st = new int[max_size];

    // Fill the allocated memory st
    constructSTUtil(0, 0, n-1, arr, st);
}

```

```

// Return the constructed segment tree
return st;
}

// Recursive version of the Euler tour of T
void eulerTour(Node *root, int l)
{
    /* if the passed node exists */
    if (root)
    {
        euler[ind] = root->key; // insert in euler array
        level[ind] = l;           // insert l in level array
        ind++;                  // increment index

        /* if unvisited, mark first occurrence */
        if (firstOccurrence[root->key] == -1)
            firstOccurrence[root->key] = ind-1;

        /* tour left subtree if exists, and remark euler
           and level arrays for parent on return */
        if (root->left)
        {
            eulerTour(root->left, l+1);
            euler[ind]=root->key;
            level[ind] = l;
            ind++;
        }

        /* tour right subtree if exists, and remark euler
           and level arrays for parent on return */
        if (root->right)
        {
            eulerTour(root->right, l+1);
            euler[ind]=root->key;
            level[ind] = l;
            ind++;
        }
    }
}

// Returns LCA of nodes n1, n2 (assuming they are
// present in the tree)
int findLCA(Node *root, int u, int v)
{
    /* Mark all nodes unvisited. Note that the size of
       firstOccurrence is 1 as node values which vary from
       1 to 9 are used as indexes */

```

```

memset(firstOccurrence, -1, sizeof(int)*(V+1));

/* To start filling euler and level arrays from index 0 */
ind = 0;

/* Start Euler tour with root node on level 0 */
eulerTour(root, 0);

/* construct segment tree on level array */
int *st = constructST(level, 2*V-1);

/* If v before u in Euler tour. For RMQ to work, first
parameter 'u' must be smaller than second 'v' */
if (firstOccurrence[u]>firstOccurrence[v])
    std::swap(u, v);

// Starting and ending indexes of query range
int qs = firstOccurrence[u];
int qe = firstOccurrence[v];

// query for index of LCA in tour
int index = RMQ(st, 2*V-1, qs, qe);

/* return LCA node */
return euler[index];
}

// Driver program to test above functions
int main()
{
    // Let us create the Binary Tree as shown in the diagram.
    Node * root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->left = newNode(6);
    root->right->right = newNode(7);
    root->left->right->left = newNode(8);
    root->left->right->right = newNode(9);

    int u = 4, v = 9;
    printf("The LCA of node %d and node %d is node %d.\n",
           u, v, findLCA(root, u, v));
    return 0;
}

```

Java

```
// Java program to find LCA of u and v by reducing problem to RMQ

import java.util.*;

// A binary tree node
class Node
{
    Node left, right;
    int data;

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

class St_class
{
    int st;
    int stt[] = new int[10000];
}

class BinaryTree
{
    Node root;

    int v = 9; // v is the highest value of node in our tree
    int euler[] = new int[2 * v - 1]; // for euler tour sequence
    int level[] = new int[2 * v - 1]; // level of nodes in tour sequence
    int f_occur[] = new int[2 * v - 1]; // to store 1st occurrence of nodes
    int fill; // variable to fill euler and level arrays
    St_class sc = new St_class();

    // log base 2 of x
    int Log2(int x)
    {
        int ans = 0;
        int y = x >>= 1;
        while (y-- != 0)
            ans++;
        return ans;
    }

    int swap(int a, int b)
    {
        return a;
    }
}
```

```

/* A recursive function to get the minimum value in a given range
of array indexes. The following are parameters for this function.

st    --> Pointer to segment tree
index --> Index of current node in the segment tree. Initially
0 is passed as root is always at index 0
ss & se --> Starting and ending indexes of the segment represented
by current node, i.e., st[index]
qs & qe --> Starting and ending indexes of query range */
int RMQUtil(int index, int ss, int se, int qs, int qe, St_class st)
{
    // If segment of this node is a part of given range, then return
    // the min of the segment
    if (qs <= ss && qe >= se)
        return st.sst[index];

    // If segment of this node is outside the given range
    else if (se < qs || ss > qe)
        return -1;

    // If a part of this segment overlaps with the given range
    int mid = (ss + se) / 2;

    int q1 = RMQUtil(2 * index + 1, ss, mid, qs, qe, st);
    int q2 = RMQUtil(2 * index + 2, mid + 1, se, qs, qe, st);

    if (q1 == -1)
        return q2;
    else if (q2 == -1)
        return q1;

    return (level[q1] < level[q2]) ? q1 : q2;
}

// Return minimum of elements in range from index qs (quey start) to
// qe (query end). It mainly uses RMQUtil()
int RMQ(St_class st, int n, int qs, int qe)
{
    // Check for erroneous input values
    if (qs < 0 || qe > n - 1 || qs > qe)
    {
        System.out.println("Invalid input");
        return -1;
    }

    return RMQUtil(0, 0, n - 1, qs, qe, st);
}

```

```

// A recursive function that constructs Segment Tree for array[ss..se].
// si is index of current node in segment tree st
void constructSTUtil(int si, int ss, int se, int arr[], St_class st)
{
    // If there is one element in array, store it in current node of
    // segment tree and return
    if (ss == se)
        st.stt[si] = ss;
    else
    {
        // If there are more than one elements, then recur for left and
        // right subtrees and store the minimum of two values in this node
        int mid = (ss + se) / 2;
        constructSTUtil(si * 2 + 1, ss, mid, arr, st);
        constructSTUtil(si * 2 + 2, mid + 1, se, arr, st);

        if (arr[st.stt[2 * si + 1]] < arr[st.stt[2 * si + 2]])
            st.stt[si] = st.stt[2 * si + 1];
        else
            st.stt[si] = st.stt[2 * si + 2];
    }
}

/* Function to construct segment tree from given array. This function
allocates memory for segment tree and calls constructSTUtil() to
fill the allocated memory */
int constructST(int arr[], int n)
{
    // Allocate memory for segment tree
    // Height of segment tree
    int x = Log2(n) + 1;

    // Maximum size of segment tree
    int max_size = 2 * (1 << x) - 1; // 2*pow(2,x) -1

    sc.stt = new int[max_size];

    // Fill the allocated memory st
    constructSTUtil(0, 0, n - 1, arr, sc);

    // Return the constructed segment tree
    return sc.st;
}

// Recursive version of the Euler tour of T
void eulerTour(Node node, int l)
{
    /* if the passed node exists */

```

```

if (node != null)
{
    euler[fill] = node.data; // insert in euler array
    level[fill] = 1;         // insert 1 in level array
    fill++;                 // increment index

    /* if unvisited, mark first occurrence */
    if (f_occur[node.data] == -1)
        f_occur[node.data] = fill - 1;

    /* tour left subtree if exists, and remark euler
       and level arrays for parent on return */
    if (node.left != null)
    {
        eulerTour(node.left, l + 1);
        euler[fill] = node.data;
        level[fill] = l;
        fill++;
    }

    /* tour right subtree if exists, and remark euler
       and level arrays for parent on return */
    if (node.right != null)
    {
        eulerTour(node.right, l + 1);
        euler[fill] = node.data;
        level[fill] = l;
        fill++;
    }
}

// returns LCA of node n1 and n2 assuming they are present in tree
int findLCA(Node node, int u, int v)
{
    /* Mark all nodes unvisited. Note that the size of
       firstOccurrence is 1 as node values which vary from
       1 to 9 are used as indexes */
    Arrays.fill(f_occur, -1);

    /* To start filling euler and level arrays from index 0 */
    fill = 0;

    /* Start Euler tour with root node on level 0 */
    eulerTour(root, 0);

    /* construct segment tree on level array */
    sc.st = constructST(level, 2 * v - 1);
}

```

```
/* If v before u in Euler tour. For RMQ to work, first
parameter 'u' must be smaller than second 'v' */
if (f_occur[u] > f_occur[v])
    u = swap(u, u = v);

// Starting and ending indexes of query range
int qs = f_occur[u];
intqe = f_occur[v];

// query for index of LCA in tour
int index = RMQ(sc, 2 * v - 1, qs, qe);

/* return LCA node */
return euler[index];

}

// Driver program to test above functions
public static void main(String args[])
{
    BinaryTree tree = new BinaryTree();

    // Let us create the Binary Tree as shown in the diagram.
    tree.root = new Node(1);
    tree.root.left = new Node(2);
    tree.root.right = new Node(3);
    tree.root.left.left = new Node(4);
    tree.root.left.right = new Node(5);
    tree.root.right.left = new Node(6);
    tree.root.right.right = new Node(7);
    tree.root.left.right.left = new Node(8);
    tree.root.left.right.right = new Node(9);

    int u = 4, v = 9;
    System.out.println("The LCA of node " + u + " and " + v + " is "
        + tree.findLCA(tree.root, u, v));
}

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

Output:

The LCA of node 4 and node 9 is node 2.

Note:

1. We assume that the nodes queried are present in the tree.
2. We also assumed that if there are V nodes in tree, then keys (or data) of these nodes are in range from 1 to V .

Time complexity:

1. Euler tour: Number of nodes is V . For a tree, $E = V-1$. Euler tour (DFS) will take $O(V+E)$ which is $O(2*V)$ which can be written as $O(V)$.
2. Segment Tree construction : $O(n)$ where $n = V + E = 2*V - 1$.
3. Range Minimum query: $O(\log(n))$

Overall this method takes $O(n)$ time for preprocessing, but takes $O(\log n)$ time for query. Therefore, it can be useful when we have a single tree on which we want to perform large number of LCA queries (Note that LCA is useful for finding shortest path between two nodes of Binary Tree)

Auxiliary Space:

1. Euler tour array: $O(n)$ where $n = 2*V - 1$
2. Node Levels array: $O(n)$
3. First Occurrences array: $O(V)$
4. Segment Tree: $O(n)$

Overall: $O(n)$

Another observation is that the adjacent elements in level array differ by 1. This can be used to convert a RMQ problem to a LCA problem.

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

Source

<https://www.geeksforgeeks.org/find-lca-in-binary-tree-using-rmq/>

Chapter 59

Find all possible interpretations of an array of digits

Find all possible interpretations of an array of digits - GeeksforGeeks

Consider a coding system for alphabets to integers where 'a' is represented as 1, 'b' as 2, .. 'z' as 26. Given an array of digits (1 to 9) as input, write a function that prints all valid interpretations of input array.

Examples

```
Input: {1, 1}
Output: ("aa", 'k")
[2 interpretations: aa(1, 1), k(11)]
```

```
Input: {1, 2, 1}
Output: ("aba", "au", "la")
[3 interpretations: aba(1,2,1), au(1,21), la(12,1)]
```

```
Input: {9, 1, 8}
Output: {"iah", "ir"}
[2 interpretations: iah(9,1,8), ir(9,18)]
```

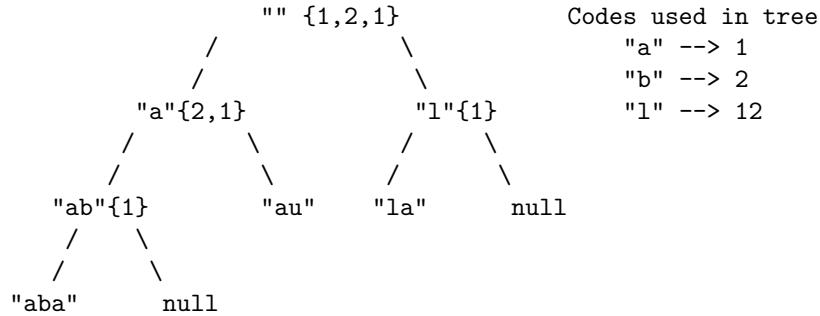
Please note we cannot change order of array. That means {1,2,1} cannot become {2,1,1}. On first look it looks like a problem of permutation/combinations. But on closer look you will figure out that this is an interesting tree problem.

The idea here is string can take at-most two paths:

1. Process single digit
2. Process two digits

That means we can use binary tree here. Processing with single digit will be left child and two digits will be right child. If value two digits is greater than 26 then our right child will be null as we don't have alphabet for greater than 26.

Let's understand with an example . Array a = {1,2,1}. Below diagram shows that how our tree grows.



Braces {} contain array still pending for processing. Note that with every level, our array size decreases. If you will see carefully, it is not hard to find that tree height is always n (array size)

How to print all strings (interpretations)? Output strings are leaf node of tree. i.e for {1,2,1}, output is {aba au la}.

We can conclude that there are mainly two steps to print all interpretations of given integer array.

Step 1: Create a binary tree with all possible interpretations in leaf nodes.

Step 2: Print all leaf nodes from the binary tree created in step 1.

Following is Java implementation of above algorithm.

```

// A Java program to print all interpretations of an integer array
import java.util.Arrays;

// A Binary Tree node
class Node {

    String dataString;
    Node left;
    Node right;

    Node(String dataString) {
        this.dataString = dataString;
        //Be default left and right child are null.
    }

    public String getDataString() {
        return dataString;
    }
}
  
```

```
public class arrayToAllInterpretations {

    // Method to create a binary tree which stores all interpretations
    // of arr[] in lead nodes
    public static Node createTree(int data, String pString, int[] arr) {

        // Invalid input as alphabets maps from 1 to 26
        if (data > 26)
            return null;

        // Parent String + String for this node
        String dataToStr = pString + alphabet[data];

        Node root = new Node(dataToStr);

        // if arr.length is 0 means we are done
        if (arr.length != 0) {
            data = arr[0];

            // new array will be from index 1 to end as we are consuming
            // first index with this node
            int newArr[] = Arrays.copyOfRange(arr, 1, arr.length);

            // left child
            root.left = createTree(data, dataToStr, newArr);

            // right child will be null if size of array is 0 or 1
            if (arr.length > 1) {

                data = arr[0] * 10 + arr[1];

                // new array will be from index 2 to end as we
                // are consuming first two index with this node
                newArr = Arrays.copyOfRange(arr, 2, arr.length);

                root.right = createTree(data, dataToStr, newArr);
            }
        }
        return root;
    }

    // To print out leaf nodes
    public static void printleaf(Node root) {
        if (root == null)
            return;

        if (root.left == null && root.right == null)
            System.out.print(root.getDataString() + " ");
    }
}
```

```
printleaf(root.left);
printleaf(root.right);
}

// The main function that prints all interpretations of array
static void printAllInterpretations(int[] arr) {

    // Step 1: Create Tree
    Node root = createTree(0, "", arr);

    // Step 2: Print Leaf nodes
    printleaf(root);

    System.out.println(); // Print new line
}

// For simplicity I am taking it as string array. Char Array will save space
private static final String[] alphabet = {"", "a", "b", "c", "d", "e",
    "f", "g", "h", "i", "j", "k", "l", "m", "n", "o", "p", "q", "r",
    "s", "t", "u", "v", "w", "x", "v", "z"};

// Driver method to test above methods
public static void main(String args[]) {

    // aacd(1,1,3,4) amd(1,13,4) kcd(11,3,4)
    // Note : 1,1,34 is not valid as we don't have values corresponding
    // to 34 in alphabet
    int[] arr = {1, 1, 3, 4};
    printAllInterpretations(arr);

    // aaa(1,1,1) ak(1,11) ka(11,1)
    int[] arr2 = {1, 1, 1};
    printAllInterpretations(arr2);

    // bf(2,6) z(26)
    int[] arr3 = {2, 6};
    printAllInterpretations(arr3);

    // ab(1,2), l(12)
    int[] arr4 = {1, 2};
    printAllInterpretations(arr4);

    // a(1,0} j(10)
    int[] arr5 = {1, 0};
    printAllInterpretations(arr5);

    // "" empty string output as array is empty
```

```
int[] arr6 = {};
printAllInterpretations(arr6);

// abba abu ava lba lu
int[] arr7 = {1, 2, 2, 1};
printAllInterpretations(arr7);
}

}
```

Output:

```
aacd  amd  kcd
aaa  ak  ka
bf  z
ab  l
a  j

abba  abu  ava  lba  lu
```

Exercise:

1. What is the time complexity of this solution? [Hint : size of tree + finding leaf nodes]
2. Can we store leaf nodes at the time of tree creation so that no need to run loop again for leaf node fetching?
3. How can we reduce extra space?

This article is compiled by [Varun 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-all-possible-interpretations/>

Chapter 60

Find last unique URL from long list of URLs in single traversal

Find last unique URL from long list of URLs in single traversal - GeeksforGeeks

Given a very long list of URLs, find out last unique URL. Only one traversal of all URLs is allowed.

Examples:

Input:

```
https://www.geeksforgeeks.org
http://quiz.geeksforgeeks.org
http://qa.geeksforgeeks.org
https://practice.geeksforgeeks.org
https://ide.geeksforgeeks.org
http://www.contribute.geeksforgeeks.org
http://quiz.geeksforgeeks.org
https://practice.geeksforgeeks.org
https://ide.geeksforgeeks.org
http://quiz.geeksforgeeks.org
http://qa.geeksforgeeks.org
https://practice.geeksforgeeks.org
```

Output:

```
http://www.contribute.geeksforgeeks.org
```

We can solve this problem in one traversal by using [Trie](#) with a Doubly Linked List (We can insert and delete in O(1) time). The idea is to insert all URLs into the Trie one by one and check if it is duplicate or not. To know if we have previously encountered the URL, we need to mark the last node of every URL as leaf node. If we encounter a URL for the first time, we insert it into the doubly Linked list and maintain a pointer to that node in linked list in

leaf node of the trie. If we encounter a URL that is already in the trie and has pointer to the url in the linked list, we delete the node from the linked list and set its pointer in the trie to null. After all URLs are processed, linked list will only contain the URLs that are distinct and the node at the beginning of the linked list will be last unique URL.

```
// C++ program to print distinct URLs using Trie
// and Doubly Linked List
#include <bits/stdc++.h>
using namespace std;

// Alphabet size (# of symbols)
const int ALPHABET_SIZE = 256;

// A linked list node
struct DLLNode
{
    string data;
    DLLNode* next, * prev;
};

// trie node
struct TrieNode
{
    TrieNode* children[ALPHABET_SIZE];

    // isLeaf is true if the node represents
    // end of a word
    bool isLeaf;

    DLLNode* LLptr;
};

/* 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(DLLNode*& head_ref, string new_data)
{
    DLLNode* new_node = new DLLNode;

    // put in the data
    new_node->data = new_data;

    // Make next of new node as head and previous
    // as NULL
    new_node->next = (head_ref);
    new_node->prev = NULL;

    // 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(DLLNode*& head_ref, DLLNode* 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
    delete(del);
    return;
}

// Returns new trie node (initialized to NULLs)
TrieNode* getNewTrieNode(void)
{
    TrieNode* pNode = new TrieNode;

    if (pNode)
    {
        pNode->isLeaf = false;

        for (int i = 0; i < ALPHABET_SIZE; i++)
            pNode->children[i] = NULL;

        pNode->LLptr = NULL;
    }
}

```

```
    }

    return pNode;
}

// If not present, inserts key into trie
// If the key is prefix of trie node, just marks leaf node
void insert(TrieNode* root, string key, DLLNode*& head)
{
    int index;
    TrieNode* pCrawl = root;

    for (int level = 0; level < key.length(); level++)
    {
        index = int(key[level]);
        if (!pCrawl->children[index])
            pCrawl->children[index] = getNewTrieNode();

        pCrawl = pCrawl->children[index];
    }

    if (pCrawl->isLeaf)
    {
        // cout << "Duplicate Found " << key << endl;
        // delete from linked list
        if (pCrawl->LLptr)
            deleteNode(head, pCrawl->LLptr);
        pCrawl->LLptr = NULL;
    }
    else
    {
        // mark last node as leaf
        pCrawl->isLeaf = true;

        // insert to linked list
        push(head, key);
        pCrawl->LLptr = head;
    }
}

// Driver function
int main()
{
    string urls[] = {
        "https://www.geeksforgeeks.org",
        "http://www.contribute.geeksforgeeks.org",
        "http://quiz.geeksforgeeks.org",
        "http://qa.geeksforgeeks.org",
```

```
"https://practice.geeksforgeeks.org",
"https://ide.geeksforgeeks.org",
"http://quiz.geeksforgeeks.org",
"https://practice.geeksforgeeks.org",
"https://ide.geeksforgeeks.org",
"http://quiz.geeksforgeeks.org",
"http://qa.geeksforgeeks.org",
"https://practice.geeksforgeeks.org"
};

TrieNode* root = getNewTrieNode();

// Start with the empty list
DLLNode* head = NULL;
int n = sizeof(urls)/sizeof(urls[0]);

// Construct Trie from given URLs
for (int i = 0; i < n; i++)
    insert(root, urls[i], head);

// head of linked list will point to last
// distinct URL
cout << head->data << endl;

return 0;
}
```

Output:

<http://www.contribute.geeksforgeeks.org>

Source

<https://www.geeksforgeeks.org/find-last-unique-url-long-list-urls-single-traversal/>

Chapter 61

Find maximum XOR of given integer in a stream of integers

Find maximum XOR of given integer in a stream of integers - GeeksforGeeks

You are given a number of queries **Q** and each query will be of the following types:

1. **Query 1** : add(x) This means add x into your data structure.
2. **Query 2** : maxXOR(y) This means print the maximum possible XOR of y with all the elements already stored in the data structure.

$1 \leq x, y \leq 10^9$

$1 \leq 10^5 \leq Q$

The data structure begins with only a 0 in it.

Example:

Input: (1 10), (1 13), (2 10), (1 9), (1 5), (2 6)

Output: 7 15

Add 10 and 13 to stream.

Find maximum XOR with 10, which is 7

Insert 9 and 5

Find maximum XOR with 6 which is 15.

A good way to solve this problem is to use a Trie. A **prerequisite** for this post is [Trie Insert and Search](#).

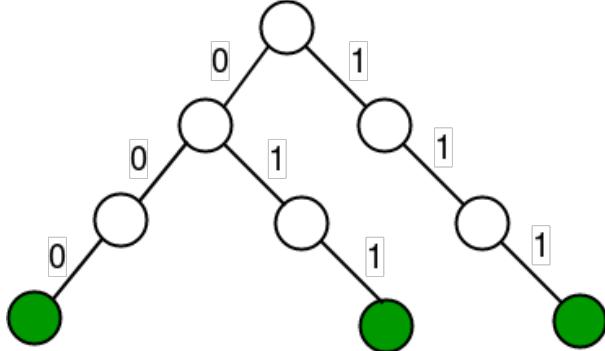
Each Trie Node will look following:

```
struct TrieNode
{
    // We use binary and hence we
    // need only 2 children
    TrieNode* Children[2];
    bool isLeaf;
};
```

Another thing to handle is that we have to pad the binary equivalent of each input number by a suitable number of zeros to the left before storing them. The maximum possible value of x or y is 10^9 and hence 32 bits will be sufficient.

So how does this work?

Assume we have to insert 3 and 7 into Trie. The Trie starts out with 0 and after these three insertions can be visualized like this:

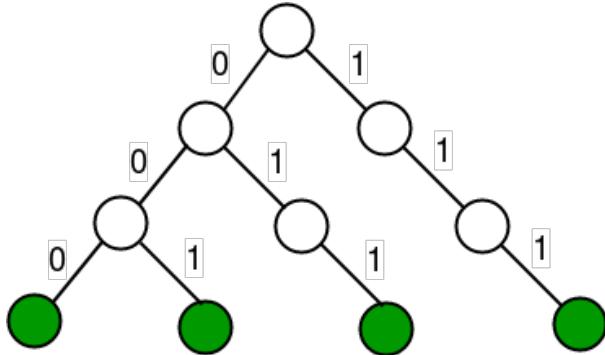


For simplification, the padding has been done to store each number using 3 bits. Note that in binary:

3 is 011

7 is 111

Now if we have to insert 1 into our Trie, we can note that 1 is 001 and we already have path for 00. So we make a new node for the last set bit and after connecting, we get this:

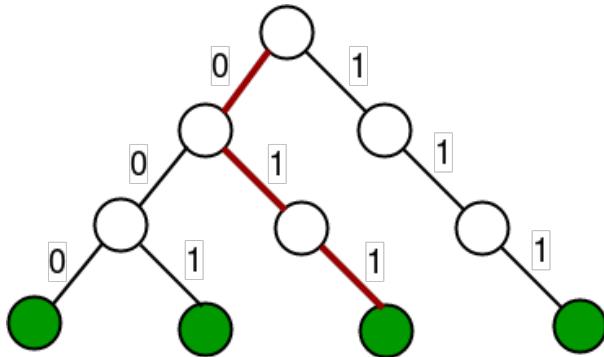


Now if we have to take XOR with 5 which is 101, we note that for the leftmost bit (position 2), we can choose a 0 starting at the root and thus we go to the left. This is the position 2 and we add 2^2 to the answer.

For position 1, we have a 0 in 5 and we see that we can choose a 1 from our current node.

Thus we go right and add 2^1 to the answer.

For position 0, we have a 1 in 5 and we see that we cannot choose a 0 from our current node, thus we go right.



The path taken for 5 is shown above. The answer is thus $2^2 + 2^1 = 6$.

```

// C++ program to find maximum XOR in
// a stream of integers
#include<bits/stdc++.h>
using namespace std;

struct TrieNode
{
    TrieNode* children[2];
    bool isLeaf;
};

// This checks if the ith position in
// binary of N is a 1 or a 0
bool check(int N, int i)
{
    return (bool)(N & (1<<i));
}

// Create a new Trie node
TrieNode* newNode()
{
    TrieNode* temp = new TrieNode;
    temp->isLeaf = false;
    temp->children[0] = NULL;
    temp->children[1] = NULL;
    return temp;
}

// Inserts x into the Trie
void insert(TrieNode* root, int x)
{

```

```
TrieNode* Crawler = root;

// padding upto 32 bits
for (int i = 31; i >= 0; i--)
{
    int f = check(x, i);
    if (!Crawler->children[f])
        Crawler->children[f] = newNode();
    Crawler = Crawler->children[f];
}
Crawler->isLeaf = true;
}

// Finds maximum XOR of x with stream of
// elements so far.
int query2(TrieNode *root, int x)
{
    TrieNode* Crawler = root;

    // Do XOR from root to a leaf path
    int ans = 0;
    for (int i = 31; i >= 0; i--)
    {
        // Find i-th bit in x
        int f = check(x, i);

        // Move to the child whose XOR with f
        // is 1.
        if ((Crawler->children[f ^ 1]))
        {
            ans = ans + (1 << i); // update answer
            Crawler = Crawler->children[f ^ 1];
        }

        // If child with XOR 1 doesn't exist
        else
            Crawler = Crawler->children[f];
    }

    return ans;
}

// Process x (Add x to the stream)
void query1(TrieNode *root, int x)
{
    insert(root, x);
}
```

```
// Driver code
int main()
{
    TrieNode* root = newNode();
    query1(root, 10);
    query1(root, 13);
    cout << query2(root, 10) << endl;
    query1(root, 9);
    query1(root, 5);
    cout << query2(root, 6) << endl;
    return 0;
}
```

7

15

The space taken by the Trie is $O(n * \log(n))$. Each query of type 1 takes $O(\log(n))$ time. Each query of type 2 takes $O(\log(n))$ time too. Here n is the largest query number.

Follow up problem: What if we are given three queries instead of two?

- 1) add(x) This means add x into your data structure (duplicates are allowed).
 - 2) maxXOR(y) This means print the maximum possible XOR of y with all the elements already stored in the data structure.
 - 3) remove(z) This means remove one instance of z from the data structure.
- What changes in the Trie solution can achieve this?

Source

<https://www.geeksforgeeks.org/find-maximum-xor-given-integer-stream-integers/>

Chapter 62

Find pair of rows in a binary matrix that has maximum bit difference

Find pair of rows in a binary matrix that has maximum bit difference - GeeksforGeeks

Given a Binary Matrix. The task is to find the pair of row in the Binary matrix that has maximum bit difference

Examples:

```
Input: mat[] [] = {{1, 1, 1, 1},  
                   {1, 1, 0, 1},  
                   {0, 0, 0, 0}};  
Output : (1, 3)  
Bit difference between row numbers 1 and 3  
is maximum with value 4. Bit difference  
between 1 and 2 is 1 and between 2 and 3  
is 3.  
  
Input: mat[] [] = {{1 ,1 ,1 ,1 }  
                   {1 ,0, 1 ,1 }  
                   {0 ,1 ,0 ,0 }  
                   {0, 0 ,0 ,0 }}  
Output : (2, 3)  
Bit difference between rows 2 and 3 is  
maximum which is 4.  
  
Input: mat[] [] = {{1 ,0 ,1 ,1 }  
                   {1 ,1 ,1 ,1 }  
                   {0 ,1 ,0 ,1 }}
```

```
{1, 0 ,0 ,0 }
Output : (1, 3) or (2 ,4 ) or (3 ,4 )
They all are having maximum bit difference
that is 3
```

Simple solution of this problem is that pick each row of binary matrix one -by -one and compute maximum bit difference with rest of the rows of matrix .at last return rows those have maximum bit difference .

An **Efficient solution** using [Trie Data Structure](#). Below is algorithm.

- 1). Create an empty Trie. Every node of Trie contains two children for 0 and 1 bits.
- 2). Insert First Row of Binary matrix into Trie
- 3). Traverse rest of the rows of given Binary Matrix
 - a). For Each Row First we search maximum bit difference with rows that we insert before that in Trie and count bit difference
 - b). For every search we update maximum bit_diff count if needed else not store pair of index that have maximum bit difference
 - c). At Last Print Pair

```
// C++ program to Find Pair of row in Binary matrix
// that has maximum Bit difference
#include<bits/stdc++.h>
using namespace std;

// Maximum size of matrix
const int MAX = 100;

struct TrieNode
{
    int leaf; //store index of visited row
    struct TrieNode *Child[2];
};

// Utility function to create a new Trie node
TrieNode * getNode()
{
    TrieNode * newNode = new TrieNode;
    newNode->leaf = 0;
    newNode->Child[0] = newNode->Child[1] = NULL;
    return newNode;
}
```

```
// utility function insert new row in trie
void insert(TrieNode *root, int Mat[][][MAX], int n,
            int row_index)
{
    TrieNode * temp = root;

    for (int i=0; i<n; i++)
    {
        // Add a new Node into trie
        if(temp->Child[ Mat[row_index][i] ] == NULL)
            temp->Child[ Mat[row_index][i] ] = getNode();

        // move current node to point next node in trie
        temp = temp->Child[ Mat[row_index][i] ];
    }

    // store index of currently inserted row
    temp->leaf = row_index +1 ;
}

// utility function calculate maximum bit difference of
// current row with previous visited row of binary matrix
pair<int, int> maxBitDiffCount(TrieNode * root,
                                 int Mat[][][MAX], int n, int row_index)
{
    TrieNode * temp = root;
    int count = 0;

    // Find previous visited row of binary matrix
    // that has starting bit same as current row
    for (int i= 0 ; i < n ; i++)
    {
        // First look for same bit in trie
        if (temp->Child[ Mat[row_index][i] ] != NULL)
            temp = temp->Child[ Mat[row_index][i] ];

        // Else looking for opposite bit
        else if (temp->Child[1 - Mat[row_index][i]] != NULL)
        {
            temp = temp->Child[1- Mat[row_index][i]];
            count++;
        }
    }

    int leaf_index = temp->leaf;
    int count1 = 0 ;
    temp = root;
```

```
// Find previous visited row of binary matrix
// that has starting bit opposite to current row
for (int i= 0 ; i < n ; i++)
{
    // First looking for opposite bit
    if (temp->Child[ 1 - Mat[row_index][i] ] !=NULL)
    {
        temp = temp->Child[ 1- Mat[row_index][i] ];
        count1++;
    }

    // Else look for same bit in trie
    else if (temp->Child[ Mat[row_index][i] ] != NULL)
        temp = temp->Child[ Mat[row_index][i] ];
}

pair <int ,int> P = count1 > count ?
    make_pair(count1, temp->leaf):
    make_pair(count, leaf_index);

// return pair that contain both bit difference
// count and index of row with we get bit
// difference
return P;
}

// Returns maximum bit difference pair of row
void maxDiff( int mat[][][MAX], int n, int m)
{
    TrieNode * root = getNode();

    // Insert first matrix row in trie
    insert(root, mat, m, 0);

    int max_bit_diff = INT_MIN;
    pair <int ,int> P, temp ;

    // Traverse all rest row of binary matrix
    for (int i = 1 ; i < n; i++)
    {
        // compute bit difference with previous visited
        // rows of matrix
        temp = maxBitDiffCount(root, mat, m ,i);

        // update maximum bit difference
        if (max_bit_diff < temp.first )
        {
            max_bit_diff = temp.first;
        }
    }
}
```

```
P = make_pair( temp.second, i+1);
}

// insert current row value into Trie
insert(root, mat, m, i );
}

// print maximum bit difference pair in row
cout << "(" << P.first <<, "<< P.second << ")";
}

// Driver program
int main()
{
    int mat[][] [MAX] = {{0 ,1 ,0 ,1, 0 },
                        {1, 0, 1 ,1 ,0 },
                        {0 ,0 ,1 ,0, 1 }
    };
    maxDiff(mat, 3, 5) ;
    return 0;
}
```

Output:

1 , 3

Source

<https://www.geeksforgeeks.org/find-pair-rows-binary-matrix-maximum-bit-difference/>

Chapter 63

Find shortest unique prefix for every word in a given list Set 1 (Using Trie)

Find shortest unique prefix for every word in a given list Set 1 (Using Trie) - GeeksforGeeks

Given an array of words, find all shortest unique prefixes to represent each word in the given array. Assume that no word is prefix of another.

Examples:

```
Input: arr[] = {"zebra", "dog", "duck", "dove"}  
Output: dog, dov, du, z  
Explanation: dog => dog  
           dove = dov  
           duck = du  
           z     => zebra
```

```
Input: arr[] = {"geeksgeeks", "geeksquiz", "geeksforgeeks"};  
Output: geeks, geeksg, geeksq}
```

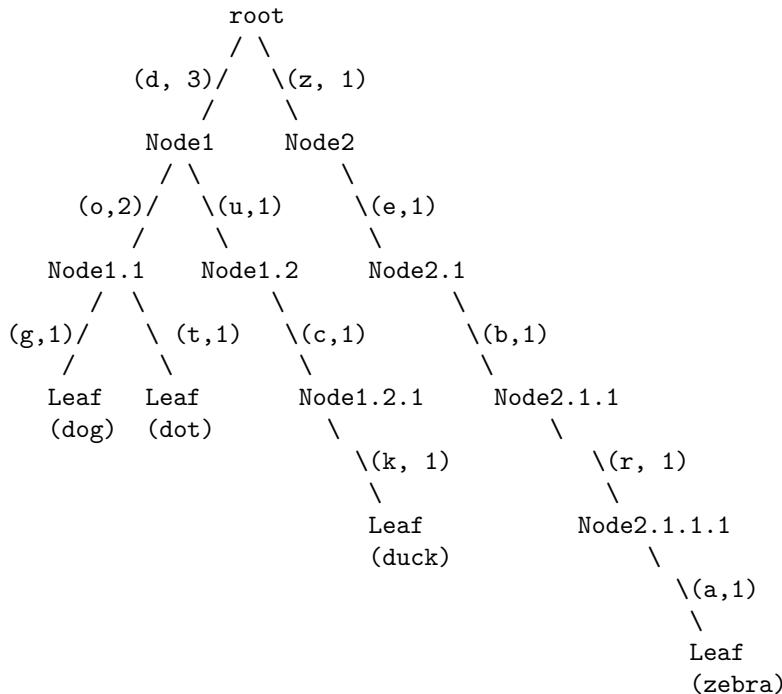
A **Simple Solution** is to consider every prefix of every word (starting from the shortest to largest), and if a prefix is not prefix of any other string, then print it.

An **Efficient Solution** is to use [Trie](#). The idea is to maintain a count in every node. Below are steps.

- 1) Construct a [Trie](#) of all words. Also maintain frequency of every node (Here frequency is number of times node is visited during insertion). Time complexity of this step is $O(N)$ where N is total number of characters in all words.

2) Now, for every word, we find the character nearest to the root with frequency as 1. The prefix of the word is path from root to this character. To do this, we can traverse Trie starting from root. For every node being traversed, we check its frequency. If frequency is one, we print all characters from root to this node and don't traverse down this node.

Time complexity if this step also is O(N) where N is total number of characters in all words.



Below is C++ implementation of above idea.

C++

```

// C++ program to print all prefixes that
// uniquely represent words.
#include<bits/stdc++.h>
using namespace std;

#define MAX 256

// Maximum length of an input word
#define MAX_WORD_LEN 500

// Trie Node.
struct trienode
{
    struct trienode *child[MAX];
}

```

```
        int freq; // To store frequency
    };

// Function to create a new trie node.
struct trienode *newTrienode(void)
{
    struct trienode *newNode = new trienode;
    newNode->freq = 1;
    for (int i = 0; i<MAX; i++)
        newNode->child[i] = NULL;
    return newNode;
}

// Method to insert a new string into Trie
void insert(struct trienode *root, string str)
{
    // Length of the URL
    int len = str.length();
    struct trienode *pCrawl = root;

    // Traversing over the length of given str.
    for (int level = 0; level<len; level++)
    {
        // Get index of child node from current character
        // in str.
        int index = str[level];

        // Create a new child if not exist already
        if (!pCrawl->child[index])
            pCrawl->child[index] = newTrienode();
        else
            (pCrawl->child[index]->freq)++;

        // Move to the child
        pCrawl = pCrawl->child[index];
    }
}

// This function prints unique prefix for every word stored
// in Trie. Prefixes one by one are stored in prefix[].
// 'ind' is current index of prefix[]
void findPrefixesUtil(struct trienode *root, char prefix[],
                      int ind)
{
    // Corner case
    if (root == NULL)
        return;
```

```
// Base case
if (root->freq == 1)
{
    prefix[ind] = '\0';
    cout << prefix << " ";
    return;
}

for (int i=0; i<MAX; i++)
{
    if (root->child[i] != NULL)
    {
        prefix[ind] = i;
        findPrefixesUtil(root->child[i], prefix, ind+1);
    }
}
}

// Function to print all prefixes that uniquely
// represent all words in arr[0..n-1]
void findPrefixes(string arr[], int n)
{
    // Construct a Trie of all words
    struct trienode *root = newTrienode();
    root->freq = 0;
    for (int i = 0; i<n; i++)
        insert(root, arr[i]);

    // Create an array to store all prefixes
    char prefix[MAX_WORD_LEN];

    // Print all prefixes using Trie Traversal
    findPrefixesUtil(root, prefix, 0);
}

// Driver function.
int main()
{
    string arr[] = {"zebra", "dog", "duck", "dove"};
    int n = sizeof(arr)/sizeof(arr[0]);
    findPrefixes(arr, n);

    return 0;
}
```

Java

```
// Java program to print all prefixes that
```

```
// uniquely represent words.
public class Unique_Prefix_Trie {

    static final int MAX = 256;

    // Maximum length of an input word
    static final int MAX_WORD_LEN = 500;

    // Trie Node.
    static class TrieNode
    {
        TrieNode[] child = new TrieNode[MAX];
        int freq; // To store frequency
        TrieNode() {
            freq = 1;
            for (int i = 0; i < MAX; i++)
                child[i] = null;
        }
    }
    static TrieNode root;

    // Method to insert a new string into Trie
    static void insert(String str)
    {
        // Length of the URL
        int len = str.length();
        TrieNode pCrawl = root;

        // Traversing over the length of given str.
        for (int level = 0; level < len; level++)
        {
            // Get index of child node from current character
            // in str.
            int index = str.charAt(level);

            // Create a new child if not exist already
            if (pCrawl.child[index] == null)
                pCrawl.child[index] = new TrieNode();
            else
                (pCrawl.child[index].freq)++;

            // Move to the child
            pCrawl = pCrawl.child[index];
        }
    }

    // This function prints unique prefix for every word stored
    // in Trie. Prefixes one by one are stored in prefix[].
}
```

```
// 'ind' is current index of prefix[]
static void findPrefixesUtil(TrieNode root, char[] prefix,
                             int ind)
{
    // Corner case
    if (root == null)
        return;

    // Base case
    if (root.freq == 1)
    {
        prefix[ind] = '\0';
        int i = 0;
        while(prefix[i] != '\0')
            System.out.print(prefix[i++]);
        System.out.print(" ");
        return;
    }

    for (int i=0; i<MAX; i++)
    {
        if (root.child[i] != null)
        {
            prefix[ind] = (char) i;
            findPrefixesUtil(root.child[i], prefix, ind+1);
        }
    }
}

// Function to print all prefixes that uniquely
// represent all words in arr[0..n-1]
static void findPrefixes(String arr[], int n)
{
    // Construct a Trie of all words
    root = new TrieNode();
    root.freq = 0;
    for (int i = 0; i<n; i++)
        insert(arr[i]);

    // Create an array to store all prefixes
    char[] prefix = new char[MAX_WORD_LEN];

    // Print all prefixes using Trie Traversal
    findPrefixesUtil(root, prefix, 0);
}

// Driver function.
public static void main(String args[])

```

```
{  
    String arr[] = {"zebra", "dog", "duck", "dove"};  
    int n = arr.length;  
    findPrefixes(arr, n);  
}  
}  
// This code is contributed by Sumit Ghosh
```

Output:

dog dov du z

Thanks to Gaurav Ahirwar for suggesting above solution.

Source

<https://www.geeksforgeeks.org/find-all-shortest-unique-prefixes-to-represent-each-word-in-a-given-list/>

Chapter 64

Find the k most frequent words from a file

Find the k most frequent words from a file - GeeksforGeeks

Given a book of words. Assume you have enough main memory to accommodate all words. design a data structure to find top K maximum occurring words. The data structure should be dynamic so that new words can be added.

A simple solution is to **use Hashing**. Hash all words one by one in a hash table. If a word is already present, then increment its count. Finally, traverse through the hash table and return the k words with maximum counts.

We can **use Trie and Min Heap** to get the k most frequent words efficiently. The idea is to use Trie for searching existing words adding new words efficiently. Trie also stores count of occurrences of words. A Min Heap of size k is used to keep track of k most frequent words at any point of time(Use of Min Heap is same as we used it to find k largest elements in [this post](#)).

Trie and Min Heap are linked with each other by storing an additional field in Trie ‘indexMinHeap’ and a pointer ‘trNode’ in Min Heap. The value of ‘indexMinHeap’ is maintained as -1 for the words which are currently not in Min Heap (or currently not among the top k frequent words). For the words which are present in Min Heap, ‘indexMinHeap’ contains, index of the word in Min Heap. The pointer ‘trNode’ in Min Heap points to the leaf node corresponding to the word in Trie.

Following is the complete process to print k most frequent words from a file.

Read all words one by one. For every word, insert it into Trie. Increase the counter of the word, if already exists. Now, we need to insert this word in min heap also. For insertion in min heap, 3 cases arise:

1. The word is already present. We just increase the corresponding frequency value in min heap and call minHeapify() for the index obtained by “indexMinHeap” field in Trie. When the min heap nodes are being swapped, we change the corresponding minHeapIndex in the Trie. Remember each node of the min heap is also having pointer to Trie leaf node.

2. The minHeap is not full. we will insert the new word into min heap & update the root node in the min heap node & min heap index in Trie leaf node. Now, call buildMinHeap().
3. The min heap is full. Two sub-cases arise.
 -3.1 The frequency of the new word inserted is less than the frequency of the word stored in the head of min heap. Do nothing.
 -3.2 The frequency of the new word inserted is greater than the frequency of the word stored in the head of min heap. Replace & update the fields. Make sure to update the corresponding min heap index of the “word to be replaced” in Trie with -1 as the word is no longer in min heap.
4. Finally, Min Heap will have the k most frequent words of all words present in given file. So we just need to print all words present in Min Heap.

```
// A program to find k most frequent words in a file
#include <stdio.h>
#include <string.h>
#include <ctype.h>

#define MAX_CHARS 26
#define MAX_WORD_SIZE 30

// A Trie node
struct TrieNode
{
    bool isEnd; // indicates end of word
    unsigned frequency; // the number of occurrences of a word
    int indexMinHeap; // the index of the word in minHeap
    TrieNode* child[MAX_CHARS]; // represents 26 slots each for 'a' to 'z'.
};

// A Min Heap node
struct MinHeapNode
{
    TrieNode* root; // indicates the leaf node of TRIE
    unsigned frequency; // number of occurrences
    char* word; // the actual word stored
};

// A Min Heap
struct MinHeap
{
    unsigned capacity; // the total size a min heap
    int count; // indicates the number of slots filled.
    MinHeapNode* array; // represents the collection of minHeapNodes
};

// A utility function to create a new Trie node
```

```
TrieNode* newTrieNode()
{
    // Allocate memory for Trie Node
    TrieNode* trieNode = new TrieNode;

    // Initialize values for new node
    trieNode->isEnd = 0;
    trieNode->frequency = 0;
    trieNode->indexMinHeap = -1;
    for( int i = 0; i < MAX_CHARS; ++i )
        trieNode->child[i] = NULL;

    return trieNode;
}

// A utility function to create a Min Heap of given capacity
MinHeap* createMinHeap( int capacity )
{
    MinHeap* minHeap = new MinHeap;

    minHeap->capacity = capacity;
    minHeap->count = 0;

    // Allocate memory for array of min heap nodes
    minHeap->array = new MinHeapNode [ minHeap->capacity ];

    return minHeap;
}

// A utility function to swap two min heap nodes. This function
// is needed in minHeapify
void swapMinHeapNodes ( MinHeapNode* a, MinHeapNode* b )
{
    MinHeapNode temp = *a;
    *a = *b;
    *b = temp;
}

// This is the standard minHeapify function. It does one thing extra.
// It updates the minHapIndex in Trie when two nodes are swapped in
// in min heap
void minHeapify( MinHeap* minHeap, int idx )
{
    int left, right, smallest;

    left = 2 * idx + 1;
    right = 2 * idx + 2;
    smallest = idx;
```

```

if ( left < minHeap->count &&
    minHeap->array[ left ]. frequency <
    minHeap->array[ smallest ]. frequency
)
    smallest = left;

if ( right < minHeap->count &&
    minHeap->array[ right ]. frequency <
    minHeap->array[ smallest ]. frequency
)
    smallest = right;

if( smallest != idx )
{
    // Update the corresponding index in Trie node.
    minHeap->array[ smallest ]. root->indexMinHeap = idx;
    minHeap->array[ idx ]. root->indexMinHeap = smallest;

    // Swap nodes in min heap
    swapMinHeapNodes ( &minHeap->array[ smallest ], &minHeap->array[ idx ] );

    minHeapify( minHeap, smallest );
}
}

// A standard function to build a heap
void buildMinHeap( MinHeap* minHeap )
{
    int n, i;
    n = minHeap->count - 1;

    for( i = ( n - 1 ) / 2; i >= 0; --i )
        minHeapify( minHeap, i );
}

// Inserts a word to heap, the function handles the 3 cases explained above
void insertInMinHeap( MinHeap* minHeap, TrieNode** root, const char* word )
{
    // Case 1: the word is already present in minHeap
    if( (*root)->indexMinHeap != -1 )
    {
        ++( minHeap->array[ (*root)->indexMinHeap ]. frequency );

        // percolate down
        minHeapify( minHeap, (*root)->indexMinHeap );
    }

    // Case 2: Word is not present and heap is not full
}

```

```

else if( minHeap->count < minHeap->capacity )
{
    int count = minHeap->count;
    minHeap->array[ count ]. frequency = (*root)->frequency;
    minHeap->array[ count ]. word = new char [strlen( word ) + 1];
    strcpy( minHeap->array[ count ]. word, word );

    minHeap->array[ count ]. root = *root;
    (*root)->indexMinHeap = minHeap->count;

    +( minHeap->count );
    buildMinHeap( minHeap );
}

// Case 3: Word is not present and heap is full. And frequency of word
// is more than root. The root is the least frequent word in heap,
// replace root with new word
else if ( (*root)->frequency > minHeap->array[0]. frequency )
{

    minHeap->array[ 0 ]. root->indexMinHeap = -1;
    minHeap->array[ 0 ]. root = *root;
    minHeap->array[ 0 ]. root->indexMinHeap = 0;
    minHeap->array[ 0 ]. frequency = (*root)->frequency;

    // delete previously allocated memory and
    delete [] minHeap->array[ 0 ]. word;
    minHeap->array[ 0 ]. word = new char [strlen( word ) + 1];
    strcpy( minHeap->array[ 0 ]. word, word );

    minHeapify ( minHeap, 0 );
}
}

// Inserts a new word to both Trie and Heap
void insertUtil ( TrieNode** root, MinHeap* minHeap,
                  const char* word, const char* dupWord )
{
    // Base Case
    if ( *root == NULL )
        *root = newTrieNode();

    // There are still more characters in word
    if ( *word != '\0' )
        insertUtil ( &(*root)->child[ tolower( *word ) - 97 ] ),
                    minHeap, word + 1, dupWord );
    else // The complete word is processed
    {

```

```
// word is already present, increase the frequency
if ( (*root)->isEnd )
    ++( (*root)->frequency );
else
{
    (*root)->isEnd = 1;
    (*root)->frequency = 1;
}

// Insert in min heap also
insertInMinHeap( minHeap, root, dupWord );
}

}

// add a word to Trie & min heap. A wrapper over the insertUtil
void insertTrieAndHeap(const char *word, TrieNode** root, MinHeap* minHeap)
{
    insertUtil( root, minHeap, word, word );
}

// A utility function to show results, The min heap
// contains k most frequent words so far, at any time
void displayMinHeap( MinHeap* minHeap )
{
    int i;

    // print top K word with frequency
    for( i = 0; i < minHeap->count; ++i )
    {
        printf( "%s : %d\n", minHeap->array[i].word,
                minHeap->array[i].frequency );
    }
}

// The main funtion that takes a file as input, add words to heap
// and Trie, finally shows result from heap
void printKMostFreq( FILE* fp, int k )
{
    // Create a Min Heap of Size k
    MinHeap* minHeap = createMinHeap( k );

    // Create an empty Trie
    TrieNode* root = NULL;

    // A buffer to store one word at a time
    char buffer[MAX_WORD_SIZE];
```

```
// Read words one by one from file. Insert the word in Trie and Min Heap
while( fscanf( fp, "%s", buffer ) != EOF )
    insertTrieAndHeap(buffer, &root, minHeap);

// The Min Heap will have the k most frequent words, so print Min Heap nodes
displayMinHeap( minHeap );
}

// Driver program to test above functions
int main()
{
    int k = 5;
    FILE *fp = fopen ("file.txt", "r");
    if (fp == NULL)
        printf ("File doesn't exist ");
    else
        printKMostFreq (fp, k);
    return 0;
}
```

Output:

```
your : 3
well : 3
and : 4
to : 4
Geeks : 6
```

The above output is for a file with following content.

```
Welcome to the world of Geeks
This portal has been created to provide well written well thought and well explained
solutions for selected questions If you like Geeks for Geeks and would like to contribute
here is your chance You can write article and mail your article to contribute at
geeksforgeeks.org See your article appearing on the Geeks for Geeks main page and help
thousands of other Geeks
```

Source

<https://www.geeksforgeeks.org/find-the-k-most-frequent-words-from-a-file/>

Chapter 65

Find the maximum subarray XOR in a given array

Find the maximum subarray XOR in a given array - GeeksforGeeks

Given an array of integers. find the maximum XOR subarray value in given array. Expected time complexity O(n).

Examples:

```
Input: arr[] = {1, 2, 3, 4}
Output: 7
The subarray {3, 4} has maximum XOR value

Input: arr[] = {8, 1, 2, 12, 7, 6}
Output: 15
The subarray {1, 2, 12} has maximum XOR value

Input: arr[] = {4, 6}
Output: 6
The subarray {6} has maximum XOR value
```

A **Simple Solution** is to use two loops to find XOR of all subarrays and return the maximum.

C++

```
// A simple C++ program to find max subarray XOR
#include<bits/stdc++.h>
using namespace std;

int maxSubarrayXOR(int arr[], int n)
```

```
{  
    int ans = INT_MIN;      // Initialize result  
  
    // Pick starting points of subarrays  
    for (int i=0; i<n; i++)  
    {  
        int curr_xor = 0; // to store xor of current subarray  
  
        // Pick ending points of subarrays starting with i  
        for (int j=i; j<n; j++)  
        {  
            curr_xor = curr_xor ^ arr[j];  
            ans = max(ans, curr_xor);  
        }  
    }  
    return ans;  
}  
  
// Driver program to test above functions  
int main()  
{  
    int arr[] = {8, 1, 2, 12};  
    int n = sizeof(arr)/sizeof(arr[0]);  
    cout << "Max subarray XOR is " << maxSubarrayXOR(arr, n);  
    return 0;  
}
```

Java

```
// A simple Java program to find max subarray XOR  
class GFG {  
    static int maxSubarrayXOR(int arr[], int n)  
    {  
        int ans = Integer.MIN_VALUE; // Initialize result  
  
        // Pick starting points of subarrays  
        for (int i=0; i<n; i++)  
        {  
            // to store xor of current subarray  
            int curr_xor = 0;  
  
            // Pick ending points of subarrays starting with i  
            for (int j=i; j<n; j++)  
            {  
                curr_xor = curr_xor ^ arr[j];  
                ans = Math.max(ans, curr_xor);  
            }  
        }  
    }
```

```
    return ans;
}

// Driver program to test above functions
public static void main(String args[])
{
    int arr[] = {8, 1, 2, 12};
    int n = arr.length;
    System.out.println("Max subarray XOR is " +
                        maxSubarrayXOR(arr, n));
}
//This code is contributed by Sumit Ghosh
```

Python3

```
# A simple Python program
# to find max subarray XOR

def maxSubarrayXOR(arr,n):

    ans = -2147483648      #Initialize result

    # Pick starting points of subarrays
    for i in range(n):

        # to store xor of current subarray
        curr_xor = 0

        # Pick ending points of
        # subarrays starting with i
        for j in range(i,n):

            curr_xor = curr_xor ^ arr[j]
            ans = max(ans, curr_xor)

    return ans

# Driver code

arr = [8, 1, 2, 12]
n = len(arr)

print("Max subarray XOR is ",
      maxSubarrayXOR(arr, n))
```

```
# This code is contributed
# by Anant Agarwal.
```

C#

```
// A simple C# program to find
// max subarray XOR
using System;

class GFG
{

    // Function to find max subarray
    static int maxSubarrayXOR(int []arr, int n)
    {
        int ans = int.MinValue;
        // Initialize result

        // Pick starting points of subarrays
        for (int i = 0; i < n; i++)
        {
            // to store xor of current subarray
            int curr_xor = 0;

            // Pick ending points of
            // subarrays starting with i
            for (int j = i; j < n; j++)
            {
                curr_xor = curr_xor ^ arr[j];
                ans = Math.Max(ans, curr_xor);
            }
        }
        return ans;
    }

    // Driver code
    public static void Main()
    {
        int []arr = {8, 1, 2, 12};
        int n = arr.Length;
        Console.WriteLine("Max subarray XOR is " +
                           maxSubarrayXOR(arr, n));
    }
}

// This code is contributed by Sam007.
```

PHP

```
<?php
// A simple PHP program to
// find max subarray XOR

function maxSubarrayXOR($arr, $n)
{

    // Initialize result
    $ans = PHP_INT_MIN;

    // Pick starting points
    // of subarrays
    for ($i = 0; $i < $n; $i++)
    {
        // to store xor of
        // current subarray
        $curr_xor = 0;

        // Pick ending points of
        // subarrays starting with i
        for ($j = $i; $j < $n; $j++)
        {
            $curr_xor = $curr_xor ^ $arr[$j];
            $ans = max($ans, $curr_xor);
        }
    }
    return $ans;
}

// Driver Code
$arr = array(8, 1, 2, 12);
$n = count($arr);
echo "Max subarray XOR is "
    , maxSubarrayXOR($arr, $n);

// This code is contributed by anuj_67.
?>
```

Output:

```
Max subarray XOR is 15
```

Time Complexity of above solution is $O(n^2)$.

An **Efficient Solution** can solve the above problem in $O(n)$ time under the assumption that integers take fixed number of bits to store. The idea is to use Trie Data Structure. Below is algorithm.

- 1) Create an empty Trie. Every node of Trie is going to contain two children, for 0 and 1 value of bit.
- 2) Initialize pre_xor = 0 and insert into the Trie.
- 3) Initialize result = minus infinite
- 4) Traverse the given array and do following for every array element arr[i].
 - a) pre_xor = pre_xor ^ arr[i]
pre_xor now contains xor of elements from arr[0] to arr[i].
 - b) Query the maximum xor value ending with arr[i] from Trie.
 - c) Update result if the value obtained in step 4.b is more than current value of result.

How does 4.b work?

We can observe from above algorithm that we build a Trie that contains XOR of all prefixes of given array. To find the maximum XOR subarray ending with arr[i], there may be two cases.

- i) The prefix itself has the maximum XOR value ending with arr[i]. For example if i=2 in {8, 2, 1, 12}, then the maximum subarray xor ending with arr[2] is the whole prefix.
- ii) We need to remove some prefix (ending at index from 0 to i-1). For example if i=3 in {8, 2, 1, 12}, then the maximum subarray xor ending with arr[3] starts with arr[1] and we need to remove arr[0].

To find the prefix to be removed, we find the entry in Trie that has maximum XOR value with current prefix. If we do XOR of such previous prefix with current prefix, we get the maximum XOR value ending with arr[i].

If there is no prefix to be removed (case i), then we return 0 (that's why we inserted 0 in Trie).

Below is the implementation of above idea :

C++

```
// C++ program for a Trie based O(n) solution to find max
// subarray XOR
#include<bits/stdc++.h>
using namespace std;

// Assumed int size
#define INT_SIZE 32

// A Trie Node
struct TrieNode
{
    int value; // Only used in leaf nodes
    TrieNode *arr[2];
};

TrieNode* insert(TrieNode* root, string s)
{
    TrieNode* curr = root;
    for (int i = 0; i < s.length(); i++) {
        int val = s[i] - '0';
        if (!curr->arr[val]) {
            curr->arr[val] = new TrieNode();
        }
        curr = curr->arr[val];
    }
    curr->value = 1;
    return root;
}

int query(TrieNode* root, string s)
{
    TrieNode* curr = root;
    int ans = 0;
    for (int i = 0; i < s.length(); i++) {
        int val = s[i] - '0';
        if (curr->arr[1 - val]) {
            curr = curr->arr[1 - val];
            ans ^= 1;
        } else {
            curr = curr->arr[val];
        }
    }
    return ans;
}
```

```
// Utility function tp create a Trie node
TrieNode *newNode()
{
    TrieNode *temp = new TrieNode;
    temp->value = 0;
    temp->arr[0] = temp->arr[1] = NULL;
    return temp;
}

// Inserts pre_xor to trie with given root
void insert(TrieNode *root, int pre_xor)
{
    TrieNode *temp = root;

    // Start from the msb, insert all bits of
    // pre_xor into Trie
    for (int i=INT_SIZE-1; i>=0; i--)
    {
        // Find current bit in given prefix
        bool val = pre_xor & (1<<i);

        // Create a new node if needed
        if (temp->arr[val] == NULL)
            temp->arr[val] = newNode();

        temp = temp->arr[val];
    }

    // Store value at leaf node
    temp->value = pre_xor;
}

// Finds the maximum XOR ending with last number in
// prefix XOR 'pre_xor' and returns the XOR of this maximum
// with pre_xor which is maximum XOR ending with last element
// of pre_xor.
int query(TrieNode *root, int pre_xor)
{
    TrieNode *temp = root;
    for (int i=INT_SIZE-1; i>=0; i--)
    {
        // Find current bit in given prefix
        bool val = pre_xor & (1<<i);

        // Traverse Trie, first look for a
        // prefix that has opposite bit
        if (temp->arr[1-val]!=NULL)
```

```
temp = temp->arr[1-val];

// If there is no prefix with opposite
// bit, then look for same bit.
else if (temp->arr[val] != NULL)
    temp = temp->arr[val];
}

return pre_xor^(temp->value);
}

// Returns maximum XOR value of a subarray in arr[0..n-1]
int maxSubarrayXOR(int arr[], int n)
{
    // Create a Trie and insert 0 into it
    TrieNode *root = newNode();
    insert(root, 0);

    // Initialize answer and xor of current prefix
    int result = INT_MIN, pre_xor = 0;

    // Traverse all input array element
    for (int i=0; i<n; i++)
    {
        // update current prefix xor and insert it into Trie
        pre_xor = pre_xor^arr[i];
        insert(root, pre_xor);

        // Query for current prefix xor in Trie and update
        // result if required
        result = max(result, query(root, pre_xor));
    }
    return result;
}

// Driver program to test above functions
int main()
{
    int arr[] = {8, 1, 2, 12};
    int n = sizeof(arr)/sizeof(arr[0]);
    cout << "Max subarray XOR is " << maxSubarrayXOR(arr, n);
    return 0;
}
```

Java

```
// Java program for a Trie based O(n) solution to
// find max subarray XOR
class GFG
```

```
{  
    // Assumed int size  
    static final int INT_SIZE = 32;  
  
    // A Trie Node  
    static class TrieNode  
    {  
        int value; // Only used in leaf nodes  
        TrieNode[] arr = new TrieNode[2];  
        public TrieNode()  
        {  
            value = 0;  
            arr[0] = null;  
            arr[1] = null;  
        }  
    }  
    static TrieNode root;  
  
    // Inserts pre_xor to trie with given root  
    static void insert(int pre_xor)  
    {  
        TrieNode temp = root;  
  
        // Start from the msb, insert all bits of  
        // pre_xor into Trie  
        for (int i=INT_SIZE-1; i>=0; i--)  
        {  
            // Find current bit in given prefix  
            int val = (pre_xor & (1<<i)) >=1 ? 1 : 0;  
  
            // Create a new node if needed  
            if (temp.arr[val] == null)  
                temp.arr[val] = new TrieNode();  
  
            temp = temp.arr[val];  
        }  
  
        // Store value at leaf node  
        temp.value = pre_xor;  
    }  
  
    // Finds the maximum XOR ending with last number in  
    // prefix XOR 'pre_xor' and returns the XOR of this  
    // maximum with pre_xor which is maximum XOR ending  
    // with last element of pre_xor.  
    static int query(int pre_xor)  
    {  
        TrieNode temp = root;  
        for (int i=INT_SIZE-1; i>=0; i--)
```

```
{  
    // Find current bit in given prefix  
    int val = (pre_xor & (1<<i)) >= 1 ? 1 : 0;  
  
    // Traverse Trie, first look for a  
    // prefix that has opposite bit  
    if (temp.arr[1-val] != null)  
        temp = temp.arr[1-val];  
  
    // If there is no prefix with opposite  
    // bit, then look for same bit.  
    else if (temp.arr[val] != null)  
        temp = temp.arr[val];  
}  
return pre_xor^(temp.value);  
}  
  
// Returns maximum XOR value of a subarray in  
// arr[0..n-1]  
static int maxSubarrayXOR(int arr[], int n)  
{  
    // Create a Trie and insert 0 into it  
    root = new TrieNode();  
    insert(0);  
  
    // Initialize answer and xor of current prefix  
    int result = Integer.MIN_VALUE;  
    int pre_xor = 0;  
  
    // Traverse all input array element  
    for (int i=0; i<n; i++)  
    {  
        // update current prefix xor and insert it  
        // into Trie  
        pre_xor = pre_xor^arr[i];  
        insert(pre_xor);  
  
        // Query for current prefix xor in Trie and  
        // update result if required  
        result = Math.max(result, query(pre_xor));  
    }  
    return result;  
}  
  
// Driver program to test above functions  
public static void main(String args[])  
{
```

```
int arr[] = {8, 1, 2, 12};  
int n = arr.length;  
System.out.println("Max subarray XOR is " +  
                    maxSubarrayXOR(arr, n));  
}  
}  
// This code is contributed by Sumit Ghosh
```

Output:

Max subarray XOR is 15

Exercise: Extend the above solution so that it also prints starting and ending indexes of subarray with maximum value (Hint: we can add one more field to Trie node to achieve this)

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

Improved By : [Sam007](#), [vt_m](#)

Source

<https://www.geeksforgeeks.org/find-the-maximum-subarray-xor-in-a-given-array/>

Chapter 66

Find the number of Islands Set 2 (Using Disjoint Set)

Find the number of Islands Set 2 (Using Disjoint Set) - GeeksforGeeks

Given a boolean 2D matrix, find the number of islands.

A group of connected 1s forms an island. For example, the below matrix contains 5 islands

```
{1, 1, 0, 0, 0},  
{0, 1, 0, 0, 1},  
{1, 0, 0, 1, 1},  
{0, 0, 0, 0, 0},  
{1, 0, 1, 0, 1}
```

A cell in 2D matrix can be connected to 8 neighbors.

This is an variation of the standard problem: “Counting number of connected components in a undirected graph”. We have discussed a DFS based solution in below set 1.

Find the number of islands

We can also solve the question using disjoint set data structure explained [here](#). The idea is to consider all 1 values as individual sets. Traverse the matrix and do union of all adjacent 1 vertices. Below are detailed steps.

Approach:

- 1) Initialize result (count of islands) as 0
- 2) Traverse each index of the 2D matrix.
- 3) If value at that index is 1, check all its 8 neighbours. If a neighbour is also equal to 1, take union of index and its neighbour.
- 4) Now define an array of size row*column to store frequencies of all sets.
- 5) Now traverse the matrix again.

- 6) If value at index is 1, find its set.
- 7) If frequency of the set in the above array is 0, increment the result be 1.

Following is Java implementation of above steps.

```
// Java program to fnd number of islands using Disjoint
// Set data structure.
import java.io.*;
import java.util.*;

public class Main
{
    public static void main(String[] args) throws IOException
    {
        int[][] a = new int[][] {{1, 1, 0, 0, 0},
                                 {0, 1, 0, 0, 1},
                                 {1, 0, 0, 1, 1},
                                 {0, 0, 0, 0, 0},
                                 {1, 0, 1, 0, 1}};
        System.out.println("Number of Islands is: " +
                           countIslands(a));
    }

    // Returns number of islands in a[] []
    static int countIslands(int a[][])
    {
        int n = a.length;
        int m = a[0].length;

        DisjointUnionSets dus = new DisjointUnionSets(n*m);

        /* The following loop checks for its neighbours
           and unites the indexes if both are 1. */
        for (int j=0; j<n; j++)
        {
            for (int k=0; k<m; k++)
            {
                // If cell is 0, nothing to do
                if (a[j][k] == 0)
                    continue;

                // Check all 8 neighbours and do a union
                // with neighbour's set if neighbour is
                // also 1
                if (j+1 < n && a[j+1][k]==1)
                    dus.union(j*(m)+k, (j+1)*(m)+k);
                if (j-1 >= 0 && a[j-1][k]==1)
                    dus.union(j*(m)+k, (j-1)*(m)+k);
            }
        }
    }
}
```

```

        if (k+1 < m && a[j][k+1]==1)
            dus.union(j*(m)+k, (j)*(m)+k+1);
        if (k-1 >= 0 && a[j][k-1]==1)
            dus.union(j*(m)+k, (j)*(m)+k-1);
        if (j+1<n && k+1<m && a[j+1][k+1]==1)
            dus.union(j*(m)+k, (j+1)*(m)+k+1);
        if (j+1<n && k-1>=0 && a[j+1][k-1]==1)
            dus.union(j*m+k, (j+1)*(m)+k-1);
        if (j-1>=0 && k+1<m && a[j-1][k+1]==1)
            dus.union(j*m+k, (j-1)*m+k+1);
        if (j-1>=0 && k-1>=0 && a[j-1][k-1]==1)
            dus.union(j*m+k, (j-1)*m+k-1);
    }
}

// Array to note down frequency of each set
int[] c = new int[n*m];
int numberOfIslands = 0;
for (int j=0; j<n; j++)
{
    for (int k=0; k<m; k++)
    {
        if (a[j][k]==1)
        {

            int x = dus.find(j*m+k);

            // If frequency of set is 0,
            // increment numberOfIslands
            if (c[x]==0)
            {
                numberOfIslands++;
                c[x]++;
            }

            else
                c[x]++;
        }
    }
}
return numberOfIslands;
}

// Class to represent Disjoint Set Data structure
class DisjointUnionSets
{
    int[] rank, parent;
}

```

```
int n;

public DisjointUnionSets(int n)
{
    rank = new int[n];
    parent = new int[n];
    this.n = n;
    makeSet();
}

void makeSet()
{
    // Initially, all elements are in their
    // own set.
    for (int i=0; i<n; i++)
        parent[i] = i;
}

// Finds the representative of the set that x
// is an element of
int find(int x)
{
    if (parent[x] != x)
    {
        // if x is not the parent of itself,
        // then x is not the representative of
        // its set.
        // so we recursively call Find on its parent
        // and move i's node directly under the
        // representative of this set
        return find(parent[x]);
    }

    return x;
}

// Unites the set that includes x and the set
// that includes y
void union(int x, int y)
{
    // Find the representatives (or the root nodes)
    // for x and y
    int xRoot = find(x);
    int yRoot = find(y);

    // Elements are in the same set, no need
    // to unite anything.
    if (xRoot == yRoot)
```

```
return;

// If x's rank is less than y's rank
// Then move x under y so that depth of tree
// remains less
if (rank[xRoot] < rank[yRoot])
    parent[xRoot] = yRoot;

// Else if y's rank is less than x's rank
// Then move y under x so that depth of tree
// remains less
else if(rank[yRoot]<rank[xRoot])
    parent[yRoot] = xRoot;

else // Else if their ranks are the same
{
    // Then move y under x (doesn't matter
    // which one goes where)
    parent[yRoot] = xRoot;

    // And increment the the result tree's
    // rank by 1
    rank[xRoot] = rank[xRoot] + 1;
}
}
```

Output:

Number of Islands is: 5

Source

<https://www.geeksforgeeks.org/find-the-number-of-islands-set-2-using-disjoint-set/>

Chapter 67

Find whether a subarray is in form of a mountain or not

Find whether a subarray is in form of a mountain or not - GeeksforGeeks

We are given an array of integers and a range, we need to find whether the subarray which falls in this range has values in form of a mountain or not. All values of the subarray are said to be in form of a mountain if either all values are increasing or first increasing and then decreasing.

More formally a subarray $[a_1, a_2, a_3 \dots a_N]$ is said to be in form of a mountain if there exist an integer K , $1 \leq K \leq N$ such that,

$a_1 \leq a_2 \leq a_3 \dots \leq a_K \geq a_{K+1} \geq a_{K+2} \dots \geq a_N$

Examples:

```
Arr[] = [2 3 2 4 4 6 3 2]
Range = [0, 2]
Output yes because [2 3 2] subarray first
increases and then decreases
```

```
Range = [2, 7]
Output yes because [2 4 4 6 3 2] subarray
first increases and then decreases
```

```
Range = [2, 3]
Output yes because [2 4] subarray increases
```

```
Range = [1, 3]
Output no because [3 2 4] is not in the form
above stated
```

We can solve this problem by first some preprocessing then we can answer for each subarray in the constant amount of time. We maintain two arrays left and right where $\text{left}[i]$ stores

the last index on left side which is increasing i.e. greater than its previous element and right[i] will store the first index on the right side which is decreasing i.e. greater than its next element. Once we maintained these arrays we can answer each subarray in constant time. Suppose range [l, r] is given then only if right[l] \geq left[r], the subarray will be in form of a mountain otherwise not because the first index in decreasing form (i.e. right[l]) should come later than last index in increasing form (i.e. left[r]).

This procedure is demonstrated for above example array.

```
Arr[] = [2 3 2 4 4 6 3 2]
Using above procedure building left and right array,
left[] = [0 1 1 3 3 5 5 5]
right[] = [1 1 5 5 5 6 7]
Range = [2, 4]
Now right[2] is 5 and left[4] is 3 that means at
index 2 first decreasing element is right to the
last increasing element at index 4, so they should
have a mountain form.

Range = [0, 3]
Now right[0] is 1 and left[3] is 3 that means at
index 0 first decreasing element is left to the
last increasing element at index 3, so the subarray
corresponding to this range does not have mountain
form. We can see this in the array itself, right[0]
is 1 which is value 3 and left[3] is 3 which is
value 4 so 4 which is in increasing form (due to
previous value 2) comes later to 3 which is in
decreasing form (due to next value 2), mountain form
was not possible here, same information is carried
out with the help of left and right array.
```

Auxiliary space for this solution is O(N) and preprocessing takes O(N) time, after that each subarray can be handled in constant time.

C/C++

```
// C++ program to check whether a subarray is in
// mountain form or not
#include <bits/stdc++.h>
using namespace std;

// Utility method to construct left and right array
int preprocess(int arr[], int N, int left[], int right[])
{
    // initialize first left index as that index only
    left[0] = 0;
    int lastIncr = 0;
```

```
for (int i = 1; i < N; i++)
{
    // if current value is greater than previous,
    // update last increasing
    if (arr[i] > arr[i - 1])
        lastIncr = i;
    left[i] = lastIncr;
}

// initialize last right index as that index only
right[N - 1] = N - 1;
int firstDecr = N - 1;

for (int i = N - 2; i >= 0; i--)
{
    // if current value is greater than next,
    // update first decreasing
    if (arr[i] > arr[i + 1])
        firstDecr = i;
    right[i] = firstDecr;
}
}

// method returns true if arr[L..R] is in mountain form
bool isSubarrayMountainForm(int arr[], int left[],
                            int right[], int L, int R)
{
    // return true only if right at starting range is
    // greater than left at ending range
    return (right[L] >= left[R]);
}

// Driver code to test above methods
int main()
{
    int arr[] = {2, 3, 2, 4, 4, 6, 3, 2};

    int left[N], right[N];
    preprocess(arr, N, left, right);

    int L = 0;
    int R = 2;
    if (isSubarrayMountainForm(arr, left, right, L, R))
        cout << "Subarray is in mountain form\n";
    else
        cout << "Subarray is not in mountain form\n";
}
```

```
L = 1;
R = 3;
if (isSubarrayMountainForm(arr, left, right, L, R))
    cout << "Subarray is in mountain form\n";
else
    cout << "Subarray is not in mountain form\n";

return 0;
}
```

Java

```
// Java program to check whether a subarray is in
// mountain form or not
class SubArray
{
    // Utility method to construct left and right array
    static void preprocess(int arr[], int N, int left[], int right[])
    {
        // initialize first left index as that index only
        left[0] = 0;
        int lastIncr = 0;

        for (int i = 1; i < N; i++)
        {
            // if current value is greater than previous,
            // update last increasing
            if (arr[i] > arr[i - 1])
                lastIncr = i;
            left[i] = lastIncr;
        }

        // initialize last right index as that index only
        right[N - 1] = N - 1;
        int firstDecr = N - 1;

        for (int i = N - 2; i >= 0; i--)
        {
            // if current value is greater than next,
            // update first decreasing
            if (arr[i] > arr[i + 1])
                firstDecr = i;
            right[i] = firstDecr;
        }
    }

    // method returns true if arr[L..R] is in mountain form
}
```

```
static boolean isSubarrayMountainForm(int arr[], int left[],
                                     int right[], int L, int R)
{
    // return true only if right at starting range is
    // greater than left at ending range
    return (right[L] >= left[R]);
}

public static void main(String[] args)
{
    int arr[] = {2, 3, 2, 4, 4, 6, 3, 2};
    int N = arr.length;
    int left[] = new int[N];
    int right[] = new int[N];
    preprocess(arr, N, left, right);
    int L = 0;
    int R = 2;

    if (isSubarrayMountainForm(arr, left, right, L, R))
        System.out.println("Subarray is in mountain form");
    else
        System.out.println("Subarray is not in mountain form");

    L = 1;
    R = 3;

    if (isSubarrayMountainForm(arr, left, right, L, R))
        System.out.println("Subarray is in mountain form");
    else
        System.out.println("Subarray is not in mountain form");
}
}

// This Code is Contributed by Saket Kumar
```

C#

```
// C# program to check whether
// a subarray is in mountain
// form or not
using System;

class GFG
{

    // Utility method to construct
    // left and right array
    static void preprocess(int []arr, int N,
                          int []left, int []right)
```

```
{  
    // initialize first left  
    // index as that index only  
    left[0] = 0;  
    int lastIncr = 0;  
  
    for (int i = 1; i < N; i++)  
    {  
        // if current value is  
        // greater than previous,  
        // update last increasing  
        if (arr[i] > arr[i - 1])  
            lastIncr = i;  
        left[i] = lastIncr;  
    }  
  
    // initialize last right  
    // index as that index only  
    right[N - 1] = N - 1;  
    int firstDecr = N - 1;  
  
    for (int i = N - 2; i >= 0; i--)  
    {  
        // if current value is  
        // greater than next,  
        // update first decreasing  
        if (arr[i] > arr[i + 1])  
            firstDecr = i;  
        right[i] = firstDecr;  
    }  
}  
  
// method returns true if  
// arr[L..R] is in mountain form  
static bool isSubarrayMountainForm(int []arr, int []left,  
                                    int []right, int L, int R)  
{  
    // return true only if right at  
    // starting range is greater  
    // than left at ending range  
    return (right[L] >= left[R]);  
}  
  
// Driver Code  
static public void Main ()  
{  
    int []arr = {2, 3, 2, 4,
```

```
        4, 6, 3, 2};  
int N = arr.Length;  
int []left = new int[N];  
int []right = new int[N];  
preprocess(arr, N, left, right);  
  
int L = 0;  
int R = 2;  
  
if (isSubarrayMountainForm(arr, left,  
                           right, L, R))  
    Console.WriteLine("Subarray is in " +  
                      "mountain form");  
else  
    Console.WriteLine("Subarray is not " +  
                      "in mountain form");  
  
L = 1;  
R = 3;  
  
if (isSubarrayMountainForm(arr, left,  
                           right, L, R))  
    Console.WriteLine("Subarray is in " +  
                      "mountain form");  
else  
    Console.WriteLine("Subarray is not " +  
                      "in mountain form");  
}  
}  
  
// This code is contributed by aj_36
```

Output:

```
Subarray is in mountain form  
Subarray is not in mountain form
```

Improved By : [jit_t](#)

Source

<https://www.geeksforgeeks.org/find-whether-subarray-form-mountain-not/>

Chapter 68

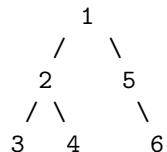
Flatten a binary tree into linked list Set-2

Flatten a binary tree into linked list Set-2 - GeeksforGeeks

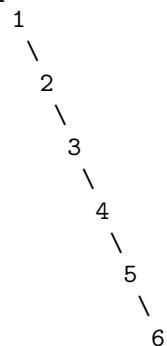
Given a binary tree, flatten it into a linked list. After flattening, the left of each node should point to NULL and right should contain next node in level order.

Example:

Input :

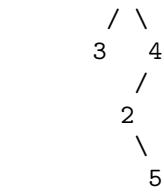


Output :

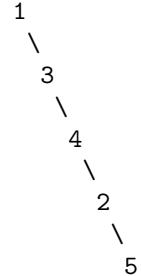


Input :

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 69

GCDs of given index ranges in an array

GCDs of given index ranges in an array - GeeksforGeeks

Given an array $a[0 \dots n-1]$. We should be able to efficiently find the GCD from index qs (query start) to qe (query end) where $0 \leq qs \leq qe \leq n-1$.

Example :

```
Input : a[] = {2, 3, 60, 90, 50};  
        Index Ranges : {1, 3}, {2, 4}, {0, 2}  
Output: GCDs of given ranges are 3, 10, 1
```

Method 1 (Simple)

A simple solution is to run a loop from qs to qe and find GCD in given range. This solution takes $O(n)$ time in worst case.

Method 2 (2D Array)

Another solution is to create a 2D array where an entry $[i, j]$ stores the GCD in range $arr[i..j]$. GCD of a given range can now be calculated in $O(1)$ time, but preprocessing takes $O(n^2)$ time. Also, this approach needs $O(n^2)$ extra space which may become huge for large input arrays.

Method 3 (Segment Tree)

Prerequisites : [Segment Tree Set 1](#), [Segment Tree Set 2](#)

Segment tree can be used to do preprocessing and query in moderate time. With segment

tree, preprocessing time is $O(n)$ and time to for GCD query is $O(\log n)$. The extra space required is $O(n)$ to store the segment tree.

Representation of Segment trees

- Leaf Nodes are the elements of the input array.
- Each internal node represents GCD of all leaves under it.

Array representation of tree is used to represent Segment Trees i.e., for each node at index i ,

- Left child is at index $2*i+1$
- Right child at $2*i+2$ and the parent is at $\text{floor}((i-1)/2)$.

Construction of Segment Tree from given array

- Begin with a segment $\text{arr}[0 \dots n-1]$ and keep dividing into two halves. Every time we divide the current segment into two halves (if it has not yet become a segment of length 1), then call the same procedure on both halves, and for each such segment, we store the GCD value in a segment tree node.
- All levels of the constructed segment tree will be completely filled except the last level. Also, the tree will be a Full Binary Tree (every node has 0 or two children) because we always divide segments in two halves at every level.
- Since the constructed tree is always full binary tree with n leaves, there will be $n-1$ internal nodes. So total number of nodes will be $2*n - 1$.
- Height of the segment tree will be $\lceil \log_2 n \rceil$. Since the tree is represented using array and relation between parent and child indexes must be maintained, size of memory allocated for segment tree will be $2^{2^{\lceil \log_2 n \rceil}} - 1$

Query for GCD of given range

```
/ qs --> query start index, qe --> query end index
int GCD(node, qs, qe)
{
    if range of node is within qs and qe
        return value in node
    else if range of node is completely
        outside qs and qe
        return INFINITE
    else
        return GCD( GCD(node's left child, qs, qe),
                    GCD(node's right child, qs, qe) )
}
```

Below is Implementation of this method.

C++

```

// C++ Program to find GCD of a number in a given Range
// using segment Trees
#include <bits/stdc++.h>
using namespace std;

// To store segment tree
int *st;

// Function to find gcd of 2 numbers.
int gcd(int a, int b)
{
    if (a < b)
        swap(a, b);
    if (b==0)
        return a;
    return gcd(b, a%b);
}

/* A recursive function to get gcd of given
range of array indexes. The following are parameters for
this function.

    st    --> Pointer to segment tree
    si --> Index of current node in the segment tree. Initially
           0 is passed as root is always at index 0
    ss & se  --> Starting and ending indexes of the segment
                  represented by current node, i.e., st[index]
    qs & qe  --> Starting and ending indexes of query range */
int findGcd(int ss, int se, int qs, int qe, int si)
{
    if (ss>qe || se < qs)
        return 0;
    if (qs<=ss && qe>=se)
        return st[si];
    int mid = ss+(se-ss)/2;
    return gcd(findGcd(ss, mid, qs, qe, si*2+1),
               findGcd(mid+1, se, qs, qe, si*2+2));
}

//Finding The gcd of given Range
int findRangeGcd(int ss, int se, int arr[],int n)
{
    if (ss<0 || se > n-1 || ss>se)
    {
        cout << "Invalid Arguments" << "\n";
    }
}

```

```
        return -1;
    }
    return findGcd(0, n-1, ss, se, 0);
}

// A recursive function that constructs Segment Tree for
// array[ss..se]. si is index of current node in segment
// tree st
int constructST(int arr[], int ss, int se, int si)
{
    if (ss==se)
    {
        st[si] = arr[ss];
        return st[si];
    }
    int mid = ss+(se-ss)/2;
    st[si] = gcd(constructST(arr, ss, mid, si*2+1),
                  constructST(arr, mid+1, se, si*2+2));
    return st[si];
}

/* Function to construct segment tree from given array.
   This function allocates memory for segment tree and
   calls constructSTUtil() to fill the allocated memory */
int *constructSegmentTree(int arr[], int n)
{
    int height = (int)(ceil(log2(n)));
    int size = 2*(int)pow(2, height)-1;
    st = new int[size];
    constructST(arr, 0, n-1, 0);
    return st;
}

// Driver program to test above functions
int main()
{
    int a[] = {2, 3, 6, 9, 5};
    int n = sizeof(a)/sizeof(a[0]);

    // Build segment tree from given array
    constructSegmentTree(a, n);

    // Starting index of range. These indexes are 0 based.
    int l = 1;

    // Last index of range. These indexes are 0 based.
    int r = 3;
    cout << "GCD of the given range is:";
```

```
    cout << findRangeGcd(l, r, a, n) << "\n";  
  
    return 0;  
}
```

Java

```
// Java Program to find GCD of a number in a given Range  
// using segment Trees  
import java.io.*;  
  
public class Main  
{  
    private static int[] st; // Array to store segment tree  
  
    /* Function to construct segment tree from given array.  
       This function allocates memory for segment tree and  
       calls constructSTUtil() to fill the allocated memory */  
    public static int[] constructSegmentTree(int[] arr)  
    {  
        int height = (int) Math.ceil(Math.log(arr.length)/Math.log(2));  
        int size = 2*(int) Math.pow(2, height)-1;  
        st = new int[size];  
        constructST(arr, 0, arr.length-1, 0);  
        return st;  
    }  
  
    // A recursive function that constructs Segment  
    // Tree for array[ss..se]. si is index of current  
    // node in segment tree st  
    public static int constructST(int[] arr, int ss,  
                                int se, int si)  
    {  
        if (ss==se)  
        {  
            st[si] = arr[ss];  
            return st[si];  
        }  
        int mid = ss+(se-ss)/2;  
        st[si] = gcd(constructST(arr, ss, mid, si*2+1),  
                    constructST(arr, mid+1, se, si*2+2));  
        return st[si];  
    }  
  
    // Function to find gcd of 2 numbers.  
    private static int gcd(int a, int b)  
    {  
        if (a < b)
```

```

{
    // If b greater than a swap a and b
    int temp = b;
    b = a;
    a = temp;
}

if (b==0)
    return a;
return gcd(b,a%b);
}

//Finding The gcd of given Range
public static int findRangeGcd(int ss, int se, int[] arr)
{
    int n = arr.length;

    if (ss<0 || se > n-1 || ss>se)
        throw new IllegalArgumentException("Invalid arguments");

    return findGcd(0, n-1, ss, se, 0);
}

/* A recursive function to get gcd of given
range of array indexes. The following are parameters for
this function.

st --> Pointer to segment tree
si --> Index of current node in the segment tree. Initially
      0 is passed as root is always at index 0
ss & se --> Starting and ending indexes of the segment
            represented by current node, i.e., st[si]
qs & qe --> Starting and ending indexes of query range */
public static int findGcd(int ss, int se, int qs, int qe, int si)
{
    if (ss>qe || se < qs)
        return 0;

    if (qs<=ss && qe>=se)
        return st[si];

    int mid = ss+(se-ss)/2;

    return gcd(findGcd(ss, mid, qs, qe, si*2+1),
              findGcd(mid+1, se, qs, qe, si*2+2));
}

// Driver Code

```

```
public static void main(String[] args) throws IOException
{
    int[] a = {2, 3, 6, 9, 5};

    constructSegmentTree(a);

    int l = 1; // Starting index of range.
    int r = 3; //Last index of range.
    System.out.print("GCD of the given range is: ");
    System.out.print(findRangeGcd(l, r, a));
}
```

Output:

```
GCD of the given range is: 3
```

Time Complexity: Time Complexity for tree construction is $O(n * \log(\min(a, b)))$, where n is the number of nodes and a and b are nodes whose GCD is calculated during merge operation. There are total $2n-1$ nodes, and value of every node is calculated only once in tree construction. Time complexity to query is $O(\log n * \log n)$.

Source

<https://www.geeksforgeeks.org/gcds-of-a-given-index-ranges-in-an-array/>

Chapter 70

Generalized Suffix Tree 1

Generalized Suffix Tree 1 - GeeksforGeeks

In earlier suffix tree articles, we created suffix tree for one string and then we queried that tree for [substring check](#), [searching all patterns](#), [longest repeated substring](#) and [built suffix array](#) (All linear time operations).

There are lots of other problems where multiple strings are involved.

e.g. pattern searching in a text file or dictionary, spell checker, phone book, [Autocomplete](#), [Longest common substring problem](#), [Longest palindromic substring](#) and [More](#).

For such operations, all the involved strings need to be indexed for faster search and retrieval. One way to do this is using suffix trie or suffix tree. We will discuss suffix tree here.

A suffix tree made of a set of strings is known as [Generalized Suffix Tree](#).

We will discuss a simple way to build [Generalized Suffix Tree](#) here for [two strings only](#). Later, we will discuss another approach to build [Generalized Suffix Tree](#) for [two or more strings](#).

Here we will use the [suffix tree implementation](#) for one string discussed already and modify that a bit to build [generalized suffix tree](#).

Lets consider two strings X and Y for which we want to build generalized suffix tree. For this we will make a new string X#Y\$ where # and \$ both are terminal symbols (must be unique). Then we will build suffix tree for X#Y\$ which will be the generalized suffix tree for X and Y. Same logic will apply for more than two strings (i.e. concatenate all strings using unique terminal symbols and then build suffix tree for concatenated string).

Lets say X = xabxa, and Y = babxba, then

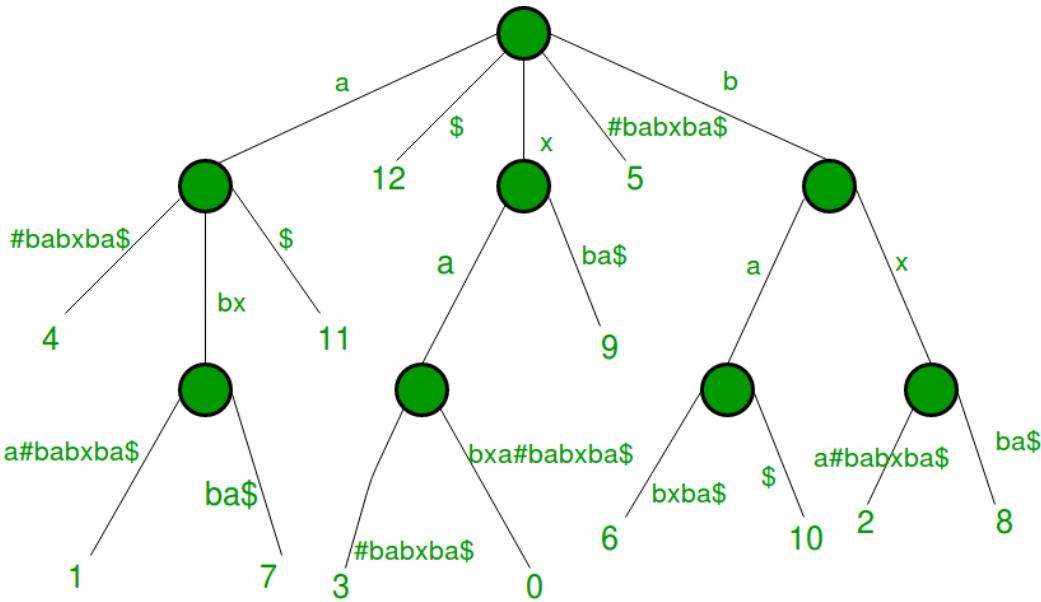
X#Y\$ = xabxa#babxba\$

If we run the code implemented at [Ukkonen's Suffix Tree Construction – Part 6](#) for string xabxa#babxba\$, we get following output:

Output:

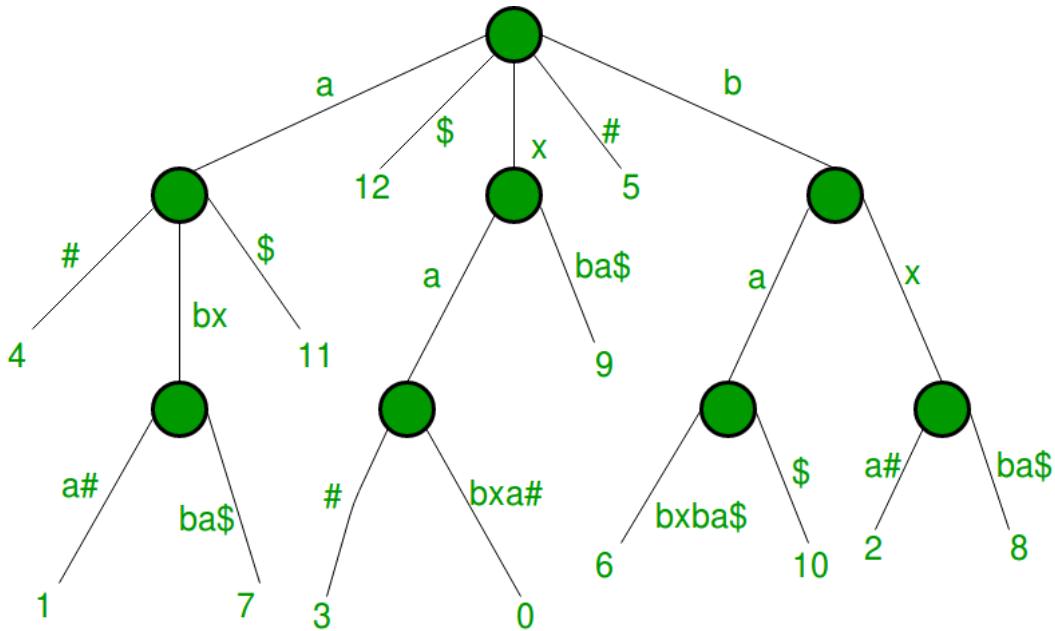
```
# babxba$ [5]
$ [12]
a [-1]
#babxba$ [4]
$ [11]
bx [-1]
a#babxba$ [1]
ba$ [7]
b [-1]
a [-1]
$ [10]
bxba$ [6]
x [-1]
a#babxba$ [2]
ba$ [8]
x [-1]
a [-1]
#babxba$ [3]
bxa#babxba$ [0]
ba$ [9]
```

Pictorial View:



Suffix tree for string xabxa#babxa\$

We can use this tree to solve some of the problems, but we can refine it a bit by removing unwanted substrings on a path label. A path label should have substring from only one input string, so if there are path labels having substrings from multiple input strings, we can keep only the initial portion corresponding to one string and remove all the later portion. For example, for path labels #babxba\$, a#babxba\$ and bxa#babxba\$, we can remove babxba\$ (belongs to 2nd input string) and then new path labels will be #, a# and bxa# respectively. With this change, above diagram will look like below:



Below implementation is built on top of [original implementation](#). Here we are removing unwanted characters on path labels. If a path label has “#” character in it, then we are trimming all characters after the “#” in that path label.

Note: This implementation builds generalized suffix tree for only two strings X and Y which are concatenated as X#Y\$

```

// A C program to implement Ukkonen's Suffix Tree Construction
// And then build generalized suffix tree
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define MAX_CHAR 256

struct SuffixTreeNode {
    struct SuffixTreeNode *children[MAX_CHAR];
    //pointer to other node via suffix link
    struct SuffixTreeNode *suffixLink;
    /*(start, end) interval specifies the edge, by which the
    node is connected to its parent node. Each edge will
    connect two nodes, one parent and one child, and
    (start, end) interval of a given edge will be stored
    in the child node. Lets say there are two nodes A and B
    connected by an edge with indices (5, 8) then this
    indices (5, 8) will be stored in node B. */
    int start;
    int *end;
}

```

```
/*for leaf nodes, it stores the index of suffix for
   the path from root to leaf*/
int suffixIndex;
};

typedef struct SuffixTreeNode Node;

char text[100]; //Input string
Node *root = NULL; //Pointer to root node

/*lastNewNode will point to newly created internal node,
   waiting for it's suffix link to be set, which might get
   a new suffix link (other than root) in next extension of
   same phase. lastNewNode will be set to NULL when last
   newly created internal node (if there is any) got it's
   suffix link reset to new internal node created in next
   extension of same phase. */
Node *lastNewNode = NULL;
Node *activeNode = NULL;

/*activeEdge is represeted as input string character
   index (not the character itself)*/
int activeEdge = -1;
int activeLength = 0;

// remainingSuffixCount tells how many suffixes yet to
// be added in tree
int remainingSuffixCount = 0;
int leafEnd = -1;
int *rootEnd = NULL;
int *splitEnd = NULL;
int size = -1; //Length of input string

Node *newNode(int start, int *end)
{
    Node *node =(Node*) malloc(sizeof(Node));
    int i;
    for (i = 0; i < MAX_CHAR; i++)
        node->children[i] = NULL;

    /*For root node, suffixLink will be set to NULL
     For internal nodes, suffixLink will be set to root
     by default in current extension and may change in
     next extension*/
    node->suffixLink = root;
    node->start = start;
    node->end = end;
```

```

/*suffixIndex will be set to -1 by default and
actual suffix index will be set later for leaves
at the end of all phases*/
node->suffixIndex = -1;
return node;
}

int edgeLength(Node *n) {
    if(n == root)
        return 0;
    return *(n->end) - (n->start) + 1;
}

int walkDown(Node *currNode)
{
    /*activePoint change for walk down (APCFWD) using
    Skip/Count Trick (Trick 1). If activeLength is greater
    than current edge length, set next internal node as
    activeNode and adjust activeEdge and activeLength
    accordingly to represent same activePoint*/
    if (activeLength >= edgeLength(currNode))
    {
        activeEdge += edgeLength(currNode);
        activeLength -= edgeLength(currNode);
        activeNode = currNode;
        return 1;
    }
    return 0;
}

void extendSuffixTree(int pos)
{
    /*Extension Rule 1, this takes care of extending all
    leaves created so far in tree*/
    leafEnd = pos;

    /*Increment remainingSuffixCount indicating that a
    new suffix added to the list of suffixes yet to be
    added in tree*/
    remainingSuffixCount++;

    /*set lastNewNode to NULL while starting a new phase,
    indicating there is no internal node waiting for
    it's suffix link reset in current phase*/
    lastNewNode = NULL;

    //Add all suffixes (yet to be added) one by one in tree
}

```

```

while(remainingSuffixCount > 0) {

    if (activeLength == 0)
        activeEdge = pos; //APCFALZ

    // There is no outgoing edge starting with
    // activeEdge from activeNode
    if (activeNode->children] == NULL)
    {
        //Extension Rule 2 (A new leaf edge gets created)
        activeNode->children] =
            newNode(pos, &leafEnd);

        /*A new leaf edge is created in above line starting
         from an existing node (the current activeNode), and
         if there is any internal node waiting for it's suffix
         link get reset, point the suffix link from that last
         internal node to current activeNode. Then set lastNewNode
         to NULL indicating no more node waiting for suffix link
         reset.*/
        if (lastNewNode != NULL)
        {
            lastNewNode->suffixLink = activeNode;
            lastNewNode = NULL;
        }
    }
    // There is an outgoing edge starting with activeEdge
    // from activeNode
    else
    {
        // Get the next node at the end of edge starting
        // with activeEdge
        Node *next = activeNode->children];
        if (walkDown(next))//Do walkdown
        {
            //Start from next node (the new activeNode)
            continue;
        }
        /*Extension Rule 3 (current character being processed
         is already on the edge)*/
        if (text[next->start + activeLength] == text[pos])
        {
            //If a newly created node waiting for it's
            //suffix link to be set, then set suffix link
            //of that waiting node to current active node
            if(lastNewNode != NULL && activeNode != root)
            {
                lastNewNode->suffixLink = activeNode;
            }
        }
    }
}

```

```

        lastNewNode = NULL;
    }

    //APCFER3
    activeLength++;
    /*STOP all further processing in this phase
    and move on to next phase*/
    break;
}

/*We will be here when activePoint is in middle of
the edge being traversed and current character
being processed is not on the edge (we fall off
the tree). In this case, we add a new internal node
and a new leaf edge going out of that new node. This
is Extension Rule 2, where a new leaf edge and a new
internal node get created*/
splitEnd = (int*) malloc(sizeof(int));
*splitEnd = next->start + activeLength - 1;

//New internal node
Node *split = newNode(next->start, splitEnd);
activeNode->children] = split;

//New leaf coming out of new internal node
split->children] = newNode(pos, &leafEnd);
next->start += activeLength;
split->children] = next;

/*We got a new internal node here. If there is any
internal node created in last extensions of same
phase which is still waiting for it's suffix link
reset, do it now.*/
if (lastNewNode != NULL)
{
/*suffixLink of lastNewNode points to current newly
created internal node*/
    lastNewNode->suffixLink = split;
}

/*Make the current newly created internal node waiting
for it's suffix link reset (which is pointing to root
at present). If we come across any other internal node
(existing or newly created) in next extension of same
phase, when a new leaf edge gets added (i.e. when
Extension Rule 2 applies is any of the next extension
of same phase) at that point, suffixLink of this node
will point to that internal node.*/

```

```

        lastNewNode = split;
    }

    /* One suffix got added in tree, decrement the count of
       suffixes yet to be added.*/
    remainingSuffixCount--;
    if (activeNode == root && activeLength > 0) //APCFER2C1
    {
        activeLength--;
        activeEdge = pos - remainingSuffixCount + 1;
    }
    else if (activeNode != root) //APCFER2C2
    {
        activeNode = activeNode->suffixLink;
    }
}
}

void print(int i, int j)
{
    int k;
    for (k=i; k<=j && text[k] != '#'; k++)
        printf("%c", text[k]);
    if(k<=j)
        printf("#");
}

//Print the suffix tree as well along with setting suffix index
//So tree will be printed in DFS manner
//Each edge along with it's suffix index will be printed
void setSuffixIndexByDFS(Node *n, int labelHeight)
{
    if (n == NULL) return;

    if (n->start != -1) //A non-root node
    {
        //Print the label on edge from parent to current node
        print(n->start, *(n->end));
    }
    int leaf = 1;
    int i;
    for (i = 0; i < MAX_CHAR; i++)
    {
        if (n->children[i] != NULL)
        {
            if (leaf == 1 && n->start != -1)
                printf(" [%d]\n", n->suffixIndex);

```

```

//Current node is not a leaf as it has outgoing
//edges from it.
leaf = 0;
setSuffixIndexByDFS(n->children[i], labelHeight +
                     edgeLength(n->children[i]));
}
}
if (leaf == 1)
{
    for(i= n->start; i<= *(n->end); i++)
    {
        if(text[i] == '#') //Trim unwanted characters
        {
            n->end = (int*) malloc(sizeof(int));
            *(n->end) = i;
        }
    }
    n->suffixIndex = size - labelHeight;
    printf(" [%d]\n", n->suffixIndex);
}
}

void freeSuffixTreeByPostOrder(Node *n)
{
    if (n == NULL)
        return;
    int i;
    for (i = 0; i < MAX_CHAR; i++)
    {
        if (n->children[i] != NULL)
        {
            freeSuffixTreeByPostOrder(n->children[i]);
        }
    }
    if (n->suffixIndex == -1)
        free(n->end);
    free(n);
}

/*Build the suffix tree and print the edge labels along with
suffixIndex. suffixIndex for leaf edges will be >= 0 and
for non-leaf edges will be -1*/
void buildSuffixTree()
{
    size = strlen(text);
    int i;
    rootEnd = (int*) malloc(sizeof(int));
    *rootEnd = - 1;
}

```

```

/*Root is a special node with start and end indices as -1,
as it has no parent from where an edge comes to root*/
root = newNode(-1, rootEnd);

activeNode = root; //First activeNode will be root
for (i=0; i<size; i++)
    extendSuffixTree(i);
int labelHeight = 0;
setSuffixIndexByDFS(root, labelHeight);

//Free the dynamically allocated memory
freeSuffixTreeByPostOrder(root);
}

// driver program to test above functions
int main(int argc, char *argv[])
{
// strcpy(text, "xabxac#abcabxabcd$"); buildSuffixTree();
strcpy(text, "xabxa#babxba$"); buildSuffixTree();
return 0;
}

```

Output: (You can see that below output corresponds to the 2nd Figure shown above)

```

# [5]
$ [12]
a [-1]
# [4]
$ [11]
bx [-1]
a# [1]
ba$ [7]
b [-1]
a [-1]
$ [10]
bxba$ [6]
x [-1]
a# [2]
ba$ [8]
x [-1]
a [-1]
# [3]
bxa# [0]
ba$ [9]

```

If two strings are of size M and N, this implementation will take O(M+N) time and space.

If input strings are not concatenated already, then it will take $2(M+N)$ space in total, $M+N$ space to store the generalized suffix tree and another $M+N$ space to store concatenated string.

Followup:

Extend above implementation for more than two strings (i.e. concatenate all strings using unique terminal symbols and then build suffix tree for concatenated string)

One problem with this approach is the need of unique terminal symbol for each input string. This will work for few strings but if there is too many input strings, we may not be able to find that many unique terminal symbols.

We will discuss another approach to build generalized suffix tree soon where we will need only one unique terminal symbol and that will resolve the above problem and can be used to build generalized suffix tree for any number of input strings.

We have published following more articles on suffix tree applications:

- [Suffix Tree Application 1 – Substring Check](#)
- [Suffix Tree Application 2 – Searching All Patterns](#)
- [Suffix Tree Application 3 – Longest Repeated Substring](#)
- [Suffix Tree Application 4 – Build Linear Time Suffix Array](#)
- [Suffix Tree Application 5 – Longest Common Substring](#)
- [Suffix Tree Application 6 – Longest Palindromic Substring](#)

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

Source

<https://www.geeksforgeeks.org/generalized-suffix-tree-1/>

Chapter 71

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 72

Given a sequence of words, print all anagrams together Set 2

Given a sequence of words, print all anagrams together Set 2 - GeeksforGeeks

Given an array of words, print all anagrams together. For example, if the given array is {"cat", "dog", "tac", "god", "act"}, then output may be "cat tac act dog god".

We have discussed two different methods in the [previous post](#). In this post, a more efficient solution is discussed.

Trie data structure can be used for a more efficient solution. Insert the sorted order of each word in the trie. Since all the anagrams will end at the same leaf node. We can start a linked list at the leaf nodes where each node represents the index of the original array of words. Finally, traverse the Trie. While traversing the Trie, traverse each linked list one line at a time. Following are the detailed steps.

- 1) Create an empty Trie
- 2) One by one take all words of input sequence. Do following for each word
 - ...a) Copy the word to a buffer.
 - ...b) Sort the buffer
 - ...c) Insert the sorted buffer and index of this word to Trie. Each leaf node of Trie is head of a Index list. The Index list stores index of words in original sequence. If sorted buffer is already present, we insert index of this word to the index list.
- 3) Traverse Trie. While traversing, if you reach a leaf node, traverse the index list. And print all words using the index obtained from Index list.

C++

```
// An efficient program to print all anagrams together
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
```

```
#define NO_OF_CHARS 26

// Structure to represent list node for indexes of words in
// the given sequence. The list nodes are used to connect
// anagrams at leaf nodes of Trie
struct IndexNode
{
    int index;
    struct IndexNode* next;
};

// Structure to represent a Trie Node
struct TrieNode
{
    bool isEnd; // indicates end of word
    struct TrieNode* child[NO_OF_CHARS]; // 26 slots each for 'a' to 'z'
    struct IndexNode* head; // head of the index list
};

// A utility function to create a new Trie node
struct TrieNode* newTrieNode()
{
    struct TrieNode* temp = new TrieNode;
    temp->isEnd = 0;
    temp->head = NULL;
    for (int i = 0; i < NO_OF_CHARS; ++i)
        temp->child[i] = NULL;
    return temp;
}

/* Following function is needed for library function qsort(). Refer
   http://www.cplusplus.com/reference/clibrary/cstdlib/qsort/ */
int compare(const void* a, const void* b)
{   return *(char*)a - *(char*)b; }

/* A utility function to create a new linked list node */
struct IndexNode* newIndexNode(int index)
{
    struct IndexNode* temp = new IndexNode;
    temp->index = index;
    temp->next = NULL;
    return temp;
}

// A utility function to insert a word to Trie
void insert(struct TrieNode** root, char* word, int index)
{
```

```

// Base case
if (*root == NULL)
    *root = newTrieNode();

if (*word != '\0')
    insert( &(*root)->child[tolower(*word) - 'a'], word+1, index );
else // If end of the word reached
{
    // Insert index of this word to end of index linked list
    if ((*root)->isEnd)
    {
        IndexNode* pCrawl = (*root)->head;
        while( pCrawl->next )
            pCrawl = pCrawl->next;
        pCrawl->next = newIndexNode(index);
    }
    else // If Index list is empty
    {
        (*root)->isEnd = 1;
        (*root)->head = newIndexNode(index);
    }
}
}

// This function traverses the built trie. When a leaf node is reached,
// all words connected at that leaf node are anagrams. So it traverses
// the list at leaf node and uses stored index to print original words
void printAnagramsUtil(struct TrieNode* root, char *wordArr[])
{
    if (root == NULL)
        return;

    // If a lead node is reached, print all anagrams using the indexes
    // stored in index linked list
    if (root->isEnd)
    {
        // traverse the list
        IndexNode* pCrawl = root->head;
        while (pCrawl != NULL)
        {
            printf( "%s ", wordArr[ pCrawl->index ] );
            pCrawl = pCrawl->next;
        }
    }

    for (int i = 0; i < NO_OF_CHARS; ++i)
        printAnagramsUtil(root->child[i], wordArr);
}

```

```
// The main function that prints all anagrams together. wordArr[] is input
// sequence of words.
void printAnagramsTogether(char* wordArr[], int size)
{
    // Create an empty Trie
    struct TrieNode* root = NULL;

    // Iterate through all input words
    for (int i = 0; i < size; ++i)
    {
        // Create a buffer for this word and copy the word to buffer
        int len = strlen(wordArr[i]);
        char *buffer = new char[len+1];
        strcpy(buffer, wordArr[i]);

        // Sort the buffer
        qsort( (void*)buffer, strlen(buffer), sizeof(char), compare );

        // Insert the sorted buffer and its original index to Trie
        insert(&root, buffer, i);
    }

    // Traverse the built Trie and print all anagrams together
    printAnagramsUtil(root, wordArr);
}

// Driver program to test above functions
int main()
{
    char* wordArr[] = {"cat", "dog", "tac", "god", "act", "gdo"};
    int size = sizeof(wordArr) / sizeof(wordArr[0]);
    printAnagramsTogether(wordArr, size);
    return 0;
}
```

Java

```
// An efficient program to print all
// anagrams together
import java.util.Arrays;
import java.util.LinkedList;

public class GFG
{
    static final int NO_OF_CHARS = 26;
```

```
// Class to represent a Trie Node
static class TrieNode
{
    boolean isEnd; // indicates end of word

    // 26 slots each for 'a' to 'z'
    TrieNode[] child = new TrieNode[NO_OF_CHARS];

    // head of the index list
    LinkedList<Integer> head;

    // constructor
    public TrieNode()
    {
        isEnd = false;
        head = new LinkedList<>();
        for (int i = 0; i < NO_OF_CHARS; ++i)
            child[i] = null;
    }
}

// A utility function to insert a word to Trie
static TrieNode insert(TrieNode root, String word,
                      int index, int i)
{
    // Base case
    if (root == null)
    {
        root = new TrieNode();
    }

    if (i < word.length() )
    {
        int index1 = word.charAt(i) - 'a';
        root.child[index1] = insert(root.child[index1],
                                    word, index, i+1 );
    }
    else // If end of the word reached
    {
        // Insert index of this word to end of
        // index linked list
        if (root.isEnd == true)
        {
            root.head.add(index);
        }
        else // If Index list is empty
        {
            root.isEnd = true;
        }
    }
}
```

```
        root.head.add(index);
    }
}
return root;
}

// This function traverses the built trie. When a leaf
// node is reached, all words connected at that leaf
// node are anagrams. So it traverses the list at leaf
// node and uses stored index to print original words
static void printAnagramsUtil(TrieNode root,
                               String wordArr[])
{
    if (root == null)
        return;

    // If a lead node is reached, print all anagrams
    // using the indexes stored in index linked list
    if (root.isEnd)
    {
        // traverse the list
        for(Integer pCrawl: root.head)
            System.out.println(wordArr[pCrawl]);
    }

    for (int i = 0; i < NO_OF_CHARS; ++i)
        printAnagramsUtil(root.child[i], wordArr);
}

// The main function that prints all anagrams together.
// wordArr[] is input sequence of words.
static void printAnagramsTogether(String wordArr[],
                                  int size)
{
    // Create an empty Trie
    TrieNode root = null;

    // Iterate through all input words
    for (int i = 0; i < size; ++i)
    {
        // Create a buffer for this word and copy the
        // word to buffer
        char[] buffer = wordArr[i].toCharArray();

        // Sort the buffer
        Arrays.sort(buffer);

        // Insert the sorted buffer and its original
```

```
// index to Trie
root = insert(root, new String(buffer), i, 0);

}

// Traverse the built Trie and print all anagrams
// together
printAnagramsUtil(root, wordArr);
}

// Driver program to test above functions
public static void main(String args[])
{
    String wordArr[] = {"cat", "dog", "tac", "god",
                        "act", "gdo"};
    int size = wordArr.length;
    printAnagramsTogether(wordArr, size);
}
}

// This code is contributed by Sumit Ghosh
```

Output:

```
cat
tac
act
dog
god
gdo
```

Improved By : [reyaz_ahmed](#)

Source

<https://www.geeksforgeeks.org/given-a-sequence-of-words-print-all-anagrams-together-set-2/>

Chapter 73

Gomory-Hu Tree Set 1 (Introduction)

Gomory-Hu Tree Set 1 (Introduction) - GeeksforGeeks

Background :

In a flow network, an s-t cut is a cut that requires the source 's' and the sink 't' to be in different subsets, and it consists of edges going from the source's side to the sink's side. The capacity of an s-t cut is defined by the sum of capacity of each edge in the cut-set. (Source: [Wiki](#)). Given a two vertices, s and t, we can [find minimum s-t cut using max flow algorithm](#).

Since there are total $O(n^2)$ possible pairs, at first look it seems that there would be total $O(n^2)$ total minimum s-t cut values. But when we use Gomory-Hu Tree, we would see that there are total $n-1$ different cut values [A Tree with n vertices has $n-1$ edges]

Popular graph problems that can be solved using Gomory-Hu Tree :

- Given a weighted and connected graph, find [minimum s-t cut](#) for all pairs of vertices.
Or a problem like find minimum of all possible minimum s-t cuts.
- [Minimum K-Cut problem](#) : Find minimum weight set of edges whose removal would partition the graph to k connected components. This is a NP-Hard problem, Gomory-Hu Tree provides an approximate solution for this problem.

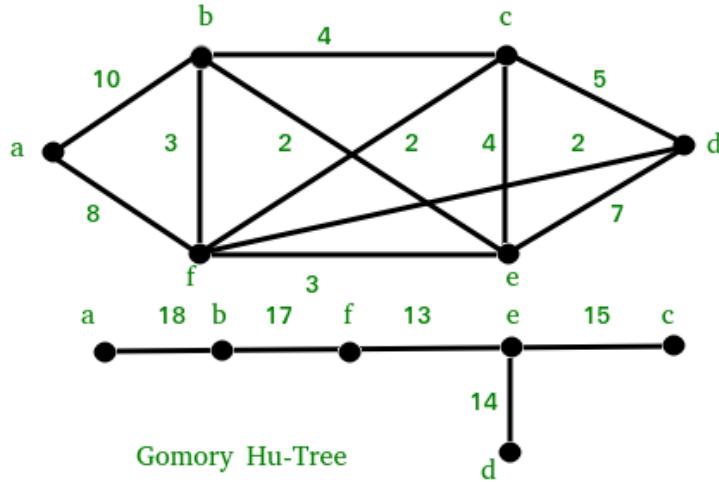
What is [Gomory-Hu Tree](#)?

A Gomory-Hu Tree is defined for a flow graph with edge capacity function c . The tree has same set of vertices as input graph and has $n-1$ (n is number of vertices) edges. Edge capacity function c' is defined using following properties:

Equivalent flow tree : For any pair of vertices s and t , the minimum s-t cut in graph is equal to the smallest capacity of the edges on the path between s and t in Tree.

Cut property : a minimum s-t cut in Tree is also a minimum cut in Graph.G

For example, consider the following Graph and Corresponding Gomory-Hu Tree.



Since there are $n-1$ edges in a tree with n nodes, we can conclude that there are at most $n-1$ different flow values in a flow network with n vertices.

We will be discussing construction of Tree in next post.

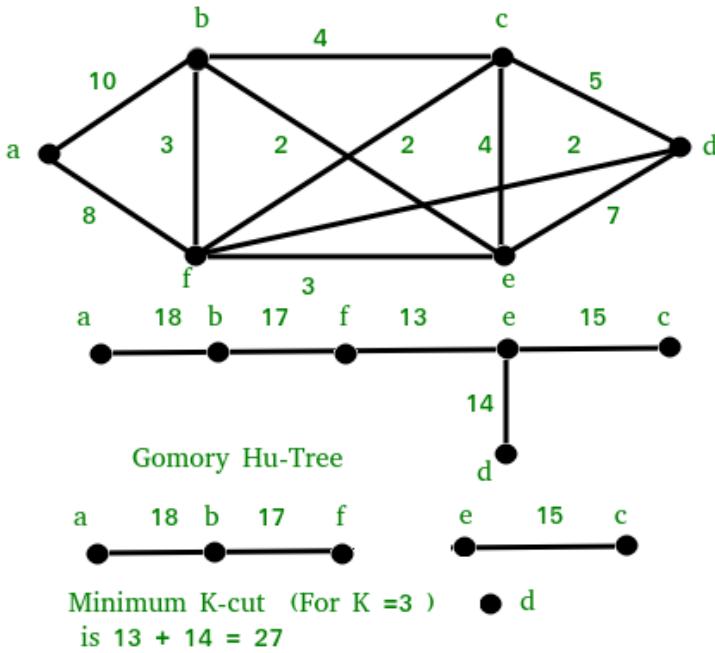
How to solve above problems using Gomory-Hu Tree is constructed?

The minimum weight edge in the tree is minimum of all s-t cuts.

We can solve the k-cut problem using below steps.

- 1) Construct Gomory-Hu Tree.
- 2) Remove $k-1$ minimum weight (or lightest) edges from the Tree.
- 3) Return union of components obtained by above removal of edges.

Below diagram illustrates above algorithm.



Note that the above solution may not always produce optimal result, but it is guaranteed to produce results within bounds of $(2-2/k)$.

References:

- <https://www.corelab.ntua.gr/seminar/material/2008-2009/2008.10.20.Gomory-Hu%20trees%20and%20applications.slides.pdf>
- https://courses.engr.illinois.edu/cs598csc/sp2009/lectures/lecture_7.pdf
- <https://cseweb.ucsd.edu/classes/fa06/cse202/Gomory-Hu%20Tree.pdf>

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

Source

<https://www.geeksforgeeks.org/gomory-hu-tree-introduction/>

Chapter 74

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 tablesize = 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*[tablesize];
        top = new hash_node*[tablesize];
        for (int i = 0; i < tablesize; i++) {
            hashtable[i] = NULL;
            top[i] = NULL;
        }
    }

    // destructor
    ~HashMap()
    {
        delete [] hashtable;
    }

    // hash function definition
    int HashFunc(int key)
    {
        return key % tablesize;
    }

    // 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 75

Heavy Light Decomposition Set 1 (Introduction)

Heavy Light Decomposition Set 1 (Introduction) - GeeksforGeeks

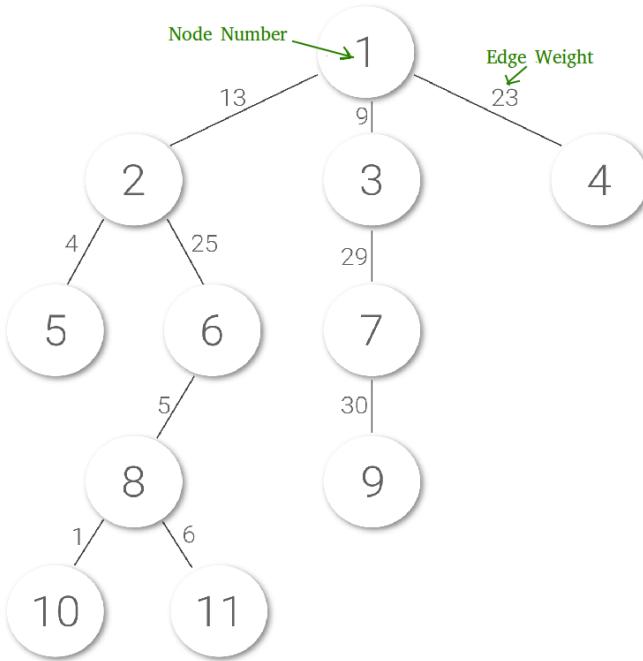
Heavy Light decomposition (HLD) is one of the [most used techniques in competitive programming](#).

Example Problem: Let us understand Heavy-light decomposition (HLD) with the help of below example.

Suppose we have **an unbalanced tree (not necessarily a Binary Tree) of n nodes**, and we have to perform operations on the tree to answer a number of queries, each can be of one of the two types:

1. **change(a, b):** Update weight of the a^{th} edge to b.
2. **maxEdge(a, b):** Print the maximum edge weight on the path from node a to node b. For example maxEdge(5, 10) should print 25.

Assume that nodes are numbered from 1 to n. There must be **n-1 edges**. Edge weights are natural numbers. Also assume that both type of queries are interspersed (approximately equal in number), and hence no type can be sidelined to compromise on complexity.



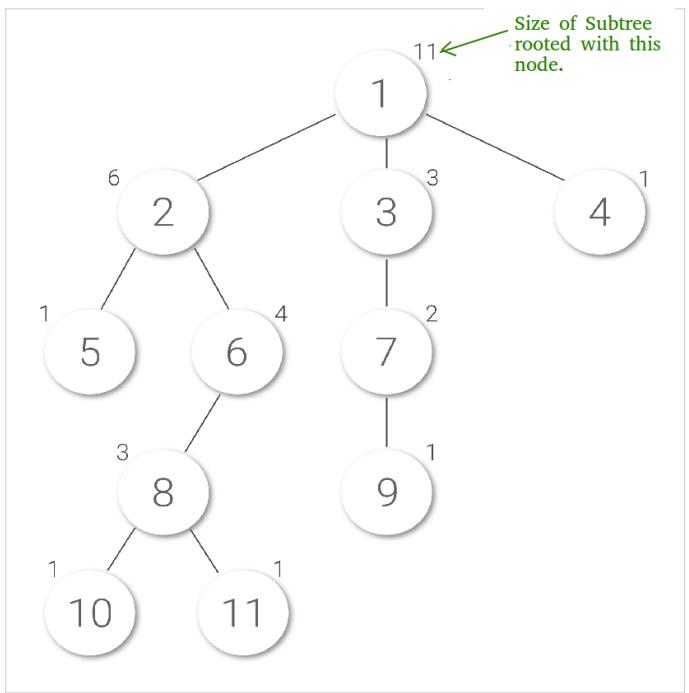
Simple Solution: A Simple solution is to traverse the complete tree for any query. Time complexity of every query in this solution is $O(n)$.

HLD Based Solution:

Upon careful observation of the two operations, we realize that we have seen them somewhere. Eureka! [Segment Trees](#). A [Segment Tree](#) can be used to perform both types in $O(\log(n))$. But wait! A [Segment Tree](#) can be built from a one-dimensional array / chain (set of nodes linked one after another), and what we have here is a tree. So, can we reduce the tree to chains?

The HLD based solution discussed in the post takes $O(\log^2(n))$ for `maxEdge()` and $O(\log n)$ for `change()`.

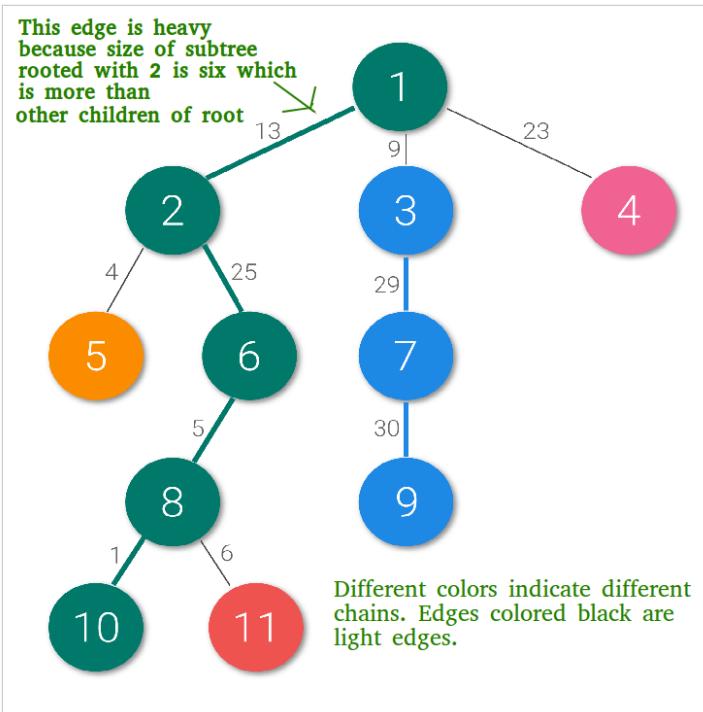
Size of a node x is number of nodes in subtree rooted with the node x . Here is an image showing subtree sizes of each node written on top of them:



HLD of a rooted tree is a method of decomposing the vertices of the tree into disjoint chains (no two chains share a node), to achieve important asymptotic time bounds for certain problems involving trees.

HLD can also be seen as ‘coloring’ of the tree’s edges. The ‘Heavy-Light’ comes from the way we segregate edges. We use size of the subtrees rooted at the nodes as our criteria.

An edge is **heavy** if $\text{size}(v) > \text{size}(u)$ where u is any sibling of v . If they come out to be equal, we pick any one such v as special.



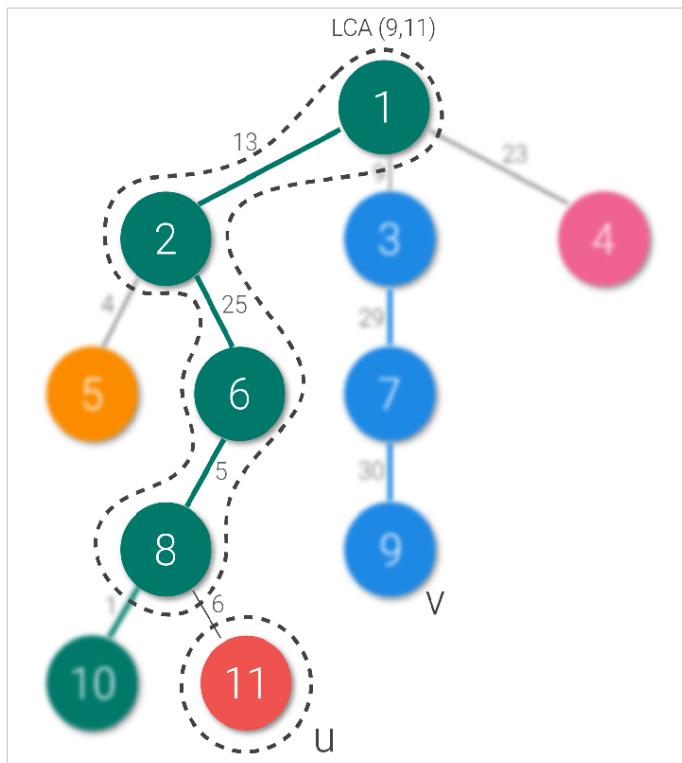
change(u, v) operation:

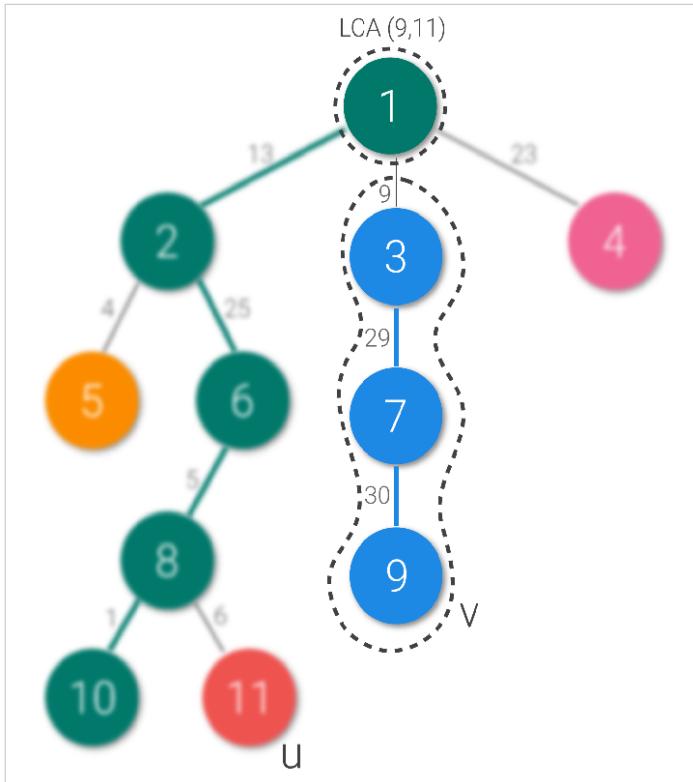
Since Segment Tree is used as underlying data structure to represent individual chains, change is done using update of segment tree. So the *time complexity of change* operation is $O(\log n)$.

maxEdge(u, v) operation:

1. We first find LCA of two nodes. Say node u is 11 and node v is 9. Their LCA is node 1.
2. Then we crawl up the tree from node u to the lca. If node u and lca belong to the same chain, we find the maximum from the range that represents the edges between them using segment tree. Else, we find the maximum from the chain to which u belongs, then change chains and repeat while we are not in the same chain.
3. We repeat the same step (as step 2) from v to lca and return maximum of two weights.

As per our example above, let's take u as node 11 and v as node 9. LCA is node 1. We move from node 11 to node 1, and we change chains once. When we change chains, we shift from our queried node to the parent of head of the chain to which it belongs (11 changes to 8 here). Similarly node 9 to node 3 queried (including the light edge), and chain changed (node changed to 1).





Time Complexity of maxEdge is $O(\log^2(n))$. Querying maximum of a chain takes $O(\log(n))$ time as chains are represented using [Segment Tree](#) and there are at-most $O(\log(n))$ chains.

How is the number of chains $O(\log(n))$?

All chains are connected by a light edge (see above examples). So the number of chains is bounded by number of light edges on any path. If we follow a light edge from the root, the subtree rooted at the resulting vertex has at most $n/2$ size. If we repeat, we land at a vertex with subtree size at most $n/4$, and so on. We conclude that **number of light edges on any path from root to leaf is at most $\log(n)$** (Source: [wcipeg](#))

The main idea of Heavy Light Decomposition of a unbalanced tree is to club vertices of long (or heavy) paths together in chains so that these linear chains can be queried in $O(\log n)$ time using a data structure like Segment Tree.

In the [next article](#), segment tree representation of chains in more detail and implementation of HLD solution for the problem is discussed as example.

[Heavy Light Decomposition Set 2 \(Implementation\)](#)

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

Source

<https://www.geeksforgeeks.org/heavy-light-decomposition-set-1-introduction/>

Chapter 76

Heavy Light Decomposition Set 2 (Implementation)

Heavy Light Decomposition Set 2 (Implementation) - GeeksforGeeks

We strongly recommend to refer below post as a prerequisite of this.

[Heavy Light Decomposition Set 1 \(Introduction\)](#)

In the above post, we discussed the Heavy-light decomposition (HLD) with the help of below example.

Suppose we have **an unbalanced tree (not necessarily a Binary Tree) of n nodes**, and we have to perform operations on the tree to answer a number of queries, each can be of one of the two types:

1. **change(a, b):** Update weight of the a^{th} edge to b.
2. **maxEdge(a, b):** Print the maximum edge weight on the path from node a to node b. For example maxEdge(5, 10) should print 25.

In this article implementation of same is discussed

Our line of attack for this problem is:

1. Creating the tree
2. Setting up the subtree size, depth and parent for each node (using a DFS)
3. Decomposing the tree into disjoint chains
4. Building up the segment tree
5. Answering queries

1. Tree Creation: Implementation uses adjacency matrix representation of the tree, for the ease of understanding. One can use adjacency list rep with some changes to the source. If edge number e with weight w exists between nodes u and v, we shall store e at $\text{tree}[u][v]$ and $\text{tree}[v][u]$, and the weight w in a separate linear array of edge weights (n-1 edges).

2. Setting up the subtree size, depth and parent for each node: Next we do a DFS on the tree to set up arrays that store parent, subtree size and depth of each node. Another important thing we do at the time of DFS is storing the deeper node of every edge we traverse. This will help us at the time of updating the tree (`change()` query) .

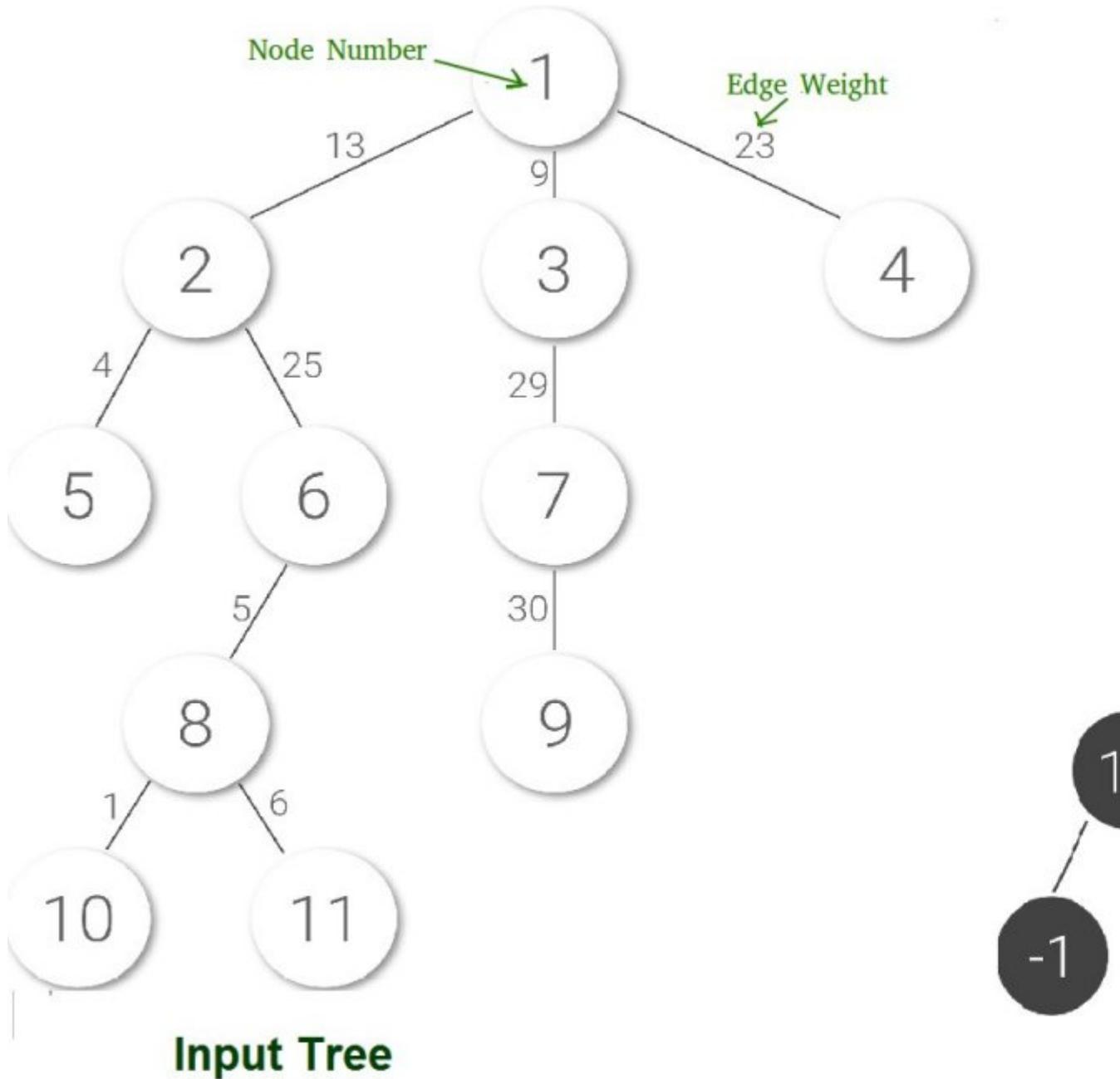
3 & 4. Decomposing the tree into disjoint chains and Building Segment Tree Now comes the most important part: HLD. As we traverse the edges and reach nodes (starting from the root), we place the edge in the segment tree base, we decide if the node will be a head to a new chain (if it is a normal child) or will the current chain extend (special child), store the chain ID to which the node belongs, and store its place in the segment tree base (for future queries). The base for segment tree is built such that all edges belonging to the same chain are together, and chains are separated by light edges.

Illustration: We start at node 1. Since there wasn't any edge by which we came to this node, we insert '-1' as the imaginary edge's weight, in the array that will act as base to the segment tree.

Next, we move to node 1's special child, which is node 2, and since we traversed edge with weight 13, we add 13 to our base array. Node 2's special child is node 6. We traverse edge with weight 25 to reach node 6. We insert in base array. And similarly we extend this chain further while we haven't reached a leaf node (node 10 in our case).

Then we shift to a normal child of parent of the last leaf node, and mark the beginning of a new chain. Parent here is node 8 and normal child is node 11. We traverse edge with weight 6 and insert it into the base array. This is how we complete the base array for the segment tree.

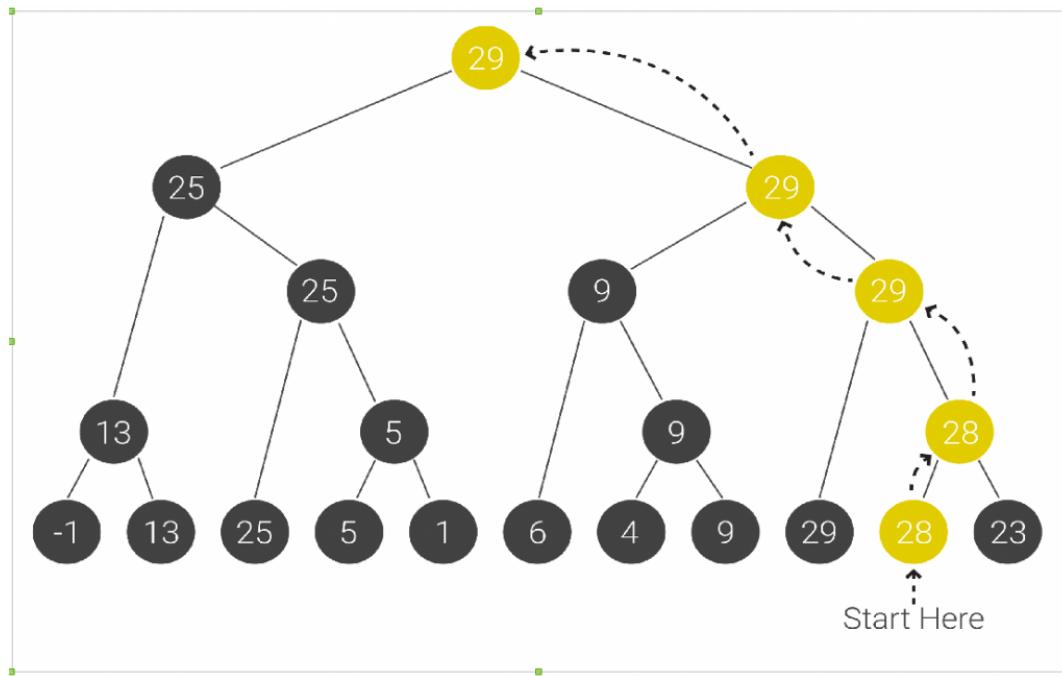
Also remember that we need to store the position of every node in segment tree for future queries. Position of node 1 is 1, node 2 is 2, node 6 is 3, node 8 is 4, ..., node 11 is 6, node 5 is 7, node 9 is 10, node 4 is 11 (1-based indexing).



5. Answering queries

We have discussed **mexEdge()** query in detail in [previous post](#). For $\text{maxEdge}(u, v)$, we find max weight edge on path from u to LCA and v to LCA and return maximum of two.

For **change()** query, we can update the segment tree by using the deeper end of the edge whose weight is to be updated. We will find the position of deeper end of the edge in the array acting as base to the segment tree and then start our update from that node and move upwards updating segment tree. Say we want to update edge 8 (between node 7 and node 9) to 28. Position of deeper node 9 in base array is 10, We do it as follows:



Below is C++ implementation of above steps.

```
/* C++ program for Heavy-Light Decomposition of a tree */
#include<bits/stdc++.h>
using namespace std;

#define N 1024

int tree[N][N]; // Matrix representing the tree

/* a tree node structure. Every node has a parent, depth,
   subtree size, chain to which it belongs and a position
   in base array*/
struct treeNode
{
    int par; // Parent of this node
    int depth; // Depth of this node
    int size; // Size of subtree rooted with this node
    int pos_segbase; // Position in segment tree base
    int chain;
}
```

```

} node[N];

/* every Edge has a weight and two ends. We store deeper end */
struct Edge
{
    int weight; // Weight of Edge
    int deeper_end; // Deeper end
} edge[N];

/* we construct one segment tree, on base array */
struct segmentTree
{
    int base_array[N], tree[6*N];
} s;

// A function to add Edges to the Tree matrix
// e is Edge ID, u and v are the two nodes, w is weight
void addEdge(int e, int u, int v, int w)
{
    /*tree as undirected graph*/
    tree[u-1][v-1] = e-1;
    tree[v-1][u-1] = e-1;

    edge[e-1].weight = w;
}

// A recursive function for DFS on the tree
// curr is the current node, prev is the parent of curr,
// dep is its depth
void dfs(int curr, int prev, int dep, int n)
{
    /* set parent of current node to predecessor*/
    node[curr].par = prev;
    node[curr].depth = dep;
    node[curr].size = 1;

    /* for node's every child */
    for (int j=0; j<n; j++)
    {
        if (j!=curr && j!=node[curr].par && tree[curr][j]!=-1)
        {
            /* set deeper end of the Edge as this child*/
            edge[tree[curr][j]].deeper_end = j;

            /* do a DFS on subtree */
            dfs(j, curr, dep+1, n);

            /* update subtree size */
        }
    }
}

```

```

        node[curr].size+=node[j].size;
    }
}
}

// A recursive function that decomposes the Tree into chains
void hld(int curr_node, int id, int *edge_counted, int *curr_chain,
         int n, int chain_heads[])
{
    /* if the current chain has no head, this node is the first node
     * and also chain head */
    if (chain_heads[*curr_chain]==-1)
        chain_heads[*curr_chain] = curr_node;

    /* set chain ID to which the node belongs */
    node[curr_node].chain = *curr_chain;

    /* set position of node in the array acting as the base to
       the segment tree */
    node[curr_node].pos_segbase = *edge_counted;

    /* update array which is the base to the segment tree */
    s.base_array[(*edge_counted)++] = edge[id].weight;

    /* Find the special child (child with maximum size) */
    int spcl_chld = -1, spcl_edg_id;
    for (int j=0; j<n; j++)
        if (j!=curr_node && j!=node[curr_node].par && tree[curr_node][j]!=-1)
            if (spcl_chld==-1 || node[spcl_chld].size < node[j].size)
                spcl_chld = j, spcl_edg_id = tree[curr_node][j];

    /* if special child found, extend chain */
    if (spcl_chld!=-1)
        hld(spcl_chld, spcl_edg_id, edge_counted, curr_chain, n, chain_heads);

    /* for every other (normal) child, do HLD on child subtree as separate
       chain*/
    for (int j=0; j<n; j++)
    {
        if (j!=curr_node && j!=node[curr_node].par &&
            j!=spcl_chld && tree[curr_node][j]!=-1)
        {
            (*curr_chain)++;
            hld(j, tree[curr_node][j], edge_counted, curr_chain, n, chain_heads);
        }
    }
}
}

```

```

// A recursive function that constructs Segment Tree for array[ss..se].
// si is index of current node in segment tree st
int construct_ST(int ss, int se, int si)
{
    // If there is one element in array, store it in current node of
    // segment tree and return
    if (ss == se-1)
    {
        s.tree[si] = s.base_array[ss];
        return s.base_array[ss];
    }

    // If there are more than one elements, then recur for left and
    // right subtrees and store the minimum of two values in this node
    int mid = (ss + se)/2;
    s.tree[si] = max(construct_ST(ss, mid, si*2),
                    construct_ST(mid, se, si*2+1));
    return s.tree[si];
}

// A recursive function that updates the Segment Tree
// x is the node to be updated to value val
// si is the starting index of the segment tree
// ss, se mark the corners of the range represented by si
int update_ST(int ss, int se, int si, int x, int val)
{
    if(ss > x || se <= x);

    else if(ss == x && se == se-1)s.tree[si] = val;

    else
    {
        int mid = (ss + se)/2;
        s.tree[si] = max(update_ST(ss, mid, si*2, x, val),
                        update_ST(mid, se, si*2+1, x, val));
    }

    return s.tree[si];
}

// A function to update Edge e's value to val in segment tree
void change(int e, int val, int n)
{
    update_ST(0, n, 1, node[edge[e].deeper_end].pos_segbase, val);

    // following lines of code make no change to our case as we are
    // changing in ST above

```

```

        // Edge_weights[e] = val;
        // segtree_Edges_weights[deeper_end_of_edge[e]] = val;
    }

    // A function to get the LCA of nodes u and v
    int LCA(int u, int v, int n)
    {
        /* array for storing path from u to root */
        int LCA_aux[n+5];

        // Set u is deeper node if it is not
        if (node[u].depth < node[v].depth)
            swap(u, v);

        /* LCA_aux will store path from node u to the root*/
        memset(LCA_aux, -1, sizeof(LCA_aux));

        while (u!=-1)
        {
            LCA_aux[u] = 1;
            u = node[u].par;
        }

        /* find first node common in path from v to root and u to
           root using LCA_aux */
        while (v)
        {
            if (LCA_aux[v]==1)break;
            v = node[v].par;
        }

        return v;
    }

    /* A recursive function to get the minimum value in a given range
       of array indexes. The following are parameters for this function.
       st    --> Pointer to segment tree
       index --> Index of current node in the segment tree. Initially
                  0 is passed as root is always at index 0
       ss & se  --> Starting and ending indexes of the segment represented
                      by current node, i.e., st[index]
       qs & qe  --> Starting and ending indexes of query range */
    int RMQUtil(int ss, int se, int qs, int qe, int index)
    {
        //printf("%d,%d,%d,%d,%d\n", ss, se, qs, qe, index);

        // If segment of this node is a part of given range, then return
        // the min of the segment
    }
}

```

```

if (qs <= ss && qe >= se-1)
    return s.tree[index];

// If segment of this node is outside the given range
if (se-1 < qs || ss > qe)
    return -1;

// If a part of this segment overlaps with the given range
int mid = (ss + se)/2;
return max(RMQUtil(ss, mid, qs, qe, 2*index),
           RMQUtil(mid, se, qs, qe, 2*index+1));
}

// Return minimum of elements in range from index qs (query start) to
// qe (query end). It mainly uses RMQUtil()
int RMQ(int qs, int qe, int n)
{
    // Check for erroneous input values
    if (qs < 0 || qe > n-1 || qs > qe)
    {
        printf("Invalid Input");
        return -1;
    }

    return RMQUtil(0, n, qs, qe, 1);
}

// A function to move from u to v keeping track of the maximum
// we move to the surface changing u and chains
// until u and v do not belong to the same
int crawl_tree(int u, int v, int n, int chain_heads[])
{
    int chain_u, chain_v = node[v].chain, ans = 0;

    while (true)
    {
        chain_u = node[u].chain;

        /* if the two nodes belong to same chain,
         * we can query between their positions in the array
         * acting as base to the segment tree. After the RMQ,
         * we can break out as we have no where further to go */
        if (chain_u==chain_v)
        {
            if (u==v);    //trivial
            else
                ans = max(RMQ(node[v].pos_segbase+1, node[u].pos_segbase, n),
                           ans);
        }
    }
}

```

```

        break;
    }

    /* else, we query between node u and head of the chain to which
       u belongs and later change u to parent of head of the chain
       to which u belongs indicating change of chain */
    else
    {
        ans = max(ans,
                   RMQ(node[chain_heads[chain_u]].pos_segbase,
                        node[u].pos_segbase, n));

        u = node[chain_heads[chain_u]].par;
    }
}

return ans;
}

// A function for MAX_EDGE query
void maxEdge(int u, int v, int n, int chain_heads[])
{
    int lca = LCA(u, v, n);
    int ans = max(crawl_tree(u, lca, n, chain_heads),
                  crawl_tree(v, lca, n, chain_heads));
    printf("%d\n", ans);
}

// driver function
int main()
{
    /* fill adjacency matrix with -1 to indicate no connections */
    memset(tree, -1, sizeof(tree));

    int n = 11;

    /* arguments in order: Edge ID, node u, node v, weight w*/
    addEdge(1, 1, 2, 13);
    addEdge(2, 1, 3, 9);
    addEdge(3, 1, 4, 23);
    addEdge(4, 2, 5, 4);
    addEdge(5, 2, 6, 25);
    addEdge(6, 3, 7, 29);
    addEdge(7, 6, 8, 5);
    addEdge(8, 7, 9, 30);
    addEdge(9, 8, 10, 1);
    addEdge(10, 8, 11, 6);
}

```

```

/* our tree is rooted at node 0 at depth 0 */
int root = 0, parent_of_root=-1, depth_of_root=0;

/* a DFS on the tree to set up:
 * arrays for parent, depth, subtree size for every node;
 * deeper end of every Edge */
dfs(root, parent_of_root, depth_of_root, n);

int chain_heads[N];

/*we have initialized no chain heads */
memset(chain_heads, -1, sizeof(chain_heads));

/* Stores number of edges for construction of segment
tree. Initially we haven't traversed any Edges. */
int edge_counted = 0;

/* we start with filling the 0th chain */
int curr_chain = 0;

/* HLD of tree */
hld(root, n-1, &edge_counted, &curr_chain, n, chain_heads);

/* ST of segregated Edges */
construct_ST(0, edge_counted, 1);

/* Since indexes are 0 based, node 11 means index 11-1,
8 means 8-1, and so on*/
int u = 11, v = 9;
cout << "Max edge between " << u << " and " << v << " is ";
maxEdge(u-1, v-1, n, chain_heads);

// Change value of edge number 8 (index 8-1) to 28
change(8-1, 28, n);

cout << "After Change: max edge between " << u << " and "
<< v << " is ";
maxEdge(u-1, v-1, n, chain_heads);

v = 4;
cout << "Max edge between " << u << " and " << v << " is ";
maxEdge(u-1, v-1, n, chain_heads);

// Change value of edge number 5 (index 5-1) to 22
change(5-1, 22, n);
cout << "After Change: max edge between " << u << " and "
<< v << " is ";
maxEdge(u-1, v-1, n, chain_heads);

```

```
    return 0;
}
```

Output:

```
Max edge between 11 and 9 is 30
After Change: max edge between 11 and 9 is 29
Max edge between 11 and 4 is 25
After Change: max edge between 11 and 4 is 23
```

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

Source

<https://www.geeksforgeeks.org/heavy-light-decomposition-set-2-implementation/>

Chapter 77

How to Implement Forward DNS Look Up Cache?

How to Implement Forward DNS Look Up Cache? - GeeksforGeeks

We have discussed [implementation of Reverse DNS Look Up Cache](#). Forward DNS look up is getting IP address for a given domain name typed in the web browser.

The cache should do the following operations :

1. Add a mapping from URL to IP address
2. Find IP address for a given URL.

There are a few changes from [reverse DNS look up cache](#) that we need to incorporate.

1. Instead of [0-9] and (.) dot we need to take care of [A-Z], [a-z] and (.) dot. As most of the domain name contains only lowercase characters we can assume that there will be [a-z] and (.) 27 children for each trie node.
2. When we type www.google.in and google.in the browser takes us to the same page. So, we need to add a domain name into trie for the words after www(.). Similarly while searching for a domain name corresponding IP address remove the www(.) if the user has provided it.

This is left as an exercise and for simplicity we have taken care of www. also.

One solution is to use [Hashing](#). In this post, a [Triebased](#) solution is discussed. One advantage of Trie based solutions is, worst case upper bound is O(1) for Trie, for hashing, the best possible average case time complexity is O(1). Also, with Trie we can implement prefix search (finding all IPs for a common prefix of URLs). The general disadvantage of Trie is large amount of memory requirement.

The idea is to store URLs in Trie nodes and store the corresponding IP address in last or leaf node.

Following is C style implementation in C++.

```
// C based program to implement reverse DNS lookup
#include<stdio.h>
```

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

// There are atmost 27 different chars in a valid URL
// assuming URL consists [a-z] and (.)
#define CHARS 27

// Maximum length of a valid URL
#define MAX 100

// A utility function to find index of child for a given character 'c'
int getIndex(char c)
{
    return (c == '.') ? 26 : (c - 'a');
}

// A utility function to find character for a given child index.
char getCharFromIndex(int i)
{
    return (i == 26) ? '.' : ('a' + i);
}

// Trie Node.
struct trienode
{
    bool isLeaf;
    char *ipAdd;
    struct trienode *child[CHARS];
};

// Function to create a new trie node.
struct trienode *newTrieNode(void)
{
    struct trienode *newNode = new trienode;
    newNode->isLeaf = false;
    newNode->ipAdd = NULL;
    for (int i = 0; i<CHARS; i++)
        newNode->child[i] = NULL;
    return newNode;
}

// This method inserts a URL and corresponding IP address
// in the trie. The last node in Trie contains the ip address.
void insert(struct trienode *root, char *URL, char *ipAdd)
{
    // Length of the URL
    int len = strlen(URL);
    struct trienode *pCrawl = root;
```

```
// Traversing over the length of the URL.
for (int level = 0; level<len; level++)
{
    // Get index of child node from current character
    // in URL[] Index must be from 0 to 26 where
    // 0 to 25 is used for alphabets and 26 for dot
    int index = getIndex(URL[level]);

    // Create a new child if not exist already
    if (!pCrawl->child[index])
        pCrawl->child[index] = newTrieNode();

    // Move to the child
    pCrawl = pCrawl->child[index];
}

//Below needs to be carried out for the last node.
//Save the corresponding ip address of the URL in the
//last node of trie.
pCrawl->isLeaf = true;
pCrawl->ipAdd = new char[strlen(ipAdd) + 1];
strcpy(pCrawl->ipAdd, ipAdd);
}

// This function returns IP address if given URL is
// present in DNS cache. Else returns NULL
char *searchDNSCache(struct trieNode *root, char *URL)
{
    // Root node of trie.
    struct trieNode *pCrawl = root;
    int len = strlen(URL);

    // Traversal over the length of URL.
    for (int level = 0; level<len; level++)
    {
        int index = getIndex(URL[level]);
        if (!pCrawl->child[index])
            return NULL;
        pCrawl = pCrawl->child[index];
    }

    // If we find the last node for a given ip address,
    // print the ip address.
    if (pCrawl != NULL && pCrawl->isLeaf)
        return pCrawl->ipAdd;

    return NULL;
```

```
}

// Driver function.
int main()
{
    char URL[] [50] = { "www.samsung.com", "www.samsung.net",
                        "www.google.in"
                      };
    char ipAdd[] [MAX] = { "107.108.11.123", "107.109.123.255",
                          "74.125.200.106"
                        };
    int n = sizeof(URL) / sizeof(URL[0]);
    struct trieNode *root = newTrieNode();

    // Inserts all the domain name and their corresponding
    // ip address
    for (int i = 0; i<n; i++)
        insert(root, URL[i], ipAdd[i]);

    // If forward DNS look up succeeds print the url along
    // with the resolved ip address.
    char url[] = "www.samsung.com";
    char *res_ip = searchDNSCache(root, url);
    if (res_ip != NULL)
        printf("Forward DNS look up resolved in cache:\n%s --> %s",
               url, res_ip);
    else
        printf("Forward DNS look up not resolved in cache ");

    return 0;
}
```

Output:

```
Forward DNS look up resolved in cache:
www.samsung.com --> 107.108.11.123
```

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/implement-forward-dns-look-cache/>

Chapter 78

How to Implement Reverse DNS Look Up Cache?

How to Implement Reverse DNS Look Up Cache? - GeeksforGeeks

Reverse DNS look up is using an internet IP address to find a domain name. For example, if you type 74.125.200.106 in browser, it automatically redirects to google.in.

How to implement Reverse DNS Look Up cache? Following are the operations needed from cache.

- 1) Add a IP address to URL Mapping in cache.
- 2) Find URL for a given IP address.

One solution is to use [Hashing](#).

In this post, a [Triebased](#) solution is discussed. One advantage of Trie based solutions is, worst case upper bound is $O(1)$ for Trie, for hashing, the best possible average case time complexity is $O(1)$. Also, with Trie we can implement prefix search (finding all urls for a common prefix of IP addresses).

The general disadvantage of Trie is large amount of memory requirement, this is not a major problem here as the alphabet size is only 11 here. Ten characters are needed for digits from '0' to '9' and one for dot ('.') .

The idea is to store IP addresses in Trie nodes and in the last node we store the corresponding domain name. Following is C style implementation in C++.

```
// C based program to implement reverse DNS lookup
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

// There are atmost 11 different chars in a valid IP address
#define CHARS 11

// Maximum length of a valid IP address
```

```
#define MAX 50

// A utility function to find index of child for a given character 'c'
int getIndex(char c) { return (c == '.')? 10: (c - '0'); }

// A utility function to find character for a given child index.
char getCharFromIndex(int i) { return (i== 10)? '.' : ('0' + i); }

// Trie Node.
struct trieNode
{
    bool isLeaf;
    char *URL;
    struct trieNode *child[CHARS];
};

// Function to create a new trie node.
struct trieNode *newTrieNode(void)
{
    struct trieNode *newNode = new trieNode;
    newNode->isLeaf = false;
    newNode->URL = NULL;
    for (int i=0; i<CHARS; i++)
        newNode->child[i] = NULL;
    return newNode;
}

// This method inserts an ip address and the corresponding
// domain name in the trie. The last node in Trie contains the URL.
void insert(struct trieNode *root, char *ipAdd, char *URL)
{
    // Length of the ip address
    int len = strlen(ipAdd);
    struct trieNode *pCrawl = root;

    // Traversing over the length of the ip address.
    for (int level=0; level<len; level++)
    {
        // Get index of child node from current character
        // in ipAdd[]. Index must be from 0 to 10 where
        // 0 to 9 is used for digits and 10 for dot
        int index = getIndex(ipAdd[level]);

        // Create a new child if not exist already
        if (!pCrawl->child[index])
            pCrawl->child[index] = newTrieNode();

        // Move to the child
        pCrawl = pCrawl->child[index];
    }
}
```

```
    pCrawl = pCrawl->child[index];
}

//Below needs to be carried out for the last node.
//Save the corresponding URL of the ip address in the
//last node of trie.
pCrawl->isLeaf = true;
pCrawl->URL = new char[strlen(URL) + 1];
strcpy(pCrawl->URL, URL);
}

// This function returns URL if given IP address is present in DNS cache.
// Else returns NULL
char *searchDNSCache(struct trieNode *root, char *ipAdd)
{
    // Root node of trie.
    struct trieNode *pCrawl = root;
    int len = strlen(ipAdd);

    // Traversal over the length of ip address.
    for (int level=0; level<len; level++)
    {
        int index = getIndex(ipAdd[level]);
        if (!pCrawl->child[index])
            return NULL;
        pCrawl = pCrawl->child[index];
    }

    // If we find the last node for a given ip address, print the URL.
    if (pCrawl!=NULL && pCrawl->isLeaf)
        return pCrawl->URL;

    return NULL;
}

//Driver function.
int main()
{
    /* Change third ipAddress for validation */
    char ipAdd[] [MAX] = {"107.108.11.123", "107.109.123.255",
                          "74.125.200.106"};
    char URL[] [50] = {"www.samsung.com", "www.samsung.net",
                       "www.google.in"};
    int n = sizeof(ipAdd)/sizeof(ipAdd[0]);
    struct trieNode *root = newTrieNode();

    // Inserts all the ip address and their corresponding
    // domain name after ip address validation.
```

```
for (int i=0; i<n; i++)
    insert(root,ipAdd[i],URL[i]);

// If reverse DNS look up succeeds print the domain
// name along with DNS resolved.
char ip[] = "107.108.11.123";
char *res_url = searchDNSCache(root, ip);
if (res_url != NULL)
    printf("Reverse DNS look up resolved in cache:\n%s --> %s",
           ip, res_url);
else
    printf("Reverse DNS look up not resolved in cache ");
return 0;
}
```

Output:

```
Reverse DNS look up resolved in cache:
107.108.11.123 --> www.samsung.com
```

Note that the above implementation of Trie assumes that the given IP address does not contain characters other than {‘0’, ‘1’,.... ‘9’, ‘?’}. What if a user gives an invalid IP address that contains some other characters? This problem can be resolved by [validating the input IP address](#) before inserting it into Trie. We can use the approach discussed [here](#) for IP address validation.

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/implement-reverse-dns-look-cache/>

Chapter 79

How to design a tiny URL or URL shortener?

How to design a tiny URL or URL shortener? - GeeksforGeeks

How to design a system that takes big URLs like “<https://www.geeksforgeeks.org/count-sum-of-digits-in-numbers-from-1-to-n/>” and converts them into a short 6 character URL. It is given that URLs are stored in database and every URL has an associated integer id.

One important thing to note is, the long url should also be uniquely identifiable from short url. So we need a [Bijective Function](#)

One **Simple Solution** could be Hashing. Use a hash function to convert long string to short string. In hashing, that may be collisions (2 long urls map to same short url) and we need a unique short url for every long url so that we can access long url back.

A **Better Solution** is to use the integer id stored in database and convert the integer to character string that is at most 6 characters long. This problem can basically seen as a base conversion problem where we have a 10 digit input number and we want to convert it into a 6 character long string.

Below is one important observation about possible characters in URL.

A URL character can be one of the following

- 1) A lower case alphabet ['a' to 'z'], total 26 characters
- 2) An upper case alphabet ['A' to 'Z'], total 26 characters
- 3) A digit ['0' to '9'], total 10 characters

There are total $26 + 26 + 10 = 62$ possible characters.

So the task is to convert a decimal number to base 62 number.

To get the original long url, we need to get url id in database. The id can be obtained using base 62 to decimal conversion.

Below is a C++ program based on this idea.

```
// C++ prgram to generate short url from intger id and
// integer id back from short url.
#include<iostream>
#include<algorithm>
#include<string>
using namespace std;

// Function to generate a short url from intger ID
string idToShortURL(long int n)
{
    // Map to store 62 possible characters
    char map[] = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ"
                 "GHIJKLMNOPQRSTUVWXYZ0123456789";

    string shorturl;

    // Convert given integer id to a base 62 number
    while (n)
    {
        // use above map to store actual character
        // in short url
        shorturl.push_back(map[n%62]);
        n = n/62;
    }

    // Reverse shortURL to complete base conversion
    reverse(shorturl.begin(), shorturl.end());

    return shorturl;
}

// Function to get integer ID back from a short url
long int shortURLtoID(string shortURL)
{
    long int id = 0; // initialize result

    // A simple base conversion logic
    for (int i=0; i < shortURL.length(); i++)
    {
        if ('a' <= shortURL[i] && shortURL[i] <= 'z')
            id = id*62 + shortURL[i] - 'a';
        if ('A' <= shortURL[i] && shortURL[i] <= 'Z')
            id = id*62 + shortURL[i] - 'A' + 26;
        if ('0' <= shortURL[i] && shortURL[i] <= '9')
            id = id*62 + shortURL[i] - '0' + 52;
    }
    return id;
}
```

```
// Driver program to test above function
int main()
{
    int n = 12345;
    string shorturl = idToShortURL(n);
    cout << "Generated short url is " << shorturl << endl;
    cout << "Id from url is " << shortURLtoID(shorturl);
    return 0;
}
```

Output:

```
Generated short url is dnh
Id from url is 12345
```

Optimization: We can avoid reverse step in idToShortURL(). To make sure that we get same ID back, we also need to change shortURLtoID() to process characters from end instead of beginning.

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

Source

<https://www.geeksforgeeks.org/how-to-design-a-tiny-url-or-url-shortener/>

Chapter 80

Image Manipulation Using Quadtrees

Image Manipulation Using Quadtrees - GeeksforGeeks

Quadtrees are an effective method to store and locate data of points in a two-dimensional plane. Another effective use of quadtrees is in the field of image manipulation.

Unlike in storage of points, in image manipulation we get a complete quadtree with the leaf nodes consisting of individual pixels of the image. Due to this, we can utilize an array to store the nodes of the tree. This leads to less memory needed (compared to linked representation) for processing.

The lowest level of the quadtree would contain N nodes equivalent to the number of pixels in the image. The next level would contain $N / 4$ nodes.

Thus, the total number of nodes necessary can be found by: $N + N / 4 + N / 16 + \dots + 1$.

To get an upperbound we can use sum of geometric progression to infinity

$\text{Nodes} = N / (1 - 1 / 4) = 4 / 3 * N$

This is the size of the array necessary.

Thus, the amount of memory needed is $O(N)$

```
class Point(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y
class Pixel(object):
    def __init__(self, color = [0, 0, 0],
                 topLeft = Point(0, 0),
                 bottomRight = Point(0, 0)):
        self.R = color[0]
        self.G = color[1]
        self.B = color[2]
        self.topLeft = topLeft
        self.bottomRight = bottomRight
```

The above code in Python shown demonstrates the class definition of a pixel.

Insertion

Unlike in a classic quadtree, for image manipulation, we can insert all the nodes in O(N) time complexity.

First, we insert all the leaf nodes directly into the last N positions of the array. The following code snippet demonstrates this:

```
# Store leaves into array
count = 0
for i in range(image.size[0] - 1, 0, -2):
    for j in range(image.size[1] - 1, 0, -2):
        self.tree[self.size - 1 - 4 * count] = Pixel(self.image[i, j],
            Point(i, j),
            Point(i, j))
        self.tree[self.size - 2 - 4 * count] = Pixel(self.image[i, j - 1],
            Point(i, j - 1),
            Point(i, j - 1))
        self.tree[self.size - 3 - 4 * count] = Pixel(self.image[i - 1, j],
            Point(i - 1, j),
            Point(i - 1, j))
        self.tree[self.size - 4 - 4 * count] = Pixel(self.image[i - 1, j - 1],
            Point(i - 1, j - 1),
            Point(i - 1, j - 1))
    count += 1
```

In the above snippet, `self.tree` is the array of nodes, `self.size` is the total size of the array, and `self.image` is the pixels of the image, `count` is used to traverse the tree.

For nodes that aren't leaves, the R, G, B values are calculated by taking the average of the values of the children.

The `topLeft` and `bottomRight` are obtained by taking the maximum and minimum values of the x and y of the children.

```
# Calculate and create parent nodes
for i in range(self.size - 4 * count - 1, -1, -1):
    self.tree[i] = Pixel(
        [(self.tree[4 * i + 1].R + self.tree[4 * i + 2].R + self.tree[4 * i + 3].R + self.tree[4
            (self.tree[4 * i + 1].G + self.tree[4 * i + 2].G + self.tree[4 * i + 3].G + self.tree[4
            (self.tree[4 * i + 1].B + self.tree[4 * i + 2].B + self.tree[4 * i + 3].B + self.tree[4
            self.tree[4 * i + 1].topLeft,
            self.tree[4 * i + 4].bottomRight)]
```

Here we can see, we take the values of R, G, B of the four children, and we divide by 4, thus getting the average.

Calculation of these values happen in O(1) time assuming values of the children are known. Since we are moving in reverse order, the values of the children are computed before that

of parents.

Thus insertion occurs in $O(N)$.

Application in Image Manipulation

Let us say we wish to **convert a high quality image to a thumbnail**. Once we have created a quadtree for the image, by selecting a height of the quadtree we can select the quality of the image we obtain. If the height is equal to the height of the quadtree, then we retain the original image. At lower heights, we obtain images of lesser quality.

In case we do not wish to reduce the quality of the image, we can try to compress the image by what is known as “pruning”. In this, leafs with colours close to that of their parents are removed. This is continuously done until no further leaves can be removed. The lowest level is then taken to form the image, using only leaf nodes. While this does not reduce the quality of the image drastically, it can lead to a small compression of the image.

Complete Code:

```

from PIL import Image
import math
class Point(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y

class Pixel(object):
    def __init__(self, color = [0, 0, 0],
                 topLeft = Point(0, 0),
                 bottomRight = Point(0, 0)):
        self.R = color[0]
        self.G = color[1]
        self.B = color[2]
        self.topLeft = topLeft
        self.bottomRight = bottomRight

class quadtree():
    def __init__(self, image):

        # Total number of nodes of tree
        self.size = 0

        # Store image pixelmap
        self.image = image.load()

        # Array of nodes
        self.tree = []
        self.x = image.size[0]
        self.y = image.size[1]

        # Total number of leaf nodes

```

```

size = image.size[0] * image.size[1]

# Count number of nodes
while(size >= 1):
    self.size += size
    size /= 4

size = image.size[0] * image.size[1]

# Initialize array elements
for i in range(0, self.size):
    self.tree.append(Pixel())

# Store leaves into array
count = 0
for i in range(image.size[0] - 1, 0, -2):
    for j in range(image.size[1] - 1, 0, -2):
        self.tree[self.size - 1 - 4 * count] = Pixel(self.image[i, j],
            Point(i, j),
            Point(i, j))
        self.tree[self.size - 2 - 4 * count] = Pixel(self.image[i, j - 1],
            Point(i, j - 1),
            Point(i, j - 1))
        self.tree[self.size - 3 - 4 * count] = Pixel(self.image[i - 1, j],
            Point(i - 1, j),
            Point(i - 1, j))
        self.tree[self.size - 4 - 4 * count] = Pixel(self.image[i - 1, j - 1],
            Point(i - 1, j - 1),
            Point(i - 1, j - 1))
        count += 1

# Calculate and create parent nodes
for i in range(self.size - 4 * count - 1, -1, -1):
    self.tree[i] = Pixel(
        [(self.tree[4 * i + 1].R + self.tree[4 * i + 2].R + self.tree[4 * i + 3].R + self.tree[4 * i + 4].R) / 4,
         (self.tree[4 * i + 1].G + self.tree[4 * i + 2].G + self.tree[4 * i + 3].G + self.tree[4 * i + 4].G) / 4,
         (self.tree[4 * i + 1].B + self.tree[4 * i + 2].B + self.tree[4 * i + 3].B + self.tree[4 * i + 4].B) / 4,
         self.tree[4 * i + 1].topLeft,
         self.tree[4 * i + 4].bottomRight])

def disp(self, level):
    start = 0

    # Calculate position of starting node of height
    for i in range(0, level):
        start = 4 * start + 1

    # Invalid height given

```

```
if (start > self.size):
    return

# Create a new image
img = Image.new("RGB", (self.x, self.y), "black")
pixels = img.load()

# Move from starting to last node on given height
for i in self.tree[start : 4 * start]:
    x1 = i.topLeft.x
    y1 = i.topLeft.y
    x2 = i.bottomRight.x
    y2 = i.bottomRight.y
    for x in range(x1, x2 + 1):
        for y in range(y1, y2 + 1):

            # Set colour
            pixels[x, y] = (i.R, i.G, i.B)

# Display image
img.show()
```

Source

<https://www.geeksforgeeks.org/image-manipulation-using-quadtrees/>

Chapter 81

Implement a Phone Directory

Implement a Phone Directory - GeeksforGeeks

Given a list of contacts which exist in a phone directory. The task is to implement search query for the phone directory. The search query on a string ‘str’ displays all the contacts which prefix as ‘str’. One special property of the search function is that, when a user searches for a contact from the contact list then suggestions (Contacts with prefix as the string entered so far) are shown after user enters each character.

Note : Contacts in the list consist of only lower case alphabets.

Example:

```
Input : contacts [] = {"gforgeeks" , "geeksquiz" }
        Query String = "gekk"

Output : Suggestions based on "g" are
         geeksquiz
         gforgeeks

         Suggestions based on "ge" are
         geeksquiz

         No Results Found for "gek"

         No Results Found for "gekk"
```

Phone Directory can be efficiently implemented using [Trie](#) Data Structure. We insert all the contacts into Trie.

Generally search query on a Trie is to determine whether the string is present or not in the trie, but in this case we are asked to find all the strings with each prefix of ‘str’. This is equivalent to doing a [DFS traversal on a graph](#). From a Trie node, visit adjacent Trie nodes and do this recursively until there are no more adjacent. This recursive function will take 2

arguments one as Trie Node which points to the current Trie Node being visited and other as the string which stores the string found so far with prefix as ‘str’.

Each Trie Node stores a boolean variable ‘isLast’ which is true if the node represents end of a contact(word).

```
// This function displays all words with given
// prefix. "node" represents last node when
// path from root follows characters of "prefix".
displayContacts (TrieNode node, string prefix)
    If (node.isLast is true)
        display prefix

        // finding adjacent nodes
    for each character 'i' in lower case Alphabets
        if (node.child[i] != NULL)
            displayContacts(node.child[i], prefix+i)
```

User will enter the string character by character and we need to display suggestions with the prefix formed after every entered character.

So one approach to find the prefix starting with the string formed is to check if the prefix exists in the Trie, if yes then call the displayContacts() function. In this approach after every entered character we check if the string exists in the Trie.

Instead of checking again and again, we can maintain a pointer **prevNode**‘ that points to the TrieNode which corresponds to the last entered character by the user, now we need to check the child node for the ‘prevNode’ when user enters another character to check if it exists in the Trie. If the new prefix is not in the Trie, then all the string which are formed by entering characters after ‘prefix’ can’t be found in Trie too. So we break the loop that is being used to generate prefixes one by one and print “No Result Found” for all remaining characters.

C++

```
// C++ Program to Implement a Phone
// Directory Using Trie Data Structure
#include <bits/stdc++.h>
using namespace std;

struct TrieNode
{
    // Each Trie Node contains a Map 'child'
    // where each alphabet points to a Trie
    // Node.
    // We can also use a fixed size array of
    // size 256.
    unordered_map<char, TrieNode*> child;

    // 'isLast' is true if the node represents
    // end of a contact
```

```
bool isLast;

// Default Constructor
TrieNode()
{
    // Initialize all the Trie nodes with NULL
    for (char i = 'a'; i <= 'z'; i++)
        child[i] = NULL;

    isLast = false;
}
};

// Making root NULL for ease so that it doesn't
// have to be passed to all functions.
TrieNode *root = NULL;

// Insert a Contact into the Trie
void insert(string s)
{
    int len = s.length();

    // 'itr' is used to iterate the Trie Nodes
    TrieNode *itr = root;
    for (int i = 0; i < len; i++)
    {
        // Check if the s[i] is already present in
        // Trie
        TrieNode *nextNode = itr->child[s[i]];
        if (nextNode == NULL)
        {
            // If not found then create a new TrieNode
            nextNode = new TrieNode();

            // Insert into the Map
            itr->child[s[i]] = nextNode;
        }

        // Move the iterator('itr') ,to point to next
        // Trie Node
        itr = nextNode;
    }

    // If its the last character of the string 's'
    // then mark 'isLast' as true
    if (i == len - 1)
        itr->isLast = true;
}
}
```

```
// This function simply displays all dictionary words
// going through current node. String 'prefix'
// represents string corresponding to the path from
// root to curNode.
void displayContactsUtil(TrieNode *curNode, string prefix)
{
    // Check if the string 'prefix' ends at this Node
    // If yes then display the string found so far
    if (curNode->isLast)
        cout << prefix << endl;

    // Find all the adjacent Nodes to the current
    // Node and then call the function recursively
    // This is similar to performing DFS on a graph
    for (char i = 'a'; i <= 'z'; i++)
    {
        TrieNode *nextNode = curNode->child[i];
        if (nextNode != NULL)
            displayContactsUtil(nextNode, prefix + (char)i);
    }
}

// Display suggestions after every character enter by
// the user for a given query string 'str'
void displayContacts(string str)
{
    TrieNode *prevNode = root;

    string prefix = "";
    int len = str.length();

    // Display the contact List for string formed
    // after entering every character
    int i;
    for (i=0; i<len; i++)
    {
        // 'prefix' stores the string formed so far
        prefix += (char)str[i];

        // Get the last character entered
        char lastChar = prefix[i];

        // Find the Node corresponding to the last
        // character of 'prefix' which is pointed by
        // prevNode of the Trie
        TrieNode *curNode = prevNode->child[lastChar];
```

```

// If nothing found, then break the loop as
// no more prefixes are going to be present.
if (curNode == NULL)
{
    cout << "nNo Results Found for " << prefix
        << "" n";
    i++;
    break;
}

// If present in trie then display all
// the contacts with given prefix.
cout << "nSuggestions based on " << prefix
    << "" are n";
displayContactsUtil(curNode, prefix);

// Change prevNode for next prefix
prevNode = curNode;
}

// Once search fails for a prefix, we print
// "Not Results Found" for all remaining
// characters of current query string "str".
for (; i<len; i++)
{
    prefix += (char)str[i];
    cout << "nNo Results Found for " << prefix
        << "" n";
}
}

// Insert all the Contacts into the Trie
void insertIntoTrie(string contacts[],int n)
{
    // Initialize root Node
    root = new TrieNode();

    // Insert each contact into the trie
    for (int i = 0; i < n; i++)
        insert(contacts[i]);
}

// Driver program to test above functions
int main()
{
    // Contact list of the User
    string contacts[] = {"gforgeeks" , "geeksquiz"};

```

```
// Size of the Contact List
int n = sizeof(contacts)/sizeof(string);

// Insert all the Contacts into Trie
insertIntoTrie(contacts, n);

string query = "gekk";

// Note that the user will enter 'g' then 'e', so
// first display all the strings with prefix as 'g'
// and then all the strings with prefix as 'ge'
displayContacts(query);

return 0;
}
```

Java

```
// Java Program to Implement a Phone
// Directory Using Trie Data Structure
import java.util.*;

class TrieNode
{
    // Each Trie Node contains a Map 'child'
    // where each alphabet points to a Trie
    // Node.
    HashMap<Character,TrieNode> child;

    // 'isLast' is true if the node represents
    // end of a contact
    boolean isLast;

    // Default Constructor
    public TrieNode()
    {
        child = new HashMap<Character,TrieNode>();

        // Initialize all the Trie nodes with NULL
        for (char i = 'a'; i <= 'z'; i++)
            child.put(i,null);

        isLast = false;
    }
}

class Trie
{
```

```
TrieNode root;

// Insert all the Contacts into the Trie
public void insertIntoTrie(String contacts[])
{
    root = new TrieNode();
    int n = contacts.length;
    for (int i = 0; i < n; i++)
    {
        insert(contacts[i]);
    }
}

// Insert a Contact into the Trie
public void insert(String s)
{
    int len = s.length();

    // 'itr' is used to iterate the Trie Nodes
    TrieNode itr = root;
    for (int i = 0; i < len; i++)
    {
        // Check if the s[i] is already present in
        // Trie
        TrieNode nextNode = itr.child.get(s.charAt(i));
        if (nextNode == null)
        {
            // If not found then create a new TrieNode
            nextNode = new TrieNode();

            // Insert into the HashMap
            itr.child.put(s.charAt(i),nextNode);
        }

        // Move the iterator('itr') ,to point to next
        // Trie Node
        itr = nextNode;

        // If its the last character of the string 's'
        // then mark 'isLast' as true
        if (i == len - 1)
            itr.isLast = true;
    }
}

// This function simply displays all dictionary words
// going through current node. String 'prefix'
// represents string corresponding to the path from
```

```
// root to curNode.
public void displayContactsUtil(TrieNode curNode,
                                String prefix)
{
    // Check if the string 'prefix' ends at this Node
    // If yes then display the string found so far
    if (curNode.isLast)
        System.out.println(prefix);

    // Find all the adjacent Nodes to the current
    // Node and then call the function recursively
    // This is similar to performing DFS on a graph
    for (char i = 'a'; i <= 'z'; i++)
    {
        TrieNode nextNode = curNode.child.get(i);
        if (nextNode != null)
        {
            displayContactsUtil(nextNode, prefix + i);
        }
    }
}

// Display suggestions after every character enter by
// the user for a given string 'str'
void displayContacts(String str)
{
    TrieNode prevNode = root;

    // 'flag' denotes whether the string entered
    // so far is present in the Contact List

    String prefix = "";
    int len = str.length();

    // Display the contact List for string formed
    // after entering every character
    int i;
    for (i = 0; i < len; i++)
    {
        // 'str' stores the string entered so far
        prefix += str.charAt(i);

        // Get the last character entered
        char lastChar = prefix.charAt(i);

        // Find the Node corresponding to the last
        // character of 'str' which is pointed by
```

```

// prevNode of the Trie
TrieNode curNode = prevNode.child.get(lastChar);

// If nothing found, then break the loop as
// no more prefixes are going to be present.
if (curNode == null)
{
    System.out.println("nNo Results Found for "
                       + prefix + "'''");
    i++;
    break;
}

// If present in trie then display all
// the contacts with given prefix.
System.out.println("nSuggestions based on "
                   + prefix + "" are");
displayContactsUtil(curNode, prefix);

// Change prevNode for next prefix
prevNode = curNode;
}

for ( ; i < len; i++)
{
    prefix += str.charAt(i);
    System.out.println("nNo Results Found for "
                       + prefix + "'''");
}
}

// Driver code
class Main
{
    public static void main(String args[])
    {
        Trie trie = new Trie();

        String contacts [] = {"gforgeeks", "geeksquiz"};

        trie.insertIntoTrie(contacts);

        String query = "gekk";

        // Note that the user will enter 'g' then 'e' so
        // first display all the strings with prefix as 'g'
        // and then all the strings with prefix as 'ge'
    }
}

```

```
        trie.displayContacts(query);
    }
}
```

Output:

```
Suggestions based on "g" are
geeksquiz
gforgeeks
```

```
Suggestions based on "ge" are
geeksquiz
```

```
No Results Found for "gek"
```

```
No Results Found for "gekk"
```

Source

<https://www.geeksforgeeks.org/implement-a-phone-directory/>

Chapter 82

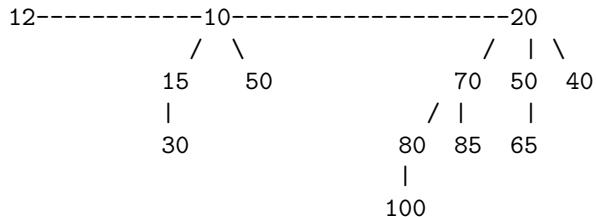
Implementation of Binomial Heap Set – 2 (delete() and decreaseKey())

Implementation of Binomial Heap Set - 2 (delete() and decreaseKey()) - GeeksforGeeks

In [previous post](#) i.e. Set 1 we have discussed that implements these below functions:

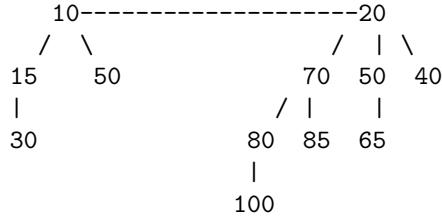
1. **insert(H, k)**: Inserts a key ‘k’ to Binomial Heap ‘H’. This operation first creates a Binomial Heap with single key ‘k’, then calls union on H and the new Binomial heap.
 2. **getMin(H)**: A simple way to getMin() is to traverse the list of root of Binomial Trees and return the minimum key. This implementation requires O(Logn) time. It can be optimized to O(1) by maintaining a pointer to minimum key root.
 3. **extractMin(H)**: This operation also uses union(). We first call getMin() to find the minimum key Binomial Tree, then we remove the node and create a new Binomial Heap by connecting all subtrees of the removed minimum node. Finally we call union() on H and the newly created Binomial Heap. This operation requires O(Logn) time.

Examples:



A Binomial Heap with 13 nodes. It is a collection of 3

Binomial Trees of orders 0, 2 and 3 from left to right.



In this post, below functions are implemented.

1. **delete(H):** Like Binary Heap, delete operation first reduces the key to minus infinite, then calls extractMin().
2. **decreaseKey(H):** decreaseKey() is also similar to Binary Heap. We compare the decreases key with it parent and if parent's key is more, we swap keys and recur for parent. We stop when we either reach a node whose parent has smaller key or we hit the root node. Time complexity of decreaseKey() is O(Logn)

```

// C++ program for implementation of
// Binomial Heap and Operations on it
#include <bits/stdc++.h>
using namespace std;

// Structure of Node
struct Node
{
    int val, degree;
    Node *parent, *child, *sibling;
};

// Making root global to avoid one extra
// parameter in all functions.
Node *root = NULL;

// link two heaps by making h1 a child
// of h2.
int binomialLink(Node *h1, Node *h2)
{
    h1->parent = h2;
    h1->sibling = h2->child;
    h2->child = h1;
    h2->degree = h2->degree + 1;
}

// create a Node
Node *createNode(int n)
  
```

```

{
    Node *new_node = new Node;
    new_node->val = n;
    new_node->parent = NULL;
    new_node->sibling = NULL;
    new_node->child = NULL;
    new_node->degree = 0;
    return new_node;
}

// This function merge two Binomial Trees
Node *mergeBHeaps(Node *h1, Node *h2)
{
    if (h1 == NULL)
        return h2;
    if (h2 == NULL)
        return h1;

    // define a Node
    Node *res = NULL;

    // check degree of both Node i.e.
    // which is greater or smaller
    if (h1->degree <= h2->degree)
        res = h1;

    else if (h1->degree > h2->degree)
        res = h2;

    // traverse till if any of heap gets empty
    while (h1 != NULL && h2 != NULL)
    {
        // if degree of h1 is smaller, increment h1
        if (h1->degree < h2->degree)
            h1 = h1->sibling;

        // Link h1 with h2 in case of equal degree
        else if (h1->degree == h2->degree)
        {
            Node *sib = h1->sibling;
            h1->sibling = h2;
            h1 = sib;
        }

        // if h2 is greater
        else
        {
            Node *sib = h2->sibling;

```

```

        h2->sibling = h1;
        h2 = sib;
    }
}
return res;
}

// This function perform union operation on two
// binomial heap i.e. h1 & h2
Node *unionBHeaps(Node *h1, Node *h2)
{
    if (h1 == NULL && h2 == NULL)
        return NULL;

    Node *res = mergeBHeaps(h1, h2);

    // Traverse the merged list and set
    // values according to the degree of
    // Nodes
    Node *prev = NULL, *curr = res,
          *next = curr->sibling;
    while (next != NULL)
    {
        if ((curr->degree != next->degree) ||
            ((next->sibling != NULL) &&
             (next->sibling)->degree ==
              curr->degree))
        {
            prev = curr;
            curr = next;
        }

        else
        {
            if (curr->val <= next->val)
            {
                curr->sibling = next->sibling;
                binomialLink(next, curr);
            }
            else
            {
                if (prev == NULL)
                    res = next;
                else
                    prev->sibling = next;
                binomialLink(curr, next);
                curr = next;
            }
        }
    }
}

```

```
        }
        next = curr->sibling;
    }
    return res;
}

// Function to insert a Node
void binomialHeapInsert(int x)
{
    // Create a new node and do union of
    // this node with root
    root = unionBHeaps(root, createNode(x));
}

// Function to display the Nodes
void display(Node *h)
{
    while (h)
    {
        cout << h->val << " ";
        display(h->child);
        h = h->sibling;
    }
}

// Function to reverse a list
// using recursion.
int revertList(Node *h)
{
    if (h->sibling != NULL)
    {
        revertList(h->sibling);
        (h->sibling)->sibling = h;
    }
    else
        root = h;
}

// Function to extract minimum value
Node *extractMinBHeap(Node *h)
{
    if (h == NULL)
        return NULL;

    Node *min_node_prev = NULL;
    Node *min_node = h;

    // Find minimum value
```

```

int min = h->val;
Node *curr = h;
while (curr->sibling != NULL)
{
    if ((curr->sibling)->val < min)
    {
        min = (curr->sibling)->val;
        min_node_prev = curr;
        min_node = curr->sibling;
    }
    curr = curr->sibling;
}

// If there is a single Node
if (min_node_prev == NULL &&
    min_node->sibling == NULL)
    h = NULL;

else if (min_node_prev == NULL)
    h = min_node->sibling;

// Remove min node from list
else
    min_node_prev->sibling = min_node->sibling;

// Set root (which is global) as children
// list of min node
if (min_node->child != NULL)
{
    revertList(min_node->child);
    (min_node->child)->sibling = NULL;
}

// Do union of root h and children
return unionBHeaps(h, root);
}

// Function to search for an element
Node *findNode(Node *h, int val)
{
    if (h == NULL)
        return NULL;

    // check if key is equal to the root's data
    if (h->val == val)
        return h;

    // Recur for child

```

```
Node *res = findNode(h->child, val);
if (res != NULL)
    return res;

    return findNode(h->sibling, val);
}

// Function to decrease the value of old_val
// to new_val
void decreaseKeyBHeap(Node *H, int old_val,
                      int new_val)
{
    // First check element present or not
    Node *node = findNode(H, old_val);

    // return if Node is not present
    if (node == NULL)
        return;

    // Reduce the value to the minimum
    node->val = new_val;
    Node *parent = node->parent;

    // Update the heap according to reduced value
    while (parent != NULL && node->val < parent->val)
    {
        swap(node->val, parent->val);
        node = parent;
        parent = parent->parent;
    }
}

// Function to delete an element
Node *binomialHeapDelete(Node *h, int val)
{
    // Check if heap is empty or not
    if (h == NULL)
        return NULL;

    // Reduce the value of element to minimum
    decreaseKeyBHeap(h, val, INT_MIN);

    // Delete the minimum element from heap
    return extractMinBHeap(h);
}

// Driver code
int main()
```

```
{  
    // Note that root is global  
    binomialHeapInsert(10);  
    binomialHeapInsert(20);  
    binomialHeapInsert(30);  
    binomialHeapInsert(40);  
    binomialHeapInsert(50);  
  
    cout << "The heap is:\n";  
    display(root);  
  
    // Delete a particular element from heap  
    root = binomialHeapDelete(root, 10);  
  
    cout << "\nAfter deleing 10, the heap is:\n";  
    display(root);  
  
    return 0;  
}
```

Output:

```
The heap is:  
50 10 30 40 20  
After deleing 10, the heap is:  
20 30 40 50
```

Source

<https://www.geeksforgeeks.org/implementation-binomial-heap-set-2/>

Chapter 83

Inclusion Exclusion principle and programming applications

Inclusion Exclusion principle and programming applications - GeeksforGeeks

Sum Rule – If a task can be done in one of A_1 ways or one of A_2 ways, where none of the set of A_1 ways is the same as any of the set of A_2 ways, then there are $A_1 + A_2$ ways to do the task.

The sum-rule mentioned above states that if there are multiple sets of ways of doing a task, there shouldn't be any way that is common between two sets of ways because if there is, it would be counted twice and the enumeration would be wrong.

The principle of **inclusion-exclusion** says that in order to count only unique ways of doing a task, we must add the number of ways to do it in one way and the number of ways to do it in another and then subtract the number of ways to do the task that are common to both sets of ways.

The principle of inclusion-exclusion is also known as the **subtraction principle**. For two sets of ways A_1 and A_2 , the enumeration would like-

Below are some examples to explain the application of inclusion-exclusion principle:

- **Example 1:** How many binary strings of length 8 either start with a '1' bit or end with two bits '00'?

Solution: If the string starts with one, there are 7 characters left which can be filled in $2^7 = 128$ ways.

If the string ends with '00' then 6 characters can be filled in $2^6 = 64$ ways.

Now if we add the above sets of ways and conclude that it is the final answer, then it would be wrong. This is because there are strings with start with '1' and end with

'00' both, and since they satisfy both criteria they are counted twice.

So we need to subtract such strings to get a correct count.

Strings that start with '1' and end with '00' have five characters that can be filled in

~~2⁵~~ = ~~32~~ ways.

So by the inclusion-exclusion principle we get-

Total strings = $128 + 64 - 32 = 160$

- **Example 2:** How many numbers between 1 and 1000, including both, are divisible by 3 or 4?

Solution: Number of numbers divisible by 3 = ~~333~~ = 333

Number of numbers divisible by 4 = ~~250~~ = 250

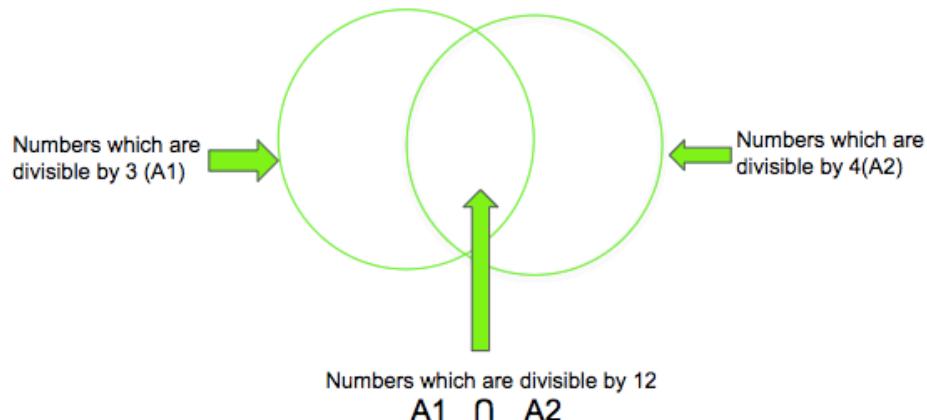
Number of numbers divisible by 3 and 4 = ~~83~~ = 83

Therefore, number of numbers divisible by 3 or 4 = ~~333 + 250 - 83~~ = 333 + 250 - 83 = 500

Implementation

Problem 1: How many numbers between 1 and 1000, including both, are divisible by 3 or 4?

The Approach will be the one discussed above, we add the number of numbers that are divisible by 3 and 4 and subtract the numbers which are divisible by 12.



C++

```
// CPP program to count the
// number of numbers between
// 1 and 1000, including both,
// that are divisible by 3 or 4
#include <bits/stdc++.h>
using namespace std;

// function to count the divisors
int countDivisors(int N, int a, int b)
{
    // Counts of numbers
    // divisible by a and b
    int count1 = N / a;
    int count2 = N / b;

    // inclusion-exclusion
    // principle applied
    int count3 = (N / (a * b));

    return count1 + count2 - count3;
}

// Driver Code
int main()
{
    int N = 1000, a = 3, b = 4;
    cout << countDivisors(N, a, b);
    return 0;
}
```

Java

```
// Java program to count the
// number of numbers between
// 1 and 1000, including both,
// that are divisible by 3 or 4
import java.io.*;

class GFG
{

    // function to count the divisors
    public static int countDivisors(int N,
                                    int a,
                                    int b)
    {
        // Counts of numbers
        // divisible by a and b
```

```
int count1 = N / a;
int count2 = N / b;

// inclusion-exclusion
// principle applied
int count3 = (N / (a * b));

return count1 + count2 - count3;
}

// Driver Code
public static void main (String[] args)
{
    int N = 1000, a = 3, b = 4;
    System.out.println (countDivisors(N, a, b));
}
}

// This code is contributed by m_kit
```

C#

```
// C# program to count the
// number of numbers between
// 1 and 1000, including both,
// that are divisible by 3 or 4
using System;

class GFG
{

    // function to count
    // the divisors
    public static int countDivisors(int N,
                                    int a,
                                    int b)
    {
        // Counts of numbers
        // divisible by a and b
        int count1 = N / a;
        int count2 = N / b;

        // inclusion-exclusion
        // principle applied
        int count3 = (N / (a * b));

        return count1 + count2 - count3;
    }
}
```

```
// Driver Code
static public void Main ()
{
    int N = 1000, a = 3, b = 4;
    Console.WriteLine(countDivisors(N, a, b));
}
}

// This code is contributed by aj_36
```

PHP

```
<?php
// PHP program to count the
// number of numbers between
// 1 and 1000, including both,
// that are divisible by 3 or 4

// function to count the divisors
function countDivisors($N, $a, $b)
{
    // Counts of numbers
    // divisible by a and b
    $count1 = $N / $a;
    $count2 = $N / $b;

    // inclusion-exclusion
    // principle applied
    $count3 = ($N / ($a * $b));

    return $count1 + $count2 - $count3;
}

// Driver Code
$N = 1000; $a = 3; $b = 4;
echo countDivisors($N, $a, $b);

// This code is contributed by aj_36
?>
```

Output :

500

.

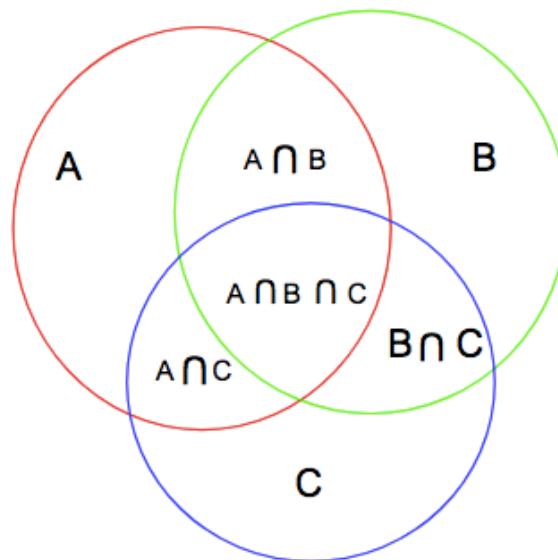
Problem 2: Given N prime numbers and a number M, find out how many numbers from 1 to M are divisible by any of the N given prime numbers.

Examples :

Input: N numbers = {2, 3, 5, 7} M = 100
Output: 78

Input: N numbers = {2, 5, 7, 11} M = 200
Output: 69

The approach for this problem will be to generate all the possible combinations of numbers using N prime numbers using power set in 2^N . For each of the given prime numbers P_i among N, it has M/P_i multiples. Suppose M=10, and we are given with 3 prime numbers(2, 3, 5), then the total count of multiples when we do $10/2 + 10/3 + 10/5$ is 11. Since we are counting 6 and 10 twice, the count of multiples in range 1-M comes 11. Using inclusion-exclusion principle, we can get the correct number of multiples. The inclusion-exclusion principle for three terms can be described as:



$$|A \cup B \cup C| = |A| + |B| + |C| - |A \cap B| - |A \cap C| - |B \cap C| + |A \cap B \cap C|$$

Similarly, for every N numbers, we can easily find the total number of multiples in range 1

to M by applying the formula for an intersection of N numbers. **The numbers that are formed by multiplication of an odd number of prime numbers will be added and the numbers formed by multiplication of even numbers will thus be subtracted to get the total number of multiples in the range 1 to M.**

Using power set we can easily get all the combination of numbers formed by the given prime numbers. To know if the number is formed by multiplication of odd or even numbers, simply count the number of set bits in all the possible combinations ($1-1<<N$).

Using power sets and adding the numbers created by combinations of odd and even prime numbers we get 123 and 45 respectively. Using **inclusion-exclusion principle** we get the number of numbers in range 1-M that is divided by any one of N prime numbers is **(odd combinations-even combinations) = (123-45) = 78**.

Below is the implementation of the above idea:

C++

```
// CPP program to count the
// number of numbers in range
// 1-M that are divisible by
// given N prime numbers
#include <bits/stdc++.h>
using namespace std;

// function to count the number
// of numbers in range 1-M that
// are divisible by given N
// prime numbers
int count(int a[], int m, int n)
{
    int odd = 0, even = 0;
    int counter, i, j, p = 1;
    int pow_set_size = (1 << n);

    // Run from counter 000..0 to 111..1
    for (counter = 1; counter < pow_set_size;
         counter++)
    {
        p = 1;
        for (j = 0; j < n; j++)
        {

            // Check if jth bit in the
            // counter is set If set
            // then pront jth element from set
            if (counter & (1 << j))
            {
                p *= a[j];
            }
        }
    }
}
```

```
// if set bits is odd, then add to
// the number of multiples
if (__builtin_popcount(counter) & 1)
    odd += (100 / p);
else
    even += 100 / p;
}

return odd - even;
}
// Driver Code
int main()
{
    int a[] = { 2, 3, 5, 7 };
    int m = 100;
    int n = sizeof(a) / sizeof(a[0]);
    cout << count(a, m, n);
    return 0;
}
```

Output:

78

Time Complexity : $O(2^N \cdot N)$

Improved By : [jit_t](#)

Source

<https://www.geeksforgeeks.org/inclusion-exclusion-principle-and-programming-applications/>

Chapter 84

Insertion at Specific Position in a Circular Doubly Linked List

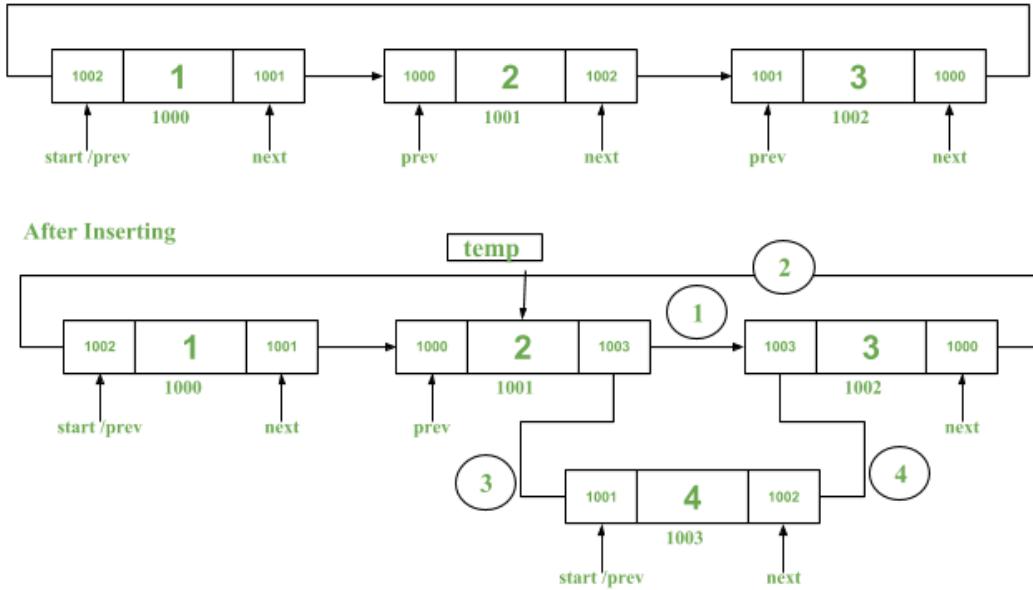
Insertion at Specific Position in a Circular Doubly Linked List - GeeksforGeeks

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;
    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: $O(n)$ => for counting the list, $O(n)$ => 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 85

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 86

Interval Tree

Interval Tree - GeeksforGeeks

Consider a situation where we have a set of intervals and we need following operations to be implemented efficiently.

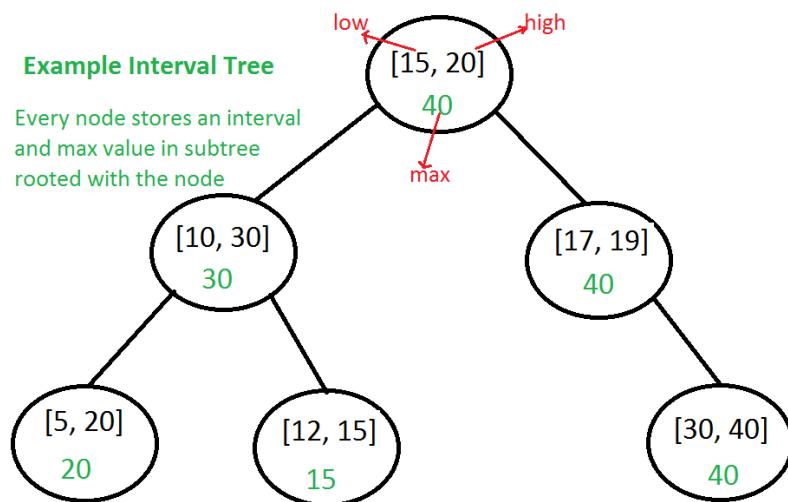
- 1) Add an interval
- 2) Remove an interval
- 3) Given an interval x , find if x overlaps with any of the existing intervals.

Interval Tree: The idea is to augment a self-balancing Binary Search Tree (BST) like [Red Black Tree](#), [AVL Tree](#), etc to maintain set of intervals so that all operations can be done in $O(\log n)$ time.

Every node of Interval Tree stores following information.

- a) **i:** An interval which is represented as a pair $[low, high]$
- b) **max:** Maximum $high$ value in subtree rooted with this node.

The low value of an interval is used as key to maintain order in BST. The insert and delete operations are same as insert and delete in self-balancing BST used.



The main operation is to search for an overlapping interval. Following is algorithm for searching an overlapping interval x in an Interval tree rooted with $root$.

```
Interval overlappingIntervalSearch(root, x)
1) If x overlaps with root's interval, return the root's interval.

2) If left child of root is not empty and the max in left child
is greater than x's low value, recur for left child

3) Else recur for right child.
```

How does the above algorithm work?

Let the interval to be searched be x . We need to prove this in for following two cases.

Case 1: When we go to right subtree, one of the following must be true.

- a) There is an overlap in right subtree: This is fine as we need to return one overlapping interval.
- b) There is no overlap in either subtree: We go to right subtree only when either left is NULL or maximum value in left is smaller than $x.low$. So the interval cannot be present in left subtree.

Case 2: When we go to left subtree, one of the following must be true.

- a) There is an overlap in left subtree: This is fine as we need to return one overlapping interval.
- b) There is no overlap in either subtree: This is the most important part. We need to consider following facts.
 - ... We went to left subtree because $x.low \leq max$ in left subtree
 - ... max in left subtree is a high of one of the intervals let us say $[a, max]$ in left subtree.
 - Since x doesn't overlap with any node in left subtree $x.low$ must be smaller than ' a '.
 - All nodes in BST are ordered by low value, so all nodes in right subtree must have low value greater than ' a '.
 - From above two facts, we can say all intervals in right subtree have low value greater than $x.low$. So x cannot overlap with any interval in right subtree.

Implementation of Interval Tree:

Following is C++ implementation of Interval Tree. The implementation uses basic [insert](#) operation of BST to keep things simple. Ideally it should be [insertion of AVL Tree](#) or [insertion of Red-Black Tree](#). [Deletion from BST](#) is left as an exercise.

```
#include <iostream>
using namespace std;

// Structure to represent an interval
struct Interval
{
    int low, high;
};
```

```

// Structure to represent a node in Interval Search Tree
struct ITNode
{
    Interval *i; // 'i' could also be a normal variable
    int max;
    ITNode *left, *right;
};

// A utility function to create a new Interval Search Tree Node
ITNode * newNode(Interval i)
{
    ITNode *temp = new ITNode;
    temp->i = new Interval(i);
    temp->max = i.high;
    temp->left = temp->right = NULL;
}

// A utility function to insert a new Interval Search Tree Node
// This is similar to BST Insert. Here the low value of interval
// is used to maintain BST property
ITNode *insert(ITNode *root, Interval i)
{
    // Base case: Tree is empty, new node becomes root
    if (root == NULL)
        return newNode(i);

    // Get low value of interval at root
    int l = root->i->low;

    // If root's low value is smaller, then new interval goes to
    // left subtree
    if (i.low < l)
        root->left = insert(root->left, i);

    // Else, new node goes to right subtree.
    else
        root->right = insert(root->right, i);

    // Update the max value of this ancestor if needed
    if (root->max < i.high)
        root->max = i.high;
}

return root;
}

// A utility function to check if given two intervals overlap
bool doOverlap(Interval i1, Interval i2)
{

```

```

        if (i1.low <= i2.high && i2.low <= i1.high)
            return true;
        return false;
    }

// The main function that searches a given interval i in a given
// Interval Tree.
Interval *overlapSearch(ITNode *root, Interval i)
{
    // Base Case, tree is empty
    if (root == NULL) return NULL;

    // If given interval overlaps with root
    if (doOverlap(*(root->i), i))
        return root->i;

    // If left child of root is present and max of left child is
    // greater than or equal to given interval, then i may
    // overlap with an interval in left subtree
    if (root->left != NULL && root->left->max >= i.low)
        return overlapSearch(root->left, i);

    // Else interval can only overlap with right subtree
    return overlapSearch(root->right, i);
}

void inorder(ITNode *root)
{
    if (root == NULL) return;

    inorder(root->left);

    cout << "[" << root->i->low << ", " << root->i->high << "]"
        << " max = " << root->max << endl;

    inorder(root->right);
}

// Driver program to test above functions
int main()
{
    // Let us create interval tree shown in above figure
    Interval ints[] = {{15, 20}, {10, 30}, {17, 19},
        {5, 20}, {12, 15}, {30, 40}};
    int n = sizeof(ints)/sizeof(ints[0]);
    ITNode *root = NULL;
    for (int i = 0; i < n; i++)
}

```

```
root = insert(root, ints[i]);  
  
cout << "Inorder traversal of constructed Interval Tree is\n";  
inorder(root);  
  
Interval x = {6, 7};  
  
cout << "\nSearching for interval [" << x.low << "," << x.high << "]";  
Interval *res = overlapSearch(root, x);  
if (res == NULL)  
    cout << "\nNo Overlapping Interval";  
else  
    cout << "\nOverlaps with [" << res->low << ", " << res->high << "]";  
return 0;  
}
```

Output:

```
Inorder traversal of constructed Interval Tree is  
[5, 20] max = 20  
[10, 30] max = 30  
[12, 15] max = 15  
[15, 20] max = 40  
[17, 19] max = 40  
[30, 40] max = 40
```

```
Searching for interval [6,7]  
Overlaps with [5, 20]
```

Applications of Interval Tree:

Interval tree is mainly a geometric data structure and often used for windowing queries, for instance, to find all roads on a computerized map inside a rectangular viewport, or to find all visible elements inside a three-dimensional scene (Source [Wiki](#)).

Interval Tree vs Segment Tree

Both segment and interval trees store intervals. Segment tree is mainly optimized for queries for a given point, and interval trees are mainly optimized for overlapping queries for a given interval.

Exercise:

- 1) Implement delete operation for interval tree.
- 2) Extend the intervalSearch() to print all overlapping intervals instead of just one.

http://en.wikipedia.org/wiki/Interval_tree

<http://www.cse.unr.edu/~mgunes/cs302/IntervalTrees.pptx>

Introduction to Algorithms 3rd Edition by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest

<https://www.youtube.com/watch?v=dQF0zyaym8A>

Source

<https://www.geeksforgeeks.org/interval-tree/>

Chapter 87

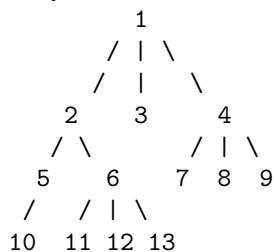
Iterative Preorder Traversal of an N-ary Tree

Iterative Preorder Traversal of an N-ary Tree - GeeksforGeeks

Given a K-ary Tree. The task is to write an iterative program to perform the [preorder traversal](#) of the given n-ary tree.

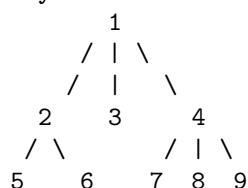
Examples:

Input: 3-Array Tree



Output: 1 2 5 10 6 11 12 13 3 4 7 8 9

Input: 3-Array Tree



Output: 1 2 5 6 3 4 7 8 9

Preorder Traversal of an N-ary Tree is similar to the preorder traversal of Binary Search Tree or Binary Tree with the only difference that is, all the child nodes of a parent are traversed from left to right in a sequence.

Iterative Preorder Traversal of Binary Tree.

Cases to handle during traversal: Two Cases have been taken care of in this Iterative Preorder Traversal Algorithm:

1. If Top of the stack is a leaf node then remove it from the stack
2. If Top of the stack is Parent with children:
 - As soon as an unvisited child is found(left to right sequence), Push it to Stack and Store it in Auxillary List and mark the following child as visited.Then, start again from Case-1, to explore this newly visited child.
 - If all Child nodes from left to right of a Parent has been visited then remove the Parent from the stack.

Note: In the below python implementation, a “dequeue” is used to implement the stack instead of a list because of its efficient append and pop operations.

Below is the implementation of the above approach:

```
# Python program for Iterative Preorder
# Traversal of N-ary Tree.
# Preorder: Root, print children
# from left to right.

from collections import deque

# Node Structure of K-ary Tree
class NewNode():

    def __init__(self, val):
        self.key = val
        # all children are stored in a list
        self.child = []

    # Utility function to print the
    # preorder of the given K-Ary Tree
    def preorderTraversal(root):

        Stack = deque([])
        # 'Preorder'-> contains all the
        # visited nodes.
```

```

Preorder = []
Preorder.append(root.key)
Stack.append(root)
while len(Stack)>0:
    # 'Flag' checks whether all the child
    # nodes have been visited.
    flag = 0
    # CASE 1- If Top of the stack is a leaf
    # node then remove it from the stack:
    if len((Stack[len(Stack)-1]).child)== 0:
        X = Stack.pop()
        # CASE 2- If Top of the stack is
        # Parent with children:
    else:
        Par = Stack[len(Stack)-1]
        # a)As soon as an unvisited child is
        # found(left to right sequence),
        # Push it to Stack and Store it in
        # Auxillary List(Marked Visited)
        # Start Again from Case-1, to explore
        # this newly visited child
        for i in range(0, len(Par.child)):
            if Par.child[i].key not in Preorder:
                flag = 1
                Stack.append(Par.child[i])
                Preorder.append(Par.child[i].key)
                break;
            # b)If all Child nodes from left to right
            # of a Parent have been visited
            # then remove the parent from the stack.
    if flag == 0:
        Stack.pop()
print(Preorder)

# Execution Start From here
if __name__=='__main__':
# input nodes
    ...
    1
    / | \
    / | \
    2 3 4
    / \ / | \
    5 6 7 8 9
    / / | \
    10 11 12 13

```

```
    ...
root = NewNode(1)
root.child.append(NewNode(2))
root.child.append(NewNode(3))
root.child.append(NewNode(4))
root.child[0].child.append(NewNode(5))
root.child[0].child[0].child.append(NewNode(10))
root.child[0].child.append(NewNode(6))
root.child[0].child[1].child.append(NewNode(11))
root.child[0].child[1].child.append(NewNode(12))
root.child[0].child[1].child.append(NewNode(13))
root.child[2].child.append(NewNode(7))
root.child[2].child.append(NewNode(8))
root.child[2].child.append(NewNode(9))

preorderTraversal(root)
```

Output:

```
[1, 2, 5, 10, 6, 11, 12, 13, 3, 4, 7, 8, 9]
```

Source

<https://www.geeksforgeeks.org/iterative-preorder-traversal-of-a-n-ary-tree/>

Chapter 88

Iterative Segment Tree (Range Maximum Query with Node Update)

Iterative Segment Tree (Range Maximum Query with Node Update) - GeeksforGeeks

Given an array $\text{arr}[0 \dots n-1]$. The task is to perform the following operation:

- Find the maximum of elements from index l to r where $0 \leq l \leq r \leq n-1$.
- Change value of a specified element of the array to a new value x . Given i and x , change $\text{A}[i]$ to x , $0 \leq i \leq n-1$.

Examples:

Input: $\text{a}[] = \{2, 6, 7, 5, 18, 86, 54, 2\}$

Query1: maximum(2, 7)

Query2: update(3, 90)

Query3: maximum(2, 6)

Output:

Maximum in range 2 to 7 is 86.

Maximum in range 2 to 6 is 90.

We have discussed [Recursive segment tree implementation](#). In this post, [iterative implementation](#) is discussed. The iterative version of the segment tree basically uses the fact, that for an index i , left child $= 2 * i$ and right child $= 2 * i + 1$ in the tree. The parent for an index i in the segment tree array can be found by $\text{parent} = i / 2$. Thus we can easily travel up and down through the levels of the tree one by one. At first we compute the maximum in the ranges while constructing the tree starting from the leaf nodes and climbing up through the levels one by one. We use the same concept while processing the queries for finding the

maximum in a range. Since there are $(\log n)$ levels in the worst case, so querying takes $\log n$ time. For update of a particular index to a given value we start updating the segment tree starting from the leaf nodes and update all those nodes which are affected by the updation of the current node by gradually moving up through the levels at every iteration. Updation also takes $\log n$ time because there we have to update all the levels starting from the leaf node where we update the exact value at the exact index given by the user.

Below is the implementation of the above approach.

```

// C++ Program to implement
// iterative segment tree.
#include <bits/stdc++.h>
using namespace std;

void construct_segment_tree(vector<int>& segtree,
                           vector<int>& a, int n)
{
    // assign values to leaves of the segment tree
    for (int i = 0; i < n; i++)
        segtree[n + i] = a[i];

    /* assign values to internal nodes
       to compute maximum in a given range */
    for (int i = n - 1; i >= 1; i--)
        segtree[i] = max(segtree[2 * i],
                         segtree[2 * i + 1]);
}

void update(vector<int>& segtree, int pos, int value,
            int n)
{
    // change the index to leaf node first
    pos += n;

    // update the value at the leaf node
    // at the exact index
    segtree[pos] = value;

    while (pos > 1) {

        // move up one level at a time in the tree
        pos >>= 1;

        // update the values in the nodes in
        // the next higher level
        segtree[pos] = max(segtree[2 * pos],
                           segtree[2 * pos + 1]);
    }
}

```

```

int range_query(vector<int>& segtree, int left, int
                    right,
                    int n)
{
    /* Basically the left and right indices will move
       towards right and left respectively and with
       every each next higher level and compute the
       maximum at each height. */
    // change the index to leaf node first
    left += n;
    right += n;

    // initialize maximum to a very low value
    int ma = INT_MIN;

    while (left < right) {

        // if left index in odd
        if (left & 1) {
            ma = max(ma, segtree[left]);

            // make left index even
            left++;
        }

        // if right index in odd
        if (right & 1) {

            // make right index even
            right--;

            ma = max(ma, segtree[right]);
        }

        // move to the next higher level
        left /= 2;
        right /= 2;
    }
    return ma;
}

// Driver code
int main()
{
    vector<int> a = { 2, 6, 10, 4, 7, 28, 9, 11, 6, 33 };
    int n = a.size();
}

```

```
/* Construct the segment tree by assigning
the values to the internal nodes*/
vector<int> segtree(2 * n);
construct_segment_tree(segtree, a, n);

// compute maximum in the range left to right
int left = 1, right = 5;
cout << "Maximum in range " << left << " to "
    << right << " is " << range_query(segtree, left,
                                         right + 1, n)
    << "\n";

// update the value of index 5 to 32
int index = 5, value = 32;

// a[5] = 32;
// Contents of array : {2, 6, 10, 4, 7, 32, 9, 11, 6, 33}
update(segtree, index, value, n);

// compute maximum in the range left to right
left = 2, right = 8;
cout << "Maximum in range " << left << " to "
    << right << " is " << range_query(segtree,
                                         left, right + 1, n)
    << "\n";

return 0;
}
```

Output:

```
Maximum in range 1 to 5 is 28
Maximum in range 2 to 8 is 32
```

Time Complexity: $(N * \log N)$
Auxiliary Space: $O(N)$

Source

<https://www.geeksforgeeks.org/iterative-segment-tree-range-maximum-query-with-node-update/>

Chapter 89

Iterative Segment Tree (Range Minimum Query)

Iterative Segment Tree (Range Minimum Query) - GeeksforGeeks

We have discussed [recursive segment tree implementation](#). In this post, iterative implementation is discussed.

Let us consider the following problem understand Segment Trees.

We have an array $\text{arr}[0 \dots n-1]$. We should be able to

- 1 Find the minimum of elements from index l to r where $0 \leq l \leq r \leq n-1$
- 2 Change value of a specified element of the array to a new value x. We need to do $\text{arr}[i] = x$ where $0 \leq i \leq n-1$.

Examples:

```
Input : 2, 6, 7, 5, 18, 86, 54, 2
        minimum(2, 7)
        update(3, 4)
        minimum(2, 6)
Output : Minimum in range 2 to 7 is 2.
         Minimum in range 2 to 6 is 4.
```

The iterative version of the segment tree basically uses the fact, that for an index i, left child $= 2 * i$ and right child $= 2 * i + 1$ in the tree. The parent for an index i in the segment tree array can be found by parent $= i / 2$. Thus we can easily travel up and down through the levels of the tree one by one. At first we compute the minimum in the ranges while constructing the tree starting from the leaf nodes and climbing up through the levels one by one. We use the same concept while processing the queries for finding the minimum in a range. Since there are $(\log n)$ levels in the worst case, so querying takes $\log n$ time. For update of a particular index to a given value we start updating the segment tree starting from the leaf nodes and update all those nodes which are affected by the updation of the

current node by gradually moving up through the levels at every iteration. Updation also takes $\log n$ time because there we have to update all the levels starting from the leaf node where we update the exact value at the exact index given by the user.

```

// CPP Program to implement iterative segment
// tree.
#include <bits/stdc++.h>
#define ll long long

using namespace std;

void construct_segment_tree(vector<int>& segtree,
                           vector<int> &a, int n)
{
    // assign values to leaves of the segment tree
    for (int i = 0; i < n; i++)
        segtree[n + i] = a[i];

    /* assign values to internal nodes
       to compute minimum in a given range */
    for (int i = n - 1; i >= 1; i--)
        segtree[i] = min(segtree[2 * i],
                         segtree[2 * i + 1]);
}

void update(vector<int>& segtree, int pos, int value,
            int n)
{
    // change the index to leaf node first
    pos += n;

    // update the value at the leaf node
    // at the exact index
    segtree[pos] = value;

    while (pos > 1) {

        // move up one level at a time in the tree
        pos >>= 1;

        // update the values in the nodes in
        // the next higher level
        segtree[pos] = min(segtree[2 * pos],
                           segtree[2 * pos + 1]);
    }
}

int range_query(vector<int>& segtree, int left, int

```

```

                    right, int n)
{
    /* Basically the left and right indices will move
       towards right and left respectively and with
       every each next higher level and compute the
       minimum at each height. */
    // change the index to leaf node first
    left += n;
    right += n;

    // initialize minimum to a very high value
    int mi = (int)1e9;

    while (left < right) {

        // if left index in odd
        if (left & 1) {
            mi = min(mi, segtree[left]);

            // make left index even
            left++;
        }

        // if right index in odd
        if (right & 1) {

            // make right index even
            right--;

            mi = min(mi, segtree[right]);
        }

        // move to the next higher level
        left /= 2;
        right /= 2;
    }
    return mi;
}

// Driver code
int main()
{
    vector<int> a = { 2, 6, 10, 4, 7, 28, 9, 11, 6, 33 };
    int n = a.size();

    /* Construct the segment tree by assigning
       the values to the internal nodes*/
    vector<int> segtree(2 * n);
}

```

```
construct_segment_tree(segtree, a, n);

// compute minimum in the range left to right
int left = 0, right = 5;
cout << "Minimum in range " << left << " to "
    << right << " is " << range_query(segtree, left,
                                         right + 1, n) << "\n";

// update the value of index 3 to 1
int index = 3, value = 1;

// a[3] = 1;
// Contents of array : {2, 6, 10, 1, 7, 28, 9, 11, 6, 33}
update(segtree, index, value, n); // point update

// compute minimum in the range left to right
left = 2, right = 6;
cout << "Minimum in range " << left << " to "
    << right << " is " << range_query(segtree,
                                         left, right + 1, n) << "\n";

return 0;
}
```

Output:

```
Minimum in range 0 to 5 is 2
Minimum in range 2 to 6 is 1
```

Time Complexity($n \log n$)
Auxiliary Space (n)

Source

<https://www.geeksforgeeks.org/iterative-segment-tree-range-minimum-query/>

Chapter 90

K Dimensional Tree Set 1 (Search and Insert)

K Dimensional Tree Set 1 (Search and Insert) - GeeksforGeeks

A K-D Tree(also called as K-Dimensional Tree) is a binary search tree where data in each node is a K-Dimensional point in space. In short, it is a space partitioning(details below) data structure for organizing points in a K-Dimensional space.

A non-leaf node in K-D tree divides the space into two parts, called as half-spaces.

Points to the left of this space are represented by the left subtree of that node and points to the right of the space are represented by the right subtree. We will soon be explaining the concept on how the space is divided and tree is formed.

For the sake of simplicity, let us understand a 2-D Tree with an example.

The root would have an x-aligned plane, the root's children would both have y-aligned planes, the root's grandchildren would all have x-aligned planes, and the root's great-grandchildren would all have y-aligned planes and so on.

Generalization:

Let us number the planes as 0, 1, 2, ...($K - 1$). From the above example, it is quite clear that a point (node) at depth D will have A aligned plane where A is calculated as:

$$A = D \bmod K$$

How to determine if a point will lie in the left subtree or in right subtree?

If the root node is aligned in planeA, then the left subtree will contain all points whose coordinates in that plane are smaller than that of root node. Similarly, the right subtree will contain all points whose coordinates in that plane are greater-equal to that of root node.

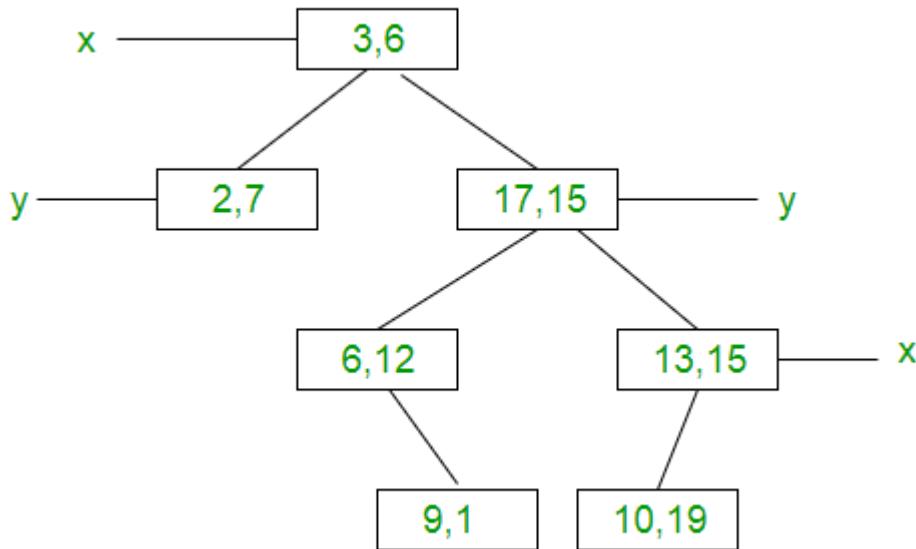
Creation of a 2-D Tree:

Consider following points in a 2-D plane:

(3, 6), (17, 15), (13, 15), (6, 12), (9, 1), (2, 7), (10, 19)

1. Insert (3, 6): Since tree is empty, make it the root node.

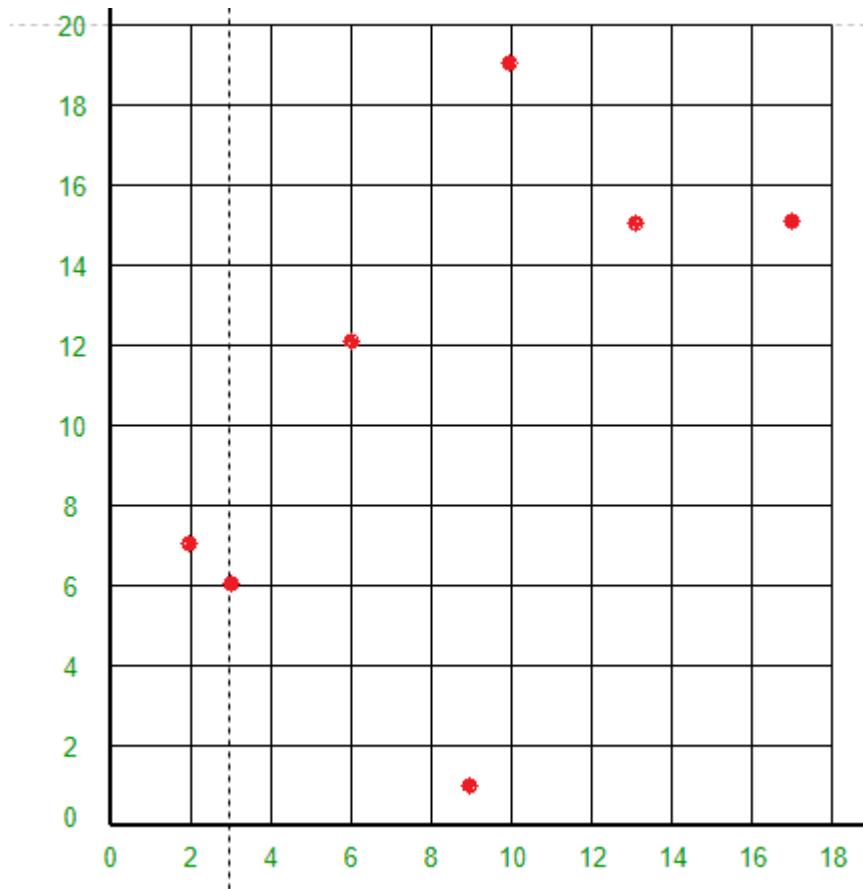
2. Insert (17, 15): Compare it with root node point. Since root node is X-aligned, the X-coordinate value will be compared to determine if it lies in the right subtree or in the left subtree. This point will be Y-aligned.
3. Insert (13, 15): X-value of this point is greater than X-value of point in root node. So, this will lie in the right subtree of (3, 6). Again Compare Y-value of this point with the Y-value of point (17, 15) (Why?). Since, they are equal, this point will lie in the right subtree of (17, 15). This point will be X-aligned.
4. Insert (6, 12): X-value of this point is greater than X-value of point in root node. So, this will lie in the right subtree of (3, 6). Again Compare Y-value of this point with the Y-value of point (17, 15) (Why?). Since, $12 < 15$, this point will lie in the left subtree of (17, 15). This point will be X-aligned.
5. Insert (9, 1): Similarly, this point will lie in the right of (6, 12).
6. Insert (2, 7): Similarly, this point will lie in the left of (3, 6).
7. Insert (10, 19): Similarly, this point will lie in the left of (13, 15).



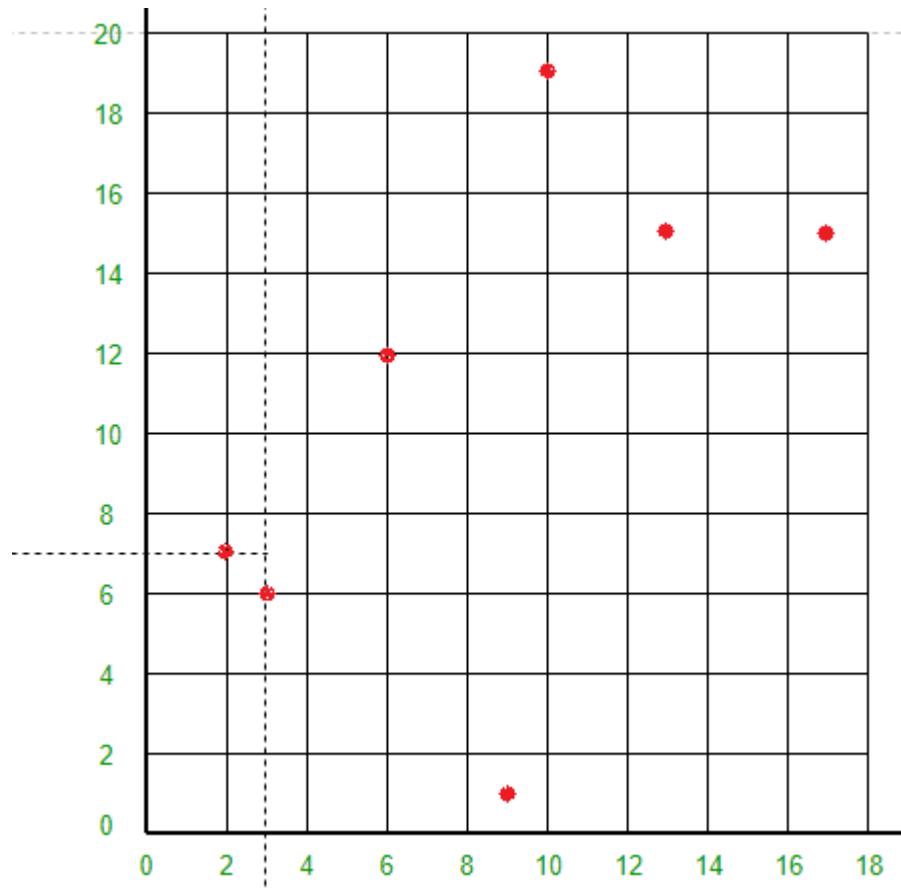
How is space partitioned?

All 7 points will be plotted in the X-Y plane as follows:

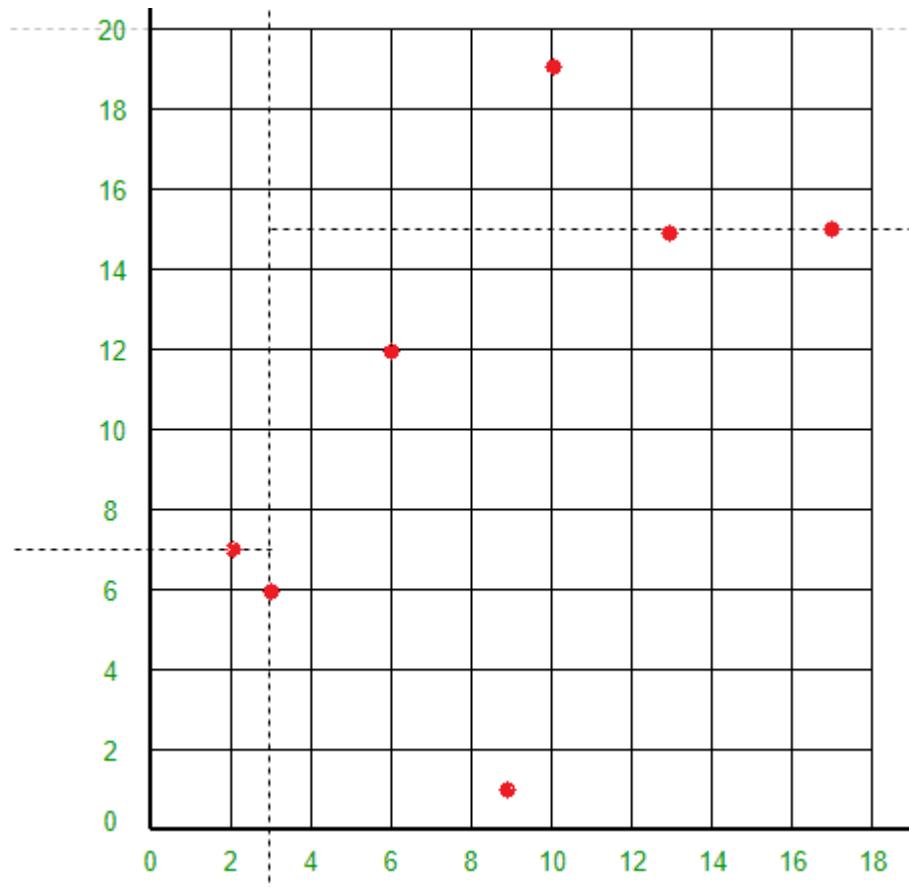
1. Point (3, 6) will divide the space into two parts: Draw line $X = 3$.



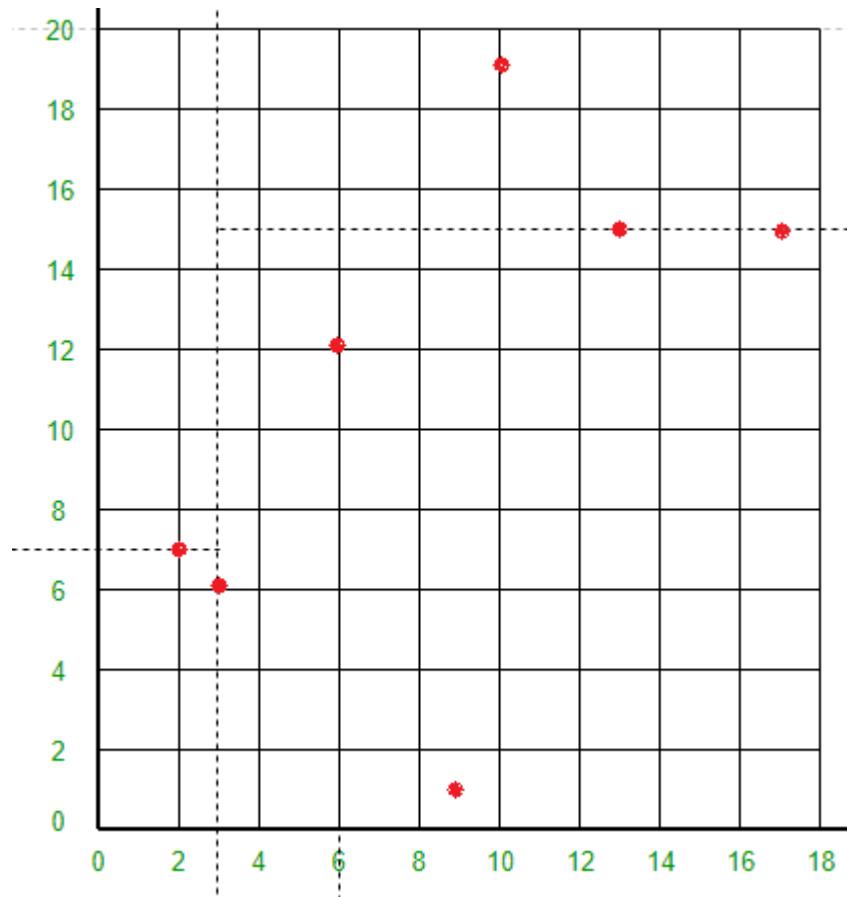
2. Point (2, 7) will divide the space to the left of line $X = 3$ into two parts horizontally.
Draw line $Y = 7$ to the left of line $X = 3$.



3. Point (17, 15) will divide the space to the right of line $X = 3$ into two parts horizontally.
Draw line $Y = 15$ to the right of line $X = 3$.

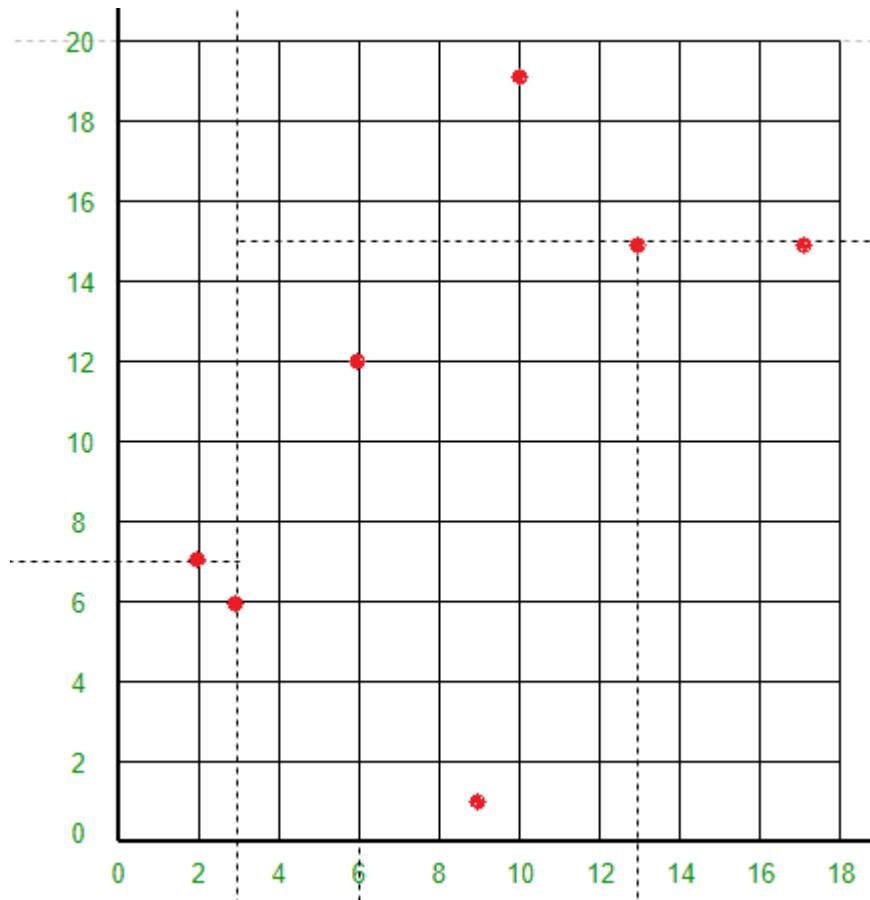


- Point (6, 12) will divide the space below line $Y = 15$ and to the right of line $X = 3$ into two parts.
Draw line $X = 6$ to the right of line $X = 3$ and below line $Y = 15$.

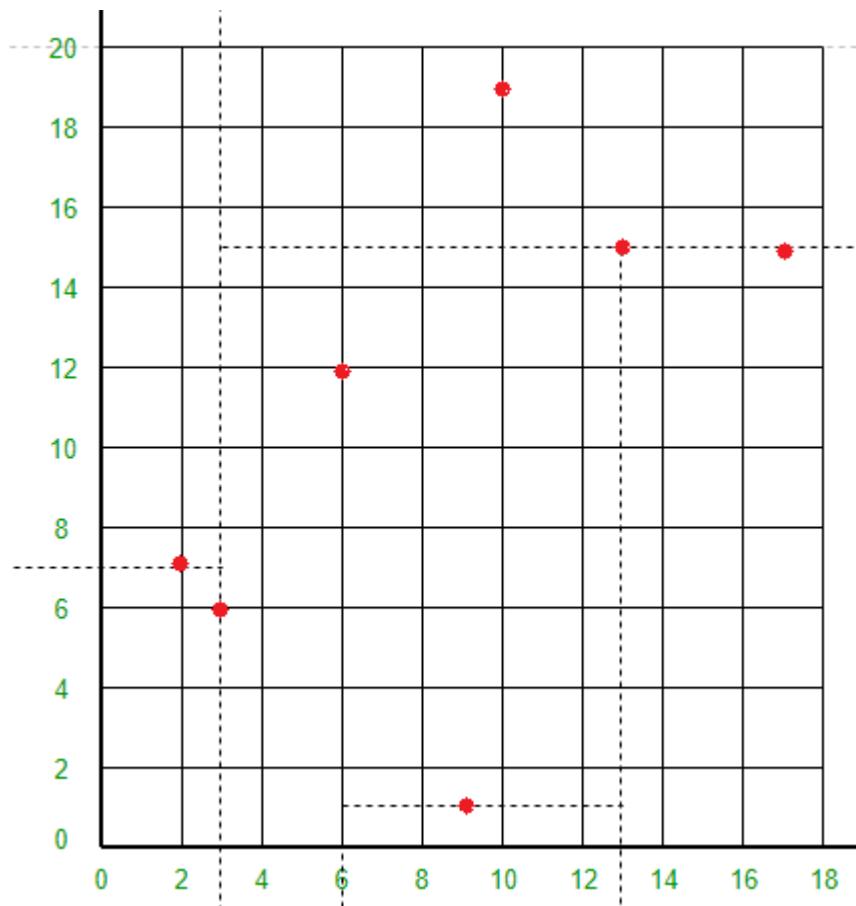


- Point (13, 15) will divide the space below line $Y = 15$ and to the right of line $X = 6$ into two parts.

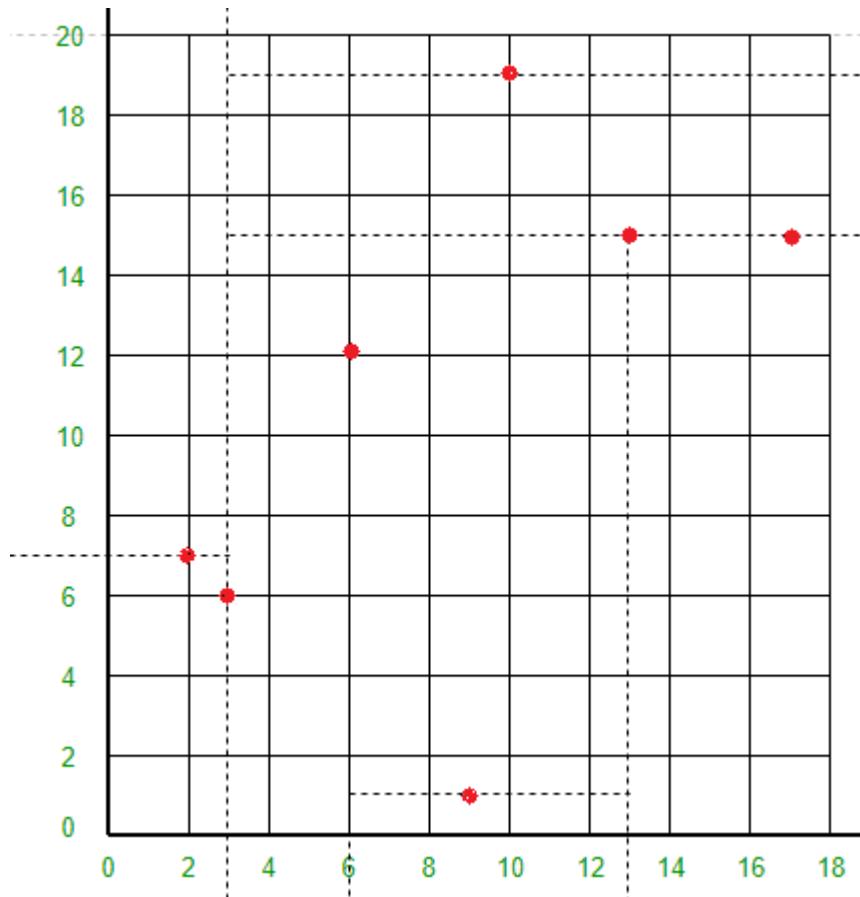
Draw line $X = 13$ to the right of line $X = 6$ and below line $Y = 15$.



- Point $(9, 1)$ will divide the space between lines $X = 3$, $X = 6$ and $Y = 15$ into two parts.
Draw line $Y = 1$ between lines $X = 3$ and $X = 6$.



- Point (10, 19) will divide the space to the right of line $X = 3$ and above line $Y = 15$ into two parts.
Draw line $Y = 19$ to the right of line $X = 3$ and above line $Y = 15$.



Following is C++ implementation of KD Tree basic operations like search, insert and delete.

```
// A C++ program to demonstrate operations of KD tree
#include<bits/stdc++.h>
using namespace std;

const int k = 2;

// A structure to represent node of kd tree
struct Node
{
    int point[k]; // To store k dimensional point
    Node *left, *right;
};

// A method to create a node of K D tree
struct Node* newNode(int arr[])
{
    struct Node* temp = new Node;
```

```

for (int i=0; i<k; i++)
    temp->point[i] = arr[i];

temp->left = temp->right = NULL;
return temp;
}

// Inserts a new node and returns root of modified tree
// The parameter depth is used to decide axis of comparison
Node *insertRec(Node *root, int point[], unsigned depth)
{
    // Tree is empty?
    if (root == NULL)
        return newNode(point);

    // Calculate current dimension (cd) of comparison
    unsigned cd = depth % k;

    // Compare the new point with root on current dimension 'cd'
    // and decide the left or right subtree
    if (point[cd] < (root->point[cd]))
        root->left = insertRec(root->left, point, depth + 1);
    else
        root->right = insertRec(root->right, point, depth + 1);

    return root;
}

// Function to insert a new point with given point in
// KD Tree and return new root. It mainly uses above recursive
// function "insertRec()"
Node* insert(Node *root, int point[])
{
    return insertRec(root, point, 0);
}

// A utility method to determine if two Points are same
// in K Dimensional space
bool arePointsSame(int point1[], int point2[])
{
    // Compare individual pointinate values
    for (int i = 0; i < k; ++i)
        if (point1[i] != point2[i])
            return false;

    return true;
}

```

```

// Searches a Point represented by "point[]" in the K D tree.
// The parameter depth is used to determine current axis.
bool searchRec(Node* root, int point[], unsigned depth)
{
    // Base cases
    if (root == NULL)
        return false;
    if (arePointsSame(root->point, point))
        return true;

    // Current dimension is computed using current depth and total
    // dimensions (k)
    unsigned cd = depth % k;

    // Compare point with root with respect to cd (Current dimension)
    if (point[cd] < root->point[cd])
        return searchRec(root->left, point, depth + 1);

    return searchRec(root->right, point, depth + 1);
}

// Searches a Point in the K D tree. It mainly uses
// searchRec()
bool search(Node* root, int point[])
{
    // Pass current depth as 0
    return searchRec(root, point, 0);
}

// Driver program to test above functions
int main()
{
    struct Node *root = NULL;
    int points[] [k] = {{3, 6}, {17, 15}, {13, 15}, {6, 12},
                       {9, 1}, {2, 7}, {10, 19}};

    int n = sizeof(points)/sizeof(points[0]);

    for (int i=0; i<n; i++)
        root = insert(root, points[i]);

    int point1[] = {10, 19};
    (search(root, point1))? cout << "Found\n": cout << "Not Found\n";

    int point2[] = {12, 19};
    (search(root, point2))? cout << "Found\n": cout << "Not Found\n";
}

```

```
    return 0;  
}
```

Output:

```
Found  
Not Found
```

Refer below articles for find minimum and delete operations.

- [K D Tree \(Find Minimum\)](#)
- [K D Tree \(Delete\)](#)

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

Source

<https://www.geeksforgeeks.org/k-dimensional-tree/>

Chapter 91

K Dimensional Tree Set 2 (Find Minimum)

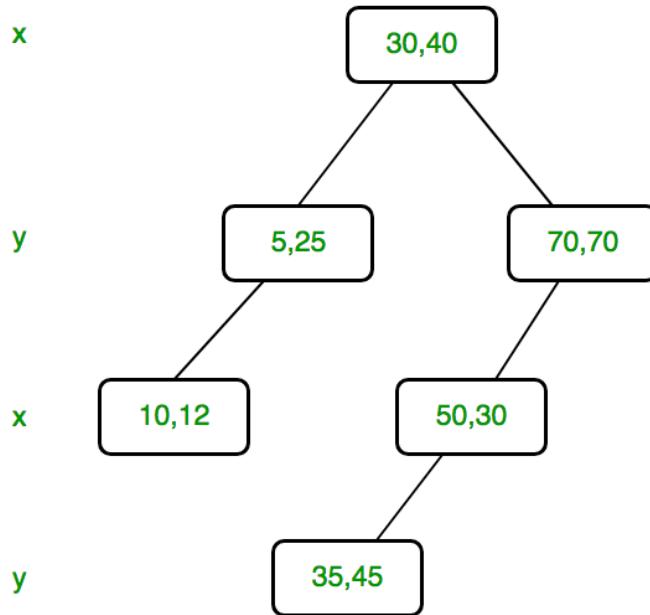
K Dimensional Tree Set 2 (Find Minimum) - GeeksforGeeks

We strongly recommend to refer below post as a prerequisite of this.

[K Dimensional Tree Set 1 \(Search and Insert\)](#)

In this post find minimum is discussed. The operation is to find minimum in the given dimension. This is especially needed in delete operation.

For example, consider below KD Tree, if given dimension is x, then output should be 5 and if given dimensions is t, then output should be 12.



In KD tree, points are divided dimension by dimension. For example, root divides keys by dimension 0, level next to root divides by dimension 1, next level by dimension 2 if k is more than 2 (else by dimension 0), and so on.

To find minimum we traverse nodes starting from root. *If dimension of current level is same as given dimension, then required minimum lies on left side if there is left child.* This is same as [Binary Search Tree Minimum](#).

Above is simple, what to do when current level's dimension is different. *When dimension of current level is different, minimum may be either in left subtree or right subtree or current node may also be minimum.* So we take minimum of three and return. This is different from Binary Search tree.

Below is C++ implementation of find minimum operation.

```

// A C++ program to demonstrate find minimum on KD tree
#include<bits/stdc++.h>
using namespace std;

const int k = 2;

// A structure to represent node of kd tree
struct Node
{
    int point[k]; // To store k dimensional point
    Node *left, *right;
};

// A method to create a node of K D tree

```

```

struct Node* newNode(int arr[])
{
    struct Node* temp = new Node;

    for (int i=0; i<k; i++)
        temp->point[i] = arr[i];

    temp->left = temp->right = NULL;
    return temp;
}

// Inserts a new node and returns root of modified tree
// The parameter depth is used to decide axis of comparison
Node *insertRec(Node *root, int point[], unsigned depth)
{
    // Tree is empty?
    if (root == NULL)
        return newNode(point);

    // Calculate current dimension (cd) of comparison
    unsigned cd = depth % k;

    // Compare the new point with root on current dimension 'cd'
    // and decide the left or right subtree
    if (point[cd] < (root->point[cd]))
        root->left = insertRec(root->left, point, depth + 1);
    else
        root->right = insertRec(root->right, point, depth + 1);

    return root;
}

// Function to insert a new point with given point in
// KD Tree and return new root. It mainly uses above recursive
// function "insertRec()"
Node* insert(Node *root, int point[])
{
    return insertRec(root, point, 0);
}

// A utility function to find minimum of three integers
int min(int x, int y, int z)
{
    return min(x, min(y, z));
}

// Recursively finds minimum of d'th dimension in KD tree
// The parameter depth is used to determine current axis.

```

```

int findMinRec(Node* root, int d, unsigned depth)
{
    // Base cases
    if (root == NULL)
        return INT_MAX;

    // Current dimension is computed using current depth and total
    // dimensions (k)
    unsigned cd = depth % k;

    // Compare point with root with respect to cd (Current dimension)
    if (cd == d)
    {
        if (root->left == NULL)
            return root->point[d];
        return findMinRec(root->left, d, depth+1);
    }

    // If current dimension is different then minimum can be anywhere
    // in this subtree
    return min(root->point[d],
               findMinRec(root->left, d, depth+1),
               findMinRec(root->right, d, depth+1));
}

// A wrapper over findMinRec(). Returns minimum of d'th dimension
int findMin(Node* root, int d)
{
    // Pass current level or depth as 0
    return findMinRec(root, d, 0);
}

// Driver program to test above functions
int main()
{
    struct Node *root = NULL;
    int points[] [k] = {{30, 40}, {5, 25}, {70, 70},
                       {10, 12}, {50, 30}, {35, 45}};

    int n = sizeof(points)/sizeof(points[0]);

    for (int i=0; i<n; i++)
        root = insert(root, points[i]);

    cout << "Minimum of 0'th dimension is " << findMin(root, 0) << endl;
    cout << "Minimum of 1'th dimension is " << findMin(root, 1) << endl;

    return 0;
}

```

}

Output:

```
Minimum of 0'th dimension is 5
Minimum of 1'th dimension is 12
```

Source:

<https://www.cs.umd.edu/class/spring2008/cmsc420/L19.kd-trees.pdf>

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

Source

<https://www.geeksforgeeks.org/k-dimensional-tree-set-2-find-minimum/>

Chapter 92

K Dimensional Tree Set 3 (Delete)

K Dimensional Tree Set 3 (Delete) - GeeksforGeeks

We strongly recommend to refer below posts as a prerequisite of this.

[K Dimensional Tree Set 1 \(Search and Insert\)](#)

[K Dimensional Tree Set 2 \(Find Minimum\)](#)

In this post delete is discussed. The operation is to delete a given point from K D Tree.

Like [Binary Search Tree Delete](#), we recursively traverse down and search for the point to be deleted. Below are steps are followed for every node visited.

1) If current node contains the point to be deleted

1. If node to be deleted is a leaf node, simply delete it (Same as [BST Delete](#))
2. If node to be deleted has right child as not NULL (Different from BST)
 - (a) Find minimum of current node's dimension in right subtree.
 - (b) Replace the node with above found minimum and recursively delete minimum in right subtree.
3. Else If node to be deleted has left child as not NULL (Different from BST)
 - (a) Find minimum of current node's dimension in left subtree.
 - (b) Replace the node with above found minimum and recursively delete minimum in left subtree.
 - (c) Make new left subtree as right child of current node.

2) If current doesn't contain the point to be deleted

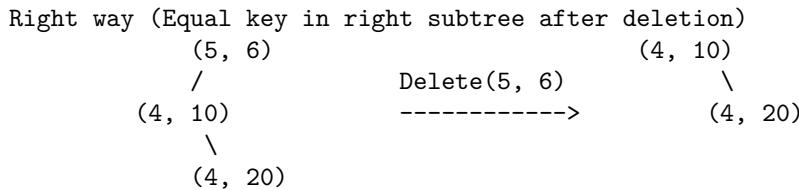
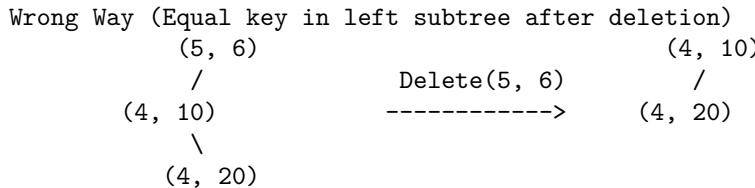
1. If node to be deleted is smaller than current node on current dimension, recur for left subtree.
2. Else recur for right subtree.

Why 1.b and 1.c are different from BST?

In BST delete, if a node's left child is empty and right is not empty, we replace the node with right child. In K D Tree, doing this would violate the KD tree property as dimension of right child of node is different from node's dimension. For example, if node divides point by x axis values. then its children divide by y axis, so we can't simply replace node with right child. Same is true for the case when right child is not empty and left child is empty.

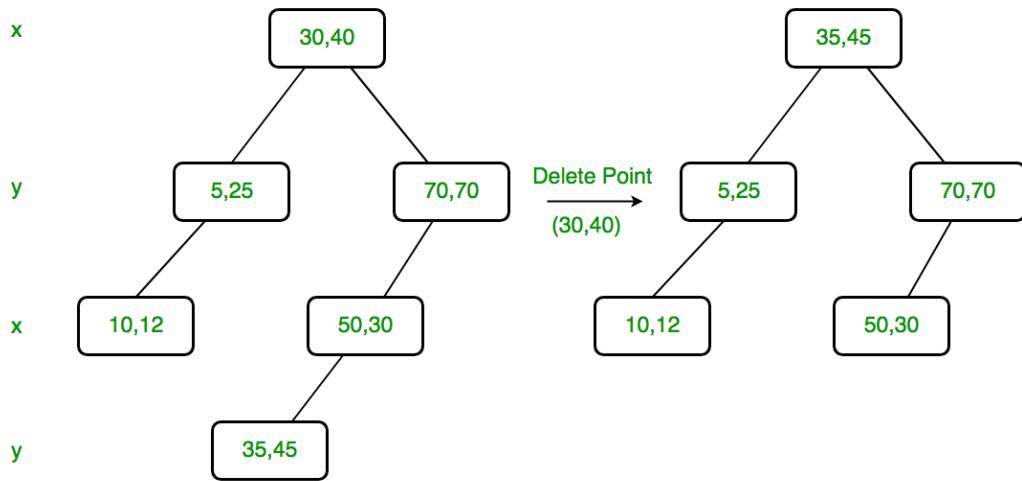
Why 1.c doesn't find max in left subtree and recur for max like 1.b?

Doing this violates the property that all equal values are in right subtree. For example, if we delete (!0, 10) in below subtree and replace if with

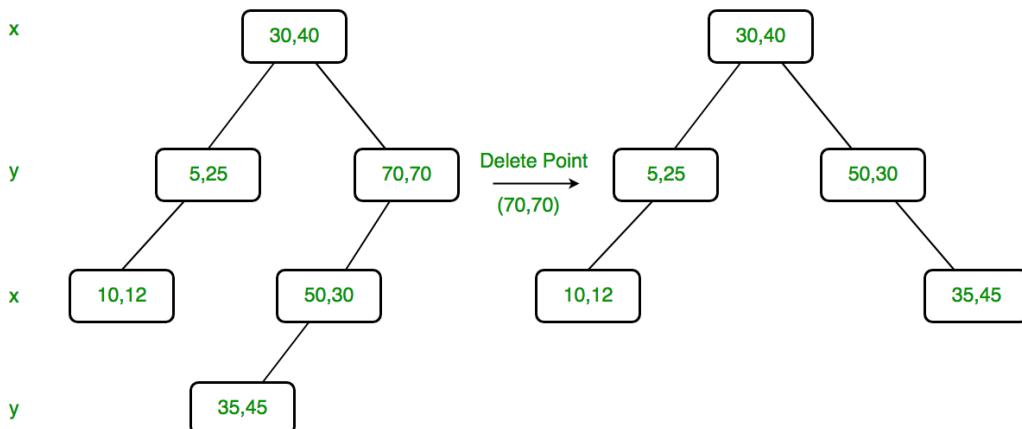


Example of Delete:

Delete (30, 40): Since right child is not NULL and dimension of node is x, we find the node with minimum x value in right child. The node is (35, 45), we replace (30, 40) with (35, 45) and delete (35, 45).



Delete (70, 70): Dimension of node is y. Since right child is NULL, we find the node with minimum y value in left child. The node is (50, 30), we replace (70, 70) with (50, 30) and recursively delete (50, 30) in left subtree. Finally we make the modified left subtree as right subtree of (50, 30).



Below is C++ implementation of K D Tree delete.

```
// A C++ program to demonstrate delete in K D tree
#include<bits/stdc++.h>
using namespace std;

const int k = 2;

// A structure to represent node of kd tree
struct Node
{
    int point[k]; // To store k dimensional point
```

```

        Node *left, *right;
    };

// A method to create a node of K D tree
struct Node* newNode(int arr[])
{
    struct Node* temp = new Node;

    for (int i=0; i<k; i++)
        temp->point[i] = arr[i];

    temp->left = temp->right = NULL;
    return temp;
}

// Inserts a new node and returns root of modified tree
// The parameter depth is used to decide axis of comparison
Node *insertRec(Node *root, int point[], unsigned depth)
{
    // Tree is empty?
    if (root == NULL)
        return newNode(point);

    // Calculate current dimension (cd) of comparison
    unsigned cd = depth % k;

    // Compare the new point with root on current dimension 'cd'
    // and decide the left or right subtree
    if (point[cd] < (root->point[cd]))
        root->left = insertRec(root->left, point, depth + 1);
    else
        root->right = insertRec(root->right, point, depth + 1);

    return root;
}

// Function to insert a new point with given point in
// KD Tree and return new root. It mainly uses above recursive
// function "insertRec()"
Node* insert(Node *root, int point[])
{
    return insertRec(root, point, 0);
}

// A utility function to find minimum of three integers
Node *minNode(Node *x, Node *y, Node *z, int d)
{
    Node *res = x;

```

```

if (y != NULL && y->point[d] < res->point[d])
    res = y;
if (z != NULL && z->point[d] < res->point[d])
    res = z;
return res;
}

// Recursively finds minimum of d'th dimension in KD tree
// The parameter depth is used to determine current axis.
Node *findMinRec(Node* root, int d, unsigned depth)
{
    // Base cases
    if (root == NULL)
        return NULL;

    // Current dimension is computed using current depth and total
    // dimensions (k)
    unsigned cd = depth % k;

    // Compare point with root with respect to cd (Current dimension)
    if (cd == d)
    {
        if (root->left == NULL)
            return root;
        return findMinRec(root->left, d, depth+1);
    }

    // If current dimension is different then minimum can be anywhere
    // in this subtree
    return minNode(root,
                  findMinRec(root->left, d, depth+1),
                  findMinRec(root->right, d, depth+1), d);
}

// A wrapper over findMinRec(). Returns minimum of d'th dimension
Node *findMin(Node* root, int d)
{
    // Pass current level or depth as 0
    return findMinRec(root, d, 0);
}

// A utility method to determine if two Points are same
// in K Dimensional space
bool arePointsSame(int point1[], int point2[])
{
    // Compare individual pointinate values
    for (int i = 0; i < k; ++i)
        if (point1[i] != point2[i])

```

```

        return false;

    return true;
}

// Copies point p2 to p1
void copyPoint(int p1[], int p2[])
{
    for (int i=0; i<k; i++)
        p1[i] = p2[i];
}

// Function to delete a given point 'point[]' from tree with root
// as 'root'. depth is current depth and passed as 0 initially.
// Returns root of the modified tree.
Node *deleteNodeRec(Node *root, int point[], int depth)
{
    // Given point is not present
    if (root == NULL)
        return NULL;

    // Find dimension of current node
    int cd = depth % k;

    // If the point to be deleted is present at root
    if (arePointsSame(root->point, point))
    {
        // 2.b) If right child is not NULL
        if (root->right != NULL)
        {
            // Find minimum of root's dimension in right subtree
            Node *min = findMin(root->right, cd);

            // Copy the minimum to root
            copyPoint(root->point, min->point);

            // Recursively delete the minimum
            root->right = deleteNodeRec(root->right, min->point, depth+1);
        }
        else if (root->left != NULL) // same as above
        {
            Node *min = findMin(root->left, cd);
            copyPoint(root->point, min->point);
            root->right = deleteNodeRec(root->left, min->point, depth+1);
        }
        else // If node to be deleted is leaf node
        {
            delete root;
    }
}

```

```
        return NULL;
    }
    return root;
}

// 2) If current node doesn't contain point, search downward
if (point[cd] < root->point[cd])
    root->left = deleteNodeRec(root->left, point, depth+1);
else
    root->right = deleteNodeRec(root->right, point, depth+1);
return root;
}

// Function to delete a given point from K D Tree with 'root'
Node* deleteNode(Node *root, int point[])
{
    // Pass depth as 0
    return deleteNodeRec(root, point, 0);
}

// Driver program to test above functions
int main()
{
    struct Node *root = NULL;
    int points[] [k] = {{30, 40}, {5, 25}, {70, 70},
                        {10, 12}, {50, 30}, {35, 45}};

    int n = sizeof(points)/sizeof(points[0]);

    for (int i=0; i<n; i++)
        root = insert(root, points[i]);

    // Delete (30, 40);
    root = deleteNode(root, points[0]);

    cout << "Root after deletion of (30, 40)\n";
    cout << root->point[0] << ", " << root->point[1] << endl;

    return 0;
}
```

Output:

```
Root after deletion of (30, 40)
35, 45
```

Source:

<https://www.cs.umd.edu/class/spring2008/cmsc420/L19.kd-trees.pdf>

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

Source

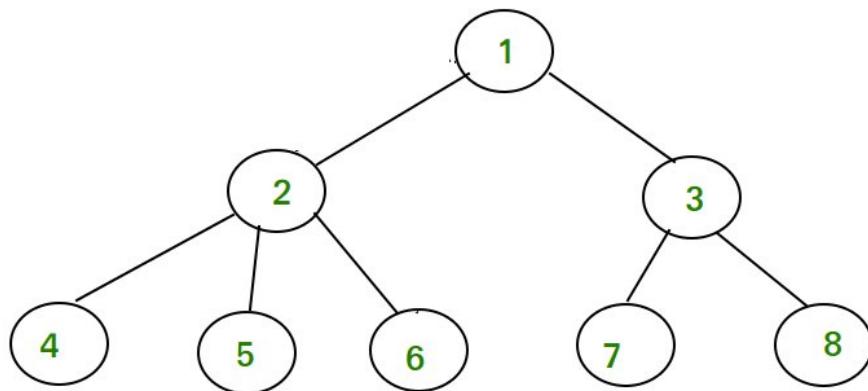
<https://www.geeksforgeeks.org/k-dimensional-tree-set-3-delete/>

Chapter 93

LCA for general or n-ary trees (Sparse Matrix DP approach < O(nlogn), O(logn)>)

LCA for general or n-ary trees (Sparse Matrix DP approach < O(nlogn), O(logn)>) - Geeks-forGeeks

In previous posts, we have discussed how to calculate the Lowest Common Ancestor (LCA) for a binary tree and a binary search tree ([this](#), [this](#) and [this](#)). Now let's look at a method that can calculate LCA for any tree (not only for binary tree). We use Dynamic Programming with Sparse Matrix Approach in our method. This method is very handy and fast when you need to answer multiple queries of LCA for a tree.



$$\begin{aligned} \text{LCA}(4, 6) &= 2 \\ \text{LCA}(5, 7) &= 1 \end{aligned}$$

Pre-requisites : -

- 1) [DFS](#)
- 2) Basic DP knowledge ([This](#) and [this](#))
- 3) [Range Minimum Query \(Square Root Decomposition and Sparse Table\)](#)

Naive Approach:- $O(n)$

The naive approach for this general tree LCA calculation will be the same as the naive approach for the LCA calculation of Binary Tree (this naive approach is already well described [here](#).

The C++ implementation for the naive approach is given below :-

```
/* Program to find LCA of n1 and n2 using one DFS on
   the Tree */
#include <iostream>
#include <vector>
using namespace std;

// Maximum number of nodes is 100000 and nodes are
// numbered from 1 to 100000
#define MAXN 100001

vector < int > tree[MAXN];
int path[3][MAXN]; // storing root to node path

// storing the path from root to node
void dfs(int cur, int prev, int pathNumber, int ptr,
         int node, bool &flag)
{
    for (int i=0; i<tree[cur].size(); i++)
    {
        if (tree[cur][i] != prev and !flag)
        {
            // pushing current node into the path
            path[pathNumber][ptr] = tree[cur][i];
            if (tree[cur][i] == node)
            {
                // node found
                flag = true;

                // terminating the path
                path[pathNumber][ptr+1] = -1;
                return;
            }
            dfs(tree[cur][i], cur, pathNumber, ptr+1,
                 node, flag);
        }
    }
}
```

```
}  
  
// This Function compares the path from root to 'a' & root  
// to 'b' and returns LCA of a and b. Time Complexity : O(n)  
int LCA(int a, int b)  
{  
    // trivial case  
    if (a == b)  
        return a;  
  
    // setting root to be first element in path  
    path[1][0] = path[2][0] = 1;  
  
    // calculating path from root to a  
    bool flag = false;  
    dfs(1, 0, 1, 1, a, flag);  
  
    // calculating path from root to b  
    flag = false;  
    dfs(1, 0, 2, 1, b, flag);  
  
    // runs till path 1 & path 2 matches  
    int i = 0;  
    while (path[1][i] == path[2][i])  
        i++;  
  
    // returns the last matching node in the paths  
    return path[1][i-1];  
}  
  
void addEdge(int a,int b)  
{  
    tree[a].push_back(b);  
    tree[b].push_back(a);  
}  
  
// Driver code  
int main()  
{  
    int n = 8; // Number of nodes  
    addEdge(1,2);  
    addEdge(1,3);  
    addEdge(2,4);  
    addEdge(2,5);  
    addEdge(2,6);  
    addEdge(3,7);  
    addEdge(3,8);  
}
```

```

cout << "LCA(4, 7) = " << LCA(4,7) << endl;
cout << "LCA(4, 6) = " << LCA(4,6) << endl;
return 0;
}

```

Output:

```

LCA(4, 7) = 1
LCA(4, 6) = 2

```

Sparse Matrix Approach ($O(n\log n)$) pre-processing, $O(\log n)$ – query

Pre-computation :- Here we store the 2^i th parent for every node, where $0 \leq i < \text{LEVEL}$, here “LEVEL” is a constant integer that tells the maximum number of 2^i th ancestor possible.

Therefore, we assume the worst case to see what is the value of the constant LEVEL. In our worst case every node in our tree will have at max 1 parent and 1 child or we can say it simply reduces to a linked list.

So, in this case $\text{LEVEL} = \text{ceil}(\log(\text{number of nodes}))$.

We also pre-compute the height for each node using one dfs in $O(n)$ time.

```

int n          // number of nodes
int parent[MAXN][LEVEL] // all initialized to -1

parent[node][0] : contains the  $2^0$ th(first)
parent of all the nodes pre-computed using DFS

// Sparse matrix Approach
for node -> 1 to n :
    for i-> 1 to LEVEL :
        if ( parent[node][i-1] != -1 ) :
            parent[node][i] =
                parent[ parent[node][i-1] ][i-1]

```

Now , as we see the above dynamic programming code runs two nested loop that runs over their complete range respectively.

Hence, it can be easily be inferred that its asymptotic Time Complexity is $O(\text{number of nodes} * \text{LEVEL}) \sim O(n * \text{LEVEL}) \sim O(n \log n)$.

Return LCA(u,v) :-

1) First Step is to bring both the nodes at the same height. As we have already pre-computed the heights for each node. We first calculate the difference in the heights of u and v (let's say $v \geq u$). Now we need the node 'v' to jump h nodes above. This can be easily done in $O(\log h)$ time (where h is the difference in the heights of u and v) as we have already stored

the 2^i parent for each node. This process is exactly same as calculating $x \wedge y$ in $O(\log y)$ time. (See the code for better understanding).

2) Now both u and v nodes are at same height. Therefore now once again we will use 2^i jumping strategy to reach the first Common Parent of u and v.

Pseudo-code:

```
For i-> LEVEL to 0 :
    If parent[u][i] != parent[v][i] :
        u = parent[u][i]
        v = parent[v][i]
```

C++ implementation of the above algorithm is given below:

```
// Sparse Matrix DP approach to find LCA of two nodes
#include <bits/stdc++.h>
using namespace std;
#define MAXN 100000
#define level 18

vector <int> tree[MAXN];
int depth[MAXN];
int parent[MAXN][level];

// pre-compute the depth for each node and their
// first parent( $2^0$ th parent)
// time complexity :  $O(n)$ 
void dfs(int cur, int prev)
{
    depth[cur] = depth[prev] + 1;
    parent[cur][0] = prev;
    for (int i=0; i<tree[cur].size(); i++)
    {
        if (tree[cur][i] != prev)
            dfs(tree[cur][i], cur);
    }
}

// Dynamic Programming Sparse Matrix Approach
// populating  $2^i$  parent for each node
// Time complexity :  $O(n\log n)$ 
void precomputeSparseMatrix(int n)
{
    for (int i=1; i<level; i++)
    {
        for (int node = 1; node <= n; node++)
        {
            if (parent[node][i-1] != -1)
```

```
        parent[node][i] =
            parent[parent[node][i-1]][i-1];
    }
}

// Returning the LCA of u and v
// Time complexity : O(log n)
int lca(int u, int v)
{
    if (depth[v] < depth[u])
        swap(u, v);

    int diff = depth[v] - depth[u];

    // Step 1 of the pseudocode
    for (int i=0; i<level; i++)
        if ((diff>>i)&1)
            v = parent[v][i];

    // now depth[u] == depth[v]
    if (u == v)
        return u;

    // Step 2 of the pseudocode
    for (int i=level-1; i>=0; i--)
        if (parent[u][i] != parent[v][i])
    {
        u = parent[u][i];
        v = parent[v][i];
    }

    return parent[u][0];
}

void addEdge(int u,int v)
{
    tree[u].push_back(v);
    tree[v].push_back(u);
}

// driver function
int main()
{
    memset(parent,-1,sizeof(parent));
    int n = 8;
    addEdge(1,2);
    addEdge(1,3);
```

```
addEdge(2,4);
addEdge(2,5);
addEdge(2,6);
addEdge(3,7);
addEdge(3,8);
depth[0] = 0;

// running dfs and precalculating depth
// of each node.
dfs(1,0);

// Precomputing the  $2^i$  th ancestor for every node
precomputeSparseMatrix(n);

// calling the LCA function
cout << "LCA(4, 7) = " << lca(4,7) << endl;
cout << "LCA(4, 6) = " << lca(4,6) << endl;
return 0;
}
```

Output:

```
LCA(4,7) = 1
LCA(4,6) = 2
```

Time Complexity: The time complexity for answering a single LCA query will be $O(\log n)$ but the overall time complexity is dominated by precalculation of the 2^i th ($0 \leq i \leq \text{level}$) ancestors for each node. Hence, the overall asymptotic Time Complexity will be $O(n * \log n)$ and Space Complexity will be $O(n \log n)$, for storing the data about the ancestors of each node.

Source

<https://www.geeksforgeeks.org/lca-for-general-or-n-ary-trees-sparse-matrix-dp-approach-onlogn-ologn/>

Chapter 94

LCA for n-ary Tree Constant Query O(1)

LCA for n-ary Tree Constant Query O(1) - GeeksforGeeks

We have seen various methods with different Time Complexities to calculate LCA in n-ary tree:-

Method 1 : Naive Method (by calculating root to node path) O(n) per query

Method 2 : Using Sqrt Decomposition O(sqrt H)

Method 3 : Using Sparse Matrix DP approach O(logn)

Lets study another method which has faster query time than all the above methods. So, our aim will be to calculate LCA in **constant time** ~ **O(1)**. Let's see how we can achieve it.

Method 4 : Using Range Minimum Query

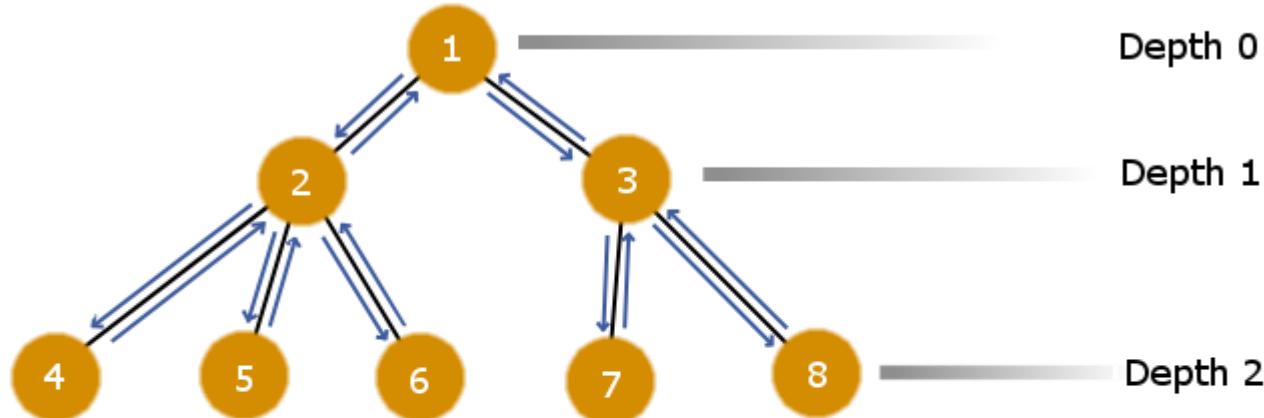
We have discussed [LCA and RMQ for binary tree](#). Here we discuss LCA problem to RMQ problem conversion for n-ary tree.

Pre-requisites:- [LCA in Binary Tree using RMQ](#)
[RMQ using sparse table](#)

Key Concept : In this method, we will be reducing our LCA problem to RMQ(Range Minimum Query) problem over a static array. Once, we do that then we will relate the Range minimum queries to the required LCA queries.

The first step will be to decompose the tree into a flat linear array. To do this we can apply the Euler walk . The Euler walk will give the pre-order traversal of the graph. So we will perform a Euler Walk on the tree and store the nodes in an array as we visit them. This process reduces the tree data-structure to a simple linear array.

Consider the below tree and the euler walk over it :-



Euler Walk for the above tree

Euler [] =

1	2	4	2	5	2	6	2	1	3	7	3	8	3	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Now lets think in general terms : Consider any two nodes on the tree. There will be exactly one path connecting both the nodes and the node that has the smallest depth value in the path will be the LCA of the two given nodes.

Now take any two distinct node say u and v in the Euler walk array. Now all the elements in the path from u to v will lie in between the index of nodes u and v in the Euler walk array. Therefore, we just need to calculate the node with the minimum depth between the index of node u and node v in the euler array.

For this we will maintain another array that will contain the depth of all the nodes corresponding to their position in the Euler walk array so that we can Apply our RMQ algorithm on it.

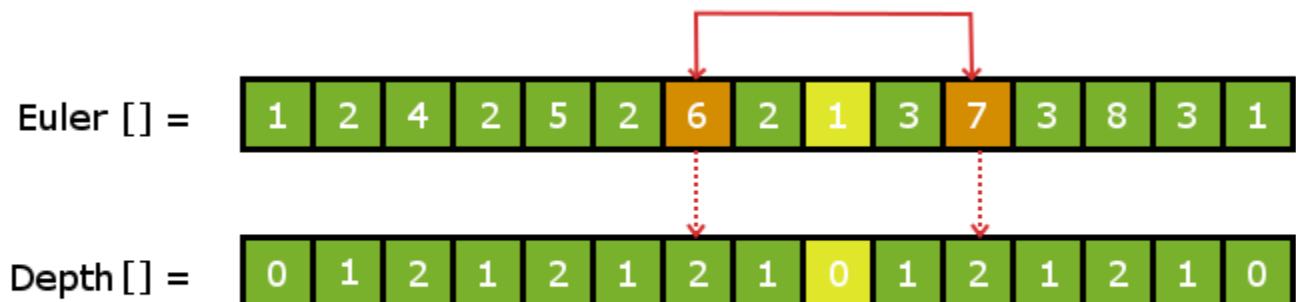
Given below is the euler walk array parallel to its depth track array.

Euler [] = 

Depth [] = 

Example :- Consider two nodes **node 6** and **node 7** in the euler array. To calculate the LCA of node 6 and node 7 we look the the smallest depth value for all the nodes in between node 6 and node 7 .

Therefore, **node 1** has the smallest *depth value* = 0 and hence, it is the LCA for node 6 and node 7.



$$\text{LCA}(6,7) = 1$$

Implementation :-

We will be maintaining three arrays 1)Euler Path
2)Depth array
3)First Appearance Index

Euler Path and Depth array are the same as described above

First Appearance Index FAI[] : The First Appearance index Array will store the index for the first position of every node in the Euler Path array. FAI[i] = First appearance of ith node in Euler Walk array.

The C++ Implementation for the above method is given below:-

```

// C++ program to demonstrate LCA of n-ary tree
// in constant time.
#include "bits/stdc++.h"
using namespace std;
#define sz 101

vector < int > adj[sz]; // stores the tree
vector < int > euler; // tracks the eulerwalk
vector < int > depthArr; // depth for each node corresponding
                        // to eulerwalk

int FAI[sz]; // stores first appearance index of every node
int level[sz]; // stores depth for all nodes in the tree
int ptr; // pointer to euler walk
int dp[sz][18]; // sparse table
int logn[sz]; // stores log values
int p2[20]; // stores power of 2

void buildSparseTable(int n)
{
    // initializing sparse table
    memset(dp,-1,sizeof(dp));

    // filling base case values
    for (int i=1; i<n; i++)
        dp[i-1][0] = (depthArr[i]>depthArr[i-1])?i-1:i;

    // dp to fill sparse table
    for (int l=1; l<15; l++)
        for (int i=0; i<n; i++)
            if (dp[i][l-1]!=-1 and dp[i+p2[l-1]][l-1]!=-1)
                dp[i][l] =
                    (depthArr[dp[i][l-1]]>depthArr[dp[i+p2[l-1]][l-1]])?
                    dp[i+p2[l-1]][l-1] : dp[i][l-1];
            else
                break;
}

int query(int l,int r)
{
    int d = r-l;
    int dx = logn[d];
    if (l==r) return l;
    if (depthArr[dp[l][dx]] > depthArr[dp[r-p2[dx]][dx]])
        return dp[r-p2[dx]][dx];
    else
        return dp[l][dx];
}

```

```

void preprocess()
{
    // memorizing powers of 2
    p2[0] = 1;
    for (int i=1; i<18; i++)
        p2[i] = p2[i-1]*2;

    // memorizing all log(n) values
    int val = 1,ptr=0;
    for (int i=1; i<sz; i++)
    {
        logn[i] = ptr-1;
        if (val==i)
        {
            val*=2;
            logn[i] = ptr;
            ptr++;
        }
    }
}

/**
 * Euler Walk ( preorder traversal)
 * converting tree to linear depthArray
 * Time Complexity : O(n)
 */
void dfs(int cur,int prev,int dep)
{
    // marking FAI for cur node
    if (FAI[cur]==-1)
        FAI[cur] = ptr;

    level[cur] = dep;

    // pushing root to euler walk
    euler.push_back(cur);

    // incrementing euler walk pointer
    ptr++;

    for (auto x:adj[cur])
    {
        if (x != prev)
        {
            dfs(x,cur,dep+1);

            // pushing cur again in backtrack
        }
    }
}

```

```
// of euler walk
euler.push_back(cur);

// increment euler walk pointer
ptr++;
}

}

// Create Level depthArray corresponding
// to the Euler walk Array
void makeArr()
{
    for (auto x : euler)
        depthArr.push_back(level[x]);
}

int LCA(int u,int v)
{
    // trivial case
    if (u==v)
        return u;

    if (FAI[u] > FAI[v])
        swap(u,v);

    // doing RMQ in the required range
    return euler[query(FAI[u], FAI[v])];
}

void addEdge(int u,int v)
{
    adj[u].push_back(v);
    adj[v].push_back(u);
}

int main(int argc, char const *argv[])
{
    // constructing the described tree
    int numberOfNodes = 8;
    addEdge(1,2);
    addEdge(1,3);
    addEdge(2,4);
    addEdge(2,5);
    addEdge(2,6);
    addEdge(3,7);
    addEdge(3,8);
```

```
// performing required precalculations
preprocess();

// doing the Euler walk
ptr = 0;
memset(FAI,-1,sizeof(FAI));
dfs(1,0,0);

// creating depthArray corresponding to euler[]
makeArr();

// building sparse table
buildSparseTable(depthArr.size());

cout << "LCA(6,7) : " << LCA(6,7) << "\n";
cout << "LCA(6,4) : " << LCA(6,4) << "\n";

return 0;
}
```

Output:

```
LCA(6,7) : 1
LCA(6,4) : 2
```

Note : We are precalculating all the required power of 2's and also precalculating the all the required log values to ensure constant time complexity per query. Else if we did log calculation for every query operation our Time complexity would have not been constant.

Time Complexity: The Conversion process from LCA to RMQ is done by Euler Walk that takes $O(n)$ time.

Pre-processing for the sparse table in RMQ takes $O(n\log n)$ time and answering each Query is a Constant time process. Therefore, overall Time Complexity is $O(n\log n)$ – preprocessing and **$O(1)$** for each query.

Source

<https://www.geeksforgeeks.org/lca-n-ary-tree-constant-query-o1/>

Chapter 95

LRU Cache Implementation

LRU Cache Implementation - GeeksforGeeks

How to implement LRU caching scheme? What data structures should be used?

We are given total possible page numbers that can be referred. We are also given cache (or memory) size (Number of page frames that cache can hold at a time). The LRU caching scheme is to remove the least recently used frame when the cache is full and a new page is referenced which is not there in cache. Please see the Galvin book for more details (see the LRU page replacement slide [here](#)).

We use two data structures to implement an LRU Cache.

1. **Queue** which is implemented using a doubly linked list. The maximum size of the queue will be equal to the total number of frames available (cache size). The most recently used pages will be near front end and least recently pages will be near rear end.
2. **A Hash** with page number as key and address of the corresponding queue node as value.

When a page is referenced, the required page may be in the memory. If it is in the memory, we need to detach the node of the list and bring it to the front of the queue.

If the required page is not in the memory, we bring that in memory. In simple words, we add a new node to the front of the queue and update the corresponding node address in the hash. If the queue is full, i.e. all the frames are full, we remove a node from the rear of queue, and add the new node to the front of queue.

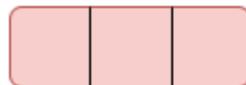
Example – Consider the following reference string :

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

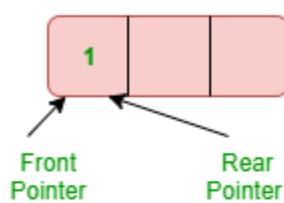
Find the number of page faults using least recently used (LRU) page replacement algorithm with 3 page frames.

Explanation –

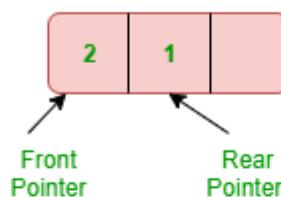
Given 3 page frames, so we take size of Queue is 3. Initially, Queue is empty.



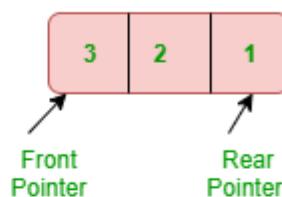
Input : 1



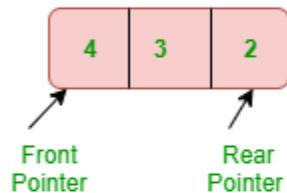
Input : 2 (every new input will be front as defined LRU)



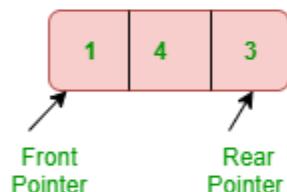
Input : 3



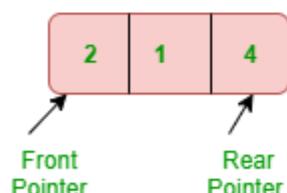
Input : 4 (since all the frames are full, we remove a node from the rear of queue, and add the new node to the front of queue.)



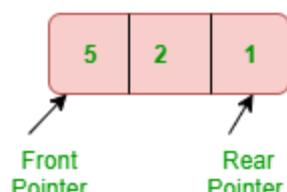
Input : 1 (since all the frames are full, we remove a node from the rear of queue, and add the new node to the front of queue.)



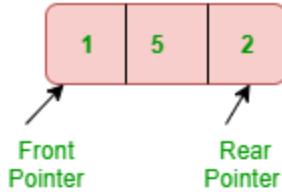
Input : 2 (since all the frames are full, we remove a node from the rear of queue, and add the new node to the front of queue.)



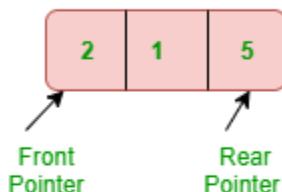
Input : 5 (since all the frames are full, we remove a node from the rear of queue, and add the new node to the front of queue.)



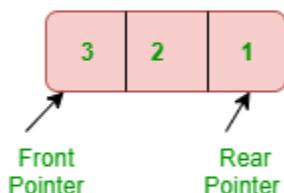
Input : 1 (since present in memory, so bring it to the front of the queue. This is called hit)



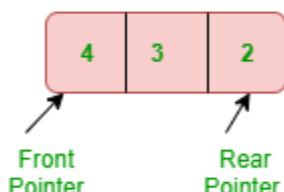
Input : 2 (since present in memory, so bring it to the front of the queue. This is called hit)



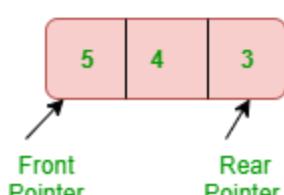
Input : 3 (since all the frames are full, we remove a node from the rear of queue, and add the new node to the front of queue.)



Input : 4 (since all the frames are full, we remove a node from the rear of queue, and add the new node to the front of queue.)



Input : 5 (since all the frames are full, we remove a node from the rear of queue, and add the new node to the front of queue.)



Note: Initially no page is in the memory.

C++ using STL

```
/* We can use stl container list as a double
   ended queue to store the cache keys, with
   the descending time of reference from front
   to back and a set container to check presence
   of a key. But to fetch the address of the key
   in the list using find(), it takes O(N) time.
   This can be optimized by storing a reference
   (iterator) to each key in a hash map. */
#include <bits/stdc++.h>
using namespace std;

class LRUCache
{
    // store keys of cache
    list<int> dq;

    // store references of key in cache
    unordered_map<int, list<int>::iterator> ma;
    int csize; //maximum capacity of cache

public:
    LRUCache(int);
    void refer(int);
    void display();
};

LRUCache::LRUCache(int n)
{
    csize = n;
}

/* Refers key x with in the LRU cache */
void LRUCache::refer(int x)
{
    // not present in cache
    if (ma.find(x) == ma.end())
    {
        // cache is full
        if (dq.size() == csize)
        {
            //delete least recently used element
            int last = dq.back();
            dq.pop_back();
            ma.erase(last);
        }
    }
}
```

```
}

// present in cache
else
    dq.erase(ma[x]);

// update reference
dq.push_front(x);
ma[x] = dq.begin();
}

// display contents of cache
void LRUCache::display()
{
    for (auto it = dq.begin(); it != dq.end();
          it++)
        cout << (*it) << " ";

    cout << endl;
}

// Driver program to test above functions
int main()
{
    LRUCache ca(4);

    ca.refer(1);
    ca.refer(2);
    ca.refer(3);
    ca.refer(1);
    ca.refer(4);
    ca.refer(5);
    ca.display();

    return 0;
}
// This code is contributed by Satish Srinivas
```

C

```
// A C program to show implementation of LRU cache
#include <stdio.h>
#include <stdlib.h>

// A Queue Node (Queue is implemented using Doubly Linked List)
typedef struct QNode
{
    struct QNode *prev, *next;
```

```

        unsigned pageNumber; // the page number stored in this QNode
    } QNode;

// A Queue (A FIFO collection of Queue Nodes)
typedef struct Queue
{
    unsigned count; // Number of filled frames
    unsigned numberOfWorks; // total number of frames
    QNode *front, *rear;
} Queue;

// A hash (Collection of pointers to Queue Nodes)
typedef struct Hash
{
    int capacity; // how many pages can be there
    QNode* *array; // an array of queue nodes
} Hash;

// A utility function to create a new Queue Node. The queue Node
// will store the given 'pageNumber'
QNode* newQNode( unsigned pageNumber )
{
    // Allocate memory and assign 'pageNumber'
    QNode* temp = (QNode *)malloc( sizeof( QNode ) );
    temp->pageNumber = pageNumber;

    // Initialize prev and next as NULL
    temp->prev = temp->next = NULL;

    return temp;
}

// A utility function to create an empty Queue.
// The queue can have at most 'numberOfWorks' nodes
Queue* createQueue( int numberOfWorks )
{
    Queue* queue = (Queue *)malloc( sizeof( Queue ) );

    // The queue is empty
    queue->count = 0;
    queue->front = queue->rear = NULL;

    // Number of frames that can be stored in memory
    queue->numberOfWorks = numberOfWorks;

    return queue;
}

```

```

// A utility function to create an empty Hash of given capacity
Hash* createHash( int capacity )
{
    // Allocate memory for hash
    Hash* hash = (Hash *) malloc( sizeof( Hash ) );
    hash->capacity = capacity;

    // Create an array of pointers for referring queue nodes
    hash->array = (QNode **) malloc( hash->capacity * sizeof( QNode* ) );

    // Initialize all hash entries as empty
    int i;
    for( i = 0; i < hash->capacity; ++i )
        hash->array[i] = NULL;

    return hash;
}

// A function to check if there is slot available in memory
int AreAllFramesFull( Queue* queue )
{
    return queue->count == queue->numberOfFrames;
}

// A utility function to check if queue is empty
int isQueueEmpty( Queue* queue )
{
    return queue->rear == NULL;
}

// A utility function to delete a frame from queue
void deQueue( Queue* queue )
{
    if( isQueueEmpty( queue ) )
        return;

    // If this is the only node in list, then change front
    if (queue->front == queue->rear)
        queue->front = NULL;

    // Change rear and remove the previous rear
    QNode* temp = queue->rear;
    queue->rear = queue->rear->prev;

    if (queue->rear)
        queue->rear->next = NULL;

    free( temp );
}

```

```

// decrement the number of full frames by 1
queue->count--;
}

// A function to add a page with given 'pageNumber' to both queue
// and hash
void Enqueue( Queue* queue, Hash* hash, unsigned pageNumber )
{
    // If all frames are full, remove the page at the rear
    if ( AreAllFramesFull ( queue ) )
    {
        // remove page from hash
        hash->array[ queue->rear->pageNumber ] = NULL;
        deQueue( queue );
    }

    // Create a new node with given page number,
    // And add the new node to the front of queue
    QNode* temp = newQNode( pageNumber );
    temp->next = queue->front;

    // If queue is empty, change both front and rear pointers
    if ( isEmpty( queue ) )
        queue->rear = queue->front = temp;
    else // Else change the front
    {
        queue->front->prev = temp;
        queue->front = temp;
    }

    // Add page entry to hash also
    hash->array[ pageNumber ] = temp;

    // increment number of full frames
    queue->count++;
}

// This function is called when a page with given 'pageNumber' is referenced
// from cache (or memory). There are two cases:
// 1. Frame is not there in memory, we bring it in memory and add to the front
//     of queue
// 2. Frame is there in memory, we move the frame to front of queue
void ReferencePage( Queue* queue, Hash* hash, unsigned pageNumber )
{
    QNode* reqPage = hash->array[ pageNumber ];

    // the page is not in cache, bring it
}

```

```

if ( reqPage == NULL )
    Enqueue( queue, hash, pageNumber );

// page is there and not at front, change pointer
else if (reqPage != queue->front)
{
    // Unlink requested page from its current location
    // in queue.
    reqPage->prev->next = reqPage->next;
    if (reqPage->next)
        reqPage->next->prev = reqPage->prev;

    // If the requested page is rear, then change rear
    // as this node will be moved to front
    if (reqPage == queue->rear)
    {
        queue->rear = reqPage->prev;
        queue->rear->next = NULL;
    }

    // Put the requested page before current front
    reqPage->next = queue->front;
    reqPage->prev = NULL;

    // Change prev of current front
    reqPage->next->prev = reqPage;

    // Change front to the requested page
    queue->front = reqPage;
}
}

// Driver program to test above functions
int main()
{
    // Let cache can hold 4 pages
    Queue* q = createQueue( 4 );

    // Let 10 different pages can be requested (pages to be
    // referenced are numbered from 0 to 9
    Hash* hash = createHash( 10 );

    // Let us refer pages 1, 2, 3, 1, 4, 5
    ReferencePage( q, hash, 1 );
    ReferencePage( q, hash, 2 );
    ReferencePage( q, hash, 3 );
    ReferencePage( q, hash, 1 );
    ReferencePage( q, hash, 4 );
}

```

```
ReferencePage( q, hash, 5);

// Let us print cache frames after the above referenced pages
printf ("%d ", q->front->pageNumber);
printf ("%d ", q->front->next->pageNumber);
printf ("%d ", q->front->next->next->pageNumber);
printf ("%d ", q->front->next->next->next->pageNumber);

return 0;
}
```

Output:

5 4 1 3

This article is compiled by [Aashish Barnwal](#) and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

<https://www.geeksforgeeks.org/lru-cache-implementation/>

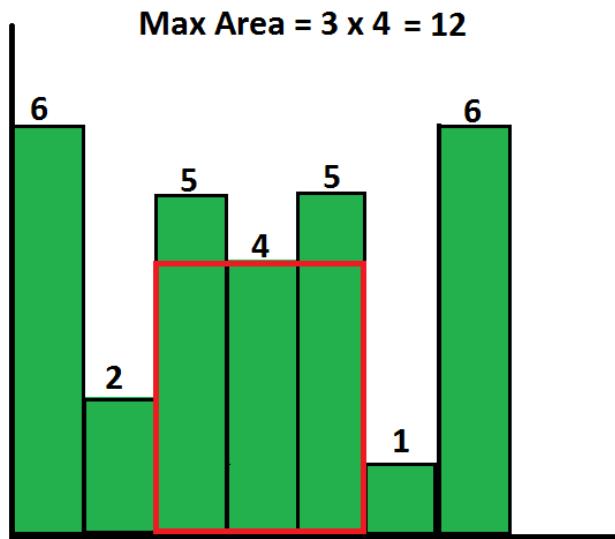
Chapter 96

Largest Rectangular Area in a Histogram Set 1

Largest Rectangular Area in a Histogram Set 1 - GeeksforGeeks

Find the largest rectangular area possible in a given histogram where the largest rectangle can be made of a number of contiguous bars. For simplicity, assume that all bars have same width and the width is 1 unit.

For example, consider the following histogram with 7 bars of heights {6, 2, 5, 4, 5, 2, 6}. The largest possible rectangle possible is 12 (see the below figure, the max area rectangle is highlighted in red)



A simple solution is to one by one consider all bars as starting points and calculate area of

all rectangles starting with every bar. Finally return maximum of all possible areas. Time complexity of this solution would be $O(n^2)$.

We can use **Divide and Conquer** to solve this in $O(n\log n)$ time. The idea is to find the minimum value in the given array. Once we have index of the minimum value, the max area is maximum of following three values.

- a) Maximum area in left side of minimum value (Not including the min value)
- b) Maximum area in right side of minimum value (Not including the min value)
- c) Number of bars multiplied by minimum value.

The areas in left and right of minimum value bar can be calculated recursively. If we use linear search to find the minimum value, then the worst case time complexity of this algorithm becomes $O(n^2)$. In worst case, we always have $(n-1)$ elements in one side and 0 elements in other side and if the finding minimum takes $O(n)$ time, we get the recurrence similar to worst case of Quick Sort.

How to find the minimum efficiently? [Range Minimum Query using Segment Tree](#) can be used for this. We build segment tree of the given histogram heights. Once the segment tree is built, all [range minimum queries take \$O\(\log n\)\$ time](#). So over all complexity of the algorithm becomes.

Overall Time = Time to build Segment Tree + Time to recursively find maximum area

[Time to build segment tree is \$O\(n\)\$](#) . Let the time to recursively find max area be $T(n)$. It can be written as following.

$$T(n) = O(\log n) + T(n-1)$$

The solution of above recurrence is $O(n\log n)$. So overall time is $O(n) + O(n\log n)$ which is $O(n\log n)$.

Following is C++ implementation of the above algorithm.

```
// A Divide and Conquer Program to find maximum rectangular area in a histogram
#include <math.h>
#include <limits.h>
#include <iostream>
using namespace std;

// A utility function to find minimum of three integers
int max(int x, int y, int z)
{   return max(max(x, y), z); }

// A utility function to get minimum of two numbers in hist[]
int minVal(int *hist, int i, int j)
{
    if (i == -1) return j;
    if (j == -1) return i;
    return (hist[i] < hist[j])? i : j;
}

// A utility function to get the middle index from corner indexes.
int getMid(int s, int e)
{   return s + (e - s)/2; }
```

```

/* A recursive function to get the index of minimum value in a given range of
indexes. The following are parameters for this function.

hist    --> Input array for which segment tree is built
st      --> Pointer to segment tree
index   --> Index of current node in the segment tree. Initially 0 is
            passed as root is always at index 0
ss & se  --> Starting and ending indexes of the segment represented by
            current node, i.e., st[index]
qs &qe   --> Starting and ending indexes of query range */
int RMQUtil(int *hist, int *st, int ss, int se, int qs, int qe, int index)
{
    // If segment of this node is a part of given range, then return the
    // min of the segment
    if (qs <= ss && qe >= se)
        return st[index];

    // If segment of this node is outside the given range
    if (se < qs || ss > qe)
        return -1;

    // If a part of this segment overlaps with the given range
    int mid = getMid(ss, se);
    return minVal(hist, RMQUtil(hist, st, ss, mid, qs, qe, 2*index+1),
                  RMQUtil(hist, st, mid+1, se, qs, qe, 2*index+2));
}

// Return index of minimum element in range from index qs (query start) to
// qe (query end). It mainly uses RMQUtil()
int RMQ(int *hist, int *st, int n, int qs, int qe)
{
    // Check for erroneous input values
    if (qs < 0 || qe > n-1 || qs > qe)
    {
        cout << "Invalid Input";
        return -1;
    }

    return RMQUtil(hist, st, 0, n-1, qs, qe, 0);
}

// A recursive function that constructs Segment Tree for hist[ss..se].
// si is index of current node in segment tree st
int constructSTUtil(int hist[], int ss, int se, int *st, int si)
{
    // If there is one element in array, store it in current node of
    // segment tree and return

```

```

if (ss == se)
    return (st[si] = ss);

// If there are more than one elements, then recur for left and
// right subtrees and store the minimum of two values in this node
int mid = getMid(ss, se);
st[si] = minVal(hist, constructSTUtil(hist, ss, mid, st, si*2+1),
                constructSTUtil(hist, mid+1, se, st, si*2+2));
return st[si];
}

/* Function to construct segment tree from given array. This function
allocates memory for segment tree and calls constructSTUtil() to
fill the allocated memory */
int *constructST(int hist[], int n)
{
    // Allocate memory for segment tree
    int x = (int)(ceil(log2(n))); //Height of segment tree
    int max_size = 2*(int)pow(2, x) - 1; //Maximum size of segment tree
    int *st = new int[max_size];

    // Fill the allocated memory st
    constructSTUtil(hist, 0, n-1, st, 0);

    // Return the constructed segment tree
    return st;
}

// A recursive function to find the maximum rectangular area.
// It uses segment tree 'st' to find the minimum value in hist[l..r]
int getMaxAreaRec(int *hist, int *st, int n, int l, int r)
{
    // Base cases
    if (l > r)  return INT_MIN;
    if (l == r)  return hist[l];

    // Find index of the minimum value in given range
    // This takes O(Logn)time
    int m = RMQ(hist, st, n, l, r);

    /* Return maximum of following three possible cases
     a) Maximum area in Left of min value (not including the min)
     a) Maximum area in right of min value (not including the min)
     c) Maximum area including min */
    return max(getMaxAreaRec(hist, st, n, l, m-1),
               getMaxAreaRec(hist, st, n, m+1, r),
               (r-l+1)*(hist[m]) );
}

```

```
// The main function to find max area
int getMaxArea(int hist[], int n)
{
    // Build segment tree from given array. This takes
    // O(n) time
    int *st = constructST(hist, n);

    // Use recursive utility function to find the
    // maximum area
    return getMaxAreaRec(hist, st, n, 0, n-1);
}

// Driver program to test above functions
int main()
{
    int hist[] = {6, 1, 5, 4, 5, 2, 6};
    int n = sizeof(hist)/sizeof(hist[0]);
    cout << "Maximum area is " << getMaxArea(hist, n);
    return 0;
}
```

Output:

```
Maximum area is 12
```

This problem can be solved in linear time. See below [set 2](#) for linear time solution.
[Linear time solution for Largest Rectangular Area in a Histogram](#)

Source

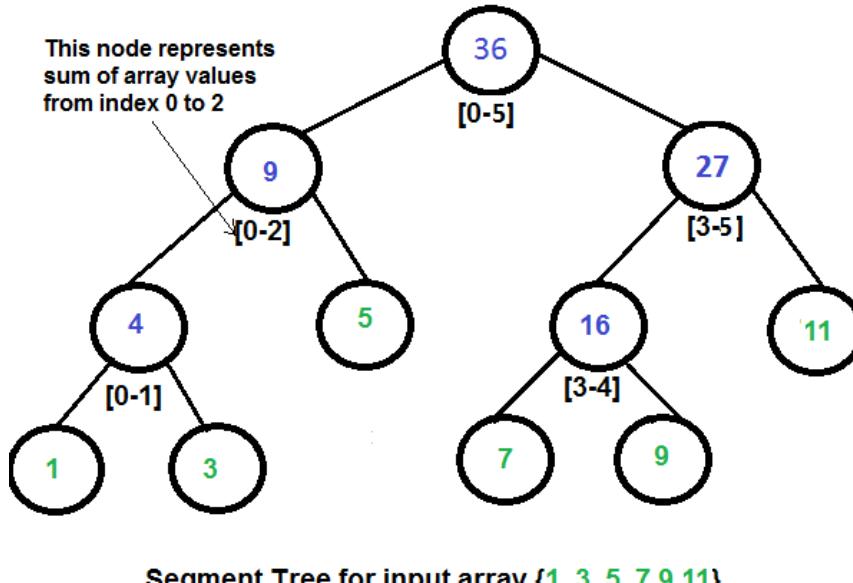
<https://www.geeksforgeeks.org/largest-rectangular-area-in-a-histogram-set-1/>

Chapter 97

Lazy Propagation in Segment Tree

Lazy Propagation in Segment Tree - GeeksforGeeks

Segment tree is introduced in [previous post](#) with an example of range sum problem. We have used the same “Sum of given Range” problem to explain Lazy propagation



How does update work in Simple Segment Tree?

In the [previous post](#), update function was called to update only a single value in array. Please note that a single value update in array may cause multiple updates in Segment Tree as there may be many segment tree nodes that have a single array element in their ranges.

Below is simple logic used in previous post.

- 1) Start with root of segment tree.
- 2) If array index to be updated is not in current node's range, then return
- 3) Else update current node and recur for children.

Below is code taken from previous post.

```
/* A recursive function to update the nodes which have the given
index in their range. The following are parameters
tree[] --> segment tree
si      --> index of current node in segment tree.
           Initial value is passed as 0.
ss and se --> Starting and ending indexes of array elements
             covered under this node of segment tree.
             Initial values passed as 0 and n-1.
i       --> index of the element to be updated. This index
           is in input array.
diff --> Value to be added to all nodes which have array
        index i in range */
void updateValueUtil(int tree[], int ss, int se, int i,
                     int diff, int si)
{
    // Base Case: If the input index lies outside the range
    // of this segment
    if (i < ss || i > se)
        return;

    // If the input index is in range of this node, then
    // update the value of the node and its children
    st[si] = st[si] + diff;
    if (se != ss)
    {
        int mid = getMid(ss, se);
        updateValueUtil(st, ss, mid, i, diff, 2*si + 1);
        updateValueUtil(st, mid+1, se, i, diff, 2*si + 2);
    }
}
```

What if there are updates on a range of array indexes?

For example add 10 to all values at indexes from 2 to 7 in array. The above update has to be called for every index from 2 to 7. We can avoid multiple calls by writing a function updateRange() that updates nodes accordingly.

```
/* Function to update segment tree for range update in input
array.
si -> index of current node in segment tree
ss and se -> Starting and ending indexes of elements for
           which current nodes stores sum.
```

```

us and ue -> starting and ending indexes of update query
diff -> which we need to add in the range us to ue */
void updateRangeUtil(int si, int ss, int se, int us,
                     int ue, int diff)
{
    // out of range
    if (ss>se || ss>ue || se<us)
        return ;

    // Current node is a leaf node
    if (ss==se)
    {
        // Add the difference to current node
        tree[si] += diff;
        return;
    }

    // If not a leaf node, recur for children.
    int mid = (ss+se)/2;
    updateRangeUtil(si*2+1, ss, mid, us, ue, diff);
    updateRangeUtil(si*2+2, mid+1, se, us, ue, diff);

    // Use the result of children calls to update this
    // node
    tree[si] = tree[si*2+1] + tree[si*2+2];
}

```

Lazy Propagation – An optimization to make range updates faster

When there are many updates and updates are done on a range, we can postpone some updates (avoid recursive calls in update) and do those updates only when required.

Please remember that a node in segment tree stores or represents result of a query for a range of indexes. And if this node's range lies within the update operation range, then all descendants of the node must also be updated. For example consider the node with value 27 in above diagram, this node stores sum of values at indexes from 3 to 5. If our update query is for range 2 to 5, then we need to update this node and all descendants of this node. With Lazy propagation, we update only node with value 27 and postpone updates to its children by storing this update information in separate nodes called lazy nodes or values. We create an array `lazy[]` which represents lazy node. Size of `lazy[]` is same as array that represents segment tree, which is `tree[]` in below code.

The idea is to initialize all elements of `lazy[]` as 0. A value 0 in `lazy[i]` indicates that there are no pending updates on node i in segment tree. A non-zero value of `lazy[i]` means that this amount needs to be added to node i in segment tree before making any query to the node.

Below is modified update method.

```

// To update segment tree for change in array
// values at array indexes from us to ue.
updateRange(us, ue)
1) If current segment tree node has any pending
   update, then first add that pending update to
   current node.
2) If current node's range lies completely in
   update query range.
....a) Update current node
....b) Postpone updates to children by setting
   lazy value for children nodes.
3) If current node's range overlaps with update
   range, follow the same approach as above simple
   update.
...a) Recur for left and right children.
...b) Update current node using results of left
   and right calls.

```

Is there any change in Query Function also?

Since we have changed update to postpone its operations, there may be problems if a query is made to a node that is yet to be updated. So we need to update our query method also which is [getSumUtil in previous post](#). The getSumUtil() now first checks if there is a pending update and if there is, then updates the node. Once it makes sure that pending update is done, it works same as the previous getSumUtil().

Below are programs to demonstrate working of Lazy Propagation.

C/C++

```

// Program to show segment tree to demonstrate lazy
// propagation
#include <stdio.h>
#include <math.h>
#define MAX 1000

// Ideally, we should not use global variables and large
// constant-sized arrays, we have done it here for simplicity.
int tree[MAX] = {0}; // To store segment tree
int lazy[MAX] = {0}; // To store pending updates

/* si -> index of current node in segment tree
   ss and se -> Starting and ending indexes of elements for
                 which current nodes stores sum.
   us and ue -> starting and ending indexes of update query
   diff -> which we need to add in the range us to ue */
void updateRangeUtil(int si, int ss, int se, int us,
                     int ue, int diff)
{
    // If lazy value is non-zero for current node of segment

```

```

// tree, then there are some pending updates. So we need
// to make sure that the pending updates are done before
// making new updates. Because this value may be used by
// parent after recursive calls (See last line of this
// function)
if (lazy[si] != 0)
{
    // Make pending updates using value stored in lazy
    // nodes
    tree[si] += (se-ss+1)*lazy[si];

    // checking if it is not leaf node because if
    // it is leaf node then we cannot go further
    if (ss != se)
    {
        // We can postpone updating children we don't
        // need their new values now.
        // Since we are not yet updating children of si,
        // we need to set lazy flags for the children
        lazy[si*2 + 1] += lazy[si];
        lazy[si*2 + 2] += lazy[si];
    }

    // Set the lazy value for current node as 0 as it
    // has been updated
    lazy[si] = 0;
}

// out of range
if (ss>se || ss>ue || se<us)
    return ;

// Current segment is fully in range
if (ss>=us && se<=ue)
{
    // Add the difference to current node
    tree[si] += (se-ss+1)*diff;

    // same logic for checking leaf node or not
    if (ss != se)
    {
        // This is where we store values in lazy nodes,
        // rather than updating the segment tree itself
        // Since we don't need these updated values now
        // we postpone updates by storing values in lazy[]
        lazy[si*2 + 1] += diff;
        lazy[si*2 + 2] += diff;
    }
}

```

```

        return;
    }

    // If not completely in rang, but overlaps, recur for
    // children,
    int mid = (ss+se)/2;
    updateRangeUtil(si*2+1, ss, mid, us, ue, diff);
    updateRangeUtil(si*2+2, mid+1, se, us, ue, diff);

    // And use the result of children calls to update this
    // node
    tree[si] = tree[si*2+1] + tree[si*2+2];
}

// Function to update a range of values in segment
// tree
/* us and eu -> starting and ending indexes of update query
   ue -> ending index of update query
   diff -> which we need to add in the range us to ue */
void updateRange(int n, int us, int ue, int diff)
{
    updateRangeUtil(0, 0, n-1, us, ue, diff);
}

/*
 * A recursive function to get the sum of values in given
 * range of the array. The following are parameters for
 * this function.
 * si --> Index of current node in the segment tree.
 *       Initially 0 is passed as root is always at'
 *       index 0
 * ss & se --> Starting and ending indexes of the
 *               segment represented by current node,
 *               i.e., tree[si]
 * qs & qe --> Starting and ending indexes of query
 *               range */
int getSumUtil(int ss, int se, int qs, int qe, int si)
{
    // If lazy flag is set for current node of segment tree,
    // then there are some pending updates. So we need to
    // make sure that the pending updates are done before
    // processing the sub sum query
    if (lazy[si] != 0)
    {
        // Make pending updates to this node. Note that this
        // node represents sum of elements in arr[ss..se] and
        // all these elements must be increased by lazy[si]
        tree[si] += (se-ss+1)*lazy[si];
    }
}

```

```

// checking if it is not leaf node because if
// it is leaf node then we cannot go further
if (ss != se)
{
    // Since we are not yet updating children os si,
    // we need to set lazy values for the children
    lazy[si*2+1] += lazy[si];
    lazy[si*2+2] += lazy[si];
}

// unset the lazy value for current node as it has
// been updated
lazy[si] = 0;
}

// Out of range
if (ss>se || ss>qe || se<qs)
    return 0;

// At this point we are sure that pending lazy updates
// are done for current node. So we can return value
// (same as it was for query in our previous post)

// If this segment lies in range
if (ss>=qs && se<=qe)
    return tree[si];

// If a part of this segment overlaps with the given
// range
int mid = (ss + se)/2;
return getSumUtil(ss, mid, qs, qe, 2*si+1) +
       getSumUtil(mid+1, se, qs, qe, 2*si+2);
}

// Return sum of elements in range from index qs (quey
// start) to qe (query end). It mainly uses getSumUtil()
int getSum(int n, int qs, int qe)
{
    // Check for erroneous input values
    if (qs < 0 || qe > n-1 || qs > qe)
    {
        printf("Invalid Input");
        return -1;
    }

    return getSumUtil(0, n-1, qs, qe, 0);
}

```

```
// A recursive function that constructs Segment Tree for
// array[ss..se]. si is index of current node in segment
// tree st.
void constructSTUtil(int arr[], int ss, int se, int si)
{
    // out of range as ss can never be greater than se
    if (ss > se)
        return ;

    // If there is one element in array, store it in
    // current node of segment tree and return
    if (ss == se)
    {
        tree[si] = arr[ss];
        return;
    }

    // If there are more than one elements, then recur
    // for left and right subtrees and store the sum
    // of values in this node
    int mid = (ss + se)/2;
    constructSTUtil(arr, ss, mid, si*2+1);
    constructSTUtil(arr, mid+1, se, si*2+2);

    tree[si] = tree[si*2 + 1] + tree[si*2 + 2];
}

/* Function to construct segment tree from given array.
   This function allocates memory for segment tree and
   calls constructSTUtil() to fill the allocated memory */
void constructST(int arr[], int n)
{
    // Fill the allocated memory st
    constructSTUtil(arr, 0, n-1, 0);
}

// Driver program to test above functions
int main()
{
    int arr[] = {1, 3, 5, 7, 9, 11};
    int n = sizeof(arr)/sizeof(arr[0]);

    // Build segment tree from given array
    constructST(arr, n);

    // Print sum of values in array from index 1 to 3
```

```

printf("Sum of values in given range = %d\n",
       getSum(n, 1, 3));

// Add 10 to all nodes at indexes from 1 to 5.
updateRange(n, 1, 5, 10);

// Find sum after the value is updated
printf("Updated sum of values in given range = %d\n",
       getSum( n, 1, 3));

return 0;
}

```

Java

```

// Java program to demonstrate lazy propagation in segment tree
class LazySegmentTree
{
    final int MAX = 1000;           // Max tree size
    int tree[] = new int[MAX];     // To store segment tree
    int lazy[] = new int[MAX];     // To store pending updates

    /* si -> index of current node in segment tree
       ss and se -> Starting and ending indexes of elements for
                      which current nodes stores sum.
       us and eu -> starting and ending indexes of update query
       ue -> ending index of update query
       diff -> which we need to add in the range us to ue */
    void updateRangeUtil(int si, int ss, int se, int us,
                         int ue, int diff)
    {
        // If lazy value is non-zero for current node of segment
        // tree, then there are some pending updates. So we need
        // to make sure that the pending updates are done before
        // making new updates. Because this value may be used by
        // parent after recursive calls (See last line of this
        // function)
        if (lazy[si] != 0)
        {
            // Make pending updates using value stored in lazy
            // nodes
            tree[si] += (se - ss + 1) * lazy[si];

            // checking if it is not leaf node because if
            // it is leaf node then we cannot go further
            if (ss != se)
            {
                // We can postpone updating children we don't

```

```

        // need their new values now.
        // Since we are not yet updating children of si,
        // we need to set lazy flags for the children
        lazy[si * 2 + 1] += lazy[si];
        lazy[si * 2 + 2] += lazy[si];
    }

    // Set the lazy value for current node as 0 as it
    // has been updated
    lazy[si] = 0;
}

// out of range
if (ss > se || ss > ue || se < us)
    return;

// Current segment is fully in range
if (ss >= us && se <= ue)
{
    // Add the difference to current node
    tree[si] += (se - ss + 1) * diff;

    // same logic for checking leaf node or not
    if (ss != se)
    {
        // This is where we store values in lazy nodes,
        // rather than updating the segment tree itself
        // Since we don't need these updated values now
        // we postpone updates by storing values in lazy[]
        lazy[si * 2 + 1] += diff;
        lazy[si * 2 + 2] += diff;
    }
    return;
}

// If not completely in rang, but overlaps, recur for
// children,
int mid = (ss + se) / 2;
updateRangeUtil(si * 2 + 1, ss, mid, us, ue, diff);
updateRangeUtil(si * 2 + 2, mid + 1, se, us, ue, diff);

// And use the result of children calls to update this
// node
tree[si] = tree[si * 2 + 1] + tree[si * 2 + 2];
}

// Function to update a range of values in segment
// tree

```

```

/* us and eu -> starting and ending indexes of update query
   ue -> ending index of update query
   diff -> which we need to add in the range us to ue */
void updateRange(int n, int us, int ue, int diff) {
    updateRangeUtil(0, 0, n - 1, us, ue, diff);
}

/* A recursive function to get the sum of values in given
range of the array. The following are parameters for
this function.
si --> Index of current node in the segment tree.
      Initially 0 is passed as root is always at'
      index 0
ss & se --> Starting and ending indexes of the
            segment represented by current node,
            i.e., tree[si]
qs & qe --> Starting and ending indexes of query
            range */

int getSumUtil(int ss, int se, int qs, int qe, int si)
{
    // If lazy flag is set for current node of segment tree,
    // then there are some pending updates. So we need to
    // make sure that the pending updates are done before
    // processing the sub sum query
    if (lazy[si] != 0)
    {
        // Make pending updates to this node. Note that this
        // node represents sum of elements in arr[ss..se] and
        // all these elements must be increased by lazy[si]
        tree[si] += (se - ss + 1) * lazy[si];

        // checking if it is not leaf node because if
        // it is leaf node then we cannot go further
        if (ss != se)
        {
            // Since we are not yet updating children os si,
            // we need to set lazy values for the children
            lazy[si * 2 + 1] += lazy[si];
            lazy[si * 2 + 2] += lazy[si];
        }

        // unset the lazy value for current node as it has
        // been updated
        lazy[si] = 0;
    }

    // Out of range
    if (ss > se || ss > qe || se < qs)

```

```

        return 0;

    // At this point sure, pending lazy updates are done
    // for current node. So we can return value (same as
    // was for query in our previous post)

    // If this segment lies in range
    if (ss >= qs && se <= qe)
        return tree[si];

    // If a part of this segment overlaps with the given
    // range
    int mid = (ss + se) / 2;
    return getSumUtil(ss, mid, qs, qe, 2 * si + 1) +
           getSumUtil(mid + 1, se, qs, qe, 2 * si + 2);
}

// Return sum of elements in range from index qs (query
// start) to qe (query end). It mainly uses getSumUtil()
int getSum(int n, int qs, int qe)
{
    // Check for erroneous input values
    if (qs < 0 || qe > n - 1 || qs > qe)
    {
        System.out.println("Invalid Input");
        return -1;
    }

    return getSumUtil(0, n - 1, qs, qe, 0);
}

/* A recursive function that constructs Segment Tree for
array[ss..se]. si is index of current node in segment
tree st. */
void constructSTUtil(int arr[], int ss, int se, int si)
{
    // out of range as ss can never be greater than se
    if (ss > se)
        return;

    /* If there is one element in array, store it in
    current node of segment tree and return */
    if (ss == se)
    {
        tree[si] = arr[ss];
        return;
    }
}

```

```

/* If there are more than one elements, then recur
   for left and right subtrees and store the sum
   of values in this node */
int mid = (ss + se) / 2;
constructSTUtil(arr, ss, mid, si * 2 + 1);
constructSTUtil(arr, mid + 1, se, si * 2 + 2);

tree[si] = tree[si * 2 + 1] + tree[si * 2 + 2];
}

/* Function to construct segment tree from given array.
   This function allocates memory for segment tree and
   calls constructSTUtil() to fill the allocated memory */
void constructST(int arr[], int n)
{
    // Fill the allocated memory st
    constructSTUtil(arr, 0, n - 1, 0);
}

// Driver program to test above functions
public static void main(String args[])
{
    int arr[] = {1, 3, 5, 7, 9, 11};
    int n = arr.length;
    LazySegmentTree tree = new LazySegmentTree();

    // Build segment tree from given array
    tree.constructST(arr, n);

    // Print sum of values in array from index 1 to 3
    System.out.println("Sum of values in given range = " +
                       tree.getSum(n, 1, 3));

    // Add 10 to all nodes at indexes from 1 to 5.
    tree.updateRange(n, 1, 5, 10);

    // Find sum after the value is updated
    System.out.println("Updated sum of values in given range = " +
                       tree.getSum(n, 1, 3));
}
}

// This Code is contributed by Ankur Narain Verma

```

Output:

```

Sum of values in given range = 15
Updated sum of values in given range = 45

```

This article is contributed by **Ankit Mittal**. 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/lazy-propagation-in-segment-tree/>

Chapter 98

Left-Child Right-Sibling Representation of Tree

Left-Child Right-Sibling Representation of Tree - GeeksforGeeks

An n-ary tree in computer science is a collection of nodes normally represented hierarchically in the following fashion.

1. The tree starts at the root node.
2. Each node of the tree holds a list of references to its child nodes.
3. The number of children a node has is less than or equal to n.

A typical representation of n-ary tree uses an array of n references (or pointers) to store children (Note that n is an upper bound on number of children). Can we do better? the idea of Left-Child Right- Sibling representation is to store only two pointers in every node.

Left-Child Right Sibling Representation

It is a different representation of an n-ary tree where instead of holding a reference to each and every child node, a node holds just two references, first a reference to it's first child, and the other to it's immediate next sibling. This new transformation not only removes the need of advance knowledge of the number of children a node has, but also limits the number of references to a maximum of two, thereby making it so much easier to code. One thing to note is that in the previous representation a link between two nodes denoted a parent-child relationship whereas in this representation a link between two nodes may denote a parent-child relationship or a sibling-sibling relationship.

Advantages :

1. This representation saves up memory by limiting the maximum number of references required per node to two.
2. It is easier to code.

Disadvantages :

1. Basic operations like searching insertion/deletion tend to take a longer time because in

order to find the appropriate position we would have to traverse through all the siblings of the node to be searched/inserted/deleted (in the worst case).

The image on the left is the normal representation of a 6-ary tree and the one on the right is its corresponding Left-Child Right-Sibling representation.

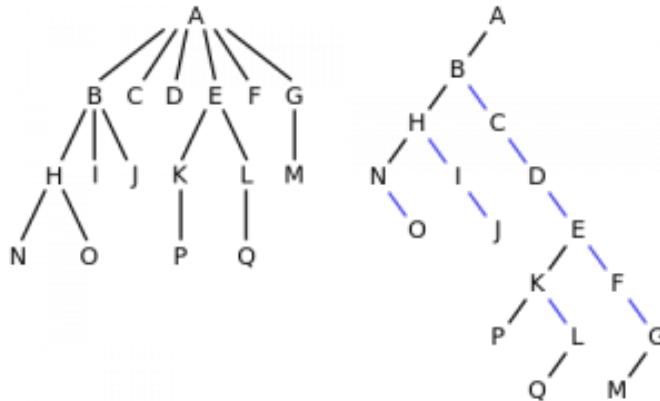


Image source : https://en.wikipedia.org/wiki/Left-child_right-sibling_binary_tree

An Example Problem :

Now let's see a problem and try to solve it using both the discussed representations for clarity.

Given a family tree. Find the k th child of some member X in the tree.

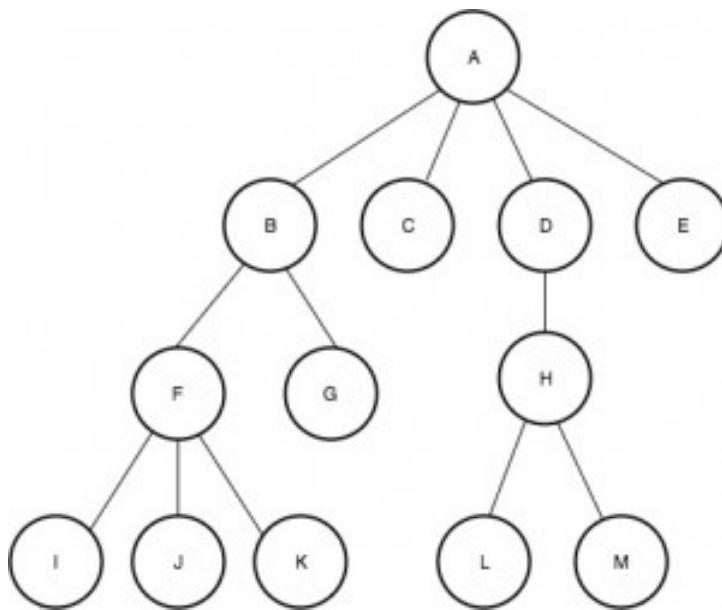
The user inputs two things.

1. A character P (representing the parent whose child is to be found)
2. An integer k (representing the child number)

The problem in itself looks pretty easy. The only issue here is that the maximum number of children a node can have is unspecified which makes it rather tricky to construct the tree.

Example:

Consider the following family tree.



Input : A 2

Output : C

In this case, the user wishes to know A's second child which according to the figure is C.

Input : F 3

Output : K

Similar to the first case, the user wishes to know F's third child which is K.

Method 1 (Storing n pointers with every node):

In this method, we assume the maximum number of children a node can have and proceed further. The only (obvious) problem with this method is the upper bound on the number of children. If the value is too low, then the code would fail for certain cases and if the value is too high, then a huge amount of memory is unnecessarily wasted.

If the programmer beforehand knows the structure of the tree, then the upper bound can be set to the maximum number of children a node has in that particular structure. But even in that case, there will be some memory wastage (all nodes may not necessarily have the same number of children, some may even have less. **Example: Leaf nodes have no children**).

```

// C++ program to find k-th child of a given
// node using typical representation that uses
// an array of pointers.
#include <iostream>
  
```

```

using namespace std;

// Maximum number of children
const int N = 10;

class Node
{
public:
    char val;
    Node * child[N];
    Node(char P)
    {
        val = P;
        for (int i=0; i<MAX; i++)
            child[i] = NULL;
    }
};

// Traverses given n-ary tree to find K-th
// child of P.
void printKthChild(Node *root, char P, int k)
{
    // If P is current root
    if (root->val == P)
    {
        if (root->child[k-1] == NULL)
            cout << "Error : Does not exist\n";
        else
            cout << root->child[k-1]->val << endl;
    }

    // If P lies in a subtree
    for (int i=0; i<N; i++)
        if (root->child[i] != NULL)
            printKthChild(root->child[i], P, k);
}

// Driver code
int main()
{
    Node *root = new Node('A');
    root->child[0] = new Node('B');
    root->child[1] = new Node('C');
    root->child[2] = new Node('D');
    root->child[3] = new Node('E');
    root->child[0]->child[0] = new Node('F');
    root->child[0]->child[1] = new Node('G');
    root->child[2]->child[0] = new Node('H');
}

```

```

root->child[0]->child[0]->child[0] = new Node('I');
root->child[0]->child[0]->child[1] = new Node('J');
root->child[0]->child[0]->child[2] = new Node('K');
root->child[2]->child[0]->child[0] = new Node('L');
root->child[2]->child[0]->child[1] = new Node('M');

// Print F's 2nd child
char P = 'F';
cout << "F's second child is : ";
printKthChild(root, P, 2);

P = 'A';
cout << "A's seventh child is : ";
printKthChild(root, P, 7);
return 0;
}

```

Output:

```

F's second child is : J
A's seventh child is : Error : Does not exist

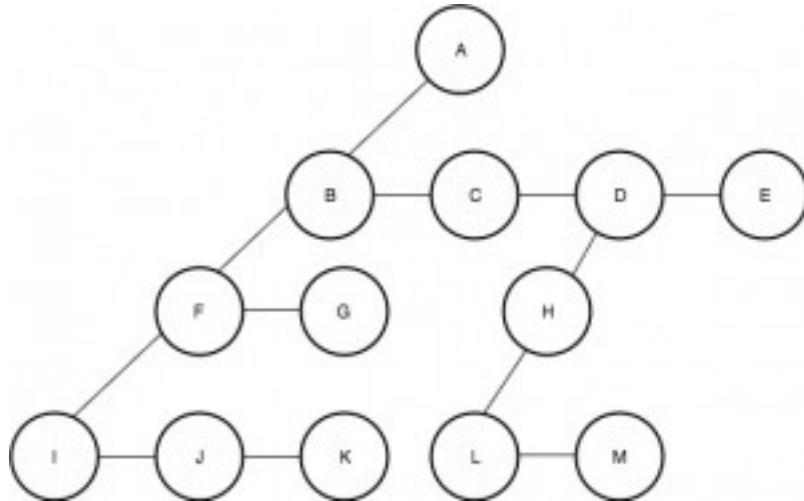
```

In the above tree, had there been a node which had say, 15 children, then this code would have given a Segmentation fault.

Method 2 : (Left-Child Right-Sibling Representation)

In this method, we change the structure of the family tree. In the standard tree, each parent node is connected to all of its children. Here as discussed above, instead of having each node store pointers to all of its children, a node will store pointer to just one of its child. Apart from this the node will also store a pointer to its immediate right sibling.

The image below is the Left-Child Right-Sibling equivalent of the example used above.



```
// C++ program to find k-th child of a given
// Node using typical representation that uses
// an array of pointers.
#include <iostream>
using namespace std;

// A Node to represent left child right sibling
// representation.
class Node
{
public:
    char val;
    Node *child;
    Node *next;
    Node(char P)
    {
        val = P;
        child = NULL;
        next = NULL;
    }
};

// Traverses given n-ary tree to find K-th
// child of P.
void printKthChild(Node *root, char P, int k)
{
    if (root == NULL)
        return;

    // If P is present at root itself
    if (root->val == P)
    {
        // Traverse children of root starting
        // from left child
        Node *t = root->child;
        int i = 1;
        while (t != NULL && i < k)
        {
            t = t->next;
            i++;
        }
        if (t == NULL)
            cout << "Error : Does not exist\n";
        else
            cout << t->val << " " << endl;
        return;
    }
}
```

```
printKthChild(root->child, P, k);
printKthChild(root->next, P, k);
}

// Driver code
int main()
{
    Node *root = new Node('A');
    root->child = new Node('B');
    root->child->next = new Node('C');
    root->child->next->next = new Node('D');
    root->child->next->next->next = new Node('E');
    root->child->child = new Node('F');
    root->child->child->next = new Node('G');
    root->child->next->next->child = new Node('H');
    root->child->next->next->child->child = new Node('L');
    root->child->next->next->child->child->next = new Node('M');
    root->child->child->child = new Node('I');
    root->child->child->child->next = new Node('J');
    root->child->child->child->next->next = new Node('K');

    // Print F's 2nd child
    char P = 'F';
    cout << "F's second child is : ";
    printKthChild(root, P, 2);

    P = 'A';
    cout << "A's seventh child is : ";
    printKthChild(root, P, 7);
    return 0;
}
```

Output:

```
F's second child is : J
A's seventh child is : Error : Does not exist
```

Related Article :

[Creating a tree with Left-Child Right-Sibling Representation](#)

Source

<https://www.geeksforgeeks.org/left-child-right-sibling-representation-tree/>

Chapter 99

Leftist Tree / Leftist Heap

Leftist Tree / Leftist Heap - GeeksforGeeks

A leftist tree or leftist heap is a priority queue implemented with a variant of a binary heap. Every node has an **s-value (or rank or distance)** which is the distance to the nearest leaf. In contrast to a binary heap (Which is always a [complete binary tree](#)), a leftist tree may be very unbalanced.

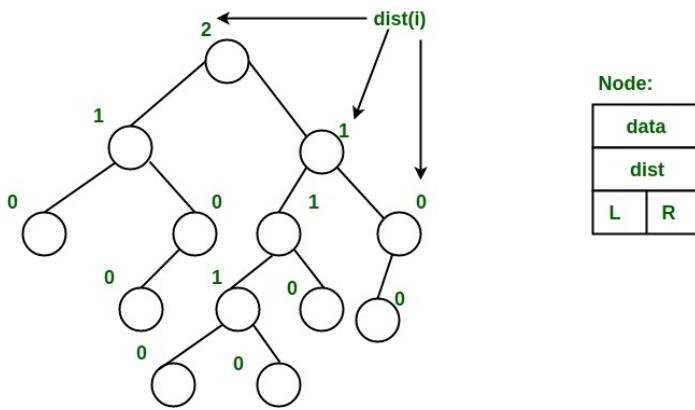
Below are [time complexities](#) of **Leftist Tree / Heap**.

Function	Complexity	Comparison
1) Get Min:	$O(1)$	[same as both Binary and Binomial]
2) Delete Min:	$O(\log n)$	[same as both Binary and Binomial]
3) Insert:	$O(\log n)$	$[O(\log n) \text{ in Binary and } O(1) \text{ in Binomial and } O(\log n) \text{ for worst case}]$
4) Merge:	$O(\log n)$	$[O(\log n) \text{ in Binomial}]$

A leftist tree is a binary tree with properties:

1. **Normal Min Heap Property :** $\text{key}(i) \geq \text{key}(\text{parent}(i))$
2. **Heavier on left side :** $\text{dist}(\text{right}(i)) \leq \text{dist}(\text{left}(i))$. Here, $\text{dist}(i)$ is the number of edges on the shortest path from node i to a leaf node in extended binary tree representation (In this representation, a null child is considered as external or leaf node). The shortest path to a descendant external node is through the right child. Every subtree is also a leftist tree and $\text{dist}(i) = 1 + \text{dist}(\text{right}(i))$.

Example: The below leftist tree is presented with its distance calculated for each node with the procedure mentioned above. The rightmost node has a rank of 0 as the right subtree of this node is null and its parent has a distance of 1 by $\text{dist}(i) = 1 + \text{dist}(\text{right}(i))$. The same is followed for each node and their s-value(or rank) is calculated.



From above second property, we can draw two conclusions :

1. The path from root to rightmost leaf is the shortest path from root to a leaf.
2. If the path to rightmost leaf has x nodes, then leftist heap has atleast $2^x - 1$ nodes.
This means the length of path to rightmost leaf is $O(\log n)$ for a leftist heap with n nodes.

Operations :

1. The main operation is `merge()`.
2. `deleteMin()` (or `extractMin()`) can be done by removing root and calling `merge()` for left and right subtrees.
3. `insert()` can be done by creating a leftist tree with single key (key to be inserted) and calling `merge()` for given tree and tree with single node.

Idea behind Merging :

Since right subtree is smaller, the idea is to merge right subtree of a tree with other tree. Below are abstract steps.

1. Put the root with smaller value as the new root.
2. Hang its left subtree on the left.
3. Recursively merge its right subtree and the other tree.
4. Before returning from recursion:
 - Update `dist()` of merged root.
 - Swap left and right subtrees just below root, if needed, to keep leftist property of merged result

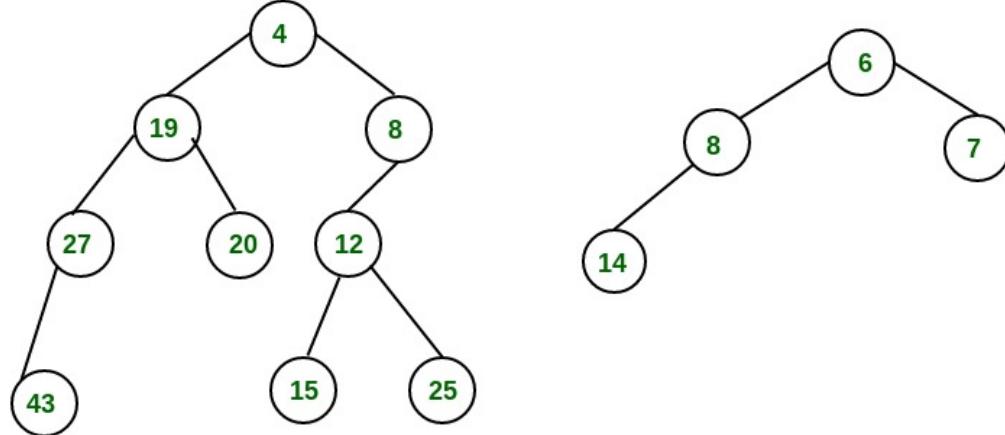
Source : <http://courses.cs.washington.edu/courses/cse326/08sp/lectures/05-leftist-heaps.pdf>

Detailed Steps for Merge:

1. Compare the roots of two heaps.
2. Push the smaller key into an empty stack, and move to the right child of smaller key.
3. Recursively compare two keys and go on pushing the smaller key onto the stack and move to its right child.
4. Repeat until a null node is reached.
5. Take the last node processed and make it the right child of the node at top of the stack, and convert it to leftist heap if the properties of leftist heap are violated.
6. Recursively go on popping the elements from the stack and making them the right child of new stack top.

Example:

Consider two leftist heaps given below:



Merge them into a single leftist heap

Compare(4,6)

Push 4

Compare(8,6)

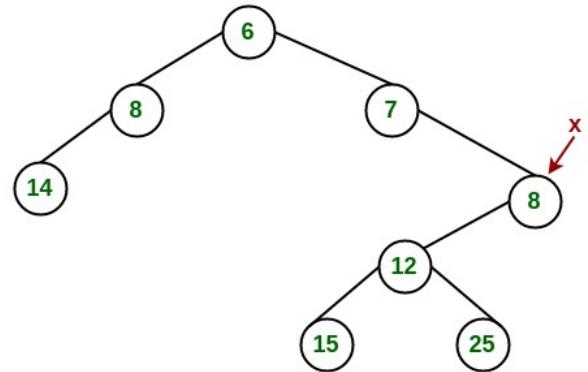
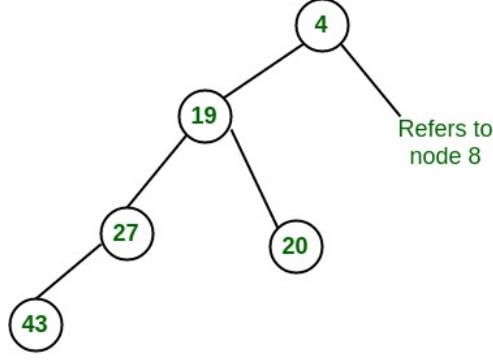
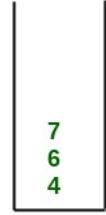
Push 6

Compare(8,7)

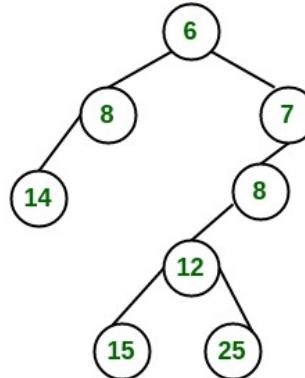
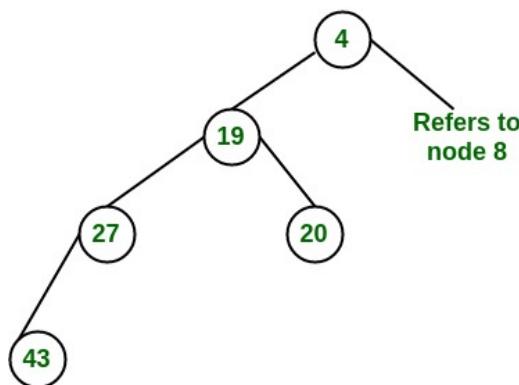
Push 7

Compare(8,null)

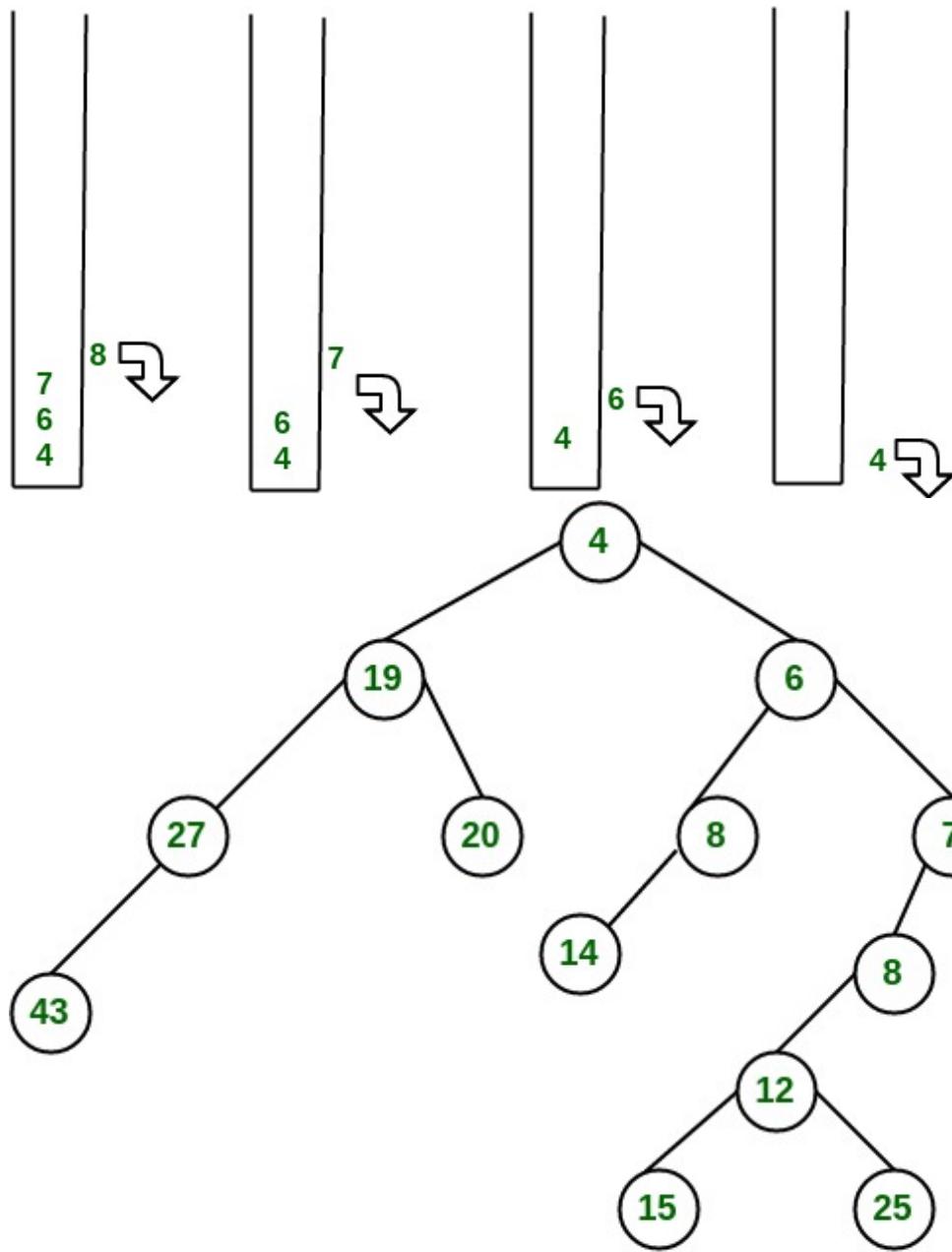
As null is encountered, we make node 8 as right sub-tree of stack top, i.e. 7



The subtree at node 7 violates the property of leftist heap so we swap it with the left child and retain the property of leftist heap.

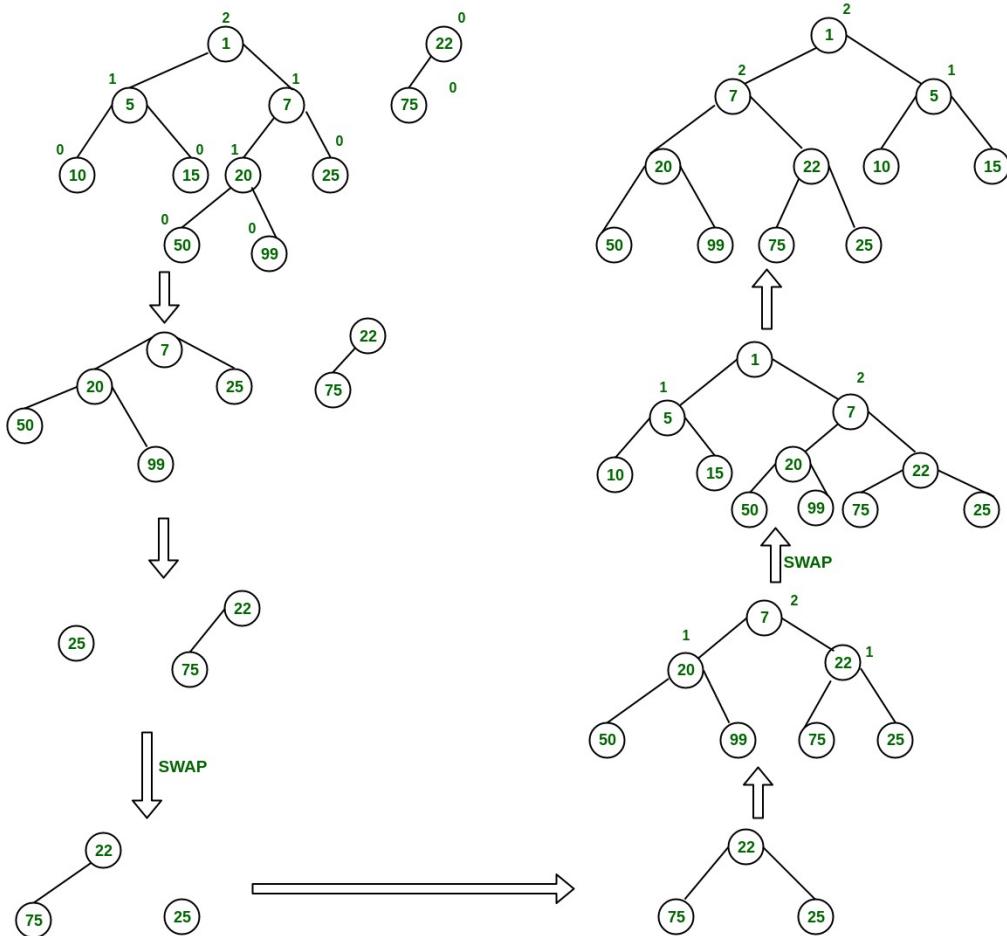


Convert to leftist heap. Repeat the process



The worst case time complexity of this algorithm is $O(\log n)$ in the worst case, where n is the number of nodes in the leftist heap.

Another example of merging two leftist heap:



Implementation of leftist Tree / leftist Heap:

```
//C++ program for leftist heap / leftist tree
#include <iostream>
#include <cstdlib>
using namespace std;

// Node Class Declaration
class LeftistNode
{
public:
    int element;
    LeftistNode *left;
    LeftistNode *right;
    int dist;
    LeftistNode(int & element, LeftistNode *lt = NULL,
               LeftistNode *rt = NULL, int np = 0)
```

```

    {
        this->element = element;
        right = rt;
        left = lt,
        dist = np;
    }
};

//Class Declaration
class LeftistHeap
{
public:
    LeftistHeap();
    LeftistHeap(LeftistHeap &rhs);
    ~LeftistHeap();
    bool isEmpty();
    bool isFull();
    int &findMin();
    void Insert(int &x);
    void deleteMin();
    void deleteMin(int &minItem);
    void makeEmpty();
    void Merge(LeftistHeap &rhs);
    LeftistHeap & operator =(LeftistHeap &rhs);
private:
    LeftistNode *root;
    LeftistNode *Merge(LeftistNode *h1,
                      LeftistNode *h2);
    LeftistNode *Merge1(LeftistNode *h1,
                      LeftistNode *h2);
    void swapChildren(LeftistNode * t);
    void reclaimMemory(LeftistNode * t);
    LeftistNode *clone(LeftistNode *t);
};

// Construct the leftist heap
LeftistHeap::LeftistHeap()
{
    root = NULL;
}

// Copy constructor.
LeftistHeap::LeftistHeap(LeftistHeap &rhs)
{
    root = NULL;
    *this = rhs;
}

```

```

// Destruct the leftist heap
LeftistHeap::~LeftistHeap()
{
    makeEmpty( );
}

/* Merge rhs into the priority queue.
rhs becomes empty. rhs must be different
from this.*/
void LeftistHeap::Merge(LeftistHeap &rhs)
{
    if (this == &rhs)
        return;
    root = Merge(root, rhs.root);
    rhs.root = NULL;
}

/* Internal method to merge two roots.
Deals with deviant cases and calls recursive Merge1.*/
LeftistNode *LeftistHeap::Merge(LeftistNode * h1,
                               LeftistNode * h2)
{
    if (h1 == NULL)
        return h2;
    if (h2 == NULL)
        return h1;
    if (h1->element < h2->element)
        return Merge1(h1, h2);
    else
        return Merge1(h2, h1);
}

/* Internal method to merge two roots.
Assumes trees are not empty, and h1's root contains
smallest item.*/
LeftistNode *LeftistHeap::Merge1(LeftistNode * h1,
                               LeftistNode * h2)
{
    if (h1->left == NULL)
        h1->left = h2;
    else
    {
        h1->right = Merge(h1->right, h2);
        if (h1->left->dist < h1->right->dist)
            swapChildren(h1);
        h1->dist = h1->right->dist + 1;
    }
    return h1;
}

```

```

}

// Swaps t's two children.
void LeftistHeap::swapChildren(LeftistNode * t)
{
    LeftistNode *tmp = t->left;
    t->left = t->right;
    t->right = tmp;
}

/* Insert item x into the priority queue, maintaining
   heap order.*/
void LeftistHeap::Insert(int &x)
{
    root = Merge(new LeftistNode(x), root);
}

/* Find the smallest item in the priority queue.
   Return the smallest item, or throw Underflow if empty.*/
int &LeftistHeap::findMin()
{
    return root->element;
}

/* Remove the smallest item from the priority queue.
   Throws Underflow if empty.*/
void LeftistHeap::deleteMin()
{
    LeftistNode *oldRoot = root;
    root = Merge(root->left, root->right);
    delete oldRoot;
}

/* Remove the smallest item from the priority queue.
   Pass back the smallest item, or throw Underflow if empty.*/
void LeftistHeap::deleteMin(int &minItem)
{
    if (isEmpty())
    {
        cout<<"Heap is Empty"<<endl;
        return;
    }
    minItem = findMin();
    deleteMin();
}

/* Test if the priority queue is logically empty.
   Returns true if empty, false otherwise*/

```

```

bool LeftistHeap::isEmpty()
{
    return root == NULL;
}

/* Test if the priority queue is logically full.
   Returns false in this implementation.*/
bool LeftistHeap::isFull()
{
    return false;
}

// Make the priority queue logically empty
void LeftistHeap::makeEmpty()
{
    reclaimMemory(root);
    root = NULL;
}

// Deep copy
LeftistHeap &LeftistHeap::operator =(LeftistHeap & rhs)
{
    if (this != &rhs)
    {
        makeEmpty();
        root = clone(rhs.root);
    }
    return *this;
}

// Internal method to make the tree empty.
void LeftistHeap::reclaimMemory(LeftistNode * t)
{
    if (t != NULL)
    {
        reclaimMemory(t->left);
        reclaimMemory(t->right);
        delete t;
    }
}

// Internal method to clone subtree.
LeftistNode *LeftistHeap::clone(LeftistNode * t)
{
    if (t == NULL)
        return NULL;
    else
        return new LeftistNode(t->element, clone(t->left),

```

```
        clone(t->right), t->dist);  
    }  
  
//Driver program  
int main()  
{  
    LeftistHeap h;  
    LeftistHeap h1;  
    LeftistHeap h2;  
    int x;  
    int arr[] = {1, 5, 7, 10, 15};  
    int arr1[] = {22, 75};  
  
    h.Insert(arr[0]);  
    h.Insert(arr[1]);  
    h.Insert(arr[2]);  
    h.Insert(arr[3]);  
    h.Insert(arr[4]);  
    h1.Insert(arr1[0]);  
    h1.Insert(arr1[1]);  
  
    h.deleteMin(x);  
    cout << x << endl;  
  
    h1.deleteMin(x);  
    cout << x << endl;  
  
    h.Merge(h1);  
    h2 = h;  
  
    h2.deleteMin(x);  
    cout << x << endl;  
  
    return 0;  
}
```

Output:

```
1  
22  
5
```

References:

[Wikipedia- Leftist Tree](#)
[CSC378: Leftist Trees](#)

Source

<https://www.geeksforgeeks.org/leftist-tree-leftist-heap/>

Chapter 100

Leftover element after performing alternate Bitwise OR and Bitwise XOR operations on adjacent pairs

Leftover element after performing alternate Bitwise OR and Bitwise XOR operations on adjacent pairs - GeeksforGeeks

Given an array of **N(always a power of 2)** elements and **Q** queries.

Every Query consists of two elements **index** and **value..** We need to write a program that assigns **value** to A_{index} and prints the single element which is left after performing below operations for each query:

- At alternate steps perform **bitwise OR** and **bitwise XOR** operations to the adjacent elements.
- In first iteration select $n/2$ pairs moving from left to right, and do a bitwise OR of all the pair values. In second iteration select $(n/2)/2$ leftover pairs and do a bitwise XOR on them. In the third iteration select, select $((n/2)/2)/2$ leftover pairs moving from left to right, and do a bitwise OR of all the pair values.
- Continue the above steps till we are left with a single element.

Examples:

```
Input : n = 4    m = 2
        arr = [1, 4, 5, 6]
        Queries-
```

```
1st: index=0 value=2
2nd: index=3 value=5
Output : 1
      3
```

Explanation:

1st query:

Assigning 2 to index 0, the sequence is now [2, 4, 5, 6].
1st iteration: There are $4/2=2$ pairs (2, 4) and (5, 6)
2 OR 4 gives 6, and 5 OR 6 gives us 7. So the sequence is now [6, 7].

2nd iteration: There is 1 pair left now (6, 7)
 $6 \wedge 7 = 1$.

Hence the last element left is 1 which is the answer to our first query.

2nd Query:

Assigning 5 to index 3, the sequence is now [2, 4, 5, 5].
1st iteration: There are $4/2=2$ pairs (2, 4) and (5, 5)
2 OR 4 gives 6, and 5 OR 5 gives us 5. So the sequence is now [6, 5].

2nd iteration: There is 1 pair left now (6, 5)
 $6 \wedge 5 = 3$.

Hence the last element left is 3 which is the answer to our second query.

Naive Approach: The naive approach is to perform every step till we are leftover with one element. Using 2-D vector we will store the resultant elements left after every step. $V[\text{steps-1}][0..size]$ gives the number of elements at previous step. If the step number is odd, we perform a bitwise OR operation, else a bitwise XOR operation is done. Repeat the steps till we are left over with one element. The last element left will be our answer.

Below is the implementation of the naive approach:

```
// CPP program to print the Leftover element after
// performing alternate Bitwise OR and Bitwise XOR
// operations to the pairs.
#include <bits/stdc++.h>
using namespace std;
```

```
#define N 1000

int lastElement(int a[],int n)
{
    // count the step number
    int steps = 1;
    vector<int>v[N];

    // if one element is there, it will be the answer
    if (n==1) return a[0];

    // at first step we do a bitwise OR
    for (int i = 0 ; i < n ; i += 2)
        v[steps].push_back(a[i] | a[i+1]);

    // keep on doing bitwise operations till the
    // last element is left
    while (v[steps].size()>1)
    {

        steps += 1;

        // perform operations
        for (int i = 0 ; i < v[steps-1].size(); i+=2)
        {
            // if step is the odd step
            if (steps&1)
                v[steps].push_back(v[steps-1][i] | v[steps-1][i+1]);
            else // even step
                v[steps].push_back(v[steps-1][i] ^ v[steps-1][i+1]);
        }
    }

    // answer when one element is left
    return v[steps][0];
}

// Driver Code
int main()
{
    int a[] = {1, 4, 5, 6};
    int n = sizeof(a)/sizeof(a[0]);

    // 1st query
    int index = 0;
    int value = 2;
```

```
a[0] = 2;
cout << lastElement(a,n) << endl;

// 2nd query
index = 3;
value = 5;
a[index] = value;
cout << lastElement(a,n) << endl;

return 0;
}
```

Output:

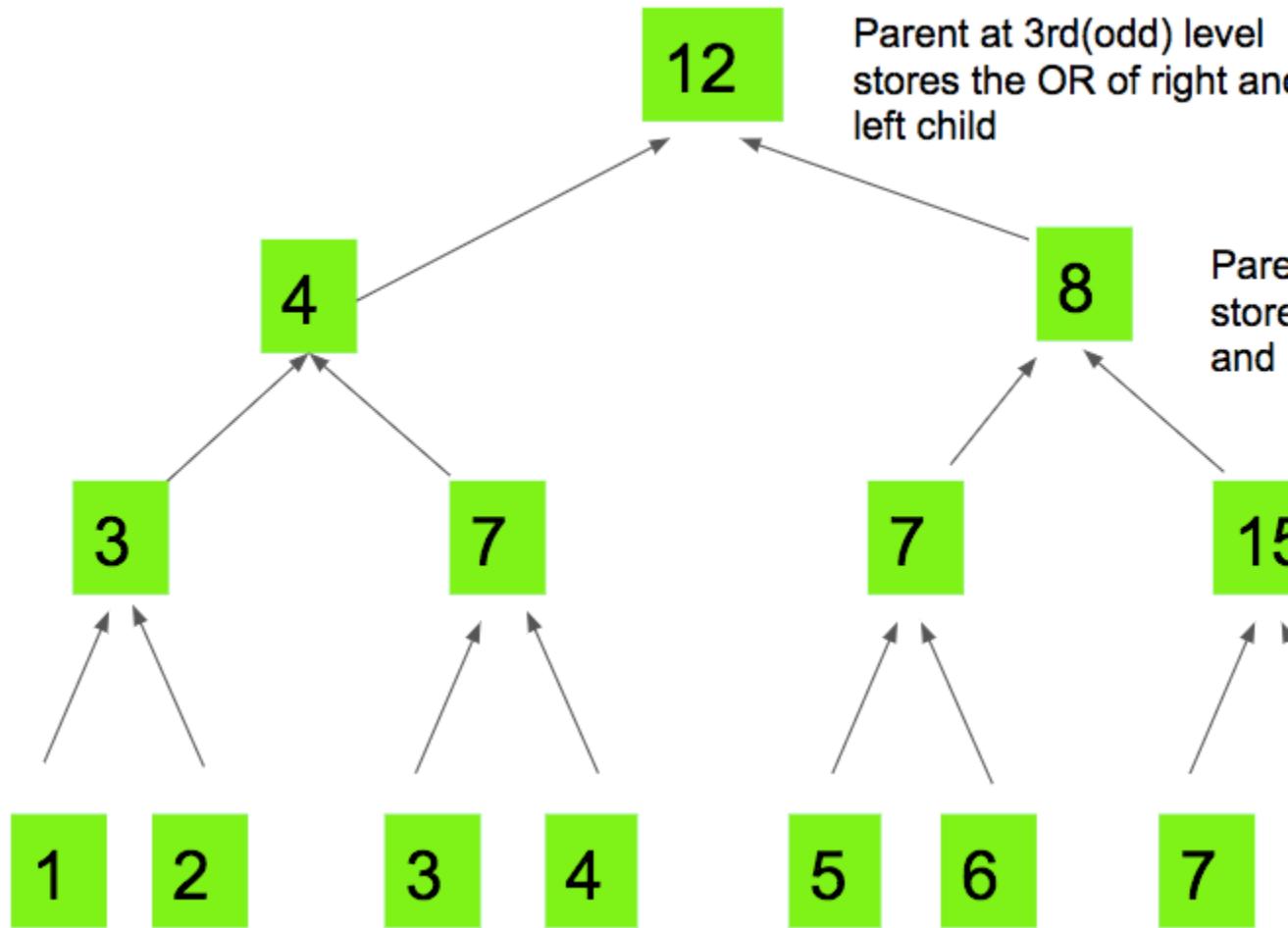
```
1
3
```

Time Complexity: $O(N * 2^N)$

Efficient Approach: The efficient approach is to use [Segment tree](#). Below is the complete segment tree approach used to solve the problem.

Building the tree

The leaves of the segment tree will store the array values and their parent will store the OR of the leaves. Moving up-ward in the tree, with every alternate step, the parent stores either of bitwise XOR or bitwise OR of left and right child. **At every odd-numbered iteration, we perform the bitwise OR of the pairs and similarly we perform bitwise XOR of pairs at every even-numbered operation.** So the odd-numbered parent will store the bitwise OR of the left and right child. Similarly, the even-numbered parent stores the bitwise XOR of the left and right child. `level[]` is an array that stores levels of every parent starting from 1, to determine if the pair(right child and left child) below it performs an OR operation or an XOR operation. **The root of the tree will be our answer to the given sequence after every update operation.**



The image above explains the construction of the tree if the sequence was $[1, 2, 3, 4, 5, 6, 7, 8]$, then after 3 iterations, we will be left over with 12 which is our answer and is stored at the root.

Answering Query

There is no need to rebuild the complete tree to perform an update operation. To do an update, we should find a **path from the root to the corresponding leaf** and recalculate the values only at the parents that are lying on the found path.

Level of parent:

Using [DP on trees](#), we can easily store the level of every parent. Initialize the leaf nodes level to 0, and keep adding as we move up to every parent.

The recurrence relation for calculating the level of parent is:

$$\text{level[parent]} = \text{level[child]} + 1$$

Here, child is $2 * \text{pos} + 1$ or $2 * \text{pos} + 2$

Below is the implementation of the above approach:

C++

```
// CPP program to print the Leftover element after
// performing alternate Bitwise OR and
// Bitwise XOR operations to the pairs.
#include <bits/stdc++.h>
using namespace std;
#define N 1000

// array to store the tree
int tree[N];

// array to store the level of every parent
int level[N];

// function to construct the tree
void constructTree(int low, int high, int pos, int a[])
{
    if (low == high)
    {
        // level of child is always 0
        level[pos] = 0;
        tree[pos] = a[high];
        return;
    }
    int mid = (low + high) / 2;

    // recursive call
    constructTree(low, mid, 2 * pos + 1, a);
    constructTree(mid + 1, high, 2 * pos + 2, a);

    // increase the level of every parent, which is
    // level of child + 1
    level[pos] = level[2 * pos + 1] + 1;

    // if the parent is at odd level, then do a
    // bitwise OR
    if (level[pos] & 1)
        tree[pos] = tree[2 * pos + 1] | tree[2 * pos + 2];

    // if the parent is at even level, then
    // do a bitwise XOR
    else
        tree[pos] = tree[2 * pos + 1] ^ tree[2 * pos + 2];
}

// function that updates the tree
```

```
void update(int low, int high, int pos, int index, int a[])
{
    // if it is a leaf and the leaf which is
    // to be updated
    if (low == high and low == index)
    {
        tree[pos] = a[low];
        return;
    }

    // out of range
    if (index < low || index > high)
        return;

    // not a leaf then recurse
    if (low != high)
    {
        int mid = (low + high) / 2;

        // recursive call
        update(low, mid, 2 * pos + 1, index, a);
        update(mid + 1, high, 2 * pos + 2, index, a);

        // check if the parent is at odd or even level
        // and perform OR or XOR according to that
        if (level[pos] & 1)
            tree[pos] = tree[2 * pos + 1] | tree[2 * pos + 2];
        else
            tree[pos] = tree[2 * pos + 1] ^ tree[2 * pos + 2];
    }
}

// function that assigns value to a[index]
// and calls update function to update the tree
void updateValue(int index, int value, int a[], int n)
{
    a[index] = value;
    update(0, n - 1, 0, index, a);
}

// Driver Code
int main()
{
    int a[] = { 1, 4, 5, 6 };
    int n = sizeof(a) / sizeof(a[0]);

    // builds the tree
    constructTree(0, n - 1, 0, a);
```

```
// 1st query
int index = 0;
int value = 2;
updateValue(index, value, a, n);
cout << tree[0] << endl;

// 2nd query
index = 3;
value = 5;
updateValue(index, value, a, n);
cout << tree[0] << endl;

return 0;
}
```

Java

```
// java program to print the Leftover
// element after performing alternate
// Bitwise OR and Bitwise XOR operations
// to the pairs.
import java.io.*;

public class GFG {

    static int N = 1000;

    // array to store the tree
    static int []tree = new int[N];

    // array to store the level of
    // every parent
    static int []level = new int[N];

    // function to construct the tree
    static void constructTree(int low, int high,
                             int pos, int []a)
    {
        if (low == high)
        {

            // level of child is
            // always 0
            level[pos] = 0;
            tree[pos] = a[high];
            return;
        }
    }
}
```

```
int mid = (low + high) / 2;

// recursive call
constructTree(low, mid, 2 * pos + 1, a);

constructTree(mid + 1, high,
              2 * pos + 2, a);

// increase the level of every parent,
// which is level of child + 1
level[pos] = level[2 * pos + 1] + 1;

// if the parent is at odd level, then
// do a bitwise OR
if ((level[pos] & 1) > 0)
    tree[pos] = tree[2 * pos + 1] |
                tree[2 * pos + 2];

// if the parent is at even level, then
// do a bitwise XOR
else
    tree[pos] = tree[2 * pos + 1] ^
                tree[2 * pos + 2];
}

// function that updates the tree
static void update(int low, int high, int pos,
                   int index, int []a)
{

    // if it is a leaf and the leaf which is
    // to be updated
    if (low == high && low == index)
    {
        tree[pos] = a[low];
        return;
    }

    // out of range
    if (index < low || index > high)
        return;

    // not a leaf then recurse
    if (low != high)
    {
        int mid = (low + high) / 2;

        // recursive call
    }
}
```

```
update(low, mid, 2 * pos + 1, index, a);

update(mid + 1, high, 2 * pos + 2,
       index, a);

// check if the parent is at odd or
// even level and perform OR or XOR
// according to that
if ((level[pos] & 1) > 0)
    tree[pos] = tree[2 * pos + 1] |
                 tree[2 * pos + 2];
else
    tree[pos] = tree[2 * pos + 1] ^
                 tree[2 * pos + 2];
}
}

// function that assigns value to a[index]
// and calls update function to update the
// tree
static void updateValue(int index, int value,
                        int []a, int n)
{
    a[index] = value;
    update(0, n - 1, 0, index, a);
}

// Driver Code
static public void main (String[] args)
{
    int []a = { 1, 4, 5, 6 };
    int n = a.length;

    // builds the tree
    constructTree(0, n - 1, 0, a);

    // 1st query
    int index = 0;
    int value = 2;
    updateValue(index, value, a, n);
    System.out.println(tree[0]);

    // 2nd query
    index = 3;
    value = 5;
    updateValue(index, value, a, n);
    System.out.println(tree[0]);
}
```

```
}
```

```
// This code is contributed by vt_m.
```

```
C#
```

```
// C# program to print the Leftover
// element after performing alternate
// Bitwise OR and Bitwise XOR
// operations to the pairs.
using System;
```

```
public class GFG {
```

```
    static int N = 1000;
```

```
    // array to store the tree
    static int []tree = new int[N];
```

```
    // array to store the level of
    // every parent
    static int []level = new int[N];
```

```
    // function to construct the
    // tree
    static void constructTree(int low, int high,
                            int pos, int []a)
    {
        if (low == high)
        {
            // level of child is always 0
            level[pos] = 0;
            tree[pos] = a[high];
            return;
        }
        int mid = (low + high) / 2;

        // recursive call
        constructTree(low, mid, 2 * pos + 1, a);

        constructTree(mid + 1, high,
                    2 * pos + 2, a);

        // increase the level of every parent,
        // which is level of child + 1
        level[pos] = level[2 * pos + 1] + 1;
    }
}
```

```
// if the parent is at odd level,
// then do a bitwise OR
if ((level[pos] & 1) > 0)
    tree[pos] = tree[2 * pos + 1] |
                tree[2 * pos + 2];

// if the parent is at even level,
// then do a bitwise XOR
else
    tree[pos] = tree[2 * pos + 1] ^
                tree[2 * pos + 2];
}

// function that updates the tree
static void update(int low, int high,
                   int pos, int index, int []a)
{

    // if it is a leaf and the leaf
    // which is to be updated
    if (low == high && low == index)
    {
        tree[pos] = a[low];
        return;
    }

    // out of range
    if (index < low || index > high)
        return;

    // not a leaf then recurse
    if (low != high)
    {
        int mid = (low + high) / 2;

        // recursive call
        update(low, mid, 2 * pos + 1,
               index, a);

        update(mid + 1, high, 2 * pos + 2,
               index, a);

        // check if the parent is at odd
        // or even level and perform OR
        // or XOR according to that
        if ((level[pos] & 1) > 0)
            tree[pos] = tree[2 * pos + 1] |
                        tree[2 * pos + 2];
    }
}
```

```
        else
            tree[pos] = tree[2 * pos + 1]
                        ^ tree[2 * pos + 2];
    }
}

// function that assigns value to a[index]
// and calls update function to update
// the tree
static void updateValue(int index, int value,
                       int []a, int n)
{
    a[index] = value;
    update(0, n - 1, 0, index, a);
}

// Driver Code
static public void Main ()
{
    int []a = { 1, 4, 5, 6 };
    int n = a.Length;;

    // builds the tree
    constructTree(0, n - 1, 0, a);

    // 1st query
    int index = 0;
    int value = 2;
    updateValue(index, value, a, n);
    Console.WriteLine(tree[0]);

    // 2nd query
    index = 3;
    value = 5;
    updateValue(index, value, a, n);
    Console.WriteLine(tree[0]);
}
}

// This code is contributed by vt_m.
```

Output:

```
1
3
```

Time Complexity:

- Tree construction: $O(N)$
- Answering Query: $O(\log_2 N)$

Improved By : [vt_m](#)

Source

<https://www.geeksforgeeks.org/leftover-element-performing-alternate-bitwise-bitwise-xor-operations-adjacent-pairs>

Chapter 101

Level Ancestor Problem

Level Ancestor Problem - GeeksforGeeks

The [level ancestor problem](#) is the problem of preprocessing a given rooted tree T into a data structure that can determine the ancestor of a given node at a given depth from the root of the tree. Here **depth** of any node in a tree is the number of edges on the shortest path from the root of the tree to the node.

Given tree is represented as **un-directed connected graph** having n nodes and **$n-1$ edges**.

The idea to solve the above query is to use **Jump Pointer Algorithm** and pre-processes the tree in $O(n \log n)$ time and answer level ancestor queries in $O(\log n)$ time. In jump pointer, there is a pointer from node N to N 's j -th ancestor, for $j = 1, 2, 4, 8, \dots$, and so on. We refer to these pointers as $\text{Jump}_N[i]$, where $\text{Jump}_u[i] = \text{LA}(N, \text{depth}(N) - 2^i)$.

When the algorithm is asked to process a query, we repeatedly jump up the tree using these jump pointers. The number of jumps will be at most $\log n$ and therefore queries can be answered in $O(\log n)$ time.

So we store **2^i th ancestor** of each node and also find the depth of each node from the root of the tree.

Now our task reduces to find the $(\text{depth}(N) - 2^i)$ th ancestor of node N . Let's denote X as $(\text{depth}(N) - 2^i)$ and let b bits of the X are **set bits** (1) denoted by $s_1, s_2, s_3, \dots, s_b$.
$$X = 2^{(s_1)} + 2^{(s_2)} + \dots + 2^{(s_b)}$$

Now the problem is how to find 2^j ancestors of each node and depth of each node from the root of the tree?

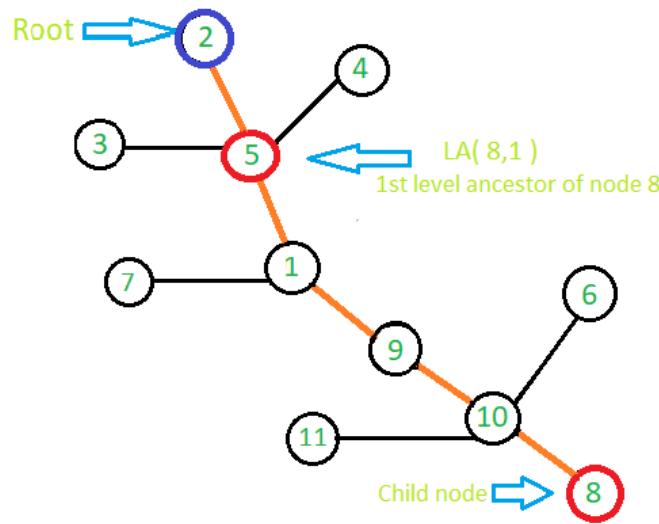
Initially, we know the 2^0 th ancestor of each node is its parent. We can recursively compute 2^j -th ancestor. We know 2^j -th ancestor is 2^{j-1} -th ancestor of 2^{j-1} -th ancestor. To calculate the depth of each node we use the ancestor matrix. If we found the root node present in the array of the k th element at j th index then the depth of that node is simply 2^j but if root node doesn't present in the array of ancestors of the k th element than the depth of k th element is $2^{(\text{index of last non zero ancestor at } k\text{th row})} + \text{depth of ancestor present at last index of } k\text{th row}$.

Below is the algorithm to fill the ancestor matrix and depth of each node using dynamic programming. Here, we denote root node as **R** and initially assume the ancestor of root node as **0**. We also initialize depth array with **-1** means the depth of the current node is not set and we need to find its depth. If the depth of the current node is not equal to **-1** means we have already computed its depth.

```
we know the first ancestor of each node so we take j>=1,
For j>=1
```

```
ancstr[k][j] = 2jth ancestor of k
              = 2j-1th ancestor of (2j-1th ancestor of k)
              = ancstr[ancstr[i][j-1][j-1]
                if ancstr[k][j] == R && depth[k] == -1
                  depth[k] = 2j
                else if ancstr[k][j] == -1 && depth[k] == -1
                  depth[k] = 2(j-1) + depth[ ancstr[k][j-1] ]
```

Let's understand this algorithm with below diagram.



In the given figure we need to compute 1st level ancestor of the node with value **8**. First, we make ancestor matrix which stores 2^{th} ancestor of nodes. Now, 2^0 ancestor of node **8** is **10** and similarly 2^0 ancestor of node **10** is **9** and for node **9** it is **1** and for node **1** it is **5**. Based on the above algorithm 1st level ancestor of node 8 is ($\text{depth}(8)-1$) th ancestor of node 8. We have pre computed depth of each node and depth of 8 is 5 so we finally need to

find (5-1) = 4th ancestor of node 8 which is equal to 2¹th ancestor of [2¹ ancestor of node 8] and 2¹th ancestor of node 8 is 2⁰th ancestor of [2⁰th ancestor of node 8]. So, 2⁰th ancestor of [2⁰th ancestor of node 8] is node with value 9 and 2¹th ancestor of node 9 is node with value 5. Thus in this way we can compute all query in O(logn) time complexity.

```

// CPP program to implement Level Ancestor Algorithm
#include <bits/stdc++.h>
using namespace std;
int R = 0;

// n -> it represent total number of nodes
// len -> it is the maximum length of array to hold
//          ancestor of each node. In worst case,
// the highest value of ancestor a node can have is n-1.
// 2 ^ len <= n-1
// len = O(log2n)
int getLen(int n)
{
    return (int)(log(n) / log(2)) + 1;
}

// ancstr represents 2D matrix to hold ancestor of node.
// Here we pass reference of 2D matrix so that the change
// made occur directly to the original matrix
// depth[] stores depth of each node
// len is same as defined above
// n is total nodes in graph
// R represent root node
void setancestor(vector<vector<int> >& ancstr,
                 vector<int>& depth, int* node, int len, int n)
{
    // depth of root node is set to 0
    depth[R] = 0;

    // if depth of a node is -1 it means its depth
    // is not set otherwise we have computed its depth
    for (int j = 1; j <= len; j++) {
        for (int i = 0; i < n; i++) {
            ancstr[node[i]][j] = ancstr[ancstr[node[i]][j - 1]][j - 1];

            // if ancestor of current node is R its height is
            // previously not set, then its height is 2^j
            if (ancstr[node[i]][j] == R && depth[node[i]] == -1) {

                // set the depth of ith node
                depth[node[i]] = pow(2, j);
            }
        }
    }
}

```

```

// if ancestor of current node is 0 means it
// does not have root node at its 2th power
// on its path so its depth is 2^(index of
// last non zero ancestor means j-1) + depth
// of 2^(j-1) th ancestor
else if (ancstr[node[i]][j] == 0 &&
         node[i] != R && depth[node[i]] == -1) {
    depth[node[i]] = pow(2, j - 1) +
                      depth[ancstr[node[i]][j - 1]];
}
}

}

}

// c -> it represent child
// p -> it represent ancestor
// i -> it represent node number
// p=0 means the node is root node
// R represent root node
// here also we pass reference of 2D matrix and depth
// vector so that the change made occur directly to
// the original matrix and original vector
void constructGraph(vector<vector<int> &> ancstr,
                     int* node, vector<int>& depth, int* isNode,
                     int c, int p, int i)
{
    // enter the node in node array
    // it stores all the nodes in the graph
    node[i] = c;

    // to confirm that no child node have 2 ancestors
    if (isNode == 0) {
        isNode = 1;

        // make ancestor of x as y
        ancstr[0] = p;

        // if its first ancestor is root than its depth is 1
        if (R == p) {
            depth = 1;
        }
    }
    return;
}

// this function will delete leaf node
// x is node to be deleted

```

```

void removeNode(vector<vector<int> >& ancstr,
                int* isNode, int len, int x)
{
    if (isNode[x] == 0)
        cout << "node does not present in graph " << endl;
    else {
        isNode[x] = 0;

        // make all ancestor of node x as 0
        for (int j = 0; j <= len; j++) {
            ancstr[x][j] = 0;
        }
    }
    return;
}

// x -> it represent new node to be inserted
// p -> it represent ancestor of new node
void addNode(vector<vector<int> >& ancstr,
             vector<int>& depth, int* isNode, int len,
             int x, int p)
{
    if (isNode[x] == 1) {
        cout << " Node is already present in array " << endl;
        return;
    }
    if (isNode[p] == 0) {
        cout << " ancestor not does not present in an array " << endl;
        return;
    }

    isNode[x] = 1;
    ancstr[x][0] = p;

    // depth of new node is 1 + depth of its ancestor
    depth[x] = depth[p] + 1;
    int j = 0;

    // while we don't reach root node
    while (ancstr[x][j] != 0) {
        ancstr[x][j + 1] = ancstr[ancstr[x][j]][j];
        j++;
    }

    // remaining array will fill with 0 after
    // we find root of tree
    while (j <= len) {
        ancstr[x][j] = 0;
    }
}

```

```

        j++;
    }
    return;
}

// LA function to find Lth level ancestor of node x
void LA(vector<vector<int> >& ancstr, vector<int> depth,
         int* isNode, int x, int L)
{
    int j = 0;
    int temp = x;

    // to check if node is present in graph or not
    if (isNode[x] == 0) {
        cout << "Node is not present in graph " << endl;
        return;
    }

    // we change L as depth of node x -
    int k = depth[x] - L;
    // int q = k;
    // in this loop we decrease the value of k by k/2 and
    // increment j by 1 after each iteration, and check for set bit
    // if we get set bit then we update x with jth ancestor of x
    // as k becomes less than or equal to zero means we
    // reach to kth level ancestor
    while (k > 0) {

        // to check if last bit is 1 or not
        if (k & 1) {
            x = ancstr[x][j];
        }

        // use of shift operator to make k = k/2
        // after every iteration
        k = k >> 1;
        j++;
    }
    cout << L << "th level ancestor of node "
        << temp << " is = " << x << endl;
}

int main()
{
    // n represent number of nodes
    int n = 12;
}

```

```
// initialization of ancestor matrix
// suppose max range of node is up to 1000
// if there are 1000 nodes than also length
// of ancestor matrix will not exceed 10
vector<vector<int>> ancestor(1000, vector<int>(10));

// this vector is used to store depth of each node.
vector<int> depth(1000);

// fill function is used to initialize depth with -1
fill(depth.begin(), depth.end(), -1);

// node array is used to store all nodes
int* node = new int[1000];

// isNode is an array to check whether a
// node is present in graph or not
int* isNode = new int[1000];

// memset function to initialize isNode array with 0
memset(isNode, 0, 1000 * sizeof(int));

// function to calculate len
// len -> it is the maximum length of array to
// hold ancestor of each node.
int len = getLen(n);

// R stores root node
R = 2;

// construction of graph
// here 0 represent that the node is root node
constructGraph(ancestor, node, depth, isNode, 2, 0, 0);
constructGraph(ancestor, node, depth, isNode, 5, 2, 1);
constructGraph(ancestor, node, depth, isNode, 3, 5, 2);
constructGraph(ancestor, node, depth, isNode, 4, 5, 3);
constructGraph(ancestor, node, depth, isNode, 1, 5, 4);
constructGraph(ancestor, node, depth, isNode, 7, 1, 5);
constructGraph(ancestor, node, depth, isNode, 9, 1, 6);
constructGraph(ancestor, node, depth, isNode, 10, 9, 7);
constructGraph(ancestor, node, depth, isNode, 11, 10, 8);
constructGraph(ancestor, node, depth, isNode, 6, 10, 9);
constructGraph(ancestor, node, depth, isNode, 8, 10, 10);

// function to pre compute ancestor matrix
setancestor(ancestor, depth, node, len, n);
```

```
// query to get 1st level ancestor of node 8
LA(ancestor, depth, isNode, 8, 1);

// add node 12 and its ancestor is 8
addNode(ancestor, depth, isNode, len, 12, 8);

// query to get 2nd level ancestor of node 12
LA(ancestor, depth, isNode, 12, 2);

// delete node 12
removeNode(ancestor, isNode, len, 12);

// query to get 5th level ancestor of node
// 12 after deletion of node
LA(ancestor, depth, isNode, 12, 1);

return 0;
}
```

Output:

```
1th level ancestor of node 8 is = 5
2th level ancestor of node 12 is = 1
Node is not present in graph
```

Source

<https://www.geeksforgeeks.org/level-ancestor-problem/>

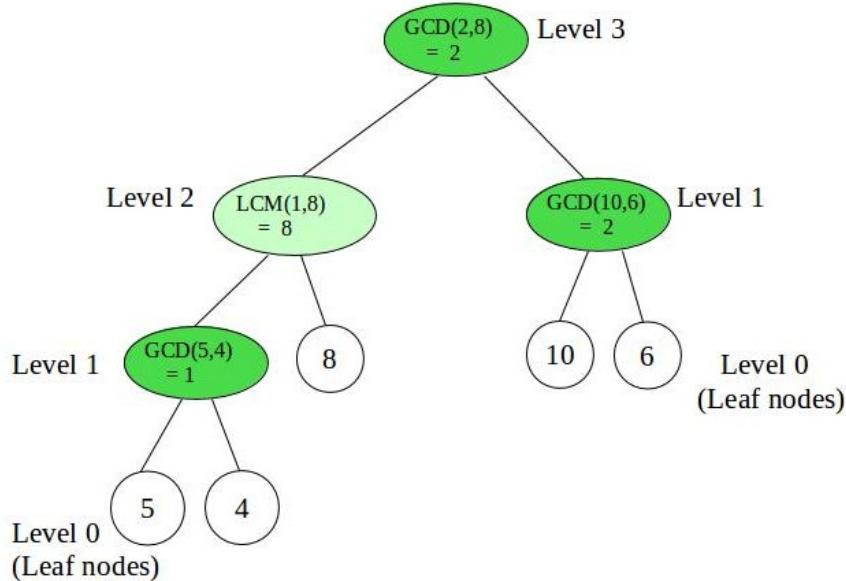
Chapter 102

Levelwise Alternating GCD and LCM of nodes in Segment Tree

Levelwise Alternating GCD and LCM of nodes in Segment Tree - GeeksforGeeks

A Levelwise GCD/LCM alternating segment tree is a segment tree, such that at every level the operations GCD and LCM alternate. In other words at Level 1 the left and right subtrees combine together by the GCD operation i.e Parent node = Left Child **GCD** Right Child and on Level 2 the left and right subtrees combine together by the LCM operation i.e Parent node = Left Child **LCM** Right Child

Such a type of Segment tree has the following type of structure:



The operations (GCD) and (LCM) indicate which operation was carried out to merge the child nodes

Given N leaf nodes, the task is to build such a segment tree and print the root node.
Examples:

Input : arr[] = { 5, 4, 8, 10, 6 }
Output : Value at Root Node = 2
Explanation : The image given above shows the segment tree corresponding to the given set leaf nodes.

Prerequisites: [Segment Trees](#)

In this Segment Tree, we carry two operations:- **GCD and LCM**.

Now, along with the information which is passed recursively for the sub-trees, information regarding the operation to be carried out at that level is also passed since these operations alternate levelwise. It is important to note that a parent node when calls its left and right children the same operation information is passed to both the children as they are on the same level.

Let's represent the two operations i.e GCD and LCM by 0 and 1 respectively. Then, if at Level i GCD operation is performed then at Level (i + 1) LCM operation will be performed. Thus if Level i has 0 as operation then level (i + 1) will have 1 as operation.

Operation at Level (i + 1) = ! (Operation at Level i)
where,
Operation at Level i {0, 1}

Careful analysis of the image suggests that if the height of the tree is even then the root node is a result of LCM operation of its left and right children else a result of GCD operation.

```
#include <bits/stdc++.h>
using namespace std;

// Recursive function to return gcd of a and b
int gcd(int a, int b)
{
    // Everything divides 0
    if (a == 0 || b == 0)
        return 0;

    // base case
    if (a == b)
        return a;
```

```

// a is greater
if (a > b)
    return gcd(a-b, b);
return gcd(a, b-a);
}

// A utility function to get the middle index from
// corner indexes.
int getMid(int s, int e) { return s + (e - s) / 2; }

void STconstructUtil(int arr[], int ss, int se, int* st,
                     int si, int op)
{
    // If there is one element in array, store it in
    // current node of segment tree and return
    if (ss == se) {
        st[si] = arr[ss];
        return;
    }

    // If there are more than one elements, then recur
    // for left and right subtrees and store the sum of
    // values in this node
    int mid = getMid(ss, se);

    // Build the left and the right subtrees by using
    // the fact that operation at level (i + 1) = !
    // (operation at level i)
    STconstructUtil(arr, ss, mid, st, si * 2 + 1, !op);
    STconstructUtil(arr, mid + 1, se, st, si * 2 + 2, !op);

    // merge the left and right subtrees by checking
    // the operation to be carried. If operation = 1,
    // then do GCD else LCM
    if (op == 1) {
        // GCD operation
        st[si] = __gcd(st[2 * si + 1], st[2 * si + 2]);
    }
    else {
        // LCM operation
        st[si] = (st[2 * si + 1] * st[2 * si + 2]) /
                  (gcd(st[2 * si + 1], st[2 * si + 2]));
    }
}

/* Function to construct segment tree from given array.
This function allocates memory for segment tree and
calls STconstructUtil() to fill the allocated memory */

```

```

int* STconstruct(int arr[], int n)
{
    // Allocate memory for segment tree

    // Height of segment tree
    int x = (int)(ceil(log2(n)));

    // maximum size of segment tree
    int max_size = 2 * (int)pow(2, x) - 1;

    // allocate memory
    int* st = new int[max_size];

    // operation = 1(GCD) if Height of tree is
    // even else it is 0(LCM) for the root node
    int opAtRoot = (x % 2 == 0 ? 0 : 1);

    // Fill the allocated memory st
    STconstructUtil(arr, 0, n - 1, st, 0, opAtRoot);

    // Return the constructed segment tree
    return st;
}

int main()
{
    int arr[] = { 5, 4, 8, 10, 6 };
    int n = sizeof(arr) / sizeof(arr[0]);

    // Build segment tree
    int* st = STconstruct(arr, n);

    // 0-based indexing in segment tree
    int rootIndex = 0;

    cout << "Value at Root Node = " << st[rootIndex];

    return 0;
}

```

Output:

```
Value at Root Node = 2
```

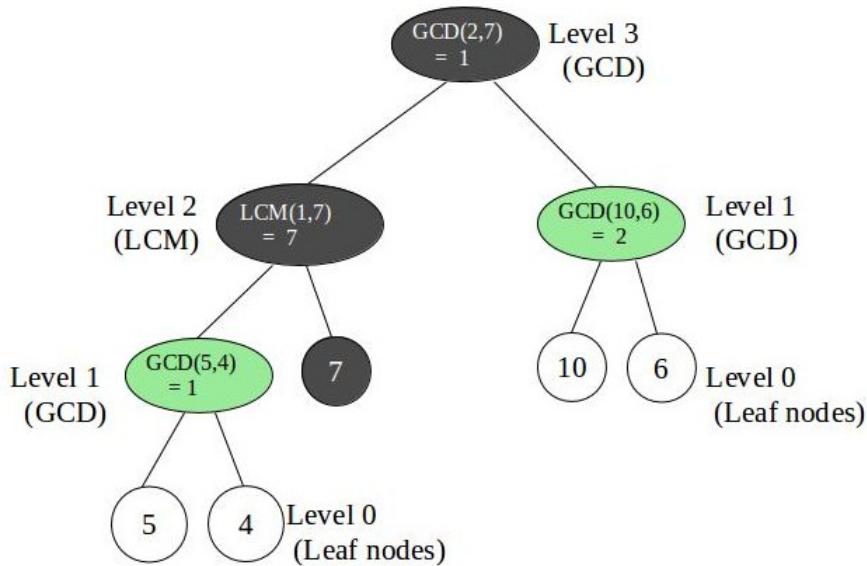
Time complexity for tree construction is $O(n)$, as there are total 2^*n-1 nodes and value at every node is calculated at once.

Now to perform point updates i.e. update the value with given index and value, can be done by traversing down the tree to the leaf node and performing the update.

While coming back to the root node we build the tree again similar to the build() function by passing the operation to be performed at every level and storing the result of applying that operation on values of its left and right children and storing the result into that node.

Consider the following Segment tree after performing the update,
 $\text{arr}[2] = 7$

Now the updated segment tree looks like this:



Here nodes in black denote the fact that they are updated.

```

#include <bits/stdc++.h>
using namespace std;

// Recursive function to return gcd of a and b
int gcd(int a, int b)
{
    // Everything divides 0
    if (a == 0 || b == 0)
        return 0;

    // base case
    if (a == b)
        return a;

    // a is greater
    if (a > b)
        return gcd(a-b, b);
    return gcd(a, b-a);
}

```

```

}

// A utility function to get the middle index from
// corner indexes.
int getMid(int s, int e) { return s + (e - s) / 2; }

void STconstructUtil(int arr[], int ss, int se, int* st,
                     int si, int op)
{
    // If there is one element in array, store it in
    // current node of segment tree and return
    if (ss == se) {
        st[si] = arr[ss];
        return;
    }

    // If there are more than one elements, then recur
    // for left and right subtrees and store the sum of
    // values in this node
    int mid = getMid(ss, se);

    // Build the left and the right subtrees by using
    // the fact that operation at level (i + 1) = !
    // (operation at level i)
    STconstructUtil(arr, ss, mid, st, si * 2 + 1, !op);
    STconstructUtil(arr, mid + 1, se, st, si * 2 + 2, !op);

    // merge the left and right subtrees by checking
    // the operation to be carried. If operation = 1,
    // then do GCD else LCM
    if (op == 1) {
        // GCD operation
        st[si] = gcd(st[2 * si + 1], st[2 * si + 2]);
    }
    else {
        // LCM operation
        st[si] = (st[2 * si + 1] * st[2 * si + 2]) /
                  (gcd(st[2 * si + 1], st[2 * si + 2]));
    }
}

void updateUtil(int* st, int ss, int se, int ind, int val,
                int si, int op)
{
    // Base Case: If the input index lies outside
    // this segment
    if (ind < ss || ind > se)
        return;
}

```

```

// If the input index is in range of this node,
// then update the value of the node and its
// children

// leaf node
if (ss == se && ss == ind) {
    st[si] = val;
    return;
}

int mid = getMid(ss, se);

// Update the left and the right subtrees by
// using the fact that operation at level
// (i + 1) = ! (operation at level i)
updateUtil(st, ss, mid, ind, val, 2 * si + 1, !op);
updateUtil(st, mid + 1, se, ind, val, 2 * si + 2, !op);

// merge the left and right subtrees by checking
// the operation to be carried. If operation = 1,
// then do GCD else LCM
if (op == 1) {

    // GCD operation
    st[si] = gcd(st[2 * si + 1], st[2 * si + 2]);
}
else {

    // LCM operation
    st[si] = (st[2 * si + 1] * st[2 * si + 2]) /
        (gcd(st[2 * si + 1], st[2 * si + 2]));
}

void update(int arr[], int* st, int n, int ind, int val)
{
    // Check for erroneous input index
    if (ind < 0 || ind > n - 1) {
        printf("Invalid Input");
        return;
    }

    // Height of segment tree
    int x = (int)(ceil(log2(n)));

    // operation = 1(GCD) if Height of tree is
    // even else it is 0(LCM) for the root node
}

```

```

int opAtRoot = (x % 2 == 0 ? 0 : 1);

arr[ind] = val;

// Update the values of nodes in segment tree
updateUtil(st, 0, n - 1, ind, val, 0, opAtRoot);
}

/* Function to construct segment tree from given array.
This function allocates memory for segment tree and
calls STconstructUtil() to fill the allocated memory */
int* STconstruct(int arr[], int n)
{
    // Allocate memory for segment tree

    // Height of segment tree
    int x = (int)(ceil(log2(n)));

    // maximum size of segment tree
    int max_size = 2 * (int)pow(2, x) - 1;

    // allocate memory
    int* st = new int[max_size];

    // operation = 1(GCD) if Height of tree is
    // even else it is 0(LCM) for the root node
    int opAtRoot = (x % 2 == 0 ? 0 : 1);

    // Fill the allocated memory st
    STconstructUtil(arr, 0, n - 1, st, 0, opAtRoot);

    // Return the constructed segment tree
    return st;
}

int main()
{
    int arr[] = { 5, 4, 8, 10, 6 };
    int n = sizeof(arr) / sizeof(arr[0]);

    // Build segment tree
    int* st = STconstruct(arr, n);

    // 0-based indexing in segment tree
    int rootIndex = 0;

    cout << "Old Value at Root Node = " <<
        st[rootIndex] << endl;
}

```

```
// perform update arr[2] = 7
update(arr, st, n, 2, 7);

cout << "New Value at Root Node = " <<
        st[rootIndex] << endl;

return 0;
}
```

Output:

```
Old Value at Root Node = 2
New Value at Root Node = 1
```

The time complexity of update is also $O(\log n)$. To update a leaf value, one node is processed at every level and number of levels is $O(\log n)$.

Source

<https://www.geeksforgeeks.org/levelwise-alternating-gcd-lcm-nodes-segment-tree/>

Chapter 103

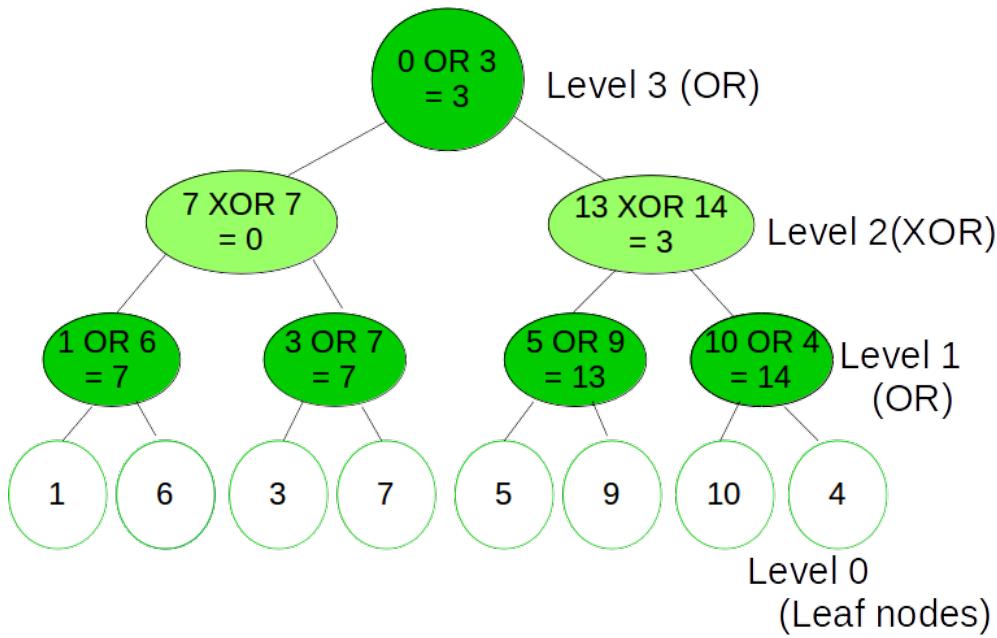
Levelwise Alternating OR and XOR operations in Segment Tree

Levelwise Alternating OR and XOR operations in Segment Tree - GeeksforGeeks

A Levelwise OR/XOR alternating segment tree is a segment tree, such that at every level the operations OR and XOR alternate. In other words at Level 1 the left and right sub-trees combine together by the OR operation i.e Parent node = Left Child **OR** Right Child and on Level 2 the left

and right sub-trees combine together by the XOR operation i.e Parent node = Left Child **XOR** Right Child

Such a type of Segment tree has the following type of structure:



The operations (OR) and (XOR) indicate which operation was carried out to merge the child node

Given 2^N leaf nodes, the task is to build such a segment tree and print the root node

Examples:

```

Input : Leaves = {1, 6, 3, 7, 5, 9, 10, 4}
Output : Value at Root Node = 3
Explanation : The image given above shows the
segment tree corresponding to the given
set leaf nodes.
  
```

This is an Extension to the [Classical Segment Tree](#) where we represent each node as an integer and the parent node is built by first building the left and the right subtrees and then combining the results of the left and the right children into the parent node. This merging of the left and right children into the parent node is done by a consistent operation. For example, MIN() / MAX() in Range Minimum Queries, Sum, XOR, OR, AND, etc. By consistent operations, we mean that this operation is performed to merge any node's left and right child into the parent node by carrying the operation say OP on their results, where OP is a consistent operation.

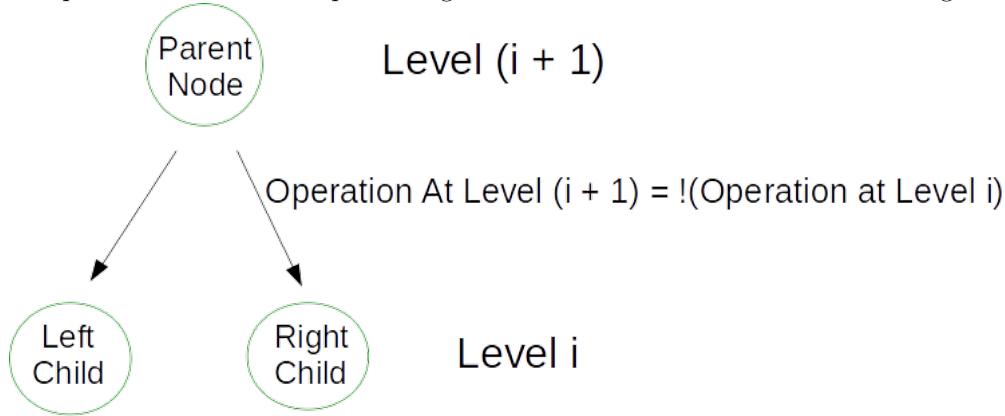
In this Segment tree, we carry two operations that are :- **OR and XOR**.

Now we build the Segment tree in a similar fashion as we do in the classical version, but now when we recursively pass the information for the sub-trees we will also pass information regarding the operation to be carried out at that level since these operations alternate levelwise. It's important to note that a parent node when calls its left and right children the same operation information is passed to both the children as they are on the same level.

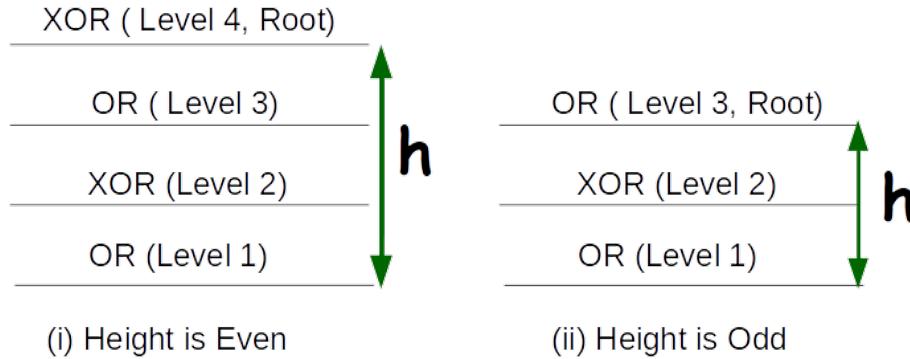
Lets represent the two operations i.e OR and XOR by 0 and 1 respectively. Then if at Level i we have an OR operation the at Level $(i + 1)$ we will have an XOR operation. There if Level i has 0 as operation then level $(i + 1)$ will have 1 as operation

```
Operation at Level (i + 1) = ! (Operation at Level i)
where,
Operation at Level i {0, 1}
```

The parent child relationship for a segment tree node is shown in the below image:



Now, we need to look at the operation to be carried out at root node.



Note: 'h' denotes height of the segment tree

If we carefully look at the image then it would be easy to figure out that if the height of the tree is even then the root node is a result of XOR operation of its left and right children else a result of OR operation.

```
// C++ program to build levelwise OR/XOR alternating
// Segment tree
#include <bits/stdc++.h>
```

```

using namespace std;

// A utility function to get the middle index from
// corner indexes.
int getMid(int s, int e) { return s + (e - s) / 2; }

// A recursive function that constructs Segment Tree
// for array[ss..se].
// si is index of current node in segment tree st
// operation denotes which operation is carried out
// at that level to merge the left and right child.
// It's either 0 or 1.
void constructSTUtil(int arr[], int ss, int se, int* st,
                     int si, int operation)
{
    // If there is one element in array, store it
    // in current node of segment tree and return
    if (ss == se) {
        st[si] = arr[ss];
        return;
    }

    // If there are more than one elements, then
    // recur for left and right subtrees and store
    // the sum of values in this node
    int mid = getMid(ss, se);

    // Build the left and the right subtrees by
    // using the fact that operation at level
    // (i + 1) = ! (operation at level i)
    constructSTUtil(arr, ss, mid, st, si * 2 + 1, !operation);
    constructSTUtil(arr, mid + 1, se, st, si * 2 + 2, !operation);

    // merge the left and right subtrees by checking
    // the operation to be carried. If operation = 1,
    // then do OR else XOR
    if (operation == 1) {

        // OR operation
        st[si] = (st[2 * si + 1] | st[2 * si + 2]);
    }
    else {

        // XOR operation
        st[si] = (st[2 * si + 1] ^ st[2 * si + 2]);
    }
}

```

```
/* Function to construct segment tree from given array.
   This function allocates memory for segment tree and
   calls constructSTUtil() to fill the allocated memory */
int* constructST(int arr[], int n)
{
    // Allocate memory for segment tree

    // Height of segment tree
    int x = (int)(ceil(log2(n)));

    // Maximum size of segment tree
    int max_size = 2 * (int)pow(2, x) - 1;

    // Allocate memory
    int* st = new int[max_size];

    // operation = 1(XOR) if Height of tree is
    // even else it is 0(OR) for the root node
    int operationAtRoot = (x % 2 == 0 ? 0 : 1);

    // Fill the allocated memory st
    constructSTUtil(arr, 0, n - 1, st, 0, operationAtRoot);

    // Return the constructed segment tree
    return st;
}

// Driver Code
int main()
{
    // leaf nodes
    int leaves[] = { 1, 6, 3, 7, 5, 9, 10, 4 };

    int n = sizeof(leaves) / sizeof(leaves[0]);

    // Build the segment tree
    int* segmentTree = constructST(leaves, n);

    // Root node is at index 0 considering
    // 0-based indexing in segment Tree
    int rootIndex = 0;

    // print value at rootIndex
    cout << "Value at Root Node = " << segmentTree[rootIndex];
}
```

Output:

Value at Root Node = 3

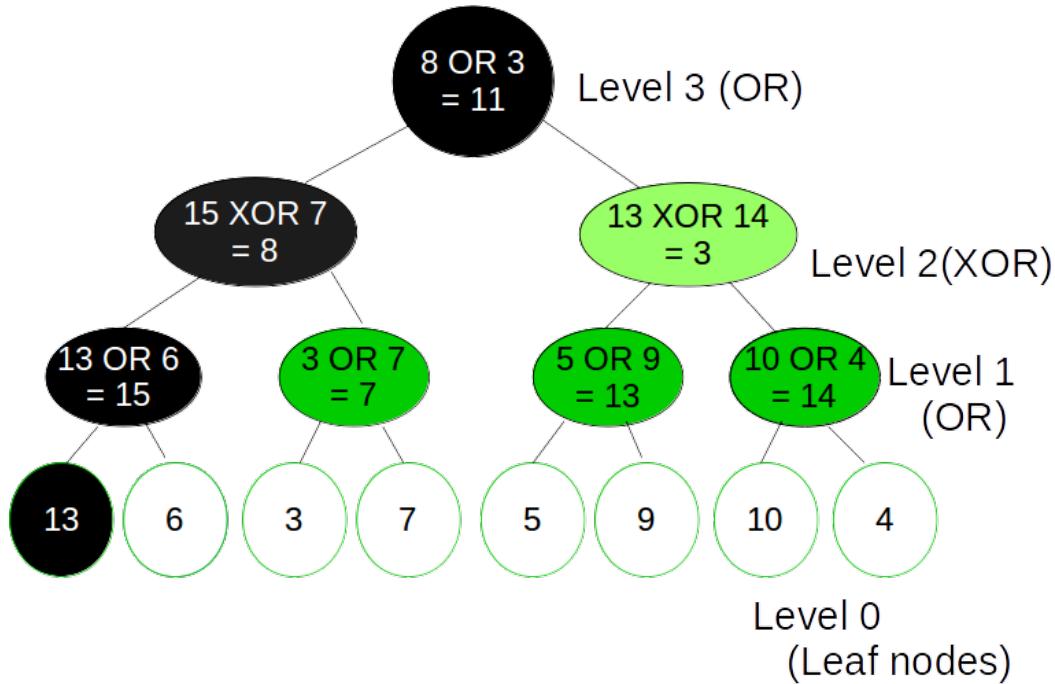
Time Complexity for tree construction is $O(n)$. There are total $2n-1$ nodes, and value of every node is calculated only once in tree construction.

We can also perform Point Updates in a similar manner. If we get an update to update the leaf at index i of the array leaves then we traverse down the tree to the leaf node and perform the update. While coming back to the root node we build the tree again similar to the build() function by passing the operation to be performed at every level and storing the result of applying that operation on values of its left and right children and storing the result into that node.

Consider the following Segment tree after performing the update,

Leaves[0] = 13

Now the updated segment tree looks like this:



Here the nodes in black denote the fact that they were updated

```

// C++ program to build levelwise OR/XOR alternating
// Segment tree (with updates)
#include <bits/stdc++.h>

using namespace std;

// A utility function to get the middle index from
// corner indexes.

```

```

int getMid(int s, int e) { return s + (e - s) / 2; }

// A recursive function that constructs Segment Tree
// for array[ss..se].
// si is index of current node in segment tree st
// operation denotes which operation is carried out
// at that level to merge the left and right child.
// Its either 0 or 1.
void constructSTUtil(int arr[], int ss, int se, int* st,
                     int si, int operation)
{
    // If there is one element in array, store it in
    // current node of segment tree and return
    if (ss == se) {
        st[si] = arr[ss];
        return;
    }

    // If there are more than one elements, then recur
    // for left and right subtrees and store the sum of
    // values in this node
    int mid = getMid(ss, se);

    // Build the left and the right subtrees by using
    // the fact that operation at level (i + 1) = !
    // (operation at level i)
    constructSTUtil(arr, ss, mid, st, si * 2 + 1, !operation);
    constructSTUtil(arr, mid + 1, se, st, si * 2 + 2, !operation);

    // merge the left and right subtrees by checking
    // the operation to be carried. If operation = 1,
    // then do OR else XOR
    if (operation == 1) {

        // OR operation
        st[si] = (st[2 * si + 1] | st[2 * si + 2]);
    }
    else {
        // XOR operation
        st[si] = (st[2 * si + 1] ^ st[2 * si + 2]);
    }
}

/* A recursive function to update the nodes which have the given
   index in their range. The following are parameters
   st, si, ss and se are same as getSumUtil()
   i      --> index of the element to be updated. This index is
             in input array.

```

```

    val --> Value to be assigned to arr[i] */
void updateValueUtil(int* st, int ss, int se, int i,
                     int val, int si, int operation)
{
    // Base Case: If the input index lies outside
    // this segment
    if (i < ss || i > se)
        return;

    // If the input index is in range of this node,
    // then update the value of the node and its
    // children

    // leaf node
    if (ss == se && ss == i) {
        st[si] = val;
        return;
    }

    int mid = getMid(ss, se);

    // Update the left and the right subtrees by
    // using the fact that operation at level
    // (i + 1) = ! (operation at level i)
    updateValueUtil(st, ss, mid, i, val, 2 * si + 1, !operation);
    updateValueUtil(st, mid + 1, se, i, val, 2 * si + 2, !operation);

    // merge the left and right subtrees by checking
    // the operation to be carried. If operation = 1,
    // then do OR else XOR
    if (operation == 1) {

        // OR operation
        st[si] = (st[2 * si + 1] | st[2 * si + 2]);
    }
    else {

        // XOR operation
        st[si] = (st[2 * si + 1] ^ st[2 * si + 2]);
    }
}

// The function to update a value in input array and
// segment tree. It uses updateValueUtil() to update the
// value in segment tree
void updateValue(int arr[], int* st, int n, int i, int new_val)
{
    // Check for erroneous input index

```

```

if (i < 0 || i > n - 1) {
    printf("Invalid Input");
    return;
}

// Height of segment tree
int x = (int)(ceil(log2(n)));

// operation = 1(XOR) if Height of tree is
// even else it is 0(OR) for the root node
int operationAtRoot = (x % 2 == 0 ? 0 : 1);

arr[i] = new_val;

// Update the values of nodes in segment tree
updateValueUtil(st, 0, n - 1, i, new_val, 0, operationAtRoot);
}

/* Function to construct segment tree from given array.
   This function allocates memory for segment tree and
   calls constructSTUtil() to fill the allocated memory */
int* constructST(int arr[], int n)
{
    // Allocate memory for segment tree

    // Height of segment tree
    int x = (int)(ceil(log2(n)));

    // Maximum size of segment tree
    int max_size = 2 * (int)pow(2, x) - 1;

    // Allocate memory
    int* st = new int[max_size];

    // operation = 1(XOR) if Height of tree is
    // even else it is 0(OR) for the root node
    int operationAtRoot = (x % 2 == 0 ? 0 : 1);

    // Fill the allocated memory st
    constructSTUtil(arr, 0, n - 1, st, 0, operationAtRoot);

    // Return the constructed segment tree
    return st;
}

// Driver Code
int main()
{

```

```
int leaves[] = { 1, 6, 3, 7, 5, 9, 10, 4 };
int n = sizeof(leaves) / sizeof(leaves[0]);

// Build the segment tree
int* segmentTree = constructST(leaves, n);

// Root node is at index 0 considering
// 0-based indexing in segment Tree
int rootIndex = 0;

// print old value at rootIndex
cout << "Old Value at Root Node = "
    << segmentTree[rootIndex] << endl;

// perform update leaves[0] = 13
updateValue(leaves, segmentTree, n, 0, 13);

cout << "New Value at Root Node = "
    << segmentTree[rootIndex];
return 0;
}
```

Output:

```
Old Value at Root Node = 3
New Value at Root Node = 11
```

The time complexity of update is also $O(\log n)$. To update a leaf value, we process one node at every level and number of levels is $O(\log n)$.

Improved By : [sirjan13](#)

Source

<https://www.geeksforgeeks.org/levelwise-alternating-xor-operations-segment-tree/>

Chapter 104

Longest Common Extension / LCE Set 2 (Reduction to RMQ)

Longest Common Extension / LCE Set 2 (Reduction to RMQ) - GeeksforGeeks

Prerequisites :

- Suffix Array Set 2
- kasai's algorithm

The Longest Common Extension (LCE) problem considers a string s and computes, for each pair (L, R) , the longest sub string of s that starts at both L and R . In LCE, in each of the query we have to answer the length of the longest common prefix starting at indexes L and R .

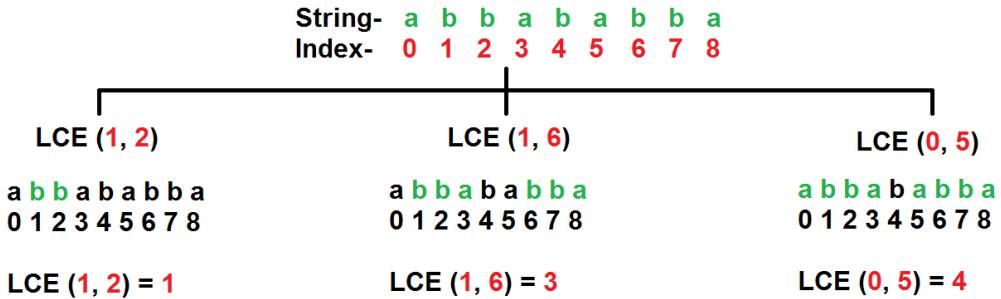
Example:

String : “abbababba”

Queries: LCE(1, 2), LCE(1, 6) and LCE(0, 5)

Find the length of the Longest Common Prefix starting at index given as, **(1, 2), (1, 6) and (0, 5)**.

The string highlighted “green” are the longest common prefix starting at index- L and R of the respective queries. We have to find the length of the longest common prefix starting at index- **(1, 2), (1, 6) and (0, 5)**.



In [Set 1](#), we explained about the naive method to find the length of the LCE of a string on many queries. In this set we will show how a LCE problem can be reduced to a RMQ problem, hence decreasing the asymptotic time complexity of the naive method.

Reduction of LCE to RMQ

Let the input string be S and queries be of the form $\text{LCE}(L, R)$. Let the suffix array for S be $\text{Suff}[]$ and the lcp array be $\text{lcp}[]$.

The longest common extension between two suffixes S_L and S_R of S can be obtained from the lcp array in the following way.

- Let low be the rank of S_L among the suffixes of S (that is, $\text{Suff}[low] = L$).
- Let high be the rank of S_R among the suffixes of S . Without loss of generality, we assume that $low < high$.
- Then the longest common extension of S_L and S_R is $\text{lcp}(low, high) = \min_{(low \leq k \leq high)} \text{lcp}[k]$.

Proof: Let $S_L = S_L \dots S_{L+c} \dots s_n$ and $S_R = S_R \dots S_{R+c} \dots s_n$, and let c be the longest common extension of S_L and S_R (i.e. $S_L \dots S_{L+c-1} = S_R \dots S_{R+c-1}$). We assume that the string S has a sentinel character so that no suffix of S is a prefix of any other suffix of S but itself.

- If $low = high - 1$ then $i = low$ and $\text{lcp}[low] = c$ is the longest common extension of S_L and S_R and we are done.
- If $low < high - 1$ then select i such $\text{lcp}[i]$ is the minimum value in the interval $[low, high]$ of the lcp array. We then have two possible cases:
 - If $c < \text{lcp}[i]$ we have a contradiction because $S_L \dots S_{L+\text{lcp}[i]-1} = S_R \dots S_{R+\text{lcp}[i]-1}$ by the definition of the LCP table, and the fact that the entries of lcp correspond to sorted suffixes of S .
 - If $c > \text{lcp}[i]$, let $high = \text{Suff}[i]$, so that S_{high} is the suffix associated with position i . S_i is such that $s_{high} \dots s_{high+\text{lcp}[i]-1} = S_L \dots S_{L+\text{lcp}[i]-1}$ and $s_{high} \dots s_{high+\text{lcp}[i]-1} = S_R \dots S_{R+\text{lcp}[i]-1}$, but since $S_L \dots S_{L+c-1} = S_R \dots S_{R+c-1}$ we have that the lcp array should be wrongly sorted which is a contradiction.

Therefore we have $c = \text{lcp}[i]$

Thus we have reduced our longest common extension query to a range minimum-query over a range in lcp.

Algorithm

- To find low and high, we must have to compute the suffix array first and then from the suffix array we compute the inverse suffix array.
- We also need lcp array, hence we use Kasai's Algorithm to find lcp array from the suffix array.
- Once the above things are done, we simply find the minimum value in lcp array from index – low to high (as proved above) for each query.

The minimum value is the length of the LCE for that query.

Implementation

```
// A C++ Program to find the length of longest common
// extension using Direct Minimum Algorithm
#include<bits/stdc++.h>
using namespace std;

// Structure to represent a query of form (L,R)
struct Query
{
    int L, R;
};

// Structure to store information of a suffix
struct suffix
{
    int index; // To store original index
    int rank[2]; // To store ranks and next rank pair
};

// A utility function to get minimum of two numbers
int minVal(int x, int y) { return (x < y)? x: y; }

// A utility function to get minimum of two numbers
int maxVal(int x, int y) { return (x > y)? x: y; }

// A comparison function used by sort() to compare
// two suffixes Compares two pairs, returns 1 if
// first pair is smaller
int cmp(struct suffix a, struct suffix b)
```

```

{
    return (a.rank[0] == b.rank[0])?
        (a.rank[1] < b.rank[1]):
        (a.rank[0] < b.rank[0]);
}

// This is the main function that takes a string 'txt'
// of size n as an argument, builds and return the
// suffix array for the given string
vector<int> buildSuffixArray(string txt, int n)
{
    // A structure to store suffixes and their indexes
    struct suffix suffixes[n];

    // Store suffixes and their indexes in an array
    // of structures.
    // The structure is needed to sort the suffixes
    // alphabetically and maintain their old indexes
    // while sorting
    for (int i = 0; i < n; i++)
    {
        suffixes[i].index = i;
        suffixes[i].rank[0] = txt[i] - 'a';
        suffixes[i].rank[1] =
            ((i+1) < n)? (txt[i + 1] - 'a'): -1;
    }

    // Sort the suffixes using the comparison function
    // defined above.
    sort(suffixes, suffixes+n, cmp);

    // At this point, all suffixes are sorted according
    // to first 2 characters. Let us sort suffixes
    // according to first 4/ characters, then first 8
    // and so on

    // This array is needed to get the index in suffixes[]
    // from original index. This mapping is needed to get
    // next suffix.
    int ind[n];

    for (int k = 4; k < 2*n; k = k*2)
    {
        // Assigning rank and index values to first suffix
        int rank = 0;
        int prev_rank = suffixes[0].rank[0];
        suffixes[0].rank[0] = rank;
        ind[suffixes[0].index] = 0;
    }
}

```

```

// Assigning rank to suffixes
for (int i = 1; i < n; i++)
{
    // If first rank and next ranks are same as
    // that of previous/ suffix in array, assign
    // the same new rank to this suffix
    if (suffixes[i].rank[0] == prev_rank &&
        suffixes[i].rank[1] == suffixes[i-1].rank[1])
    {
        prev_rank = suffixes[i].rank[0];
        suffixes[i].rank[0] = rank;
    }
    else // Otherwise increment rank and assign
    {
        prev_rank = suffixes[i].rank[0];
        suffixes[i].rank[0] = ++rank;
    }
    ind[suffixes[i].index] = i;
}

// Assign next rank to every suffix
for (int i = 0; i < n; i++)
{
    int nextindex = suffixes[i].index + k/2;
    suffixes[i].rank[1] = (nextindex < n)?
                           suffixes[ind[nextindex]].rank[0]: -1;
}

// Sort the suffixes according to first k characters
sort(suffixes, suffixes+n, cmp);
}

// Store indexes of all sorted suffixes in the suffix array
vector<int>suffixArr;
for (int i = 0; i < n; i++)
    suffixArr.push_back(suffixes[i].index);

// Return the suffix array
return suffixArr;
}

/* To construct and return LCP */
vector<int> kasai(string txt, vector<int> suffixArr,
                  vector<int> &invSuff)
{
    int n = suffixArr.size();

```

```

// To store LCP array
vector<int> lcp(n, 0);

// Fill values in invSuff[]
for (int i=0; i < n; i++)
    invSuff[suffixArr[i]] = i;

// Initialize length of previous LCP
int k = 0;

// Process all suffixes one by one starting from
// first suffix in txt[]
for (int i=0; i<n; i++)
{
    /* If the current suffix is at n-1, then we don't
       have next substring to consider. So lcp is not
       defined for this substring, we put zero. */
    if (invSuff[i] == n-1)
    {
        k = 0;
        continue;
    }

    /* j contains index of the next substring to
       be considered to compare with the present
       substring, i.e., next string in suffix array */
    int j = suffixArr[invSuff[i]+1];

    // Directly start matching from k'th index as
    // at-least k-1 characters will match
    while (i+k<n && j+k<n && txt[i+k]==txt[j+k])
        k++;

    lcp[invSuff[i]] = k; // lcp for the present suffix.

    // Deleting the starting character from the string.
    if (k>0)
        k--;
}

// return the constructed lcp array
return lcp;
}

// A utility function to find longest common extension
// from index - L and index - R
int LCE(vector<int> lcp, vector<int>invSuff, int n,
        int L, int R)

```

```

{
    // Handle the corner case
    if (L == R)
        return (n-L);

    int low = minVal(invSuff[L], invSuff[R]);
    int high = maxVal(invSuff[L], invSuff[R]);

    int length = lcp[low];

    for (int i=low+1; i<high; i++)
    {
        if (lcp[i] < length)
            length = lcp[i];
    }

    return (length);
}

// A function to answer queries of longest common extension
void LCEQueries(string str, int n, Query q[],
                int m)
{
    // Build a suffix array
    vector<int>suffixArr = buildSuffixArray(str, str.length());

    // An auxiliary array to store inverse of suffix array
    // elements. For example if suffixArr[0] is 5, the
    // invSuff[5] would store 0. This is used to get next
    // suffix string from suffix array.
    vector<int> invSuff(n, 0);

    // Build a lcp vector
    vector<int>lcp = kasai(str, suffixArr, invSuff);

    for (int i=0; i<m; i++)
    {
        int L = q[i].L;
        int R = q[i].R;

        printf ("LCE (%d, %d) = %d\n",
               L, R,
               LCE(lcp, invSuff, n, L, R));
    }

    return;
}

```

```
// Driver Program to test above functions
int main()
{
    string str = "abbababba";
    int n = str.length();

    // LCA Queries to answer
    Query q[] = {{1, 2}, {1, 6}, {0, 5}};
    int m = sizeof(q)/sizeof(q[0]);

    LCEQueries(str, n, q, m);

    return (0);
}
```

Output:

```
LCE (1, 2) = 1
LCE (1, 6) = 3
LCE (0, 5) = 4
```

Analysis of Reduction to RMQ method

Time Complexity :

- To construct the lcp and the suffix array it takes **O(N.logN)** time.
- To answer each query it takes **O(invSuff[R] – invSuff[L])**.
- <http://www.sciencedirect.com/science/article/pii/S1570866710000377>

Source

<https://www.geeksforgeeks.org/longest-common-extension-lce-set-2-reduction-rmq/>

Chapter 105

Longest Common Prefix using Trie

Longest Common Prefix using Trie - GeeksforGeeks

Given a set of strings, find the longest common prefix.

```
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](#).

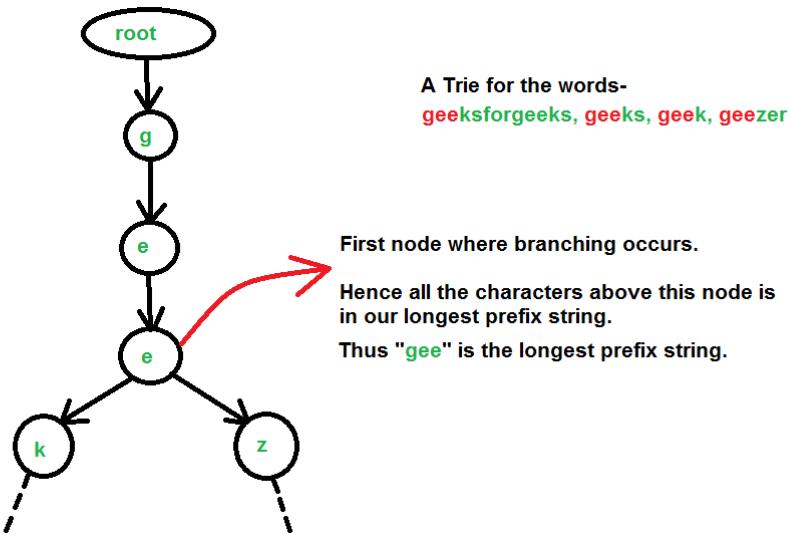
In this article, an approach using [Trie date structure](#) is discussed.

Steps:

- Insert all the words one by one in the trie. After inserting we perform a walk on the trie.
- In this walk, go deeper until we find a node having more than **1 children(branching occurs)** or **0 children (one of the string gets exhausted)**.

This is because the characters (nodes in trie) which are present in the longest common prefix must be the single child of its parent, i.e- there should not be a branching in any of these nodes.

Algorithm Illustration considering strings as – “geeksforgeeks”, “geeks”, “geek”, “geezer”



C++

```

// A Program to find the longest common
// prefix of the given words

#include<bits/stdc++.h>
using namespace std;

// Alphabet size (# of symbols)
#define ALPHABET_SIZE (26)

// Converts key current character into index
// use only 'a' through 'z' and lower case
#define CHAR_TO_INDEX(c) ((int)c - (int)'a')

// Trie node
struct TrieNode
{
    struct TrieNode *children[ALPHABET_SIZE];

    // isLeaf is true if the node represents
    // end of a word
    bool isLeaf;
};

// Returns new trie node (initialized to NULLs)
struct TrieNode *getNode(void)
{
    struct TrieNode *pNode = new TrieNode;

    if (pNode)

```

```

{
    int i;

    pNode->isLeaf = false;

    for (i = 0; i < ALPHABET_SIZE; i++)
        pNode->children[i] = NULL;
}

return pNode;
}

// If not present, inserts key into trie
// If the key is prefix of trie node, just marks leaf node
void insert(struct TrieNode *root, string key)
{
    int length = key.length();
    int index;

    struct TrieNode *pCrawl = root;

    for (int level = 0; level < length; level++)
    {
        index = CHAR_TO_INDEX(key[level]);
        if (!pCrawl->children[index])
            pCrawl->children[index] = getNode();

        pCrawl = pCrawl->children[index];
    }

    // mark last node as leaf
    pCrawl->isLeaf = true;
}

// Counts and returns the number of children of the
// current node
int countChildren(struct TrieNode *node, int *index)
{
    int count = 0;
    for (int i=0; i<ALPHABET_SIZE; i++)
    {
        if (node->children[i] != NULL)
        {
            count++;
            *index = i;
        }
    }
    return (count);
}

```

```
}

// Perform a walk on the trie and return the
// longest common prefix string
string walkTrie(struct TrieNode *root)
{
    struct TrieNode *pCrawl = root;
    int index;
    string prefix;

    while (countChildren(pCrawl, &index) == 1 &&
           pCrawl->isLeaf == false)
    {
        pCrawl = pCrawl->children[index];
        prefix.push_back('a'+index);
    }
    return (prefix);
}

// A Function to construct trie
void constructTrie(string arr[], int n, struct TrieNode *root)
{
    for (int i = 0; i < n; i++)
        insert (root, arr[i]);
    return;
}

// A Function that returns the longest common prefix
// from the array of strings
string commonPrefix(string arr[], int n)
{
    struct TrieNode *root = getNode();
    constructTrie(arr, n, root);

    // Perform a walk on the trie
    return walkTrie(root);
}

// Driver program to test above function
int main()
{
    string arr[] = {"geeksforgeeks", "geeks",
                    "geek", "geezer"};
    int n = sizeof (arr) / sizeof (arr[0]);

    string ans = commonPrefix(arr, n);

    if (ans.length())
```

```
        cout << "The longest common prefix is "
        << ans;
    else
        cout << "There is no common prefix";
    return (0);
}
```

Java

```
// Java Program to find the longest common
// prefix of the given words
public class Longest_common_prefix {

    // Alphabet size (# of symbols)
    static final int ALPHABET_SIZE = 26;

    // Trie node
    static class TrieNode
    {
        TrieNode[] children = new TrieNode[ALPHABET_SIZE];

        // isLeaf is true if the node represents
        // end of a word
        boolean isLeaf;

        // constructor
        public TrieNode() {
            isLeaf = false;
            for (int i = 0; i < ALPHABET_SIZE; i++)
                children[i] = null;
        }
    };

    static TrieNode root;
    static int indexes;

    // If not present, inserts key into trie
    // If the key is prefix of trie node, just marks
    // leaf node
    static void insert(String key)
    {
        int length = key.length();
        int index;

        TrieNode pCrawl = root;

        for (int level = 0; level < length; level++)
        {
```

```

index = key.charAt(level) - 'a';
if (pCrawl.children[index] == null)
    pCrawl.children[index] = new TrieNode();

pCrawl = pCrawl.children[index];
}

// mark last node as leaf
pCrawl.isLeaf = true;
}

// Counts and returns the number of children of the
// current node
static int countChildren(TrieNode node)
{
    int count = 0;
    for (int i=0; i<ALPHABET_SIZE; i++)
    {
        if (node.children[i] != null)
        {
            count++;
            indexs = i;
        }
    }
    return (count);
}

// Perform a walk on the trie and return the
// longest common prefix string
static String walkTrie()
{
    TrieNode pCrawl = root;
    indexs = 0;
    String prefix = "";

    while (countChildren(pCrawl) == 1 &&
           pCrawl.isLeaf == false)
    {
        pCrawl = pCrawl.children[indexs];
        prefix += (char)('a' + indexs);
    }
    return prefix;
}

// A Function to construct trie
static void constructTrie(String arr[], int n)
{
    for (int i = 0; i < n; i++)

```

```

        insert (arr[i]);
        return;
    }

    // A Function that returns the longest common prefix
    // from the array of strings
    static String commonPrefix(String arr[], int n)
    {
        root = new TrieNode();
        constructTrie(arr, n);

        // Perform a walk on the trie
        return walkTrie();
    }

    // Driver program to test above function
    public static void main(String args[])
    {
        String arr[] = {"geeksforgeeks", "geeks",
                        "geek", "geezer"};
        int n = arr.length;

        String ans = commonPrefix(arr, n);

        if (ans.length() != 0)
            System.out.println("The longest common prefix is "+ans);
        else
            System.out.println("There is no common prefix");
    }
}
// This code is contributed by Sumit Ghosh

```

Output :

The longest common prefix is gee

Time Complexity : Inserting all the words in the trie takes $O(MN)$ time and performing a walk on the trie takes $O(M)$ time, where-

N = Number of strings

M = Length of the largest string string

Auxiliary Space: To store all the strings we need to allocate $O(26^M \cdot N) \sim O(MN)$ space for the Trie.

Source

<https://www.geeksforgeeks.org/longest-common-prefix-using-trie/>

Chapter 106

Longest prefix matching – A Trie based solution in Java

Longest prefix matching - A Trie based solution in Java - GeeksforGeeks

Given a dictionary of words and an input string, find the longest prefix of the string which is also a word in dictionary.

Examples:

Let the dictionary contains the following words:
{are, area, base, cat, cater, children, basement}

Below are some input/output examples:

Input String	Output
caterer	cater
basemexy	base
child	< Empty >

Solution

We build a Trie of all dictionary words. Once the Trie is built, traverse through it using characters of input string. If prefix matches a dictionary word, store current length and look for a longer match. Finally, return the longest match.

Following is Java implementation of the above solution based.

```
import java.util.HashMap;

// Trie Node, which stores a character and the children in a HashMap
class TrieNode {
    public TrieNode(char ch) {
        value = ch;
        children = new HashMap<Character, TrieNode>();
    }

    Character value;
    Map<Character, TrieNode> children;
}
```

```

        children = new HashMap<>();
        bIsEnd = false;
    }
    public HashMap<Character,TrieNode> getChildren() { return children; }
    public char getValue() { return value; }
    public void setIsEnd(boolean val) { bIsEnd = val; }
    public boolean isEnd() { return bIsEnd; }

    private char value;
    private HashMap<Character,TrieNode> children;
    private boolean bIsEnd;
}

// Implements the actual Trie
class Trie {
    // Constructor
    public Trie() { root = new TrieNode((char)0); }

    // Method to insert a new word to Trie
    public void insert(String word) {

        // Find length of the given word
        int length = word.length();
        TrieNode crawl = root;

        // Traverse through all characters of given word
        for( int level = 0; level < length; level++ )
        {
            HashMap<Character,TrieNode> child = crawl.getChildren();
            char ch = word.charAt(level);

            // If there is already a child for current character of given word
            if( child.containsKey(ch) )
                crawl = child.get(ch);
            else // Else create a child
            {
                TrieNode temp = new TrieNode(ch);
                child.put( ch, temp );
                crawl = temp;
            }
        }

        // Set bIsEnd true for last character
        crawl.setIsEnd(true);
    }

    // The main method that finds out the longest string 'input'
    public String getMatchingPrefix(String input) {

```

```
String result = ""; // Initialize resultant string
int length = input.length(); // Find length of the input string

// Initialize reference to traverse through Trie
TrieNode crawl = root;

// Iterate through all characters of input string 'str' and traverse
// down the Trie
int level, prevMatch = 0;
for( level = 0 ; level < length; level++ )
{
    // Find current character of str
    char ch = input.charAt(level);

    // HashMap of current Trie node to traverse down
    HashMap<Character,TrieNode> child = crawl.getChildren();

    // See if there is a Trie edge for the current character
    if( child.containsKey(ch) )
    {
        result += ch; //Update result
        crawl = child.get(ch); //Update crawl to move down in Trie

        // If this is end of a word, then update prevMatch
        if( crawl.isEnd() )
            prevMatch = level + 1;
    }
    else break;
}

// If the last processed character did not match end of a word,
// return the previously matching prefix
if( !crawl.isEnd() )
    return result.substring(0, prevMatch);

else return result;
}

private TrieNode root;
}

// Testing class
public class Test {
    public static void main(String[] args) {
        Trie dict = new Trie();
        dict.insert("are");
        dict.insert("area");
        dict.insert("base");
```

```
dict.insert("cat");
dict.insert("cater");
dict.insert("basement");

String input = "caterer";
System.out.print(input + ":    ");
System.out.println(dict.getMatchingPrefix(input));

input = "basement";
System.out.print(input + ":    ");
System.out.println(dict.getMatchingPrefix(input));

input = "are";
System.out.print(input + ":    ");
System.out.println(dict.getMatchingPrefix(input));

input = "arex";
System.out.print(input + ":    ");
System.out.println(dict.getMatchingPrefix(input));

input = "base";
System.out.print(input + ":    ");
System.out.println(dict.getMatchingPrefix(input));

input = "xyz";
System.out.print(input + ":    ");
System.out.println(dict.getMatchingPrefix(input));
}

}
```

Output:

```
caterer:   cater
basement:   basement
are:   are
arex:   are
base:   base
xyz:
```

Time Complexity: Time complexity of finding the longest prefix is $O(n)$ where n is length of the input string. Refer [this](#) for time complexity of building the Trie.

This article is compiled by **Ravi Chandra Enaganti**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

<https://www.geeksforgeeks.org/longest-prefix-matching-a-trie-based-solution-in-java/>

Chapter 107

Longest word in ternary search tree

Longest word in ternary search tree - GeeksforGeeks

Given a set of words represented in a ternary search tree, find the length of largest word among them.

Examples:

```
Input : {"Prakriti", "Raghav",
         "Rashi", "Sunidhi"}
Output : Length of largest word in
          ternary search tree is: 8

Input : {"Boats", "Boat", "But", "Best"}
Output : Length of largest word in
          ternary search tree is: 5
```

Prerequisite : [Ternary Search Tree](#)

The idea is to recursively search the max of left subtree, right subtree and equal tree.
If the current character is same as the root's character increment with 1.

C

```
// C program to find the length of largest word
// in ternary search tree
#include <stdio.h>
#include <stdlib.h>
#define MAX 50

// A node of ternary search tree
```

```
struct Node
{
    char data;

    // True if this character is last
    // character of one of the words
    unsigned isEndOfString: 1;

    struct Node *left, *eq, *right;
};

// A utility function to create a new
// ternary search tree node
struct Node* newNode(char data)
{
    struct Node* temp =
        (struct Node*) malloc(sizeof( struct Node ));
    temp->data = data;
    temp->isEndOfString = 0;
    temp->left = temp->eq = temp->right = NULL;
    return temp;
}

// Function to insert a new word in a Ternary
// Search Tree
void insert(struct Node** root, char *word)
{
    // Base Case: Tree is empty
    if (!(*root))
        *root = newNode(*word);

    // If current character of word is smaller
    // than root's character, then insert this
    // word in left subtree of root
    if ((*word) < (*root)->data)
        insert(&(*root)->left), word);

    // If current character of word is greater
    // than root's character, then insert this
    // word in right subtree of root
    else if ((*word) > (*root)->data)
        insert(&(*root)->right), word);

    // If current character of word is same as
    // root's character,
    else
    {
        if (*(word+1))
```

```
        insert(&(*root)->eq), word+1);

    // the last character of the word
    else
        (*root)->isEndOfString = 1;
    }
}

// Function to find max of three numbers
int max(int a, int b, int c)
{
    int max;
    if (a >= b && a >= c)
        max = a;
    else if (b >= a && b >= c)
        max = b;
    else
        max = c;
}

// Function to find length of largest word in TST
int maxLengthTST(struct Node *root)
{
    if (root == NULL)
        return 0;
    return max(maxLengthTST(root->left),
               maxLengthTST(root->eq)+1,
               maxLengthTST(root->right));
}

// Driver program to test above functions
int main()
{
    struct Node *root = NULL;
    insert(&root, "Prakriti");
    insert(&root, "Raghav");
    insert(&root, "Rashi");
    insert(&root, "Sunidhi");
    int value = maxLengthTST(root);
    printf("Length of largest word in "
           "ternary search tree is: %d\n", value);

    return 0;
}
```

Java

```
// Java program to find the length of largest word
// in ternary search tree
public class GFG {

    static final int MAX = 50;

    // A node of ternary search tree
    static class Node
    {
        char data;

        // True if this character is last
        // character of one of the words
        int isEndOfString = 1;

        Node left, eq, right;

        // constructor
        Node(char data)
        {
            this.data = data;
            isEndOfString = 0;
            left = null;
            eq = null;
            right = null;
        }
    }

    // Function to insert a new word in a Ternary
    // Search Tree
    static Node insert(Node root, String word, int i)
    {
        // Base Case: Tree is empty
        if (root == null)
            root = new Node(word.charAt(i));

        // If current character of word is smaller
        // than root's character, then insert this
        // word in left subtree of root
        if (word.charAt(i) < root.data)
            root.left = insert(root.left, word, i);

        // If current character of word is greater
        // than root's character, then insert this
        // word in right subtree of root
        else if (word.charAt(i) > root.data)
            root.right = insert(root.right, word, i);
    }
}
```

```

// If current character of word is same as
// root's character,
else
{
    if (i + 1 < word.length())
        root.eq = insert(root.eq, word, i + 1);

    // the last character of the word
    else
        root.isEndOfString = 1;
}
return root;
}

// Function to find max of three numbers
static int max(int a, int b, int c)
{
    int max;
    if (a >= b && a >= c)
        max = a;
    else if (b >= a && b >= c)
        max = b;
    else
        max = c;
    return max;
}

// Function to find length of largest word in TST
static int maxLengthTST(Node root)
{
    if (root == null)
        return 0;
    return max(maxLengthTST(root.left),
               maxLengthTST(root.eq)+1,
               maxLengthTST(root.right));
}

// Driver program to test above functions
public static void main(String args[])
{
    Node root = null;
    root = insert(root, "Prakriti", 0);
    root = insert(root, "Raghav", 0);
    root = insert(root, "Rashi", 0);
    root = insert(root, "Sunidhi", 0);
    int value = maxLengthTST(root);
    System.out.println("Length of largest word in "+
```

```
        "ternary search tree is: "+ value);
    }
}

// This code is contributed by Sumit Ghosh
```

Output:

```
Length of largest word in ternary search tree is: 8
```

Source

<https://www.geeksforgeeks.org/longest-word-ternary-search-tree/>

Chapter 108

Maximum Occurrence in a Given Range

Maximum Occurrence in a Given Range - GeeksforGeeks

Given an array of n integers in non-decreasing order. Find the number of occurrences of the most frequent value within a given range.

Examples:

```
Input : arr[] = {-5, -5, 2, 2, 2, 2, 3, 7, 7, 7}
        Query 1: start = 0, end = 9
        Query 2: start = 4, end = 9
Output : 4
         3
Explanation:
Query 1: '2' occurred the most number of times
with a frequency of 4 within given range.
Query 2: '7' occurred the most number of times
with a frequency of 3 within given range.
```

Segment Trees can be used to solve this problem efficiently.

Refer [here](#) for the implementation of segment trees

The key idea behind this problem is that the given array is in non-decreasing order which means that all occurrences of a number are consecutively placed in the array as the array is in sorted order.

A segment tree can be constructed where each node would store the maximum count of its respective range $[i, j]$. For that we will build the frequency array and call RMQ (Range Maximum Query) on this array. For e.g.

```
arr[] = {-5, -5, 2, 2, 2, 2, 3, 7, 7, 7}
freq_arr[] = {2, 2, 4, 4, 4, 1, 3, 3, 3}
where, freq_arr[i] = frequency(arr[i])
```

Now there are two cases to be considered,

Case 1: The value of the numbers at index i and j for the given range are same, i.e. $arr[i] = arr[j]$.

Solving this case is very easy. Since $arr[i] = arr[j]$, all numbers between these indices are same (since the array is non-decreasing). Hence answer for this case is simply count of all numbers between i and j (inclusive both) i.e. $(j - i + 1)$

For e.g.

```
arr[] = {-5, -5, 2, 2, 2, 2, 3, 7, 7, 7}
if the given query range is [3, 5], answer would
be  $(5 - 3 + 1) = 3$ , as 2 occurs 3 times within
given range
```

Case 2: The value of the numbers at index i and j for the given range are different, i.e. $arr[i] \neq arr[j]$

If $arr[i] \neq arr[j]$, then there exists an index k where $arr[i] = arr[k]$ and $arr[i] \neq arr[k + 1]$. This may be a case of partial overlap where some occurrences of a particular number lie in the leftmost part of the given range and some lie just before range starts. Here simply calling RMQ would result into an incorrect answer.

For e.g.

```
arr[] = {-5, -5, 2, 2, 2, 2, 3, 7, 7, 7}
freq_arr[] = {2, 2, 4, 4, 4, 1, 3, 3, 3}
if the given query is [4, 9], calling RMQ on
freq_arr[] will give us 4 as answer which
is incorrect as some occurrences of 2 are
lying outside the range. Correct answer
is 3.
```

Similar situation can happen at the rightmost part of the given range where some occurrences of a particular number lies inside the range and some lies just after the range ends.

Hence for this case, inside the given range we have to count the leftmost same numbers upto some index say i and rightmost same numbers from index say j to the end of the range. And then calling RMQ (Range Maximum Query) between indices i and j and taking maximum of all these three.

For e.g.

```
arr[] = {-5, -5, 2, 2, 2, 2, 3, 7, 7, 7}
freq_arr[] = {2, 2, 4, 4, 4, 1, 3, 3, 3}
```

if the given query is [4, 7], counting leftmost same numbers i.e 2 which occurs 2 times inside the range and rightmost same numbers i.e. 3 which occur only 1 time and RMQ on [6, 6] is 1. Hence maximum would be 2.

Below is the implementation of the above approach

```

// C++ Program to find the occurrence
// of the most frequent number within
// a given range
#include <bits/stdc++.h>
using namespace std;

// A utility function to get the middle index
// from corner indexes.
int getMid(int s, int e) { return s + (e - s) / 2; }

/* A recursive function to get the maximum value in
   a given range of array indexes. The following
   are parameters for this function.

   st    --> Pointer to segment tree
   index --> Index of current node in the segment
              tree. Initially 0 is passed as root is
              always at index 0
   ss & se  --> Starting and ending indexes of the
              segment represented by current node,
              i.e., st[index]
   qs & qe  --> Starting and ending indexes of query
              range */

int RMQUtil(int* st, int ss, int se, int qs, int qe,
            int index)
{
    // If segment of this node is a part of given range,
    // then return the min of the segment
    if (qs <= ss && qe >= se)
        return st[index];

    // If segment of this node is outside the
    // given range
    if (se < qs || ss > qe)
        return 0;

    // If a part of this segment overlaps
    // with the given range
    int mid = getMid(ss, se);
    return max(RMQUtil(st, ss, mid, qs, qe, 2 * index + 1),

```

```

        RMQUtil(st, mid + 1, se, qs, qe, 2 * index + 2));
    }

    // Return minimum of elements in range from
    // index qs (query start) to
    // qe (query end). It mainly uses RMQUtil()
    int RMQ(int* st, int n, int qs, int qe)
    {
        // Check for erroneous input values
        if (qs < 0 || qe > n - 1 || qs > qe) {
            printf("Invalid Input");
            return -1;
        }

        return RMQUtil(st, 0, n - 1, qs, qe, 0);
    }

    // A recursive function that constructs Segment Tree
    // for array[ss..se]. si is index of current node in
    // segment tree st
    int constructSTUtil(int arr[], int ss, int se, int* st,
                        int si)
    {
        // If there is one element in array, store it in
        // current node of segment tree and return
        if (ss == se) {
            st[si] = arr[ss];
            return arr[ss];
        }

        // If there are more than one elements, then
        // recur for left and right subtrees and store
        // the minimum of two values in this node
        int mid = getMid(ss, se);
        st[si] = max(constructSTUtil(arr, ss, mid, st, si * 2 + 1),
                     constructSTUtil(arr, mid + 1, se, st, si * 2 + 2));
        return st[si];
    }

    /* Function to construct segment tree from given
       array. This function allocates memory for segment
       tree and calls constructSTUtil() to fill the
       allocated memory */
    int* constructST(int arr[], int n)
    {
        // Allocate memory for segment tree

        // Height of segment tree

```

```

int x = (int)(ceil(log2(n)));

// Maximum size of segment tree
int max_size = 2 * (int)pow(2, x) - 1;

int* st = new int[max_size];

// Fill the allocated memory st
constructSTUtil(arr, 0, n - 1, st, 0);

// Return the constructed segment tree
return st;
}

int maximumOccurrence(int arr[], int n, int qs, int qe)
{
    // Declaring a frequency array
    int freq_arr[n + 1];

    // Counting frequencies of all array elements.
    unordered_map<int, int> cnt;
    for (int i = 0; i < n; i++)
        cnt[arr[i]]++;

    // Creating frequency array by replacing the
    // number in array to the number of times it
    // has appeared in the array
    for (int i = 0; i < n; i++)
        freq_arr[i] = cnt[arr[i]];

    // Build segment tree from this frequency array
    int* st = constructST(freq_arr, n);

    int maxOcc; // to store the answer

    // Case 1: numbers are same at the starting
    // and ending index of the query
    if (arr[qs] == arr[qe])
        maxOcc = (qe - qs + 1);

    // Case 2: numbers are different
    else {
        int leftmost_same = 0, rightmost_same = 0;

        // Partial Overlap Case of a number with some
        // occurrences lying inside the leftmost
        // part of the range and some just before the
        // range starts
    }
}

```

```

        while (qs > 0 && qs <= qe && arr[qs] == arr[qs - 1]) {
            qs++;
            leftmost_same++;
        }

        // Partial Overlap Case of a number with some
        // occurrences lying inside the rightmost part of
        // the range and some just after the range ends
        while (qe >= qs && qe < n - 1 && arr[qe] == arr[qe + 1]) {
            qe--;
            righmost_same++;
        }

        // Taking maximum of all three
        maxOcc = max({leftmost_same, righmost_same,
                      RMQ(st, n, qs, qe)});
    }
    return maxOcc;
}

// Driver Code
int main()
{
    int arr[] = { -5, -5, 2, 2, 2, 3, 7, 7, 7 };
    int n = sizeof(arr) / sizeof(arr[0]);

    int qs = 0; // Starting index of query range
    int qe = 9; // Ending index of query range

    // Print occurrence of most frequent number
    // within given range
    cout << "Maximum Occurrence in range is = "
        << maximumOccurrence(arr, n, qs, qe) << endl;

    qs = 4; // Starting index of query range
    qe = 9; // Ending index of query range

    // Print occurrence of most frequent number
    // within given range
    cout << "Maximum Occurrence in range is = "
        << maximumOccurrence(arr, n, qs, qe) << endl;

    return 0;
}

```

Output:

Maximum Occurrence in range is = 4

Maximum Occurrence in range is = 3

Further Optimization: For the partial overlapping case we have to run a loop to calculate the count of same numbers on both sides. To avoid this loop and perform this operation in O(1), we can store the index of the first occurrence of every number in the given array and hence by doing some precomputation we can find the required count in O(1).

Time Complexity:

Time Complexity for tree construction is O(n). Time complexity to query is O(Log n).

Source

<https://www.geeksforgeeks.org/maximum-occurrence-given-range/>

Chapter 109

Maximum Subarray Sum in a given Range

Maximum Subarray Sum in a given Range - GeeksforGeeks

Given an array of n numbers, the task is to answer the following queries:

```
maximumSubarraySum(start, end) : Find the maximum
subarray sum in the range from array index 'start'
to 'end'.
```

Also see : [Range Query With Update Required](#)

Examples:

```
Input : arr[] = {1, 3, -4, 5, -2}
        Query 1: start = 0, end = 4
        Query 2: start = 0, end = 2
```

```
Output : 5
        4
```

Explanation:

For Query 1, [1, 3, -4, 5] or ([5]) represent the maximum sum sub arrays with sum = 5.

For Query 2, [1, 3] represents the maximum sum subarray in the query range with sum = 4

Segment Trees can be used to solve this problem. Here, we need to keep information regarding various cumulative sums. At every Node we store the following:

- 1) Maximum Prefix Sum,
- 2) Maximum Suffix Sum,
- 3) Total Sum,
- 4) Maximum Subarray Sum

A classical Segment Tree with each Node storing the above information should be enough to answer each query. The only focus here is on how the left and the right Nodes of the tree are merged together. Now, we will discuss how each of the information is constructed in each of the segment tree Nodes using the information of its left and right child.

Constructing the Maximum Prefix Sum using Left and Right child

There can be two cases for maximum prefix sum of a Node:

1. The maximum prefix sum occurs in the left child,

Left Child	Right Child
3 -4 2 1	4 -2 1 1

Case 1: Maximum Prefix Sum = 3

In this Case,

Maximum Prefix Sum = Maximum Prefix Sum of Left Child

2. The maximum prefix sum contains every array element of the left child and the elements contributing to the maximum prefix sum of the right child,

Left Child	Right Child
3 -4 2 1	4 -2 1 1

Case 2: Maximum Prefix Sum = $3 + (-4) + 2 + 1 + 4 = 6$

In this Case,

Maximum Prefix Sum = Total Sum of Left Child +
Maximum Prefix Sum of Right Child

Constructing the Maximum Suffix Sum using Left and Right child

There can be two cases for maximum suffix sum of a Node:

1. The maximum suffix sum occurs in the right child,

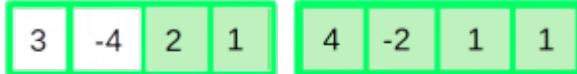
3 -4 2 1	4 -2 1 1
----------------	----------------

Case 1: Maximum Suffix Sum = 4

In this Case,

Maximum Suffix Sum = Maximum Suffix Sum of Right Child

2. The maximum suffix sum contains every array element of the Right child and the elements contributing to the maximum suffix sum of the left child,



$$\text{Case 2: Maximum Suffix Sum} = 2 + 1 + 4 + (-2) + 1 + 1 \\ = 7$$

In this Case,

Maximum Suffix Sum = Total Sum of Right Child +
Maximum Suffix Sum of Left Child

Constructing the Maximum Subarray Sum using Left and Right child

There can be three cases for the maximum sub-array sum of a Node:

1. The maximum sub-array sum occurs in the left child,



$$\text{Case 2: Maximum Subarray Sum} = 3$$

In this Case,

Maximum Sub-array Sum = Maximum Subarray Sum of Left Child

2. The maximum sub-array sum occurs in the right child,



$$\text{Case 1: Maximum Subarray Sum} = 4$$

In this Case,

Maximum Sub-array Sum = Maximum Subarray Sum of Right Child

3. The maximum subarray sum, contains array elements of the right child contributing to the maximum prefix sum of the right child, and the array elements of the Left child

contributing to the maximum suffix sum of the left child,



$$\text{Case 3: Maximum Subarray Sum} = 2 + 1 + 4 \\ = 7$$

In this Case,

$$\begin{aligned} \text{Maximum Subarray Sum} &= \text{Maximum Prefix Sum of Right Child} \\ &\quad + \\ &\quad \text{Maximum Suffix Sum of Left Child} \end{aligned}$$

```
// C++ Program to Implement Maximum Sub-Array Sum in a range
#include <bits/stdc++.h>
using namespace std;

#define inf 0x3f3f

/* Node of the segment tree consisting of:
1. Maximum Prefix Sum,
2. Maximum Suffix Sum,
3. Total Sum,
4. Maximum Sub-Array Sum */
struct Node {
    int maxPrefixSum;
    int maxSuffixSum;
    int totalSum;
    int maxSubarraySum;

    Node()
    {
        maxPrefixSum = maxSuffixSum = maxSubarraySum = -inf;
        totalSum = -inf;
    }
};

// Returns Parent Node after merging its left and right child
Node merge(Node leftChild, Node rightChild)
{
    Node parentNode;
    parentNode.maxPrefixSum = max(leftChild.maxPrefixSum,
                                  leftChild.totalSum +
                                  rightChild.maxPrefixSum);

    parentNode.maxSuffixSum = max(rightChild.maxSuffixSum,
                                  rightChild.totalSum +
                                  leftChild.totalSum);
}

int main()
{
    int arr[] = {3, -4, 2, 1, 4, -2, 1, 1};
    int n = sizeof(arr)/sizeof(arr[0]);
    Node root = merge(arr, n);
    cout << "Maximum Subarray Sum is " << root.maxSubarraySum;
}
```

```

        leftChild.maxSuffixSum);

parentNode.totalSum = leftChild.totalSum +
                    rightChild.totalSum;

parentNode.maxSubarraySum = max({leftChild.maxSubarraySum,
                                 rightChild.maxSubarraySum,
                                 leftChild.maxSuffixSum +
                                 rightChild.maxPrefixSum});

return parentNode;
}

// Builds the Segment tree recursively
void constructTreeUtil(Node* tree, int arr[], int start,
                      int end, int index)
{

/* Leaf Node */
if (start == end) {

    // single element is covered under this range
    tree[index].totalSum = arr[start];
    tree[index].maxSuffixSum = arr[start];
    tree[index].maxPrefixSum = arr[start];
    tree[index].maxSubarraySum = arr[start];
    return;
}

// Recursively Build left and right children
int mid = (start + end) / 2;
constructTreeUtil(tree, arr, start, mid, 2 * index);
constructTreeUtil(tree, arr, mid + 1, end, 2 * index + 1);

// Merge left and right child into the Parent Node
tree[index] = merge(tree[2 * index], tree[2 * index + 1]);
}

/* Function to construct segment tree from given array.
   This function allocates memory for segment tree and
   calls constructTreeUtil() to fill the allocated
   memory */
Node* constructTree(int arr[], int n)
{
    // Allocate memory for segment tree
    int x = (int)(ceil(log2(n))); // Height of the tree

    // Maximum size of segment tree
}

```

```

int max_size = 2 * (int)pow(2, x) - 1;
Node* tree = new Node[max_size];

// Fill the allocated memory tree
constructTreeUtil(tree, arr, 0, n - 1, 1);

// Return the constructed segment tree
return tree;
}

/* A Recursive function to get the desired
Maximum Sum Sub-Array,
The following are parameters of the function-
tree      --> Pointer to segment tree
index --> Index of the segment tree Node
ss & se  --> Starting and ending indexes of the
                segment represented by
                current Node, i.e., tree[index]
qs & qe  --> Starting and ending indexes of query range */
Node queryUtil(Node* tree, int ss, int se, int qs,
               int qe, int index)
{
    // No overlap
    if (ss > qe || se < qs) {

        // returns a Node for out of bounds condition
        Node nullNode;
        return nullNode;
    }

    // Complete overlap
    if (ss >= qs && se <= qe) {
        return tree[index];
    }

    // Partial Overlap Merge results of Left
    // and Right subtrees
    int mid = (ss + se) / 2;
    Node left = queryUtil(tree, ss, mid, qs, qe,
                          2 * index);
    Node right = queryUtil(tree, mid + 1, se, qs,
                           qe, 2 * index + 1);

    // merge left and right subtree query results
    Node res = merge(left, right);
    return res;
}

```

```
/* Returns the Maximum Subarray Sum between start and end
   It mainly uses queryUtil(). */
int query(Node* tree, int start, int end, int n)
{
    Node res = queryUtil(tree, 0, n - 1, start, end, 1);
    return res.maxSubarraySum;
}

int main()
{
    int arr[] = { 1, 3, -4, 5, -2 };
    int n = sizeof(arr) / sizeof(arr[0]);

    // Construct Segment Tree
    Node* Tree = constructTree(arr, n);
    int start, end, maxSubarraySum;

    // Answering query 1:
    start = 0;
    end = 4;
    maxSubarraySum = query(Tree, start, end, n);
    cout << "Maximum Sub-Array Sum between "
        << start << " and " << end
        << " = " << maxSubarraySum << "\n";

    // Answering query 2:
    start = 0;
    end = 2;
    maxSubarraySum = query(Tree, start, end, n);
    cout << "Maximum Sub-Array Sum between "
        << start << " and " << end
        << " = " << maxSubarraySum << "\n";

    return 0;
}
```

Output:

```
Maximum Sub-Array Sum between 0 and 4 = 5
Maximum Sub-Array Sum between 0 and 2 = 4
```

Time Complexity: O(logn) for each query.

Improved By : [atulim](#)

Source

<https://www.geeksforgeeks.org/maximum-subarray-sum-given-range/>

Chapter 110

Maximum Sum Increasing Subsequence using Binary Indexed Tree

Maximum Sum Increasing Subsequence using Binary Indexed Tree - GeeksforGeeks

Given an array of size n. Find the maximum sum an increasing subsequence.

Examples:

```
Input : arr[] = { 1, 20, 4, 2, 5 }
Output : Maximum sum of increasing subsequence is = 21
The subsequence 1, 20 gives maximum sum which is 21
```

```
Input : arr[] = { 4, 2, 3, 1, 5, 8 }
Output : Maximum sum of increasing subsequence is = 18
The subsequence 2, 3, 5, 8 gives maximum sum which is 18
```

Prerequisite

The solution makes the use of [Binary Indexed Tree](#) and map .

Dynamic Programming Approach : [DP approach](#) which is in $O(n^2)$.

Solution

Step 1 :

The first step is to insert all values in a map, later we can map these array values to the indexes of Binary indexed Tree.

Step 2 :

Iterate the map and assign indexes . What this would do is for an array { 4, 2, 3, 8, 5, 2 }
2 will be assigned index 1
3 will be assigned index 2

4 will be assigned index 3
5 will be assigned index 4
8 will be assigned index 5

Step 3 :

Construct the Binary Indexed Tree.

Step 4 :

For every value in the given array do the following.

Find the maximum sum till that position using BIT and then update the BIT with New Maximum Value

Step 5 :

Returns the maximum sum which is present at last position in Binary Indexed Tree.

CPP

```
// CPP code for Maximum Sum
// Increasing Subsequence
#include <bits/stdc++.h>
using namespace std;

// Returns the maximum value of
// the increasing subsequence
// till that index
// Link to understand getSum function
// https://www.geeksforgeeks.org/binary-indexed-tree-or-fenwick-tree-2/
int getSum(int BITree[], int index)
{
    int sum = 0;
    while (index > 0) {
        sum = max(sum, BITree[index]);
        index -= index & (-index);
    }
    return sum;
}

// Updates a node in Binary Index
// Tree (BITree) at given index in
// BITree. The max value is updated
// by taking max of 'val' and the
// already present value in the node.
void updateBIT(int BITree[], int newIndex,
               int index, int val)
{
    while (index <= newIndex) {
        BITree[index] = max(val, BITree[index]);
        index += index & (-index);
    }
}
```

```
// maxSumIS() returns the maximum
// sum of increasing subsequence
// in arr[] of size n
int maxSumIS(int arr[], int n)
{
    int newindex = 0, max_sum;

    map<int, int> uniqueArr;

    // Inserting all values in map uniqueArr
    for (int i = 0; i < n; i++) {
        uniqueArr[arr[i]] = 0;
    }

    // Assigning indexes to all
    // the values present in map
    for (map<int, int>::iterator it = uniqueArr.begin();
         it != uniqueArr.end(); it++) {

        // newIndex is actually the count of
        // unique values in the array.
        newindex++;

        uniqueArr[it->first] = newindex;
    }

    // Constructing the BIT
    int* BITree = new int[newindex + 1];

    // Initializing the BIT
    for (int i = 0; i <= newindex; i++) {
        BITree[i] = 0;
    }

    for (int i = 0; i < n; i++) {
        // Finding maximum sum till this element
        max_sum = getSum(BITree, uniqueArr[arr[i]] - 1);

        // Updating the BIT with new maximum sum
        updateBIT(BITree, newindex,
                  uniqueArr[arr[i]], max_sum + arr[i]);
    }

    // return maximum sum
    return getSum(BITree, newindex);
}
```

```
// Driver program
int main()
{
    int arr[] = { 1, 101, 2, 3, 100, 4, 5 };
    int n = sizeof(arr) / sizeof(arr[0]);
    cout << "Maximum sum is = " << maxSumIS(arr, n);

    return 0;
}
```

Output:

```
Maximum sum of increasing subsequence is = 106
```

Note

Time Complexity of the solution

$O(n \log n)$ for the map and $O(n \log n)$ for updating and getting sum . So over all complexity is still $O(n \log n)$.

Source

<https://www.geeksforgeeks.org/maximum-sum-increasing-subsequence-using-binary-indexed-tree/>

Chapter 111

Merge Sort Tree (Smaller or equal elements in given row range)

Merge Sort Tree (Smaller or equal elements in given row range) - GeeksforGeeks

Given an array where each element is a vector containing integers in sorted order. The task is to answer following queries:

```
count(start, end, k) : Count the numbers smaller than or equal  
                      to k in range from array index 'start'  
                      to 'end'.
```

For convenience we consider an $n * n$ 2-D array where each row corresponds to an integer vector.

Examples:

```
Input : ar[][] = {{2, 4, 5},  
                  {3, 4, 9},  
                  {6, 8, 10}}  
  
Queries[] = (0, 1, 5)  
            (1, 2, 1)  
            (0, 2, 6)  
  
Output : 5  
        0  
        6  
Count of elements (smaller than or equal to 5) from
```

```
1st row (index 0) to 2nd row (index 1) is 5.  
Count of elements (smaller than or equal to 1) from  
2nd row to 3rd row is 0  
Count of elements (smaller than or equal to 6) from  
1st row to 3rd row is 6.
```

The key idea is to build a [Segment Tree](#) with a vector at every node and the vector contains all the elements of the sub-range in a sorted order. And if we observe this segment tree structure this is somewhat similar to the tree formed during the [merge sort algorithm](#)(that is why it is called merge sort tree)

```
// C++ program to count number of smaller or  
// equal to given number and given row range.  
#include<bits/stdc++.h>  
using namespace std;  
  
const int MAX = 1000;  
  
// Constructs a segment tree and stores sTree[]  
void buildTree(int idx, int ss, int se, vector<int> a[],  
                vector<int> sTree[])  
{  
    /*leaf node*/  
    if (ss == se)  
    {  
        sTree[idx] = a[ss];  
        return;  
    }  
  
    int mid = (ss+se)/2;  
  
    /* building left subtree */  
    buildTree(2*idx+1, ss, mid, a, sTree);  
  
    /* building right subtree */  
    buildTree(2*idx+2, mid+1, se, a, sTree);  
  
    /* merging left and right child in sorted order */  
    merge(sTree[2*idx+1].begin(), sTree[2*idx+1].end(),  
          sTree[2*idx+2].begin(), sTree[2*idx+2].end(),  
          back_inserter(sTree[idx]));  
}  
  
// Recursive function to count smaller elements from row  
// a[ss] to a[se] and value smaller than or equal to k.  
int queryRec(int node, int start, int end, int ss, int se,  
             int k, vector<int> a[], vector<int> sTree[])  
{
```

```
/* If out of range return 0 */
if (ss > end || start > se)
    return 0;

/* if inside the range return count */
if (ss <= start && se >= end)
{
    /* binary search over the sorted vector
       to return count >= X */
    return upper_bound(sTree[node].begin(),
                        sTree[node].end(), k) -
           sTree[node].begin();
}

int mid = (start+end)/2;

/*searching in left subtree*/
int p1 = queryRec(2*node+1, start, mid, ss, se, k, a, sTree);

/*searching in right subtree*/
int p2 = queryRec(2*node+2, mid+1, end, ss, se, k, a, sTree);

/*adding both the result*/
return p1 + p2;
}

// A wrapper over query().
int query(int start, int end, int k, vector<int> a[], int n,
          vector<int> sTree[])
{
    return queryRec(0, 0, n-1, start, end, k, a, sTree);
}

// Driver code
int main()
{
    int n = 3;
    int arr[][][3] = { {2, 4, 5},
                      {3, 4, 9},
                      {6, 8, 10}};

    // build an array of vectors from above input
    vector<int> a[n];
    for (int i=0; i<n; i++)
        for (int j=0; j<n; j++)
            a[i].push_back(arr[i][j]);

    // Construct segment tree
```

```
vector<int> sTree[MAX];
buildTree(0, 0, n-1, a, sTree);

/* un-comment to print merge sort tree*/
/*for (int i=0;i<2*n-1;i++)
{
    cout << i << " ";
    for (int j=0;j<sTree[i].size();j++)
        cout << sTree[i][j]<<" ";
    cout << endl;
}*/
```

// Answer queries
cout << query(0, 1, 5, a, n, sTree) << endl;
cout << query(1, 2, 1, a, n, sTree) << endl;
cout << query(0, 2, 6, a, n, sTree) << endl;

```
return 0;
}
```

Output:

```
5
0
6
```

buildTree() analysis : Build a merge sort tree takes $O(N \log N)$ time which is same as Merge Sort Algorithm. It also takes $O(n \log n)$ extra space.

query() analysis : A range ‘start’ to ‘end’ can divided into at most $\log(n)$ parts, where we will perform binary search on each part . Binary search requires $O(\log n)$. Therefore total complexity $O(\log n * \log n)$.

Source

<https://www.geeksforgeeks.org/merge-sort-tree-smaller-or-equal-elements-in-given-row-range/>

Chapter 112

Merge Sort Tree for Range Order Statistics

Merge Sort Tree for Range Order Statistics - GeeksforGeeks

Given an array of n numbers, the task is to answer the following queries:

```
kthSmallest(start, end, k) : Find the Kth smallest
                                number in the range from array
                                index 'start' to 'end'.
```

Examples:

```
Input : arr[] = {3, 2, 5, 1, 8, 9|
    Query 1: start = 2, end = 5, k = 2
    Query 2: start = 1, end = 6, k = 4
Output : 2
        5
```

Explanation:

[2, 5, 1, 8] represents the range from 2 to 5 and 2 is the 2nd smallest number in the range[3, 2, 5, 1, 8, 9] represents the range from 1 to 6 and 5 is the 4th smallest number in the range

The key idea is to build a [Segment Tree](#) with a vector at every node and the vector contains all the elements of the sub-range in a sorted order. And if we observe this segment tree structure this is somewhat similar to the tree formed during the [merge sort algorithm](#)(that is why it is called merge sort tree)

We use same implementation as discussed in [Merge Sort Tree \(Smaller or equal elements in given row range\)](#)

Firstly, we maintain a vector of pairs where each pair {value, index} is such that first element of pair represents the element of the input array and the second element of the pair represents the index at which it occurs.

Now we sort this vector of pairs on the basis of the first element of each pair.

After this we build a Merge Sort Tree where each node has a vector of indices in the sorted range.

When we have to answer a query we find if the K^{th} smallest number lies in the left sub-tree or in the right sub-tree. The idea is to use two binary searches and find the number of elements in the left sub-tree such that the indices lie within the given query range.

Let the number of such indices be M .

If $M \geq K$, it means we will be able to find the K^{th} smallest Number in the left sub-tree thus we call on the left sub-tree.

Else the K^{th} smallest number lies in the right sub-tree but this time we don't have to look for the K^{th} smallest number as we already have first M smallest numbers of the range in the left sub-tree thus we should look for the remaining part ie the $(K-M)^{\text{th}}$ number in the right sub-tree.

This is the Index of K^{th} smallest number the value at this index is the required number.

```
// CPP program to implement k-th order statistics
#include <bits/stdc++.h>
using namespace std;

const int MAX = 1000;

// Constructs a segment tree and stores tree[]
void buildTree(int treeIndex, int l, int r,
               vector<pair<int, int>> a, vector<int> tree[])
{
    /* l => start of range,
       r => ending of a range
       treeIndex => index in the Segment Tree/Merge
       Sort Tree */
    /* leaf node */
    if (l == r) {
        tree[treeIndex].push_back(a[l].second);
        return;
    }

    int mid = (l + r) / 2;
```

```

/* building left subtree */
buildTree(2 * treeIndex, l, mid, a, tree);

/* building left subtree */
buildTree(2 * treeIndex + 1, mid + 1, r, a, tree);

/* merging left and right child in sorted order */
merge(tree[2 * treeIndex].begin(),
      tree[2 * treeIndex].end(),
      tree[2 * treeIndex + 1].begin(),
      tree[2 * treeIndex + 1].end(),
      back_inserter(tree[treeIndex]));
}

// Returns the Kth smallest number in query range
int queryRec(int segmentStart, int segmentEnd,
             int queryStart, int queryEnd, int treeIndex,
             int K, vector<int> tree[])
{
    /*
        segmentStart => start of a Segment,
        segmentEnd   => ending of a Segment,
        queryStart   => start of a query range,
        queryEnd     => ending of a query range,
        treeIndex     => index in the Segment
                           Tree/Merge Sort Tree,
        K   => kth smallest number to find  */

    if (segmentStart == segmentEnd)
        return tree[treeIndex][0];

    int mid = (segmentStart + segmentEnd) / 2;

    // finds the last index in the segment
    // which is <= queryEnd
    int last_in_query_range =
        (upper_bound(tree[2 * treeIndex].begin(),
                    tree[2 * treeIndex].end(),
                    queryEnd)
         - tree[2 * treeIndex].begin());

    // finds the first index in the segment
    // which is >= queryStart
    int first_in_query_range =
        (lower_bound(tree[2 * treeIndex].begin(),
                    tree[2 * treeIndex].end(),
                    queryStart)
         - tree[2 * treeIndex].begin());

```

```
int M = last_in_query_range - first_in_query_range;

if (M >= K) {

    // Kth smallest is in left subtree,
    // so recursively call left subtree for Kth
    // smallest number
    return queryRec(segmentStart, mid, queryStart,
                    queryEnd, 2 * treeIndex, K, tree);
}

else {

    // Kth smallest is in right subtree,
    // so recursively call right subtree for the
    // (K-M)th smallest number
    return queryRec(mid + 1, segmentEnd, queryStart,
                    queryEnd, 2 * treeIndex + 1, K - M, tree);
}

// A wrapper over query()
int query(int queryStart, int queryEnd, int K, int n,
          vector<pair<int, int> > a, vector<int> tree[])
{

    return queryRec(0, n - 1, queryStart - 1, queryEnd - 1,
                   1, K, tree);
}

// Driver code
int main()
{
    int arr[] = { 3, 2, 5, 1, 8, 9 };
    int n = sizeof(arr)/sizeof(arr[0]);

    // vector of pairs of form {element, index}
    vector<pair<int, int> > v;
    for (int i = 0; i < n; i++) {
        v.push_back(make_pair(arr[i], i));
    }

    // sort the vector
    sort(v.begin(), v.end());

    // Construct segment tree in tree[]
    vector<int> tree[MAX];
```

```
buildTree(1, 0, n - 1, v, tree);

// Answer queries
// kSmallestIndex hold the index of the kth smallest number
int kSmallestIndex = query(2, 5, 2, n, v, tree);
cout << arr[kSmallestIndex] << endl;

kSmallestIndex = query(1, 6, 4, n, v, tree);
cout << arr[kSmallestIndex] << endl;

return 0;
}
```

Output:

```
2
5
```

Thus, we can get the K^{th} smallest number query in range L to R, in $O(n(\log n)^2)$ by building the merge sort tree on indices.

Source

<https://www.geeksforgeeks.org/merge-sort-tree-for-range-order-statistics/>

Chapter 113

Min-Max Range Queries in Array

Min-Max Range Queries in Array - GeeksforGeeks

Given an array $\text{arr}[0 \dots n-1]$. We need to efficiently find the minimum and maximum value from index qs (query start) to qe (query end) where $0 \leq qs \leq qe \leq n-1$. We are given multiple queries.

Examples:

```
Input : arr[] = {1, 8, 5, 9, 6, 14, 2, 4, 3, 7}
        queries = 5
        qs = 0 qe = 4
        qs = 3 qe = 7
        qs = 1 qe = 6
        qs = 2 qe = 5
        qs = 0 qe = 8
Output: Minimum = 1 and Maximum = 9
        Minimum = 2 and Maximum = 14
        Minimum = 2 and Maximum = 14
        Minimum = 5 and Maximum = 14
        Minimum = 1 and Maximum = 14
```

Simple Solution : We solve this problem using [Tournament Method](#) for each query. Complexity for this approach will be $O(\text{queries} * n)$.

Efficient solution : This problem can be solved more efficiently by using [Segment Tree](#). First read given segment tree link then start solving this problem.

```
// C++ program to find minimum and maximum using segment tree
#include<bits/stdc++.h>
```

```

using namespace std;

// Node for storing minimum nd maximum value of given range
struct node
{
    int minimum;
    int maximum;
};

// A utility function to get the middle index from corner indexes.
int getMid(int s, int e) { return s + (e - s)/2; }

/* A recursive function to get the minimum and maximum value in
   a given range of array indexes. The following are parameters
   for this function.

   st    --> Pointer to segment tree
   index --> Index of current node in the segment tree. Initially
              0 is passed as root is always at index 0
   ss & se  --> Starting and ending indexes of the segment
                 represented by current node, i.e., st[index]
   qs & qe  --> Starting and ending indexes of query range */
struct node MaxMinUntill(struct node *st, int ss, int se, int qs,
                         int qe, int index)
{
    // If segment of this node is a part of given range, then return
    // the minimum and maximum node of the segment
    struct node tmp, left, right;
    if (qs <= ss && qe >= se)
        return st[index];

    // If segment of this node is outside the given range
    if (se < qs || ss > qe)
    {
        tmp.minimum = INT_MAX;
        tmp.maximum = INT_MIN;
        return tmp;
    }

    // If a part of this segment overlaps with the given range
    int mid = getMid(ss, se);
    left = MaxMinUntill(st, ss, mid, qs, qe, 2*index+1);
    right = MaxMinUntill(st, mid+1, se, qs, qe, 2*index+2);
    tmp.minimum = min(left.minimum, right.minimum);
    tmp.maximum = max(left.maximum, right.maximum);
    return tmp;
}

```

```

// Return minimum and maximum of elements in range from index
// qs (quey start) to qe (query end). It mainly uses
// MaxMinUtil()
struct node MaxMin(struct node *st, int n, int qs, int qe)
{
    struct node tmp;

    // Check for erroneous input values
    if (qs < 0 || qe > n-1 || qs > qe)
    {
        printf("Invalid Input");
        tmp.minimum = INT_MIN;
        tmp.maximum = INT_MAX;
        return tmp;
    }

    return MaxMinUtil(st, 0, n-1, qs, qe, 0);
}

// A recursive function that constructs Segment Tree for array[ss..se].
// si is index of current node in segment tree st
void constructSTUtil(int arr[], int ss, int se, struct node *st,
                     int si)
{
    // If there is one element in array, store it in current node of
    // segment tree and return
    if (ss == se)
    {
        st[si].minimum = arr[ss];
        st[si].maximum = arr[ss];
        return ;
    }

    // If there are more than one elements, then recur for left and
    // right subtrees and store the minimum and maximum of two values
    // in this node
    int mid = getMid(ss, se);
    constructSTUtil(arr, ss, mid, st, si*2+1);
    constructSTUtil(arr, mid+1, se, st, si*2+2);

    st[si].minimum = min(st[si*2+1].minimum, st[si*2+2].minimum);
    st[si].maximum = max(st[si*2+1].maximum, st[si*2+2].maximum);
}

/* Function to construct segment tree from given array. This function
   allocates memory for segment tree and calls constructSTUtil() to
   fill the allocated memory */
struct node *constructST(int arr[], int n)

```

```
{
    // Allocate memory for segment tree

    // Height of segment tree
    int x = (int)(ceil(log2(n)));

    // Maximum size of segment tree
    int max_size = 2*(int)pow(2, x) - 1;

    struct node *st = new struct node[max_size];

    // Fill the allocated memory st
    constructSTUtil(arr, 0, n-1, st, 0);

    // Return the constructed segment tree
    return st;
}

// Driver program to test above functions
int main()
{
    int arr[] = {1, 8, 5, 9, 6, 14, 2, 4, 3, 7};
    int n = sizeof(arr)/sizeof(arr[0]);

    // Build segment tree from given array
    struct node *st = constructST(arr, n);

    int qs = 0; // Starting index of query range
    intqe = 8; // Ending index of query range
    struct node result=MaxMin(st, n, qs, qe);

    // Print minimum and maximum value in arr[qs..qe]
    printf("Minimum = %d and Maximum = %d ",
           result.minimum, result.maximum);

    return 0;
}
```

Output:

```
Minimum = 1 and Maximum = 14
```

Time Complexity : $O(\text{queries} * \log n)$

Can we do better if there are no updates on array?

The above segment tree based solution also allows array updates also to happen in $O(\log n)$ time. Assume a situation when there are no updates (or array is static). We can actually process all queries in $O(1)$ time with some preprocessing. One simple solution is to make

a 2D table of nodes that stores all range minimum and maximum. This solution requires $O(1)$ query time, but requires $O(n^2)$ preprocessing time and $O(n^2)$ extra space which can be a problem for large n . We can solve this problem in $O(1)$ query time, $O(n \log n)$ space and $O(n \log n)$ preprocessing time using [Sparse Table](#).

This article is contributed by [Shashank Mishra](#). This article is reviewed by team GeeksForGeeks.

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/min-max-range-queries-array/>

Chapter 114

Minimum Word Break

Minimum Word Break - GeeksforGeeks

Given a string s, break s such that every substring of the partition can be found in the dictionary. Return the minimum break needed.

Examples:

Given a dictionary ["Cat", "Mat", "Ca",
"tM", "at", "C", "Dog", "og", "Do"]

Input : Pattern "CatMat"
Output : 1
Explanation: we can break the sentences
in three ways, as follows:
CatMat = [Cat Mat] break 1
CatMat = [Ca tM at] break 2
CatMat = [C at Mat] break 2 so the
output is: 1

Input : Dogcat
Output : 1

Asked in: **Facebook**

Solution of this problem is based on the [WordBreak Trie solution](#) and level ordered graph. We start traversing given pattern and start finding a character of pattern in a **trie**. If we reach a node(leaf) of a trie from where we can traverse a new word of a trie(dictionary), we increment level by one and call search function for rest of the pattern character in a trie. In the end, we return minimum Break.

```
MinBreak(Trie, key, level, start = 0 )
```

```

.... If start == key.length()
...update min_break
for i = start to keylength
....If we found a leaf node in trie
    MinBreak( Trie, key, level+1, i )

```

Below is the implementation of above idea

C++

```

// C++ program to find minimum breaks needed
// to break a string in dictionary words.
#include <bits/stdc++.h>
using namespace std;

const int ALPHABET_SIZE = 26;

// trie node
struct TrieNode {
    struct TrieNode* children[ALPHABET_SIZE];

    // isEndOfWord is true if the node
    // represents end of a word
    bool isEndOfWord;
};

// Returns new trie node (initialized to NULLs)
struct TrieNode* getNode(void)
{
    struct TrieNode* pNode = new TrieNode;

    pNode->isEndOfWord = false;

    for (int i = 0; i < ALPHABET_SIZE; i++)
        pNode->children[i] = NULL;

    return pNode;
}

// If not present, inserts the key into the trie
// If the key is the prefix of trie node, just
// marks leaf node
void insert(struct TrieNode* root, string key)
{
    struct TrieNode* pCrawl = root;

    for (int i = 0; i < key.length(); i++) {
        int index = key[i] - 'a';
        if (!pCrawl->children[index])

```

```

pCrawl->children[index] = getNode();

pCrawl = pCrawl->children[index];
}

// mark last node as leaf
pCrawl->isEndOfWord = true;
}

// function break the string into minimum cut
// such the every substring after breaking
// in the dictionary.
void minWordBreak(struct TrieNode* root,
                  string key, int start, int* min_Break,
                  int level = 0)
{
    struct TrieNode* pCrawl = root;

    // base case, update minimum Break
    if (start == key.length()) {
        *min_Break = min(*min_Break, level - 1);
        return;
    }

    // traverse given key(pattern)
    int minBreak = 0;
    for (int i = start; i < key.length(); i++) {
        int index = key[i] - 'a';
        if (!pCrawl->children[index])
            return;

        // if we find a condition were we can
        // move to the next word in a trie
        // dictionary
        if (pCrawl->children[index]->isEndOfWord)
            minWordBreak(root, key, i + 1,
                         min_Break, level + 1);

        pCrawl = pCrawl->children[index];
    }
}

// Driver program to test above functions
int main()
{
    string dictionary[] = { "Cat", "Mat",
                           "Ca", "Ma", "at", "C", "Dog", "og", "Do" };
    int n = sizeof(dictionary) / sizeof(dictionary[0]);
}

```

```
struct TrieNode* root = getNode();

// Construct trie
for (int i = 0; i < n; i++)
    insert(root, dictionary[i]);
int min_Break = INT_MAX;

minWordBreak(root, "CatMatat", 0, &min_Break, 0);
cout << min_Break << endl;
return 0;
}
```

Java

```
// Java program to find minimum breaks needed
// to break a string in dictionary words.
public class Trie {

TrieNode root = new TrieNode();
int minWordBreak = Integer.MAX_VALUE;

// Trie node
class TrieNode {
    boolean endOfTree;
    TrieNode children[] = new TrieNode[26];
    TrieNode(){
        endOfTree = false;
        for(int i=0;i<26;i++){
            children[i]=null;
        }
    }
}

// If not present, inserts a key into the trie
// If the key is the prefix of trie node, just
// marks leaf node
void insert(String key){
    int length = key.length();

    int index;

    TrieNode pcrawl = root;

    for(int i = 0; i < length; i++)
    {
        index = key.charAt(i)- 'a';

        if(pcrawl.children[index] == null)
```

```

pcrawl.children[index] = new TrieNode();

pcrawl = pcrawl.children[index];
}

// mark last node as leaf
pcrawl.endOfTree = true;

}

// function break the string into minimum cut
// such the every substring after breaking
// in the dictionary.
void minWordBreak(String key)
{
    minWordBreak = Integer.MAX_VALUE;

    minWordBreakUtil(root, key, 0, Integer.MAX_VALUE, 0);
}

void minWordBreakUtil(TrieNode node, String key,
                      int start, int min_Break, int level)
{
    TrieNode pCrawl = node;

    // base case, update minimum Break
    if (start == key.length()) {
        min_Break = Math.min(min_Break, level - 1);
        if(min_Break<minWordBreak){
            minWordBreak = min_Break;
        }
        return;
    }

    // traverse given key(pattern)
    for (int i = start; i < key.length(); i++) {
        int index = key.charAt(i) - 'a';
        if (pCrawl.children[index]==null)
            return;

        // if we find a condition were we can
        // move to the next word in a trie
        // dictionary
        if (pCrawl.children[index].endOfTree) {
            minWordBreakUtil(root, key, i + 1,
                            min_Break, level + 1);
        }
    }
}

```

```
        pCrawl = pCrawl.children[index];
    }
}

// Driver code
public static void main(String[] args)
{
    String keys[] = {"cat", "mat", "ca", "ma",
                     "at", "c", "dog", "og", "do" };

    Trie trie = new Trie();

    // Construct trie

    int i;
    for (i = 0; i < keys.length ; i++)
        trie.insert(keys[i]);

    trie.minWordBreak("catmatat");

    System.out.println(trie.minWordBreak());
}
}

// This code is contributed by Pavan Koli.
```

Output:

2

Improved By : [pkoli](#)

Source

<https://www.geeksforgeeks.org/minimum-word-break/>

Chapter 115

Number of elements greater than K in the range L to R using Fenwick Tree (Offline queries)

Number of elements greater than K in the range L to R using Fenwick Tree (Offline queries)
- GeeksforGeeks

Prerequisites: [Fenwick Tree \(Binary Indexed Tree\)](#)

Given an array of N numbers, and a number of queries where each query will contain three numbers(l, r and k). The task is to calculate the number of array elements which are greater than K in the subarray[L, R].

Examples:

```
Input: n=6
      q=2
      arr[ ] = { 7, 3, 9, 13, 5, 4 }
      Query1: l=1, r=4, k=6
      Query2: l=2, r=6, k=8
```

```
Output: 3
      2
```

For the first query, [7, 3, 9, 13] represents the subarray from index 1 till 4, in which there are 3 numbers which are greater than k=6 that are {7, 9, 13}.

For the second query, there are only two numbers in the query range which are greater than k.

Naive Approach is to find the answer for each query by simply traversing the array from index l till r and keep adding 1 to the count whenever the array element is greater than k.
Time Complexity: $O(n*q)$

A Better Approach is to use Merge Sort Tree. In this approach, build a Segment Tree with a vector at each node containing all the elements of the sub-range in a sorted order. Answer each query using the segment tree where Binary Search can be used to calculate how many numbers are present in each node whose sub-range lies within the query range which are greater than k.

Time complexity: $O(q * \log(n) * \log(n))$

An **Efficient Approach** is to solve the problem using offline queries and [Fenwick Trees](#). Below are the steps:

- First store all the array elements and the queries in the same array. For this, we can create a self-structure or class.
- Then sort the structural array in descending order (in case of collision the query will come first then the array element).
- Process the whole array of structure again, but before that create another BIT array (Binary Indexed Tree) whose $\text{query}(i)$ function will return the count of all the elements which are present in the array till i'th index.
- Initially, fill the whole array with 0.
- Create an answer array, in which the answers of each query are stored.
- Process the array of structure.
- If it is an array element, then update the BIT array with +1 from the index of that element.
- If it is a query, then subtract the $\text{query}(r) - \text{query}(l-1)$ and this will be the answer for that query which will be stored in answer array at the index corresponding to the query number.
- Finally output the answer array.

The key observation here is that since the array of the structure has been sorted in descending order. Whenever we encounter any query only the elements which are greater than 'k' comprises the count in the BIT array which is the answer that is needed.

Below is the explanation of structure used in the program:

Pos: stores the order of query. In case of array elements it is kept as 0.

L: stores the starting index of the query's subarray. In case of array elements it is also 0.

R: stores the ending index of the query's subarray. In case of array element it is used to store the position of element in the array.

Val: store 'k' of the query and all the array elements.

Below is the implementation of the above approach:

```
// C++ program to print the number of elements
// greater than k in a subarray of range L-R.
#include <bits/stdc++.h>
using namespace std;

// Structure which will store both
// array elements and queries.
struct node {
    int pos;
    int l;
    int r;
    int val;
};

// Boolean comparator that will be used
// for sorting the structural array.
bool comp(node a, node b)
{
    // If 2 values are equal the query will
    // occur first then array element
    if (a.val == b.val)
        return a.l > b.l;

    // Otherwise sorted in descending order.
    return a.val > b.val;
}

// Updates the node of BIT array by adding
// 1 to it and its ancestors.
void update(int* BIT, int n, int idx)
{
    while (idx <= n) {
        BIT[idx]++;
        idx += idx & (-idx);
    }
}

// Returns the count of numbers of elements
// present from starting till idx.
int query(int* BIT, int idx)
```

```
{  
    int ans = 0;  
    while (idx) {  
        ans += BIT[idx];  
  
        idx -= idx & (-idx);  
    }  
    return ans;  
}  
  
// Function to solve the queries offline  
void solveQuery(int arr[], int n, int QueryL[],  
                 int QueryR[], int QueryK[], int q)  
{  
    // create node to store the elements  
    // and the queries  
    node a[n + q + 1];  
    // 1-based indexing.  
  
    // traverse for all array numbers  
    for (int i = 1; i <= n; ++i) {  
        a[i].val = arr[i - 1];  
        a[i].pos = 0;  
        a[i].l = 0;  
        a[i].r = i;  
    }  
  
    // iterate for all queries  
    for (int i = n + 1; i <= n + q; ++i) {  
        a[i].pos = i - n;  
        a[i].val = QueryK[i - n - 1];  
        a[i].l = QueryL[i - n - 1];  
        a[i].r = QueryR[i - n - 1];  
    }  
  
    // In-built sort function used to  
    // sort node array using comp function.  
    sort(a + 1, a + n + q + 1, comp);  
  
    // Binary Indexed tree with  
    // initially 0 at all places.  
    int BIT[n + 1];  
  
    // initially 0  
    memset(BIT, 0, sizeof(BIT));  
  
    // For storing answers for each query( 1-based indexing ).  
    int ans[q + 1];
```

```
// traverse for numbers and query
for (int i = 1; i <= n + q; ++i) {
    if (a[i].pos != 0) {

        // call function to returns answer for each query
        int cnt = query(BIT, a[i].r) - query(BIT, a[i].l - 1);

        // This will ensure that answer of each query
        // are stored in order it was initially asked.
        ans[a[i].pos] = cnt;
    }
    else {
        // a[i].r contains the position of the
        // element in the original array.
        update(BIT, n, a[i].r);
    }
}
// Output the answer array
for (int i = 1; i <= q; ++i) {
    cout << ans[i] << endl;
}

// Driver Code
int main()
{
    int arr[] = { 7, 3, 9, 13, 5, 4 };
    int n = sizeof(arr) / sizeof(arr[0]);

    // 1-based indexing
    int QueryL[] = { 1, 2 };
    int QueryR[] = { 4, 6 };

    // k for each query
    int QueryK[] = { 6, 8 };

    // number of queries
    int q = sizeof(QueryL) / sizeof(QueryL[0]);

    // Function call to get
    solveQuery(arr, n, QueryL, QueryR, QueryK, q);

    return 0;
}
```

Output:

3

2

Time Complexity: $O(N * \log N)$ where $N = (n+q)$

What is offline query?

In some questions, it is hard to answer queries in any random order. So instead of answering each query separately, store all the queries and then order them accordingly to calculate answer for them efficiently. Store all the answers and then output it in the order it was initially given.

This technique is called *Offline Query*.

Note: Instead of Fenwick Tree, segment tree can also be used where each node of the segment tree will store the number of elements inserted till that iteration. The update and query functions will change, rest of the implementation will remain same.

Necessary Condition For Offline Query: This technique can be used only when the answer of one query does not depend on the answers of previous queries since after sorting the order of queries may change.

Improved By : [dhruvgupta167](#)

Source

<https://www.geeksforgeeks.org/number-of-elements-greater-than-k-in-the-range-l-to-r-using-fenwick-tree-offline-queries/>

Chapter 116

Number of elements less than or equal to a given number in a given subarray

Number of elements less than or equal to a given number in a given subarray - GeeksforGeeks

Given an array ‘a[]’ and number of queries q. Each query can be represented by l, r, x. Your task is to print the number of elements less than or equal to x in the subarray represented by l to r.

Examples:

```
Input : arr[] = {2, 3, 4, 5}
        q = 2
        0 3 5
        0 2 2
Output : 4
        1
Number of elements less than or equal to
5 in arr[0..3] is 4 (all elements)

Number of elements less than or equal to
2 in arr[0..2] is 1 (only 2)
```

Naive approach The naive approach for each query traverse the subarray and count the number of elements which are in the given range.

Efficient Approach The idea is to use-[Binary Index Tree](#).

Note in the following steps x is the number according to which you have to find the elements and the subarray is represented by l, r.

Step 1: Sort the array in ascending order.

Step 2: Sort the queries according to x in ascending order, initialize bit array as 0.

Step 3: Start from the first query and traverse the array till the value in the array is less than equal to x. For each such element update the BIT with value equal to 1

Step 4: Query the BIT array in the range l to r

```
// C++ program to answer queries to count number
// of elements smaller than or equal to x.
#include<bits/stdc++.h>
using namespace std;

// structure to hold queries
struct Query
{
    int l, r, x, idx;
};

// structure to hold array
struct ArrayElement
{
    int val, idx;
};

// bool function to sort queries according to k
bool cmp1(Query q1, Query q2)
{
    return q1.x < q2.x;
}

// bool function to sort array according to its value
bool cmp2(ArrayElement x, ArrayElement y)
{
    return x.val < y.val;
}

// updating the bit array
void update(int bit[], int idx, int val, int n)
{
    for (; idx<=n; idx += idx&-idx)
        bit[idx] += val;
}

// querying the bit array
int query(int bit[], int idx, int n)
{
    int sum = 0;
    for (; idx > 0; idx -= idx&-idx)
        sum += bit[idx];
```

```
    return sum;
}

void answerQueries(int n, Query queries[], int q,
                    ArrayElement arr[])
{
    // initialising bit array
    int bit[n+1];
    memset(bit, 0, sizeof(bit));

    // sorting the array
    sort(arr, arr+n, cmp2);

    // sorting queries
    sort(queries, queries+q, cmp1);

    // current index of array
    int curr = 0;

    // array to hold answer of each Query
    int ans[q];

    // looping through each Query
    for (int i=0; i<q; i++)
    {
        // traversing the array values till it
        // is less than equal to Query number
        while (arr[curr].val <= queries[i].x && curr<n)
        {
            // updating the bit array for the array index
            update(bit, arr[curr].idx+1, 1, n);
            curr++;
        }

        // Answer for each Query will be number of
        // values less than equal to x upto r minus
        // number of values less than equal to x
        // upto l-1
        ans[queries[i].idx] = query(bit, queries[i].r+1, n) -
                                query(bit, queries[i].l, n);
    }

    // printing answer for each Query
    for (int i=0 ; i<q; i++)
        cout << ans[i] << endl;
}

// driver function
```

```
int main()
{
    // size of array
    int n = 4;

    // initialising array value and index
    ArrayElement arr[n];
    arr[0].val = 2;
    arr[0].idx = 0;
    arr[1].val = 3;
    arr[1].idx = 1;
    arr[2].val = 4;
    arr[2].idx = 2;
    arr[3].val = 5;
    arr[3].idx = 3;

    // number of queries
    int q = 2;
    Query queries[q];
    queries[0].l = 0;
    queries[0].r = 2;
    queries[0].x = 2;
    queries[0].idx = 0;
    queries[1].l = 0;
    queries[1].r = 3;
    queries[1].x = 5;
    queries[1].idx = 1;

    answerQueries(n, queries, q, arr);

    return 0;
}
```

Output:

```
1
4
```

Source

<https://www.geeksforgeeks.org/number-elements-less-equal-given-number-given-subarray/>

Chapter 117

Number of elements less than or equal to a given number in a given subarray Set 2 (Including Updates)

Number of elements less than or equal to a given number in a given subarray Set 2 (Including Updates) - GeeksforGeeks

Given an array 'a[]' and number of queries q there will be two type of queries

1. **Query 0 update(i, v)** : Two integers i and v which means set $a[i] = v$
2. **Query 1 count(l, r, k)**: We need to print number of integers less than equal to k in the subarray l to r.

Given $a[i]$, $v \leq 10000$ Examples :

```
Input : arr[] = {5, 1, 2, 3, 4}
        q = 6
        1 1 3 1 // First value 1 means type of query is count()
        0 3 10 // First value 0 means type of query is update()
        1 3 3 4
        0 2 1
        0 0 2
        1 0 4 5
Output :
1
0
```

4

For first query number of values less than equal to 1 in arr[1..3] is 1(1 only), update a[3] = 10
There is no value less than equal to 4 in the a[3..3]
and similarly other queries are answered

We have discussed a solution that handles only count() queries in below post.[Number of elements less than or equal to a given number in a given subarray](#)

Here update() query also needs to be handled.

Naive Approach The naive approach is whenever there is update operation update the array and whenever type 2 query is there traverse the subarray and count the valid elements.

Efficient Approach

The idea is to use [square root decomposition](#)

1. **Step 1 :** Divide the array in \sqrt{n} equal sized blocks. For each block keep a binary index tree of size equal to 1 more than the maximum possible element in the array of the elements in that block.
2. Step 2: For each element of the array find out the block to which it belongs and update the bit array of that block with the value 1 at $arr[i]$.
3. Step 3: Whenever there is a update query, update the bit array of the corresponding block at the original value of the array at that index with value equal to -1 and update the bit array of the same block with value 1 at the new value of the array at that index.
4. Step 4: For type 2 query you can make a single query to the BIT (to count elements less than or equal to k) for each complete block in the range, and for the two partial blocks on the end, just loop through the elements.

C++

```
// Number of elements less than or equal to a given
// number in a given subarray and allowing update
// operations.
#include<bits/stdc++.h>
using namespace std;

const int MAX = 10001;

// updating the bit array of a valid block
void update(int idx, int blk, int val, int bit[][] [MAX])
{
    for (; idx<MAX; idx += (idx&-idx))
        bit[blk][idx] += val;
}
```

```

// answering the query
int query(int l, int r, int k, int arr[], int blk_sz,
          int bit[][] [MAX])
{
    // traversing the first block in range
    int sum = 0;
    while (l < r && l % blk_sz != 0 && l != 0)
    {
        if (arr[l] <= k)
            sum++;
        l++;
    }

    // Traversing completely overlapped blocks in
    // range for such blocks bit array of that block
    // is queried
    while (l + blk_sz <= r)
    {
        int idx = k;
        for (; idx > 0 ; idx -= idx&-idx)
            sum += bit[l/bk_sz][idx];
        l += blk_sz;
    }

    // Traversing the last block
    while (l <= r)
    {
        if (arr[l] <= k)
            sum++;
        l++;
    }
    return sum;
}

// Preprocessing the array
void preprocess(int arr[], int blk_sz, int n, int bit[][] [MAX])
{
    for (int i=0; i<n; i++)
        update(arr[i], i/bk_sz, 1, bit);
}

void preprocessUpdate(int i, int v, int blk_sz,
                      int arr[], int bit[][] [MAX])
{
    // updating the bit array at the original
    // and new value of array
    update(arr[i], i/bk_sz, -1, bit);
    update(v, i/bk_sz, 1, bit);
}

```

```

        arr[i] = v;
    }

// driver function
int main()
{
    int arr[] = {5, 1, 2, 3, 4};
    int n = sizeof(arr)/sizeof(arr[0]);

    // size of block size will be equal to square root of n
    int blk_sz = sqrt(n);

    // initialising bit array of each block
    // as elements of array cannot exceed 10^4 so size
    // of bit array is accordingly
    int bit[blk_sz+1][MAX];
    memset(bit, 0, sizeof(bit));

    preprocess(arr, blk_sz, n, bit);
    cout << query(1, 3, 1, arr, blk_sz, bit) << endl;

    preprocessUpdate(3, 10, blk_sz, arr, bit);
    cout << query(3, 3, 4, arr, blk_sz, bit) << endl;
    preprocessUpdate(2, 1, blk_sz, arr, bit);
    preprocessUpdate(0, 2, blk_sz, arr, bit);
    cout << query(0, 4, 5, arr, blk_sz, bit) << endl;
    return 0;
}

```

Java

```

// Number of elements less than or equal to a given
// number in a given subarray and allowing update
// operations.

class Test
{
    static final int MAX = 10001;

    // updating the bit array of a valid block
    static void update(int idx, int blk, int val, int bit[][])
    {
        for (; idx<MAX; idx += (idx&-idx))
            bit[blk][idx] += val;
    }

    // answering the query
    static int query(int l, int r, int k, int arr[], int blk_sz,

```

```

int bit[][])
{
    // traversing the first block in range
    int sum = 0;
    while (l < r && l%blk_sz!=0 && l!=0)
    {
        if (arr[l] <= k)
            sum++;
        l++;
    }

    // Traversing completely overlapped blocks in
    // range for such blocks bit array of that block
    // is queried
    while (l + blk_sz <= r)
    {
        int idx = k;
        for (; idx > 0 ; idx -= idx&-idx)
            sum += bit[l/bk_sz][idx];
        l += blk_sz;
    }

    // Traversing the last block
    while (l <= r)
    {
        if (arr[l] <= k)
            sum++;
        l++;
    }
    return sum;
}

// Preprocessing the array
static void preprocess(int arr[], int blk_sz, int n, int bit[][])
{
    for (int i=0; i<n; i++)
        update(arr[i], i/bk_sz, 1, bit);
}

static void preprocessUpdate(int i, int v, int blk_sz,
                           int arr[], int bit[][])
{
    // updating the bit array at the original
    // and new value of array
    update(arr[i], i/bk_sz, -1, bit);
    update(v, i/bk_sz, 1, bit);
    arr[i] = v;
}

```

```

// Driver method
public static void main(String args[])
{
    int arr[] = {5, 1, 2, 3, 4};

    // size of block size will be equal to square root of n
    int blk_sz = (int) Math.sqrt(arr.length);

    // initialising bit array of each block
    // as elements of array cannot exceed 10^4 so size
    // of bit array is accordingly
    int bit[][] = new int[blk_sz+1][MAX];

    preprocess(arr, blk_sz, arr.length, bit);
    System.out.println(query(1, 3, 1, arr, blk_sz, bit));

    preprocessUpdate(3, 10, blk_sz, arr, bit);
    System.out.println(query(3, 3, 4, arr, blk_sz, bit));
    preprocessUpdate(2, 1, blk_sz, arr, bit);
    preprocessUpdate(0, 2, blk_sz, arr, bit);
    System.out.println(query(0, 4, 5, arr, blk_sz, bit));
}
}

```

C#

```

// Number of elements less than or equal
// to a given number in a given subarray
// and allowing update operations.
using System;

class GFG
{
    static int MAX = 10001;

    // updating the bit array of a valid block
    static void update(int idx, int blk,
                       int val, int [,]bit)
    {
        for (; idx < MAX; idx += (idx& - idx))
            bit[blk, idx] += val;
    }

    // answering the query
    static int query(int l, int r, int k, int []arr,
                    int blk_sz, int [,]bit)
    {

```

```
// traversing the first block in range
int sum = 0;
while (l < r && l % blk_sz != 0 && l != 0)
{
    if (arr[l] <= k)
        sum++;
    l++;
}

// Traversing completely overlapped blocks in
// range for such blocks bit array of that block
// is queried
while (l + blk_sz <= r)
{
    int idx = k;
    for (; idx > 0 ; idx -= idx&-idx)
        sum += bit[l/bk_sz, idx];
    l += blk_sz;
}

// Traversing the last block
while (l <= r)
{
    if (arr[l] <= k)
        sum++;
    l++;
}
return sum;
}

// Preprocessing the array
static void preprocess(int []arr, int blk_sz,
                      int n, int [,]bit)
{
    for (int i=0; i<n; i++)
        update(arr[i], i / blk_sz, 1, bit);
}

static void preprocessUpdate(int i, int v, int blk_sz,
                           int []arr, int [,]bit)
{
    // updating the bit array at the original
    // and new value of array
    update(arr[i], i/bk_sz, -1, bit);
    update(v, i/bk_sz, 1, bit);
    arr[i] = v;
}
```

```
// Driver method
public static void Main()
{
    int []arr = {5, 1, 2, 3, 4};

    // size of block size will be
    // equal to square root of n
    int blk_sz = (int) Math.Sqrt(arr.Length);

    // initialising bit array of each block
    // as elements of array cannot exceed 10^4 so size
    // of bit array is accordingly
    int [,]bit = new int[blk_sz+1,MAX];

    preprocess(arr, blk_sz, arr.Length, bit);
    Console.WriteLine(query(1, 3, 1, arr, blk_sz, bit));

    preprocessUpdate(3, 10, blk_sz, arr, bit);
    Console.WriteLine(query(3, 3, 4, arr, blk_sz, bit));
    preprocessUpdate(2, 1, blk_sz, arr, bit);
    preprocessUpdate(0, 2, blk_sz, arr, bit);
    Console.WriteLine(query (0, 4, 5, arr, blk_sz, bit));
}
}

// This code is contributed by Sam007
```

Output:

```
1
0
4
```

The question is know why not to use this method when there were no update operations the answer lies in space complexity in this method 2-d bit array is used as well as its size depends upon the maximum possible value of the array but when there was no update operation our bit array was only dependent on the size of array.

Improved By : [Sam007](#)

Source

<https://www.geeksforgeeks.org/number-elements-less-equal-given-number-given-subarray-set-2-including-updates/>

Chapter 118

Number of primes in a subarray (with updates)

Number of primes in a subarray (with updates) - GeeksforGeeks

Given an array of N integers, the task is to perform the following two queries:

query(start, end) : Print the number of prime numbers in the subarray from start to end
update(i, x) : update the value at index i to x, i.e arr[i] = x

Examples:

```
Input : arr = {1, 2, 3, 5, 7, 9}
        Query 1: query(start = 0, end = 4)
        Query 2: update(i = 3, x = 6)
        Query 3: query(start = 0, end = 4)
Output :4
         3
```

Explanation

In Query 1, the subarray [0...4]
has 4 primes viz. {2, 3, 5, 7}

In Query 2, the value at index 3
is updated to 6, the array arr now is, {1, 2, 3,
6, 7, 9}
In Query 3, the subarray [0...4]
has 4 primes viz. {2, 3, 7}

Method 1 (Brute Force)

A similar problem can be found [here](#). Here there are no updates. We can modify this to

handle updates but for this we need to build the prefix array always when we perform an update which makes the time complexity of this approach $O(Q * N)$

Method 2 (Efficient)

Since, we need to handle both range queries and point updates, a segment tree is best suited for this purpose.

We can use [Sieve of Eratosthenes](#) to preprocess all the primes till the maximum value arr_i can take say MAX in $O(\text{MAX} \log(\log(\text{MAX})))$

Building the segment tree:

We basically reduce the problem to [subarray sum using segment tree](#).

Now, we can build the segment tree where a leaf node is represented as either 0 (if it is not a prime number) or 1 (if it is a prime number).

The internal nodes of the segment tree equal to the sum of its child nodes, thus a node represents the total primes in the range from L to R where the range L to R falls under this node and the sub-tree below it.

Handling Queries and Point Updates:

Whenever we get a query from start to end, then we can query the segment tree for the sum of nodes in range start to end, which in turn represent the number of primes in range start to end.

If we need to perform a point update and update the value at index i to x, then we check for the following cases:

Let the old value of arr_i be y and the new value be x

Case 1: If x and y both are primes

Count of primes in the subarray does not change so we just update array and do not modify the segment tree

Case 2: If x and y both are non primes

Count of primes in the subarray does not change so we just update array and do not modify the segment tree

Case 3: If y is prime but x is non prime

Count of primes in the subarray decreases so we update array and add -1 to every range, the index i which is to be updated, is a part of in the segment tree

Case 4: If y is non prime but x is prime

Count of primes in the subarray increases so we update array and add 1 to every range, the index i which is to be updated, is a part of in the segment tree

```
// C++ program to find number of prime numbers in a
// subarray and performing updates
#include <bits/stdc++.h>
using namespace std;
```

```

#define MAX 1000

void sieveOfEratosthenes(bool isPrime[])
{
    isPrime[1] = false;

    for (int p = 2; p * p <= MAX; p++) {

        // If prime[p] is not changed, then
        // it is a prime
        if (isPrime[p] == true) {

            // Update all multiples of p
            for (int i = p * 2; i <= MAX; i += p)
                isPrime[i] = false;
        }
    }
}

// A utility function to get the middle index from corner indexes.
int getMid(int s, int e) { return s + (e - s) / 2; }

/* A recursive function to get the number of primes in a given range
   of array indexes. The following are parameters for this function.

   st    --> Pointer to segment tree
   index --> Index of current node in the segment tree. Initially
              0 is passed as root is always at index 0
   ss & se  --> Starting and ending indexes of the segment represented
                 by current node, i.e., st[index]
   qs & qe  --> Starting and ending indexes of query range */
int queryPrimesUtil(int* st, int ss, int se, int qs, int qe, int index)
{
    // If segment of this node is a part of given range, then return
    // the number of primes in the segment
    if (qs <= ss && qe >= se)
        return st[index];

    // If segment of this node is outside the given range
    if (se < qs || ss > qe)
        return 0;

    // If a part of this segment overlaps with the given range
    int mid = getMid(ss, se);
    return queryPrimesUtil(st, ss, mid, qs, qe, 2 * index + 1) +
           queryPrimesUtil(st, mid + 1, se, qs, qe, 2 * index + 2);
}

```

```

/* A recursive function to update the nodes which have the given
   index in their range. The following are parameters
   st, si, ss and se are same as getSumUtil()
   i    --> index of the element to be updated. This index is
           in input array.
   diff --> Value to be added to all nodes which have i in range */
void updateValueUtil(int* st, int ss, int se, int i, int diff, int si)
{
    // Base Case: If the input index lies outside the range of
    // this segment
    if (i < ss || i > se)
        return;

    // If the input index is in range of this node, then update
    // the value of the node and its children
    st[si] = st[si] + diff;
    if (se != ss) {
        int mid = getMid(ss, se);
        updateValueUtil(st, ss, mid, i, diff, 2 * si + 1);
        updateValueUtil(st, mid + 1, se, i, diff, 2 * si + 2);
    }
}

// The function to update a value in input array and segment tree.
// It uses updateValueUtil() to update the value in segment tree
void updateValue(int arr[], int* st, int n, int i, int new_val,
                bool isPrime[])
{
    // Check for erroneous input index
    if (i < 0 || i > n - 1) {
        printf("Invalid Input");
        return;
    }

    int diff, oldValue;

    oldValue = arr[i];

    // Update the value in array
    arr[i] = new_val;

    // Case 1: Old and new values both are primes
    if (isPrime[oldValue] && isPrime[new_val])
        return;

    // Case 2: Old and new values both non primes
    if ((!isPrime[oldValue]) && (!isPrime[new_val]))
        return;
}

```

```

// Case 3: Old value was prime, new value is non prime
if (isPrime[oldValue] && !isPrime[new_val]) {
    diff = -1;
}

// Case 4: Old value was non prime, new_val is prime
if (!isPrime[oldValue] && isPrime[new_val]) {
    diff = 1;
}

// Update the values of nodes in segment tree
updateValueUtil(st, 0, n - 1, i, diff, 0);
}

// Return number of primes in range from index qs (query start) to
//qe (query end). It mainly uses queryPrimesUtil()
void queryPrimes(int* st, int n, int qs, int qe)
{
    int primesInRange = queryPrimesUtil(st, 0, n - 1, qs, qe, 0);

    cout << "Number of Primes in subarray from " << qs << " to "
        << qe << " = " << primesInRange << "\n";
}

// A recursive function that constructs Segment Tree
// for array[ss..se].
// si is index of current node in segment tree st
int constructSTUtil(int arr[], int ss, int se, int* st,
                    int si, bool isPrime[])
{
    // If there is one element in array, check if it
    // is prime then store 1 in the segment tree else
    // store 0 and return
    if (ss == se) {

        // if arr[ss] is prime
        if (isPrime[arr[ss]])
            st[si] = 1;
        else
            st[si] = 0;

        return st[si];
    }

    // If there are more than one elements, then recur
    // for left and right subtrees and store the sum
    // of the two values in this node
}

```

```
int mid = getMid(ss, se);
st[si] = constructSTUtil(arr, ss, mid, st,
                         si * 2 + 1, isPrime) +
constructSTUtil(arr, mid + 1, se, st,
                         si * 2 + 2, isPrime);
return st[si];
}

/* Function to construct segment tree from given array.
   This function allocates memory for segment tree and
   calls constructSTUtil() to fill the allocated memory */
int* constructST(int arr[], int n, bool isPrime[])
{
    // Allocate memory for segment tree

    // Height of segment tree
    int x = (int)(ceil(log2(n)));

    // Maximum size of segment tree
    int max_size = 2 * (int)pow(2, x) - 1;

    int* st = new int[max_size];

    // Fill the allocated memory st
    constructSTUtil(arr, 0, n - 1, st, 0, isPrime);

    // Return the constructed segment tree
    return st;
}

// Driver program to test above functions
int main()
{

    int arr[] = { 1, 2, 3, 5, 7, 9 };
    int n = sizeof(arr) / sizeof(arr[0]);

    /* Preprocess all primes till MAX.
       Create a boolean array "isPrime[0..MAX]".
       A value in prime[i] will finally be false
       if i is Not a prime, else true. */

    bool isPrime[MAX + 1];
    memset(isPrime, true, sizeof isPrime);
    sieveOfEratosthenes(isPrime);

    // Build segment tree from given array
    int* st = constructST(arr, n, isPrime);
```

```
// Query 1: Query(start = 0, end = 4)
int start = 0;
int end = 4;
queryPrimes(st, n, start, end);

// Query 2: Update(i = 3, x = 6), i.e Update
// a[i] to x
int i = 3;
int x = 6;
updateValue(arr, st, n, i, x, isPrime);

// uncomment to see array after update
// for(int i = 0; i < n; i++) cout << arr[i] << " ";

// Query 3: Query(start = 0, end = 4)
start = 0;
end = 4;
queryPrimes(st, n, start, end);

return 0;
}
```

Output:

```
Number of Primes in subarray from 0 to 4 = 4
Number of Primes in subarray from 0 to 4 = 3
```

The time complexity of each query and update is $O(\log n)$ and that of building the segment tree is $O(n)$

Note: Here, the time complexity of pre-processing primes till MAX using the sieve of Eratosthenes is $O(\text{MAX} \log(\log(\text{MAX})))$ where MAX is the maximum value arr_i can take

Source

<https://www.geeksforgeeks.org/number-primes-subarray-updates/>

Chapter 119

Order statistic tree using fenwick tree (BIT)

Order statistic tree using fenwick tree (BIT) - GeeksforGeeks

Given an array of integers with limited range (0 to 1000000). We need to implement an Order statistic tree using fenwick tree.

It should support four operations: Insert, Delete, Select and Rank. Here n denotes the size of Fenwick tree and q denotes number of queries.

Each query should be one of the following 4 operations.

- `insertElement(x)` – Insert element x into Fenwick tree, with $O(\log n)$ worst case time complexity
- `deleteElement(x)` – Delete element x from fenwick tree, with $O(\log n)$ worse case time complexity
- `findKthSmallest(k)` – Find the k-th smallest element stored in the tree, with $O(\log n * \log n)$ worst case time complexity
- `findRank(x)` – Find the rank of element x in the tree, i.e. its index in the sorted list of elements of the tree, with $O(\log n)$ time complexity

Prerequisite : [Binary Indexed Tree or Fenwick Tree](#)

The idea is to create a BIT of size with maximum limit. We insert an element in BIT using it as an index. When we insert an element x, we increment values of all ancestors of x by 1. To delete an element, we decrement values of ancestors by 1. We basically call standard function `update()` of BIT for both insert and delete. To find rank, we simply call standard function `sum()` of BIT. To find k-th smallest element, we do binary search in BIT.

```
// C++ program to find rank of an element  
// and k-th smallest element.
```

```
#include <bits/stdc++.h>
using namespace std;

const int MAX_VAL = 1000001;

/* Updates element at index 'i' of BIT. */
void update(int i, int add, vector<int>& BIT)
{
    while (i > 0 && i < BIT.size())
    {
        BIT[i] += add;
        i = i + (i & (-i));
    }
}

/* Returns cumulative sum of all elements of
fenwick tree/BIT from start upto and
including element at index 'i'. */
int sum(int i, vector<int>& BIT)
{
    int ans = 0;
    while (i > 0)
    {
        ans += BIT[i];
        i = i - (i & (-i));
    }

    return ans;
}

// Returns lower bound for k in BIT.
int findKthSmallest(int k, vector<int> &BIT)
{
    // Do binary search in BIT[] for given
    // value k.
    int l = 0;
    int h = BIT.size();
    while (l < h)
    {
        int mid = (l + h) / 2;
        if (k <= sum(mid, BIT))
            h = mid;
        else
            l = mid+1;
    }

    return l;
}
```

```
// Insert x into BIT. We basically increment
// rank of all elements greater than x.
void insertElement(int x, vector<int> &BIT)
{
    update(x, 1, BIT);
}

// Delete x from BIT. We basically decreases
// rank of all elements greater than x.
void deleteElement(int x, vector<int> &BIT)
{
    update(x, -1, BIT);
}

// Returns rank of element. We basically
// return sum of elements from start to
// index x.
int findRank(int x, vector<int> &BIT)
{
    return sum(x, BIT);
}

// Driver code
int main()
{
    vector<int> BIT(MAX_VAL);
    insertElement(20, BIT);
    insertElement(50, BIT);
    insertElement(30, BIT);
    insertElement(40, BIT);

    cout << "2nd Smallest element is "
        << findKthSmallest(2, BIT) << endl;

    cout << "Rank of 40 is "
        << findRank(40, BIT) << endl;

    deleteElement(40, BIT);

    cout << "Rank of 50 is "
        << findRank(50, BIT) << endl;

    return 0;
}
```

Output:

2nd Smallest element is 30

Rank of 40 is 3

Rank of 50 is 3

Source

<https://www.geeksforgeeks.org/order-statistic-tree-using-fenwick-tree-bit/>

Chapter 120

Ordered Set and GNU C++ PBDS

Ordered Set and GNU C++ PBDS - GeeksforGeeks

Prerequisite :Basic knowledge of [STL](#) and [Sets Data structure](#).

About ordered set

Ordered set is a [policy based data structure in g++](#) that keeps the **unique** elements in sorted order. It performs all the operations as performed by the set data structure in STL in $\log(n)$ complexity and performs two additional operations also in $\log(n)$ complexity .

- **order_of_key (k)** : Number of items strictly smaller than k .
- **find_by_order(k)** : K-th element in a set (counting from zero).

Required header files to implement ordered set and their description

For implementing ordered_set and GNU C++ library contains other Policy based data structures we need to include :

- // Common file
include <ext/pb_ds/assoc_container.hpp>
- // Including tree_order_statistics_node_update
include <ext/pb_ds/tree_policy.hpp>

The first one is used to include the associative containers or group of templates such as **set**, **multimap**, **map** etc. The tree-based data structures which we will be using below is present in this header file.

The second header file is used to include the *tree_order_statistics_node update* which is explained below:

```
using namespace __gnu_pbds;
```

It is a namespace necessary for the **GNU based Policy based data structures**.

The tree based container has a concrete structure but the necessary structure required for the ordered set implementation is :

```
tree < int , null_type , less , rb_tree_tag , tree_order_statistics_node_update >
```

1. **int** : It is the type of the data that we want to insert (KEY).It can be integer, float or pair of int etc.
2. **null_type** : It is the mapped policy. It is null here to use it as a set.If we want to get map but not the set, as the second argument type must be used mapped type.
3. **less** : It is the basis for comparison of two functions.
4. **rb_tree_tag** : type of tree used. It is generally Red black trees because it takes $\log(n)$ time for insertion and deletion while other take linear time such as splay_tree.
5. **tree_order_statistics_node_update** : It is included in tree_policy.hpp and contains various operations for updating the node variants of a tree-based container, so we can keep track of metadata like the number of nodes in a subtree

Additional functions in the ordered set other than the set

Along with the previous operations of the set, it supports *two* main important operations

- **find_by_order(k)**: It returns to an iterator to the kth element (**counting from zero**) in the set in $O(\log n)$ time.To find the first element k must be zero.

Let us assume we have a set s : {1, 5, 6, 17, 88}, then :

```
*(s.find_by_order(2)) : 3rd element in the set i.e. 6  
*(s.find_by_order(4)) : 5th element in the set i.e. 88
```

- **order_of_key(k)** : It returns to the number of items that are **strictly** smaller than our item k in $O(\log n)$ time.

Let us assume we have a set s : {1, 5, 6, 17, 88}, then :

```
s.order_of_key(6) : Count of elements strictly smaller than 6 is 2.  
s.order_of_key(25) : Count of elements strictly smaller than 25 is 4.
```

Difference between set and ordered set

There is not so much difference between the set and ordered set although ordered set can be assumed as an extended version of set capable of performing some more advanced functions(stated above) that are extensively used in competitive programming.

NOTE : **ordered_set** is used here as a macro given to `tree<int, null_type, less, rb_tree_tag, tree_order_statistics_node_update>`. Therefore it can be given any name as

macro other than `ordered_set` but generally in the world of competitive programming it is commonly referred as ordered set as it is a set with additional operations.

Practical applications:

Suppose we have a situation where the elements are inserted one by one in an array and after each insertion, we are given a range $[l, r]$ and we have to determine the number of elements in the array greater than equal to l and less than equal to r . Initially, the array is empty.

Examples:

```
Input :    5
          1 2
          1
          2 5
          2
          1 5

Output :   0
          1
          3
```

Explanation:

- 5 is inserted.
- Count of elements greater than equal to 1 and less than equal to 2 is 0.
- 1 is inserted.
- Count of elements greater than equal to 2 and less than equal to 5 is 1 i.e. 5.
- 2 is inserted.
- Count of elements greater than equal to 1 and less than equal to 5 is 3 i.e. 5, 1, 2.

```
Input :    1
          1 2
          2
          3 5
          5
          1 4
Output :   1
          0
          2
```

- 1 is inserted.
- Count of elements greater than equal to 1 and less than equal to 2 is 1 i.e 1.

- 2 is inserted.
- Count of elements greater than equal to 3 and less than equal to 5 is 0.
- 5 is inserted.
- Count of elements greater than equal to 1 and less than equal to 4 is 2 i.e. 1, 2.

2
2
2
5
1

Thus we can now solve the above problem easily i.e. count of elements between l and r can be found by:

`o_set.order_of_key(r+1) - o_set.order_of_key(l)`

NOTE : As the set contains only the **UNIQUE** elements, so to perform the operations on an array having repeated elements we can take the KEY as a pair of elements instead of integer in which the first element is our required element of the array and only the second element of the pair must be unique so that the whole pair is unique.

For more details refer to :

https://gcc.gnu.org/onlinedocs/libstdc++/manual/policy_data_structures.html

Source

<https://www.geeksforgeeks.org/ordered-set-gnu-c-pbds/>

Chapter 121

Overview of Data Structures Set 3 (Graph, Trie, Segment Tree and Suffix Tree)

Overview of Data Structures Set 3 (Graph, Trie, Segment Tree and Suffix Tree) - Geeks-forGeeks

We have discussed below data structures in previous two sets.

[Set 1 : Overview of Array, Linked List, Queue and Stack.](#)
[Set 2 : Overview of Binary Tree, BST, Heap and Hash.](#)

9. Graph

10. Trie

11. Segment Tree

12. Suffix Tree

Graph

Graph is a data structure that consists of following two components:

1. A finite set of vertices also called as nodes.
2. A finite set of ordered pair of the form (u, v) called as edge. The pair is ordered because (u, v) is not same as (v, u) in case of directed graph(di-graph). The pair of form (u, v) indicates that there is an edge from vertex u to vertex v . The edges may contain weight/value/cost.

$V \rightarrow$ Number of Vertices.

$E \rightarrow$ Number of Edges.

Graph can be classified on the basis of many things, below are the two most common classifications :

1. Direction :

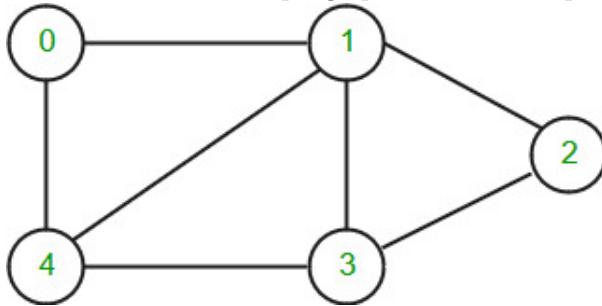
Undirected Graph : The graph in which all the edges are bidirectional.
Directed Graph : The graph in which all the edges are unidirectional.

2. Weight :

Weighted Graph : The Graph in which weight is associated with the edges.
Unweighted Graph : The Graph in which there is no weight associated to the edges.

Graph can be represented in many ways, below are the two most common representations :

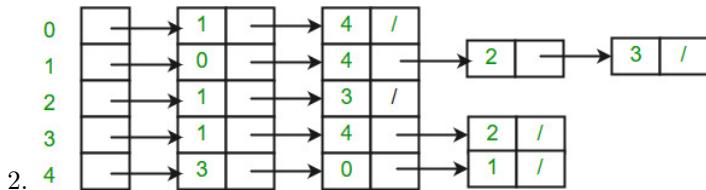
Let us take below example graph to see two representations of graph.



	0	1	2	3	4
0	0	1	0	0	1
1	1	0	1	1	1
2	0	1	0	1	0
3	0	1	1	0	1
4	1	1	0	1	0

1.

Adjacency Matrix Representation of the above graph



Adjacency List Representation of the above Graph

Time Complexities in case of Adjacency Matrix :

Traversal : (By BFS or DFS) $O(V^2)$
Space : $O(V^2)$

Time Complexities in case of Adjacency List :
Traversal : (By BFS or DFS) $O(E \log V)$
Space : $O(V+E)$

Examples : The most common example of the graph is to find shortest path in any network. Used in google maps or bing. Another common use application of graph are social networking websites where the friend suggestion depends on number of intermediate suggestions and other things.

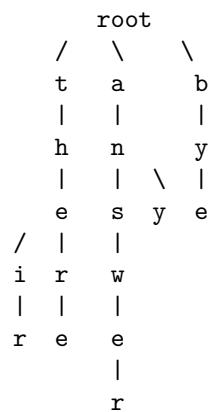
Trie

Trie is an efficient data structure for searching words in dictionaries, search complexity with Trie is linear in terms of word (or key) length to be searched. If we store keys in binary search tree, a well balanced BST will need time proportional to $M * \log N$, where M is maximum string length and N is number of keys in tree. Using trie, we can search the key in $O(M)$ time. So it is much faster than BST.

Hashing also provides word search in $O(n)$ time on average. But the advantages of Trie are there are no collisions (like hashing) so worst case time complexity is $O(n)$. Also, the most important thing is Prefix Search. With Trie, we can find all words beginning with a prefix (This is not possible with Hashing). The only problem with Tries is they require a lot of extra space. Tries are also known as radix tree or prefix tree.

The Trie structure can be defined as follows :

```
struct trie_node
{
    int value; /* Used to mark leaf nodes */
    trie_node_t *children[ALPHABET_SIZE];
};
```



The leaf nodes are in blue.

Insert time : $O(M)$ where M is the length of the string.

Search time : $O(M)$ where M is the length of the string.

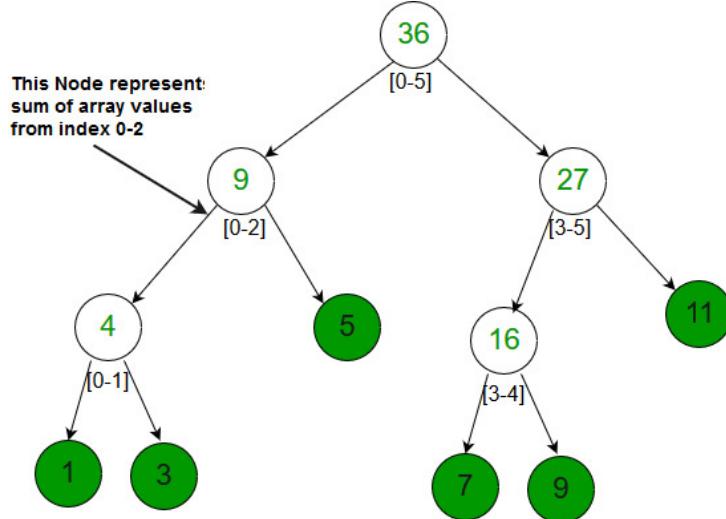
Space : $O(ALPHABET_SIZE * M * N)$ where N is number of keys in trie, $ALPHABET_SIZE$ is 26 if we are only considering upper case Latin characters.

Deletion time : $O(M)$

Example : The most common use of Tries is to implement dictionaries due to prefix search capability. Tries are also well suited for implementing approximate matching algorithms, including those used in spell checking. It is also used for searching Contact from Mobile Contact list OR Phone Directory.

Segment Tree

This data structure is usually implemented when there are a lot of queries on a set of values. These queries involve minimum, maximum, sum, .. etc on a input range of given set. Queries also involve updation of values in given set. Segment Trees are implemented using array.



Construction of segment tree : $O(N)$

Query : $O(\log N)$

Update : $O(\log N)$

Space : $O(N)$ [Exact space = $2N-1$]

Example : It is used when we need to find Maximum/Minimum/Sum/Product of numbers in a range.

Suffix Tree

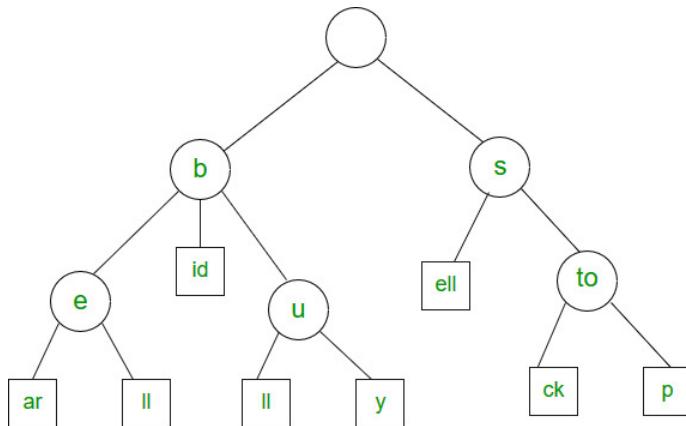
Suffix Tree is mainly used to search a pattern in a text. The idea is to preprocess the text so that search operation can be done in time linear in terms of pattern length. The pattern

searching algorithms like KMP, Z, etc take time proportional to text length. This is really a great improvement because length of pattern is generally much smaller than text.

Imagine we have stored complete work of William Shakespeare and preprocessed it. You can search any string in the complete work in time just proportional to length of the pattern. But using Suffix Tree may not be a good idea when text changes frequently like text editor, etc.

Suffix Tree is compressed trie of all suffixes, so following are very abstract steps to build a suffix tree from given text.

- 1) Generate all suffixes of given text.
- 2) Consider all suffixes as individual words and build a compressed trie.



Example : Used to find all occurrences of the pattern in string. It is also used to find the longest repeated substring (when text doesn't change often), the longest common substring and the longest palindrome in a string.

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

Improved By : [HenrySpivey](#)

Source

<https://www.geeksforgeeks.org/overview-of-data-structures-set-3-graph-trie-segment-tree-and-suffix-tree/>

Chapter 122

Palindrome pair in an array of words (or strings)

Palindrome pair in an array of words (or strings) - GeeksforGeeks

Given a list of words, find if any of the two words can be joined to form a palindrome.

Examples:

```
Input : list[] = {"geekf", "geeks", "or",
                  "keeg", "abc", "bc"}
```

Output : Yes

There is a pair "geekf" and "keeg"

```
Input : list[] = {"abc", "xyxcba", "geekst", "or",
                  "keeg", "bc"}
```

Output : Yes

There is a pair "abc" and "xyxcba"

Asked in : Google Interview

Simple approach:

- 1- Consider each pair one by one.
- 2- Check if any of the pairs forms a palindrome after concatenating them.
- 3- Return true, if any such pair exists.
- 4- Else, return false.

C++

```
// C++ program to find if there is a pair that
// can form a palindrome.
#include<bits/stdc++.h>
using namespace std;

// Utility function to check if a string is a
// palindrome
bool isPalindrome(string str)
{
    int len = str.length();

    // compare each character from starting
    // with its corresponding character from last
    for (int i = 0; i < len/2; i++)
        if (str[i] != str[len-i-1])
            return false;

    return true;
}

// Function to check if a palindrome pair exists
bool checkPalindromePair(vector <string> vect)
{
    // Consider each pair one by one
    for (int i = 0; i < vect.size()-1; i++)
    {
        for (int j = i+1; j < vect.size() ; j++)
        {
            string check_str = "";

            // concatenate both strings
            check_str = check_str + vect[i] + vect[j];

            // check if the concatenated string is
            // palindrome
            if (isPalindrome(check_str))
                return true;
        }
    }
    return false;
}

// Driver code
int main()
{
    vector <string> vect = {"geekf", "geeks", "or",
                           "keeg", "abc", "bc"};
}
```

```
checkPalindromePair(vect)? cout << "Yes" :
                           cout << "No";
return 0;
}
```

Java

```
// Java program to find if there is a pair that
// can form a palindrome.
import java.util.Arrays;
import java.util.List;
public class Palin_pair1 {

    // Utility function to check if a string is a
    // palindrome
    static boolean isPalindrome(String str)
    {
        int len = str.length();

        // compare each character from starting
        // with its corresponding character from last
        for (int i = 0; i < len/2; i++)
            if (str.charAt(i) != str.charAt(len-i-1))
                return false;

        return true;
    }

    // Function to check if a palindrome pair exists
    static boolean checkPalindromePair(List<String> vect)
    {
        // Consider each pair one by one
        for (int i = 0; i < vect.size()-1; i++)
        {
            for (int j = i+1; j < vect.size(); j++)
            {
                String check_str = "";

                // concatenate both strings
                check_str = check_str + vect.get(i) + vect.get(j);

                // check if the concatenated string is
                // palindrome
                if (isPalindrome(check_str))
                    return true;
            }
        }
    }
}
```

```
        return false;
    }

// Driver code
public static void main(String args[])
{
    List<String> vect = Arrays.asList("geekf", "geeks", "or",
                                      "keeg", "abc", "bc");

    if (checkPalindromePair(vect) == true)
        System.out.println("Yes");
    else
        System.out.println("No");
}
//This code is contributed by Sumit Ghosh
```

Output:

Yes

Time Complexity : $O(n^2k)$

Here n is the number of the words in the list and k is the maximum length that is checked for a palindrome.

Efficient method

It can be done in an efficient manner by using the Trie data structure. The idea is to maintain a Trie of the reverse of all words.

- 1) Create an empty Trie.
- 2) Do following for every word:-
 - a) Insert reverse of current word.
 - b) Also store up to which index it is a palindrome.
- 3) Traverse list of words again and do following for every word.
 - a) If it is available in Trie then return true
 - b) If it is partially available
Check the remaining word is palindrome or not
If yes then return true that means a pair forms a palindrome.
Note: Position upto which the word is palindrome is stored because of these type of cases.

C++

```
// C++ program to check if there is a pair that
// of above method using Trie
#include<bits/stdc++.h>
using namespace std;
#define ARRAY_SIZE(a) sizeof(a)/sizeof(a[0])

// Alphabet size (# of symbols)
#define ALPHABET_SIZE (26)

// Converts key current character into index
// use only 'a' through 'z' and lower case
#define CHAR_TO_INDEX(c) ((int)c - (int)'a')

// Trie node
struct TrieNode
{
    struct TrieNode *children[ALPHABET_SIZE];
    vector<int> pos; // To store palindromic
                      // positions in str
    int id;

    // isLeaf is true if the node represents
    // end of a word
    bool isLeaf;
};

// Returns new Trie node (initialized to NULLs)
struct TrieNode *getNode(void)
{
    struct TrieNode *pNode = new TrieNode;
    pNode->isLeaf = false;
    for (int i = 0; i < ALPHABET_SIZE; i++)
        pNode->children[i] = NULL;

    return pNode;
}

// Utility function to check if a string is a
// palindrome
bool isPalindrome(string str, int i, int len)
{
    // compare each character from starting
    // with its corresponding character from last
    while (i < len)
    {
        if (str[i] != str[len])

```

```
        return false;
    i++, len--;
}

return true;
}

// If not present, inserts reverse of key into Trie. If
// the key is prefix of a Trie node, just mark leaf node
void insert(struct TrieNode* root, string key, int id)
{
    struct TrieNode *pCrawl = root;

    // Start traversing word from the last
    for (int level = key.length()-1; level >=0; level--)
    {
        // If it is not available in Trie, then
        // store it
        int index = CHAR_TO_INDEX(key[level]);
        if (!pCrawl->children[index])
            pCrawl->children[index] = getNode();

        // If current word is palindrome till this
        // level, store index of current word.
        if (isPalindrome(key, 0, level))
            (pCrawl->pos).push_back(id);

        pCrawl = pCrawl->children[index];
    }
    pCrawl->id = id;
    pCrawl->pos.push_back(id);

    // mark last node as leaf
    pCrawl->isLeaf = true;
}

// Returns true if key presents in Trie, else false
void search(struct TrieNode *root, string key,
            int id, vector<vector<int> > &result)
{
    struct TrieNode *pCrawl = root;
    for (int level = 0; level < key.length(); level++)
    {
        int index = CHAR_TO_INDEX(key[level]);

        // If it is present also check upto which index
        // it is palindrome
        if (pCrawl->id >= 0 && pCrawl->id != id &&
```

```
isPalindrome(key, level, key.size()-1))
    result.push_back({id, pCrawl->id});

    // If not present then return
    if (!pCrawl->children[index])
        return;

    pCrawl = pCrawl->children[index];
}

for (int i: pCrawl->pos)
{
    if (i == id)
        continue;
    result.push_back({id, i});
}
}

// Function to check if a palindrome pair exists
bool checkPalindromePair(vector <string> vect)
{
    // Construct trie
    struct TrieNode *root = getNode();
    for (int i = 0; i < vect.size(); i++)
        insert(root, vect[i], i);

    // Search for different keys
    vector<vector<int> > result;
    for (int i=0; i<vect.size(); i++)
    {
        search(root, vect[i], i, result);
        if (result.size() > 0)
            return true;
    }

    return false;
}

// Driver code
int main()
{
    vector <string> vect = {"geekf", "geeks", "or",
                           "keeg", "abc", "bc"};

    checkPalindromePair(vect)? cout << "Yes" :
                                cout << "No";
    return 0;
}
```

}

Java

```
//Java program to check if there is a pair that
//of above method using Trie
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class Palin_pair2 {

    // Alphabet size (# of symbols)
    static final int ALPHABET_SIZE = 26;

    // Trie node
    static class TrieNode {
        TrieNode[] children = new TrieNode[ALPHABET_SIZE];
        List<Integer> pos; // To store palindromic
                           // positions in str
        int id;

        // isLeaf is true if the node represents
        // end of a word
        boolean isLeaf;

        // constructor
        public TrieNode() {
            isLeaf = false;
            pos = new ArrayList<>();
            for (int i = 0; i < ALPHABET_SIZE; i++)
                children[i] = null;
        }
    }

    // Utility function to check if a string is a
    // palindrome
    static boolean isPalindrome(String str, int i, int len) {
        // compare each character from starting
        // with its corresponding character from last
        while (i < len) {
            if (str.charAt(i) != str.charAt(len))
                return false;

            i++;
            len--;
        }
        return true;
    }
}
```

```
}

// If not present, inserts reverse of key into Trie. If
// the key is prefix of a Trie node, just mark leaf node
static void insert(TrieNode root, String key, int id) {
    TrieNode pCrawl = root;

    // Start traversing word from the last
    for (int level = key.length() - 1; level >= 0; level--) {
        // If it is not available in Trie, then
        // store it
        int index = key.charAt(level) - 'a';
        if (pCrawl.children[index] == null)
            pCrawl.children[index] = new TrieNode();

        // If current word is palindrome till this
        // level, store index of current word.
        if (isPalindrome(key, 0, level))
            (pCrawl.pos).add(id);

        pCrawl = pCrawl.children[index];
    }
    pCrawl.id = id;
    pCrawl.pos.add(id);

    // mark last node as leaf
    pCrawl.isLeaf = true;
}

// list to store result
static List<List<Integer>> result;

// Returns true if key presents in Trie, else false
static void search(TrieNode root, String key, int id) {
    TrieNode pCrawl = root;
    for (int level = 0; level < key.length(); level++) {
        int index = key.charAt(level) - 'a';

        // If it is present also check upto which index
        // it is palindrome
        if (pCrawl.id >= 0 && pCrawl.id != id
            && isPalindrome(key, level, key.length() - 1)) {
            List<Integer> l = new ArrayList<>();
            l.add(id);
            l.add(pCrawl.id);
            result.add(l);
        }
    }
}
```

```
// If not present then return
if (pCrawl.children[index] == null)
    return;

pCrawl = pCrawl.children[index];
}

for (int i : pCrawl.pos) {
    if (i == id)
        continue;
    List<Integer> l = new ArrayList<>();
    l.add(id);
    l.add(i);
    result.add(l);
}
}

// Function to check if a palindrome pair exists
static boolean checkPalindromePair(List<String> vect) {

    // Construct trie
    TrieNode root = new TrieNode();
    for (int i = 0; i < vect.size(); i++)
        insert(root, vect.get(i), i);

    // Search for different keys
    result = new ArrayList<>();
    for (int i = 0; i < vect.size(); i++) {
        search(root, vect.get(i), i);

        if (result.size() > 0)
            return true;
    }

    return false;
}

// Driver code
public static void main(String args[]) {
    List<String> vect = Arrays.asList("geekf", "geeks",
                                      "or", "keeg", "abc", "bc");

    if (checkPalindromePair(vect) == true)
        System.out.println("Yes");
    else
        System.out.println("No");
}
```

//This code is contributed by Sumit Ghosh

Output:

Yes

Time Complexity: $O(nk^2)$

Where n is the number of words in the list and k is the maximum length that is checked for palindrome.

Reference : <https://www.careercup.com/question?id=4879869638868992>

Source

<https://www.geeksforgeeks.org/palindrome-pair-in-an-array-of-words-or-strings/>

Chapter 123

Palindromic Tree Introduction & Implementation

Palindromic Tree Introduction & Implementation - GeeksforGeeks

We encounter various problems like Maximum length palindrome in a string, number of palindromic substrings and many more interesting problems on palindromic substrings . Mostly of these palindromic substring problems have some DP $O(n^2)$ solution (n is length of the given string) or then we have a complex algorithm like [Manacher's algorithm](#) which solves the Palindromic problems in linear time.

In this article, we will study an interesting Data Structure, which will solve all the above similar problems in much more simpler way. This data structure is invented by [Mikhail Rubinchik](#).

Features of Palindromic Tree : Online query and updation
Easy to implement
Very Fast

Structure of Palindromic Tree

Palindromic Tree's actual structure is **close to directed graph**. It is actually a merged structure of two Trees which share some common nodes(see the figure below for better understanding). Tree nodes store palindromic substrings of given string by storing their indices.

This tree consists of two types of edges :

- 1) Insertion edge (weighted edge)
- 2) Maximum Palindromic Suffix (un-weighted)

Insertion Edge :

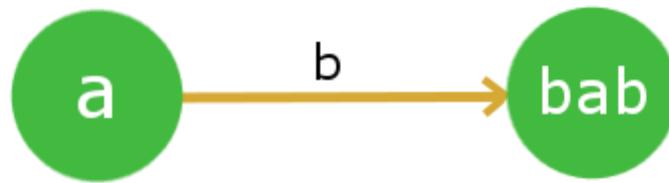
Insertion edge from a node **u** to **v** with some weight **x** means that the node v is formed by inserting x at the front and end of the string at u. As u is already a palindrome, hence the resulting string at node v will also be a palindrome.

x will be a single character for every edge. Therefore, a node can have max 26 insertion edges (considering lower letter string). We will use orange color for this edge in our pictorial representation.

Maximum Palindromic Suffix Edge:

As the name itself indicates that for a node this edge will point to its Maximum Palindromic Suffix String node. We will not be considering the complete string itself as the Maximum Palindromic Suffix as this will make no sense(self loops). For simplicity purpose, we will call it as Suffix edge(by which we mean maximum suffix except the complete string). It is quite obvious that every node will have only 1 Suffix Edge as we will not store duplicate strings in the tree. We will use Blue dashed edges for its Pictorial representation.

Insertion Edge



Maxi

aba

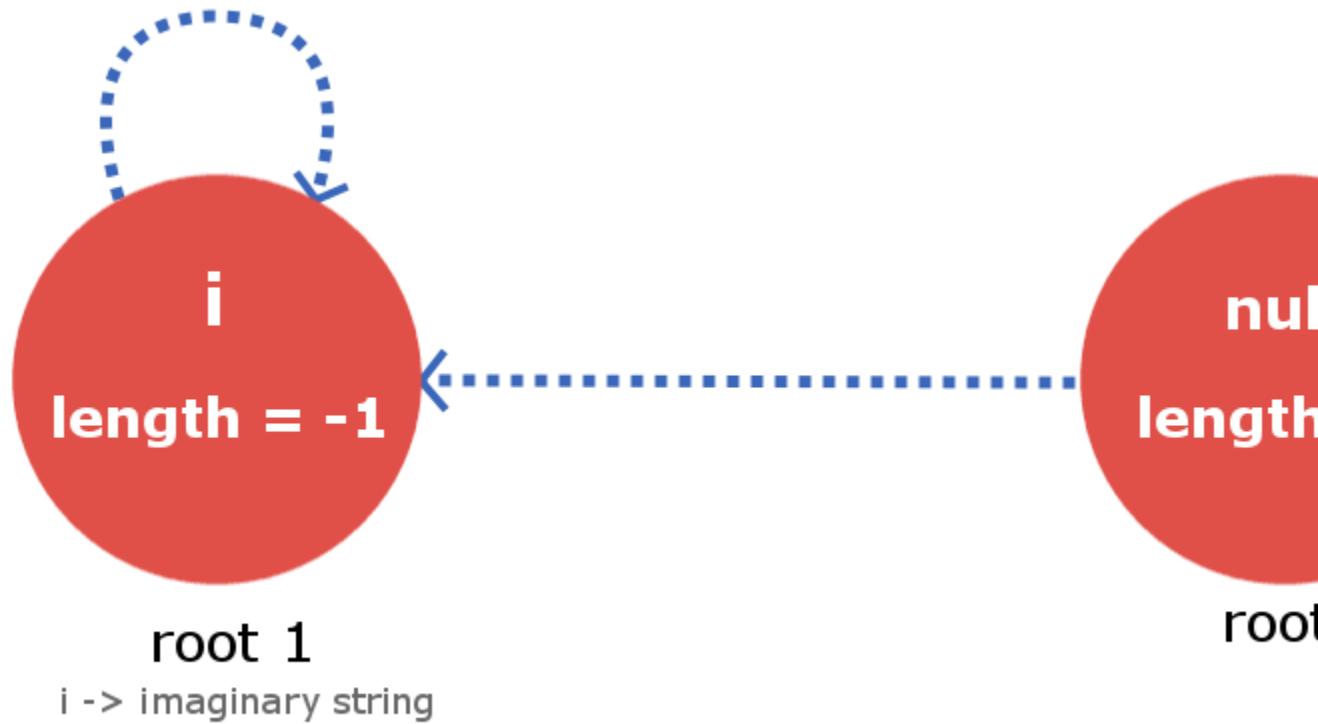
Root Nodes and their convention:

This tree/graph data structure will contain **2 root dummy nodes**. More, precisely consider it as roots of two separate trees, which are linked together.

Root-1 will be a dummy node which will describe a string for $length = -1$ (you can easily infer from the implementation point of view that why we used so). *Root-2* will be a node which will describe a null string of $length = 0$.

Root-1 has a suffix edge connected to itself(self-loop) as for any imaginary string of length -1 , its Maximum palindromic suffix will also be imaginary, so this is justified. Now *Root-2*

will also have its suffix edge connected to Root-1 as for a null string (length 0) there is no real palindromic suffix string of length less than 0.



Building the Palindromic Tree

To build a Palindromic Tree, we will simply insert the characters in our string one by one till we reach its end and when we are done inserting we will be with our palindromic tree which will contain all the distinct palindromic substrings of the given strings. All we need to ensure is that, at every insertion of a new character, our palindromic tree maintains the above discussed feature. Let's see how we can accomplish it.

Let's say we are given a string s with length l and we have inserted the string up till index k ($k < l-1$). Now, we need to insert the $(k+1)$ th character. Insertion of $(k+1)$ th character means insertion of a node that is longest palindrome ending at index $(k+1)$. So, the longest palindromic string will be of form (' $s[k+1]$ ' + "X" + ' $s[k+1]$ ') and X will itself be a palindrome. Now the fact is that the string X lies at index $< k+1$ and is palindrome. So, it will already exist in our palindromic tree as we have maintained the very basic property of it saying that it will contain all the distinct palindromic substrings.

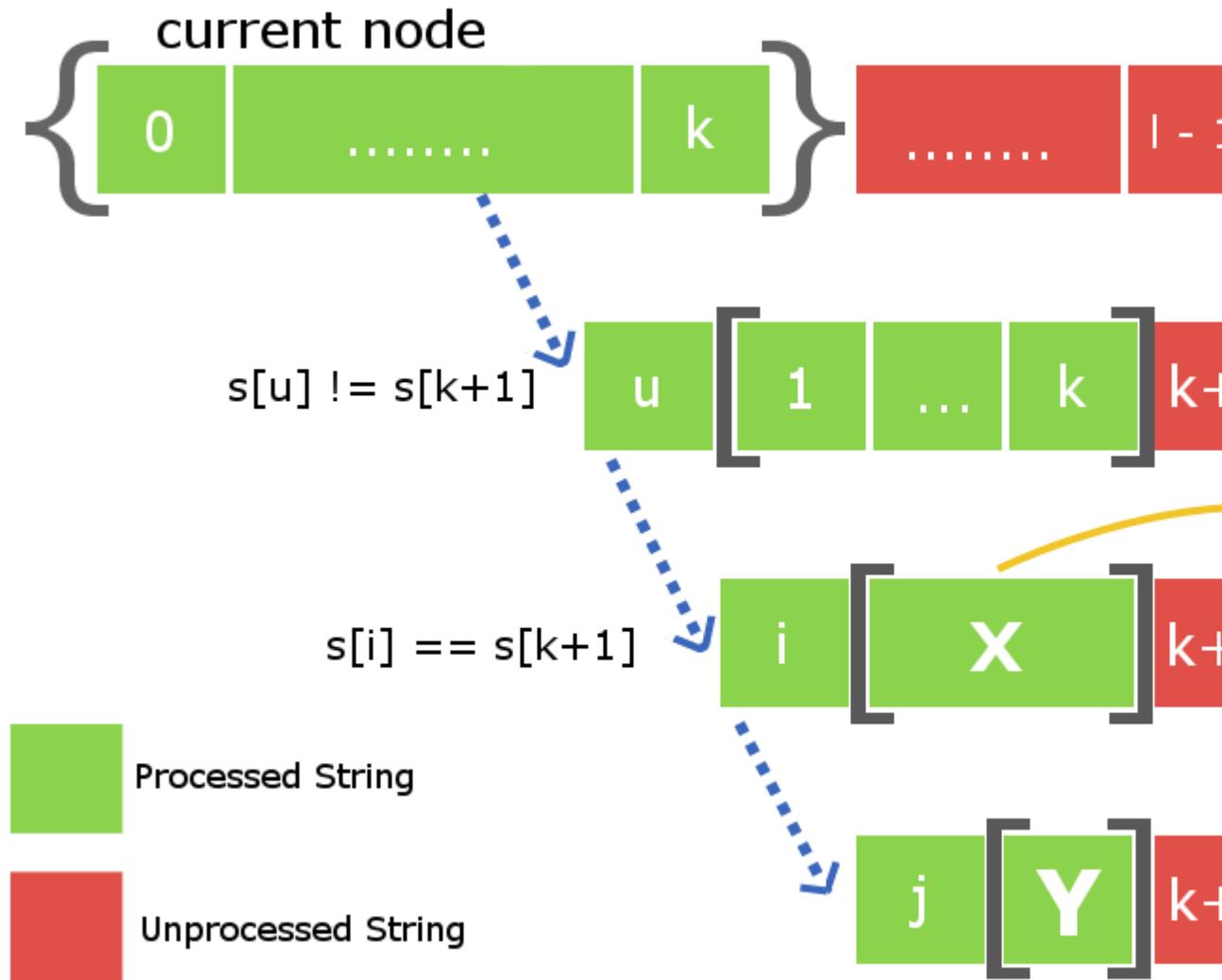
So, to insert the character $s[k+1]$, we only need to find the String X in our tree and direct the insertion edge from X with weight $s[k+1]$ to a new node, which contains $s[k+1]+X+s[k+1]$. The main job now is to find the string X in efficient time. As we know that we are storing the suffix link for all the nodes. Therefore to track the node with

string X we just need to move down the suffix link for the current node i.e the node which contains insertion of $s[k]$. See the below image for better understanding.

The current node in the below figure tells that it is the largest palindrome that ends at index k after processing all the indices from 0 to k. The blue dotted path is the link of suffix edges from current node to other processed nodes in the tree. String X will exist in one of these nodes that lie on this chain of suffix link. All we need is to find it by iterating over it down the chain.

To find the required node that contains the string X we will place the $k+1$ th character at the end of every node that lies in suffix link chain and check if first character of the corresponding suffix link string is equal to the $k+1$ th character.

Once, we find the X string we will direct an insertion edge with weight $s[k+1]$ and link it to the new node that contains largest palindrome ending at index $k+1$. The array elements between the brackets as described in the figure below are the nodes that are stored in the tree.



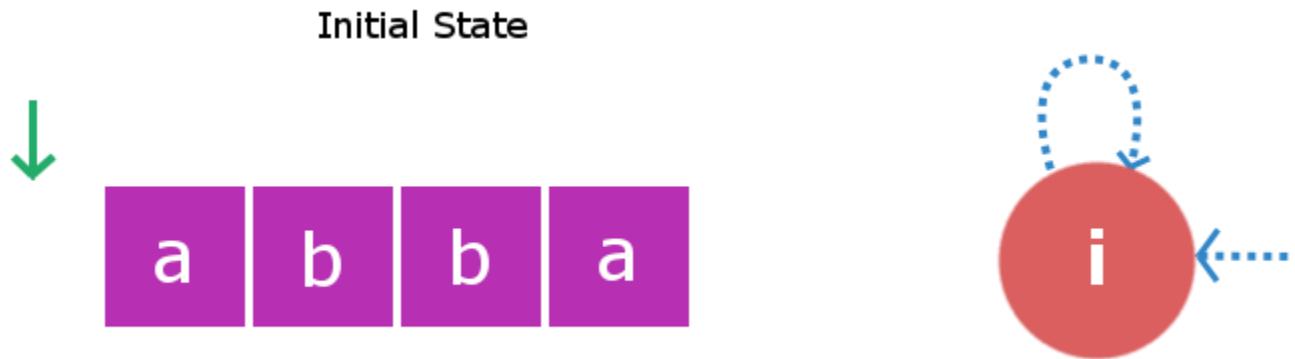
There is one more thing left that is to be done. As we have created a new node at this $s[k+1]$ insertion, therefore we will also have to connect it with its suffix link child. Once, again to do so we will use the above down the suffix link iteration from node X to find some new string Y such that $s[k+1] + Y + s[k+1]$ is a largest palindromic suffix for the newly created node. Once, we find it we will then connect the suffix link of our newly created node with the node Y.

Note : There are two possibilities when we find the string X. First possibility is that string $s[k+1]Xs[k+1]$ do not exist in the tree and second possibility is if it already exists in the tree. In first case we will proceed the same way but in second case we will not

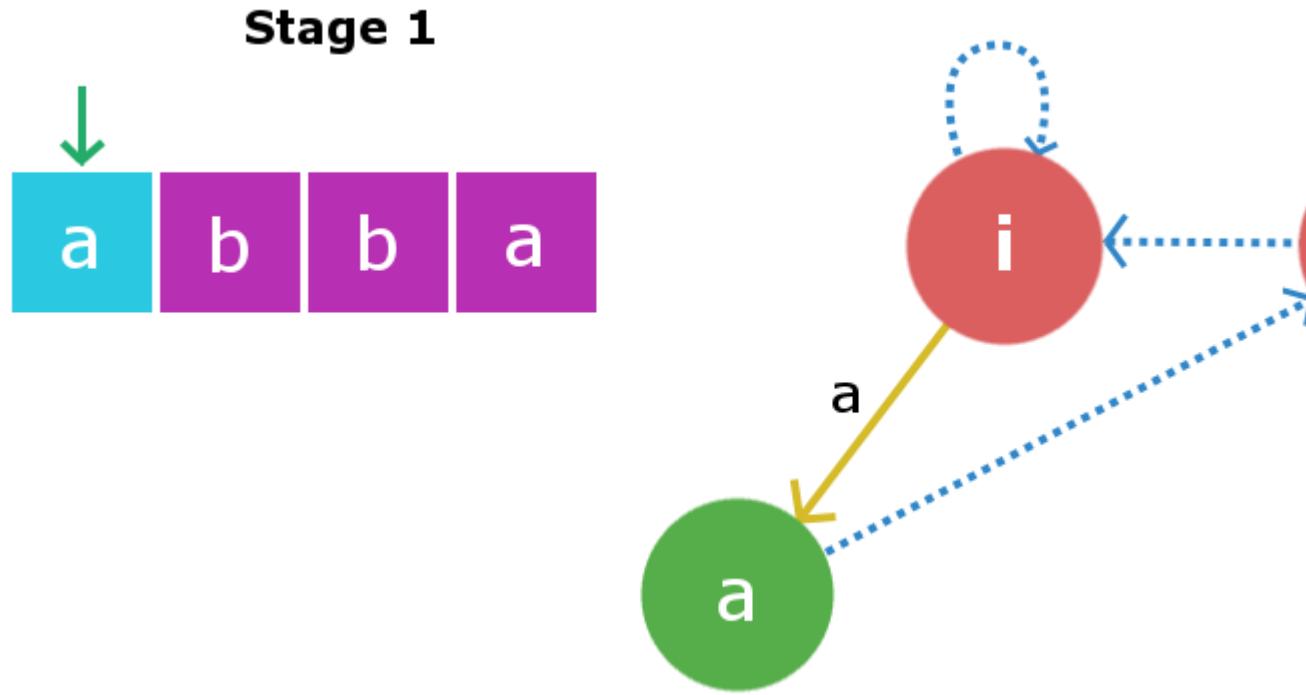
create a new node separately but will just link the insertion edge from X to already existing $S[k+1]+X+S[k+1]$ node in the tree. We also need not to add the suffix link because the node will already contain its suffix link.

Consider a string $s = \text{"abba"}$ with $length = 4$.

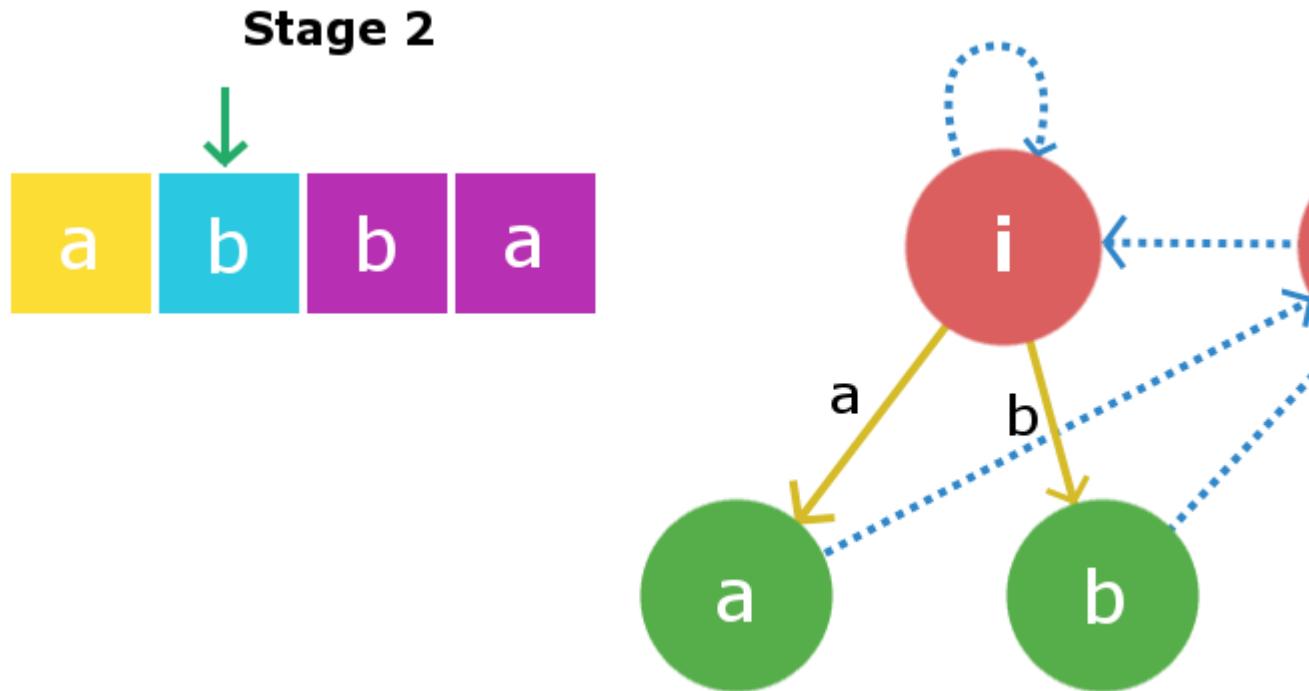
At initial state we will have our two dummy root nodes one with length -1 (some imaginary string **i**) and second a **null** string with length 0. At this point we haven't inserted any character in the tree. Root1 i.e root node with length -1 will be the current node from which insertion takes place.



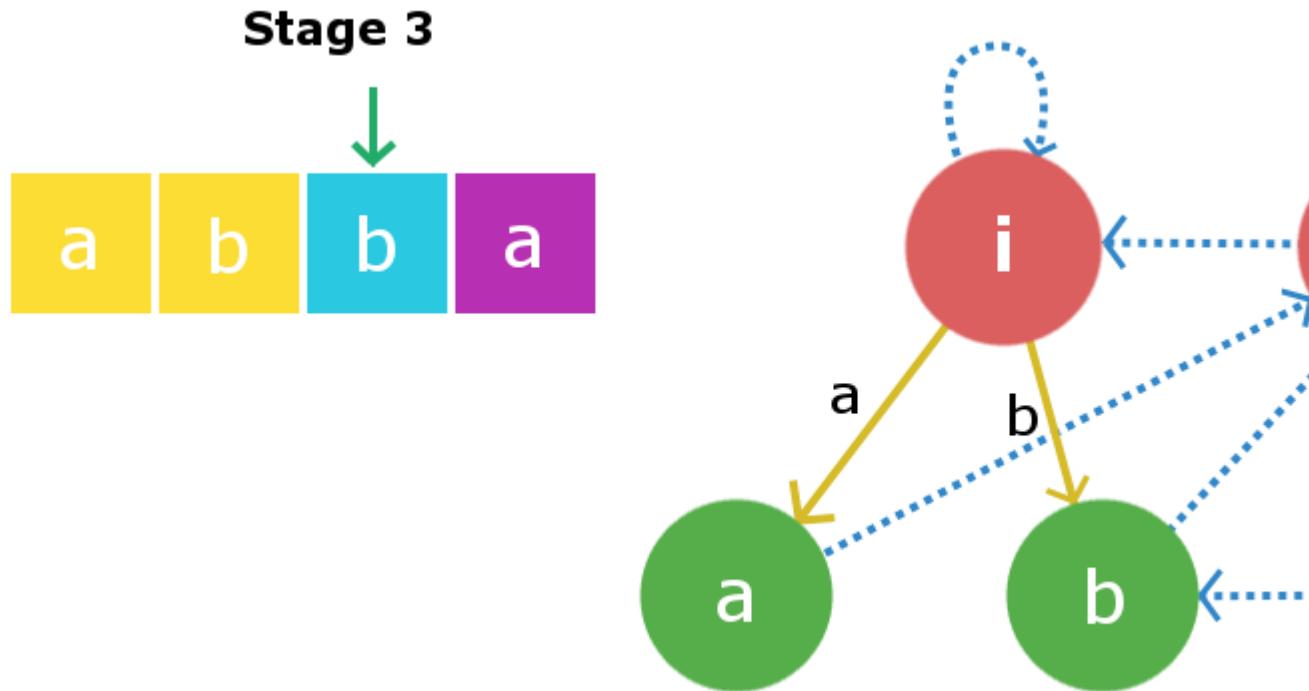
Stage 1: We will insert $s[0]$ i.e 'a'. We will start checking from the current node i.e Root1. Inserting 'a' at start and end of a string with length -1 will yield to a string with length 1 and this string will be "a". Therefore, we create a new node "a" and direct insertion edge from root1 to this new node. Now, largest palindromic suffix string for string of length 1 will be a null string so its suffix link will be directed to root2 i.e null string. Now the current node will be this new node "a".



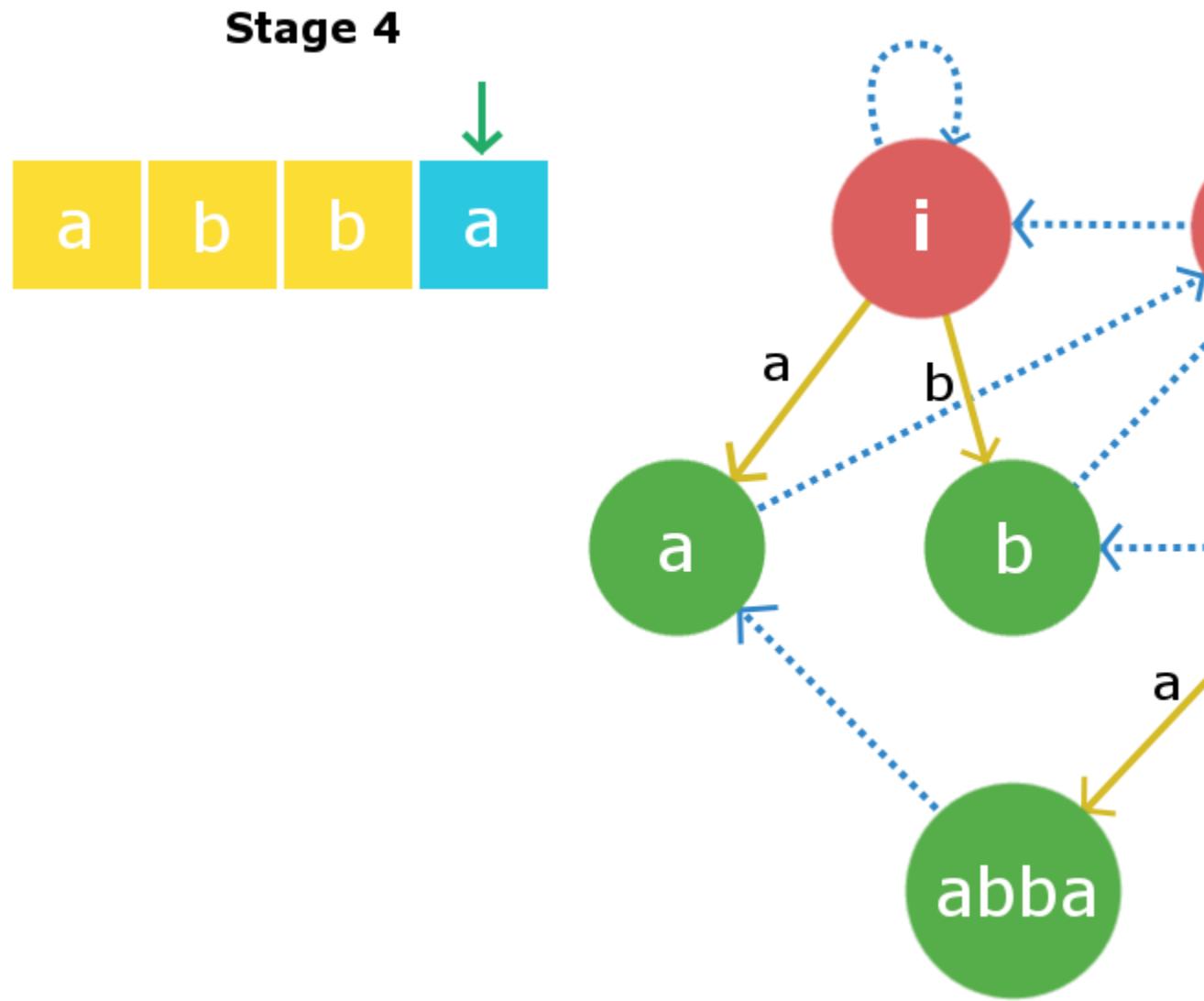
Stage 2: We will insert $s[1]$ i.e ‘b’. Insertion process will start from current node i.e “a” node. We will traverse the suffix link chain starting from current node till we find suitable X string , So here traversing the suffix link we again found root1 as X string. Once again inserting ‘b’ to string of length -1 will yield a string of of length 1 i.e string “b”. Suffix link for this node will go to null string as described in above insertion. Now the current node will be this new node “b”.



Stage 3: We will insert $s[2]$ i.e ‘b’. Once again starting from current node we will traverse its suffix link to find required X string. In this case it founds to be root2 i.e null string as adding ‘b’ at front and end of null string yields a palindrome “bb” of length 2. Therefore, we will create a new node “**b**b” and will direct the insertion edge from the null string to the newly created string. Now, the largest suffix palindrome for this current node will be node “b”. So, we will link the suffix edge from this newly created node to node “b”. Current node now becomes node “bb”.



Stage 4: We will insert $s[3]$ i.e 'a'. Insertion process begins with current node and in this case the current node itself is the largest X string such that $s[0] + X + s[3]$ is palindrome. Therefore, we will create a new node "abba" and link the insertion edge from the current node "bb" to this newly created node with edge weight 'a'. Now, the suffix the link from this newly created node will be linked to node "a" as that is the largest palindromic suffix.



The C++ implementation for the above implementation is given below :

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

#define MAXN 1000

struct Node
```

```
{
    // store start and end indexes of current
    // Node inclusively
    int start, end;

    // stores length of substring
    int length;

    // stores insertion Node for all characters a-z
    int insertEdg[26];

    // stores the Maximum Palindromic Suffix Node for
    // the current Node
    int suffixEdg;
};

// two special dummy Nodes as explained above
Node root1, root2;

// stores Node information for constant time access
Node tree[MAXN];

// Keeps track the current Node while insertion
int currNode;
string s;
int ptr;

void insert(int idx)
{
    //STEP 1//

    /* search for Node X such that s[idx] X S[idx]
       is maximum palindrome ending at position idx
       iterate down the suffix link of currNode to
       find X */
    int tmp = currNode;
    while (true)
    {
        int curLength = tree[tmp].length;
        if (idx - curLength >= 1 and s[idx] == s[idx-curLength-1])
            break;
        tmp = tree[tmp].suffixEdg;
    }

    /* Now we have found X ....
     * X = string at Node tmp
     * Check : if s[idx] X s[idx] already exists or not*/
    if(tree[tmp].insertEdg[s[idx]-'a'] != 0)
}
}
```

```

{
    // s[idx] X s[idx] already exists in the tree
    currNode = tree[tmp].insertEdg[s[idx]-'a'];
    return;
}

// creating new Node
ptr++;

// making new Node as child of X with
// weight as s[idx]
tree[tmp].insertEdg[s[idx]-'a'] = ptr;

// calculating length of new Node
tree[ptr].length = tree[tmp].length + 2;

// updating end point for new Node
tree[ptr].end = idx;

// updating the start for new Node
tree[ptr].start = idx - tree[ptr].length + 1;

//STEP 2//

/* Setting the suffix edge for the newly created
Node tree[ptr]. Finding some String Y such that
s[idx] + Y + s[idx] is longest possible
palindromic suffix for newly created Node.*/

tmp = tree[tmp].suffixEdg;

// making new Node as current Node
currNode = ptr;
if (tree[currNode].length == 1)
{
    // if new palindrome's length is 1
    // making its suffix link to be null string
    tree[currNode].suffixEdg = 2;
    return;
}
while (true)
{
    int curLength = tree[tmp].length;
    if (idx-curLength >= 1 and s[idx] == s[idx-curLength-1])
        break;
    tmp = tree[tmp].suffixEdg;
}

```

```
// Now we have found string Y
// linking current Nodes suffix link with s[idx]+Y+s[idx]
tree[currNode].suffixEdg = tree[tmp].insertEdg[s[idx]-'a'];
}

// driver program
int main()
{
    // initializing the tree
    root1.length = -1;
    root1.suffixEdg = 1;
    root2.length = 0;
    root2.suffixEdg = 1;

    tree[1] = root1;
    tree[2] = root2;
    ptr = 2;
    currNode = 1;

    // given string
    s = "abcbab";
    int l = s.length();

    for (int i=0; i<l; i++)
        insert(i);

    // printing all of its distinct palindromic
    // substring
    cout << "All distinct palindromic substring for "
        << s << " : \n";
    for (int i=3; i<=ptr; i++)
    {
        cout << i-2 << " ) ";
        for (int j=tree[i].start; j<=tree[i].end; j++)
            cout << s[j];
        cout << endl;
    }

    return 0;
}
```

Output:

```
All distinct palindromic substring for abcbab :
1)a
2)b
```

- 3)c
- 4)bcb
- 5)abcba
- 6)bab

Time Complexity

The time complexity for the building process will be **O(k*n)**, here “n” is the length of the string and ‘k’ is the extra iterations required to find the string X and string Y in the suffix links every time we insert a character. Let’s try to approximate the constant ‘k’. We shall consider a worst case like $s = \text{"aaaaaaaaabcccccccccdeeeeeeeeef"}$. In this case for similar streak of continuous characters it will take extra 2 iterations per index to find both string X and Y in the suffix links , but as soon as it reaches some index i such that $s[i] \neq s[i-1]$ the left most pointer for the maximum length suffix will reach its rightmost limit. Therefore, for all i when $s[i] \neq s[i-1]$, it will cost in total n iterations(summing over each iteration) and for rest i when $s[i] == s[i-1]$ it takes 2 iteration which sums up over all such i and takes $2*n$ iterations. Hence, approximately our complexity in this case will be $O(3*n) \sim O(n)$. So, we can roughly say that the constant factor ‘k’ will be very less. Therefore, we can consider the overall complexity to be linear **O(length of string)**. You may refer the reference links for better understanding.

References :

- <http://codeforces.com/blog/entry/13959>
- <http://adilet.org/blog/25-09-14/>

Source

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

Chapter 124

Pattern Searching using Suffix Tree

Pattern Searching using Suffix Tree - GeeksforGeeks

Given a text $txt[0..n-1]$ and a pattern $pat[0..m-1]$, write a function $search(\text{char } pat[], \text{char } txt[])$ that prints all occurrences of $pat[]$ in $txt[]$. You may assume that $n > m$.

Preprocess Pattern or Preprocess Text?

We have discussed the following algorithms in the previous posts:

[KMP Algorithm](#)

[Rabin Karp Algorithm](#)

[Finite Automata based Algorithm](#)

[Boyer Moore Algorithm](#)

All of the above algorithms preprocess the pattern to make the pattern searching faster. The best time complexity that we could get by preprocessing pattern is $O(n)$ where n is length of the text. In this post, we will discuss an approach that preprocesses the text. A suffix tree is built of the text. After preprocessing text (building suffix tree of text), we can search any pattern in $O(m)$ time where m is length of the pattern.

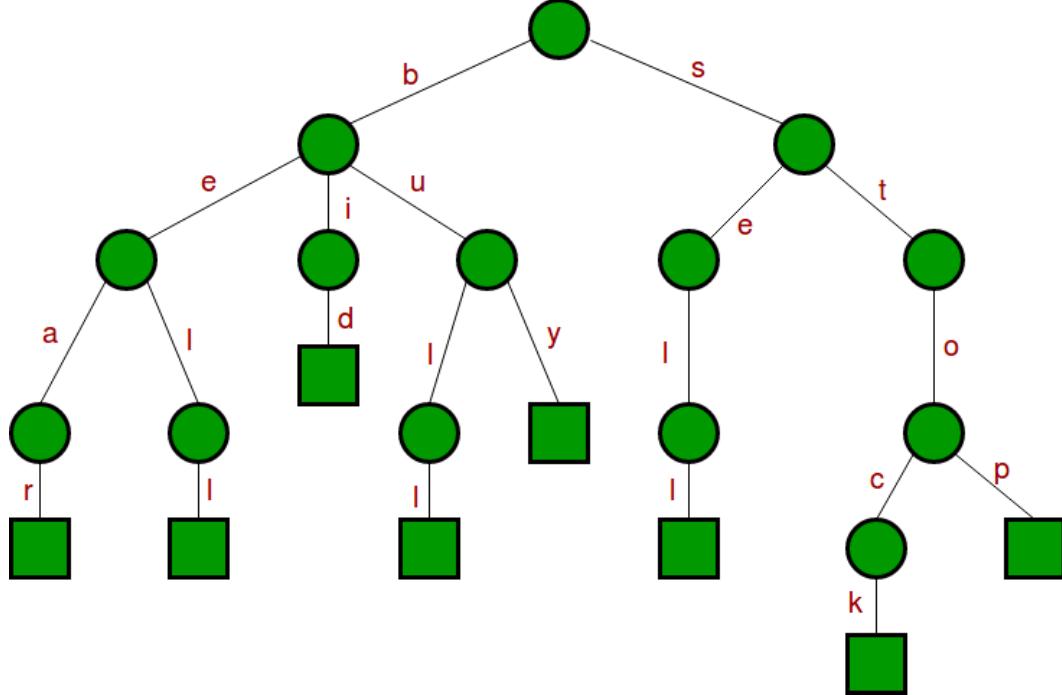
Imagine you have stored complete work of [William Shakespeare](#) and preprocessed it. You can search any string in the complete work in time just proportional to length of the pattern. This is really a great improvement because length of pattern is generally much smaller than text.

Preprocessing of text may become costly if the text changes frequently. It is good for fixed text or less frequently changing text though.

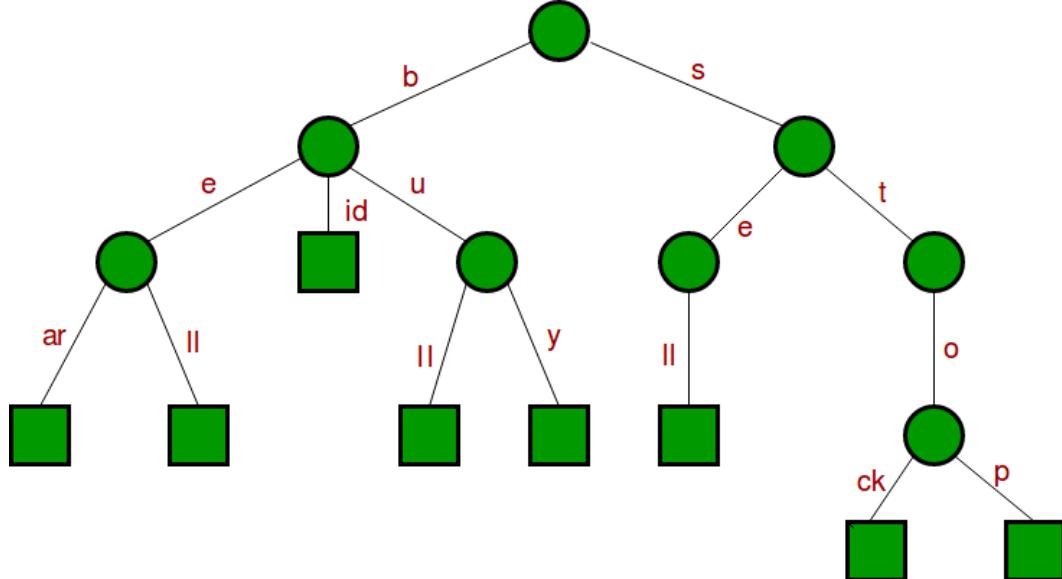
A Suffix Tree for a given text is a compressed trie for all suffixes of the given text. We have discussed [Standard Trie](#). Let us understand **Compressed Trie** with the following array of words.

```
{bear, bell, bid, bull, buy, sell, stock, stop}
```

Following is standard trie for the above input set of words.



Following is the compressed trie. Compressed Trie is obtained from standard trie by joining chains of single nodes. The nodes of a compressed trie can be stored by storing index ranges at the nodes.



How to build a Suffix Tree for a given text?

As discussed above, Suffix Tree is compressed trie of all suffixes, so following are very abstract

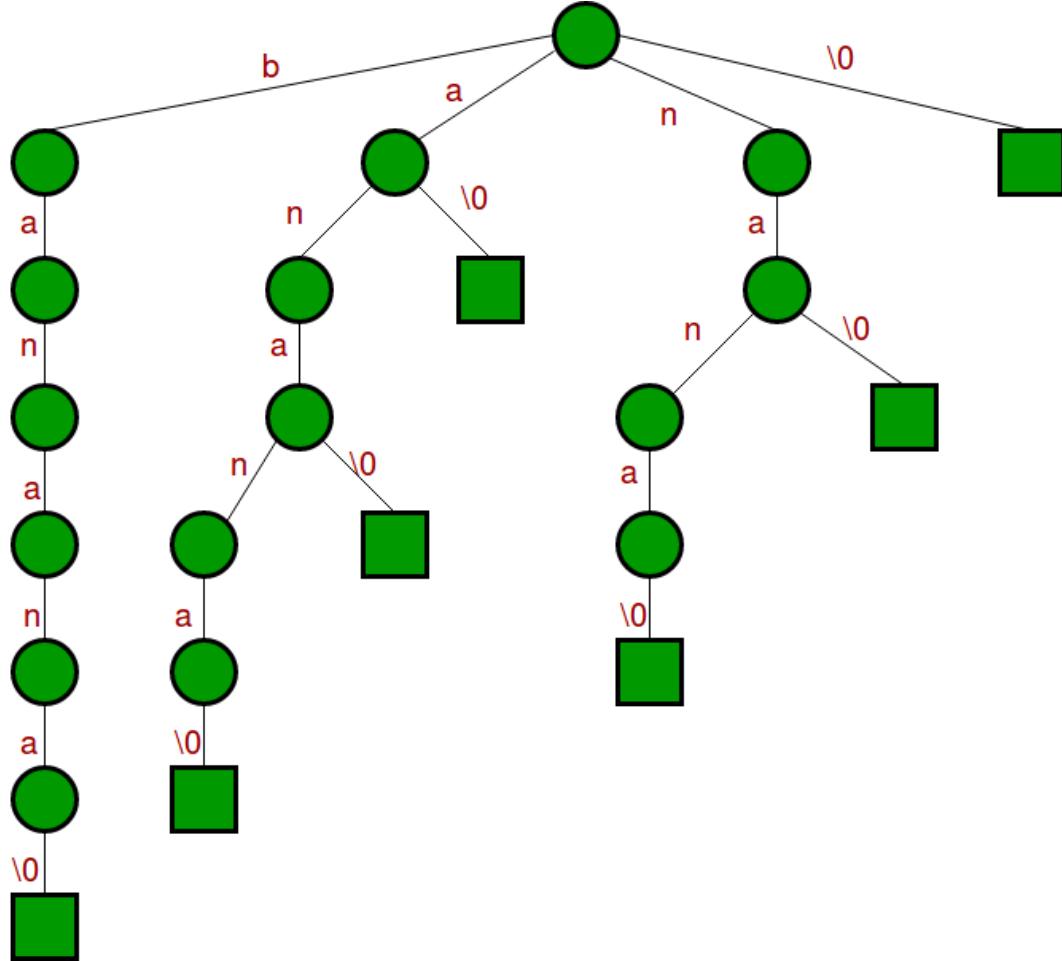
steps to build a suffix tree from given text.

- 1) Generate all suffixes of given text.
- 2) Consider all suffixes as individual words and build a compressed trie.

Let us consider an example text “banana\0” where ‘\0’ is string termination character. Following are all suffixes of “banana\0”

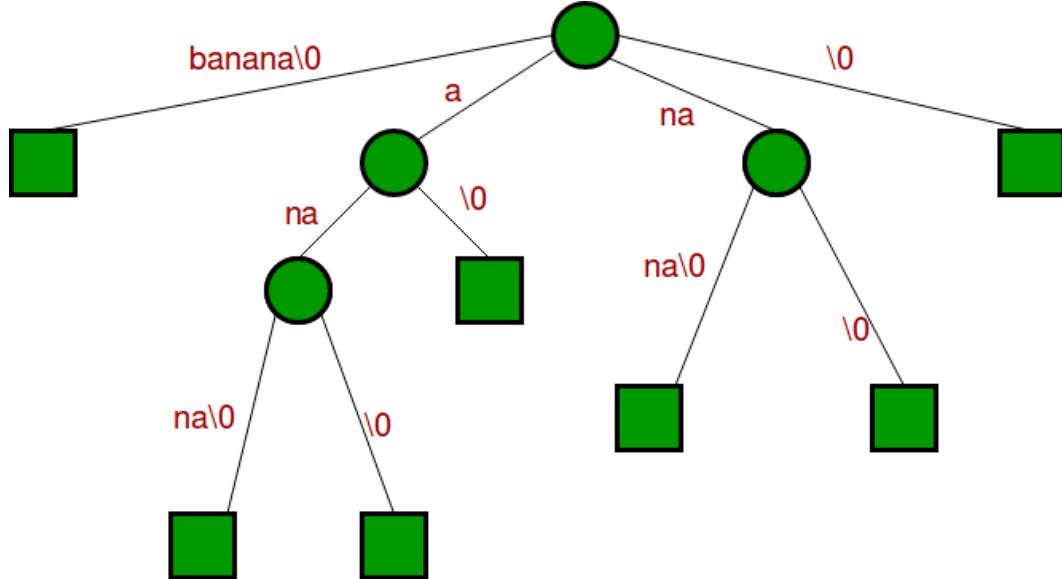
```
banana\0
anana\0
nana\0
ana\0
na\0
a\0
\0
```

If we consider all of the above suffixes as individual words and build a trie, we get following.



If we join chains of single nodes, we get the following compressed trie, which is the Suffix

Tree for given text “banana\0”



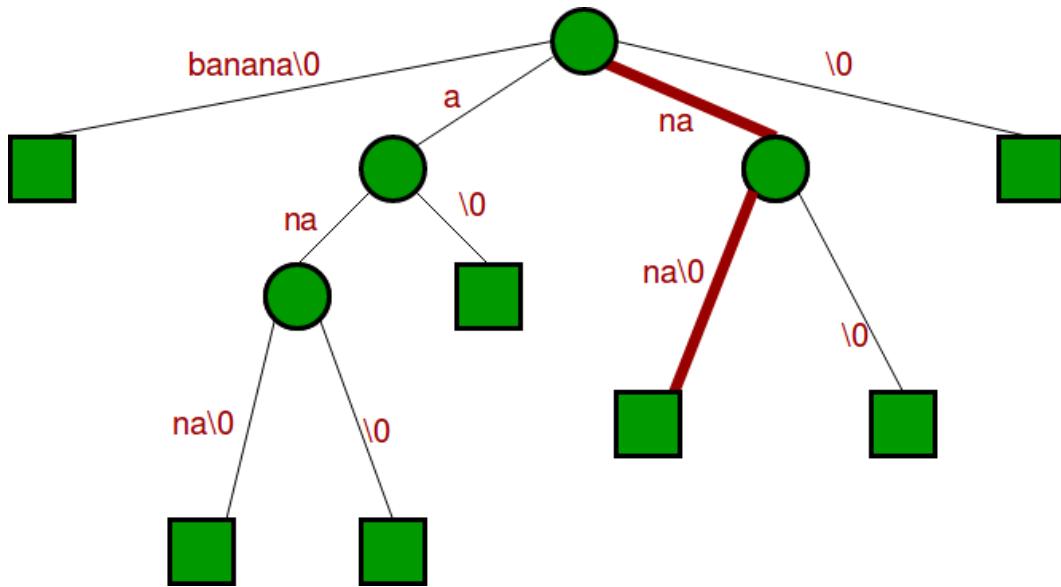
Please note that above steps are just to manually create a Suffix Tree. We will be discussing actual algorithm and implementation in a separate post.

How to search a pattern in the built suffix tree?

We have discussed above how to build a Suffix Tree which is needed as a preprocessing step in pattern searching. Following are abstract steps to search a pattern in the built Suffix Tree.

- 1) Starting from the first character of the pattern and root of Suffix Tree, do following for every character.
 - a) For the current character of pattern, if there is an edge from the current node of suffix tree, follow the edge.
 - b) If there is no edge, print “pattern doesn’t exist in text” and return.
- 2) If all characters of pattern have been processed, i.e., there is a path from root for characters of the given pattern, then print “Pattern found”.

Let us consider the example pattern as “nan” to see the searching process. Following diagram shows the path followed for searching “nan” or “nana”.



How does this work?

Every pattern that is present in text (or we can say every substring of text) must be a prefix of one of all possible suffixes. The statement seems complicated, but it is a simple statement, we just need to take an example to check validity of it.

Applications of Suffix Tree

Suffix tree can be used for a wide range of problems. Following are some famous problems where Suffix Trees provide optimal time complexity solution.

- 1) [Pattern Searching](#)
- 2) [Finding the longest repeated substring](#)
- 3) [Finding the longest common substring](#)
- 4) [Finding the longest palindrome in a string](#)

There are many more applications. See [this](#)for more details.

Ukkonen's Suffix Tree Construction is discussed in following articles:

- [Ukkonen's Suffix Tree Construction – Part 1](#)
- [Ukkonen's Suffix Tree Construction – Part 2](#)
- [Ukkonen's Suffix Tree Construction – Part 3](#)
- [Ukkonen's Suffix Tree Construction – Part 4](#)
- [Ukkonen's Suffix Tree Construction – Part 5](#)
- [Ukkonen's Suffix Tree Construction – Part 6](#)

Source

<https://www.geeksforgeeks.org/pattern-searching-using-suffix-tree/>

Chapter 125

Pattern Searching using a Trie of all Suffixes

Pattern Searching using a Trie of all Suffixes - GeeksforGeeks

Problem Statement: Given a text $txt[0..n-1]$ and a pattern $pat[0..m-1]$, write a function $search(char pat[], char txt[])$ that prints all occurrences of $pat[]$ in $txt[]$. You may assume that $n > m$.

As discussed in the [previous post](#), we discussed that there are two ways efficiently solve the above problem.

1) Preprocess Pattern: [KMP Algorithm](#), [Rabin Karp Algorithm](#), [Finite Automata](#), [Boyer Moore Algorithm](#).

2) Preprocess Text: [Suffix Tree](#)

The best possible time complexity achieved by first (preprocessing pattern) is $O(n)$ and by second (preprocessing text) is $O(m)$ where m and n are lengths of pattern and text respectively.

Note that the second way does the searching only in $O(m)$ time and it is preferred when text doesn't change very frequently and there are many search queries. We have discussed [Suffix Tree \(A compressed Trie of all suffixes of Text\)](#).

Implementation of Suffix Tree may be time consuming for problems to be coded in a technical interview or programming contexts. In this post simple implementation of a [Standard Trie](#) of all Suffixes is discussed. The implementation is close to suffix tree, the only thing is, it's a [simple Trie](#) instead of compressed Trie.

As discussed in [Suffix Tree](#) post, the idea is, every pattern that is present in text (or we can say every substring of text) must be a prefix of one of all possible suffixes. So if we build a Trie of all suffixes, we can find the pattern in $O(m)$ time where m is pattern length.

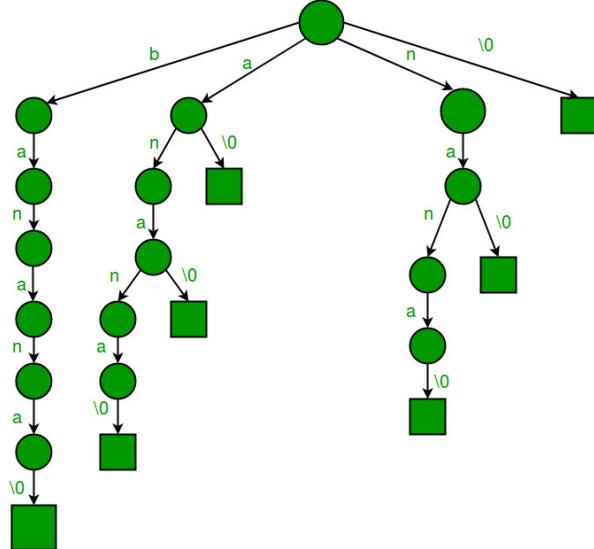
Building a Trie of Suffixes

- 1) Generate all suffixes of given text.
- 2) Consider all suffixes as individual words and build a trie.

Let us consider an example text “banana\0” where ‘\0’ is string termination character. Following are all suffixes of “banana\0”

```
banana\0
anana\0
nana\0
ana\0
na\0
a\0
\0
```

If we consider all of the above suffixes as individual words and build a Trie, we get following.



How to search a pattern in the built Trie?

Following are steps to search a pattern in the built Trie.

1) Starting from the first character of the pattern and root of the Trie, do following for every character.

.....**a)** For the current character of pattern, if there is an edge from the current node, follow the edge.

.....**b)** If there is no edge, print “pattern doesn’t exist in text” and return.

2) If all characters of pattern have been processed, i.e., there is a path from root for characters of the given pattern, then print print all indexes where pattern is present. To store indexes, we use a list with every node that stores indexes of suffixes starting at the node.

Following is C++ implementation of the above idea.

C++

```
// A simple C++ implementation of substring search using trie of suffixes
#include<iostream>
#include<list>
#define MAX_CHAR 256
```

```
using namespace std;

// A Suffix Trie (A Trie of all suffixes) Node
class SuffixTreeNode
{
private:
    SuffixTreeNode *children[MAX_CHAR];
    list<int> *indexes;
public:
    SuffixTreeNode() // Constructor
    {
        // Create an empty linked list for indexes of
        // suffixes starting from this node
        indexes = new list<int>;

        // Initialize all child pointers as NULL
        for (int i = 0; i < MAX_CHAR; i++)
            children[i] = NULL;
    }

    // A recursive function to insert a suffix of the txt
    // in subtree rooted with this node
    void insertSuffix(string suffix, int index);

    // A function to search a pattern in subtree rooted
    // with this node. The function returns pointer to a linked
    // list containing all indexes where pattern is present.
    // The returned indexes are indexes of last characters
    // of matched text.
    list<int>* search(string pat);
};

// A Trie of all suffixes
class SuffixTrie
{
private:
    SuffixTreeNode root;
public:
    // Constructor (Builds a trie of suffies of the given text)
    SuffixTrie(string txt)
    {
        // Consider all suffixes of given string and insert
        // them into the Suffix Trie using recursive function
        // insertSuffix() in SuffixTreeNode class
        for (int i = 0; i < txt.length(); i++)
            root.insertSuffix(txt.substr(i), i);
    }
}
```

```

// Function to searches a pattern in this suffix trie.
void search(string pat);
};

// A recursive function to insert a suffix of the txt in
// subtree rooted with this node
void SuffixTreeNode::insertSuffix(string s, int index)
{
    // Store index in linked list
    indexes->push_back(index);

    // If string has more characters
    if (s.length() > 0)
    {
        // Find the first character
        char cIndex = s.at(0);

        // If there is no edge for this character, add a new edge
        if (children[cIndex] == NULL)
            children[cIndex] = new SuffixTreeNode();

        // Recur for next suffix
        children[cIndex]->insertSuffix(s.substr(1), index+1);
    }
}

// A recursive function to search a pattern in subtree rooted with
// this node
list<int>* SuffixTreeNode::search(string s)
{
    // If all characters of pattern have been processed,
    if (s.length() == 0)
        return indexes;

    // if there is an edge from the current node of suffix trie,
    // follow the edge.
    if (children[s.at(0)] != NULL)
        return (children[s.at(0)])->search(s.substr(1));

    // If there is no edge, pattern doesn't exist in text
    else return NULL;
}

/* Prints all occurrences of pat in the Suffix Trie S (built for text)*/
void SuffixTrie::search(string pat)
{
    // Let us call recursive search function for root of Trie.
    // We get a list of all indexes (where pat is present in text) in

```

```
// variable 'result'  
list<int> *result = root.search(pat);  
  
// Check if the list of indexes is empty or not  
if (result == NULL)  
    cout << "Pattern not found" << endl;  
else  
{  
    list<int>::iterator i;  
    int patLen = pat.length();  
    for (i = result->begin(); i != result->end(); ++i)  
        cout << "Pattern found at position " << *i - patLen << endl;  
}  
}  
  
// driver program to test above functions  
int main()  
{  
    // Let us build a suffix trie for text "geeksforgeeks.org"  
    string txt = "geeksforgeeks.org";  
    SuffixTrie S(txt);  
  
    cout << "Search for 'ee'" << endl;  
    S.search("ee");  
  
    cout << "\nSearch for 'geek'" << endl;  
    S.search("geek");  
  
    cout << "\nSearch for 'quiz'" << endl;  
    S.search("quiz");  
  
    cout << "\nSearch for 'forgeeks'" << endl;  
    S.search("forgeeks");  
  
    return 0;  
}
```

Java

```
import java.util.LinkedList;  
import java.util.List;  
class SuffixTreeNode {  
  
    final static int MAX_CHAR = 256;  
  
    SuffixTreeNode[] children = new SuffixTreeNode[MAX_CHAR];  
    List<Integer> indexes;
```

```
SuffixTreeNode() // Constructor
{
    // Create an empty linked list for indexes of
    // suffixes starting from this node
    indexes = new LinkedList<Integer>();

    // Initialize all child pointers as NULL
    for (int i = 0; i < MAX_CHAR; i++)
        children[i] = null;
}

// A recursive function to insert a suffix of
// the text in subtree rooted with this node
void insertSuffix(String s, int index) {

    // Store index in linked list
    indexes.add(index);

    // If string has more characters
    if (s.length() > 0) {

        // Find the first character
        char cIndex = s.charAt(0);

        // If there is no edge for this character,
        // add a new edge
        if (children[cIndex] == null)
            children[cIndex] = new SuffixTreeNode();

        // Recur for next suffix
        children[cIndex].insertSuffix(s.substring(1),
                                      index + 1);
    }
}

// A function to search a pattern in subtree rooted
// with this node. The function returns pointer to a
// linked list containing all indexes where pattern
// is present. The returned indexes are indexes of
// last characters of matched text.
List<Integer> search(String s) {

    // If all characters of pattern have been
    // processed,
    if (s.length() == 0)
        return indexes;

    // if there is an edge from the current node of
```

```
// suffix tree, follow the edge.
if (children[s.charAt(0)] != null)
    return (children[s.charAt(0)]).search(s.substring(1));

// If there is no edge, pattern doesnt exist in
// text
else
    return null;
}

// A Trie of all suffixes
class Suffix_tree{

    SuffixTreeNode root = new SuffixTreeNode();

    // Constructor (Builds a trie of suffies of the
    // given text)
    Suffix_tree(String txt) {

        // Consider all suffixes of given string and
        // insert them into the Suffix Trie using
        // recursive function insertSuffix() in
        // SuffixTreeNode class
        for (int i = 0; i < txt.length(); i++)
            root.insertSuffix(txt.substring(i), i);
    }

    /* Prints all occurrences of pat in the Suffix Trie S
     * (built for text) */
    void search_tree(String pat) {

        // Let us call recursive search function for
        // root of Trie.
        // We get a list of all indexes (where pat is
        // present in text) in variable 'result'
        List<Integer> result = root.search(pat);

        // Check if the list of indexes is empty or not
        if (result == null)
            System.out.println("Pattern not found");
        else {

            int patLen = pat.length();

            for (Integer i : result)
                System.out.println("Pattern found at position " +
                    (i - patLen));
        }
    }
}
```

```
        }
    }

// driver program to test above functions
public static void main(String args[]) {

    // Let us build a suffix trie for text
    // "geeksforgeeks.org"
    String txt = "geeksforgeeks.org";
    Suffix_tree S = new Suffix_tree(txt);

    System.out.println("Search for 'ee'");
    S.search_tree("ee");

    System.out.println("\nSearch for 'geek'");
    S.search_tree("geek");

    System.out.println("\nSearch for 'quiz'");
    S.search_tree("quiz");

    System.out.println("\nSearch for 'forgeeks'");
    S.search_tree("forgeeks");
}
}

// This code is contributed by Sumit Ghosh
```

Output:

```
Search for 'ee'
Pattern found at position 1
Pattern found at position 9
```

```
Search for 'geek'
Pattern found at position 0
Pattern found at position 8
```

```
Search for 'quiz'
Pattern not found
```

```
Search for 'forgeeks'
Pattern found at position 5
```

Time Complexity of the above search function is $O(m+k)$ where m is length of the pattern and k is the number of occurrences of pattern in text.

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

Improved By : [smodi2007](#)

Source

<https://www.geeksforgeeks.org/pattern-searching-using-trie-suffixes/>

Chapter 126

Persistent Segment Tree Set 1 (Introduction)

Persistent Segment Tree Set 1 (Introduction) - GeeksforGeeks

Prerequisite : Segment Tree
Persistency in Data Structure

Segment Tree is itself a great data structure that comes into play in many cases. In this post we will introduce the concept of Persistency in this data structure. Persistency, simply means to retain the changes. But obviously, retaining the changes cause extra memory consumption and hence affect the Time Complexity.

Our aim is to apply persistency in segment tree and also to ensure that it does not take more than **O(log n)** time and space for each change.

Let's think in terms of versions i.e. for each change in our segment tree we create a new version of it.

We will consider our initial version to be Version-0. Now, as we do any update in the segment tree we will create a new version for it and in similar fashion track the record for all versions.

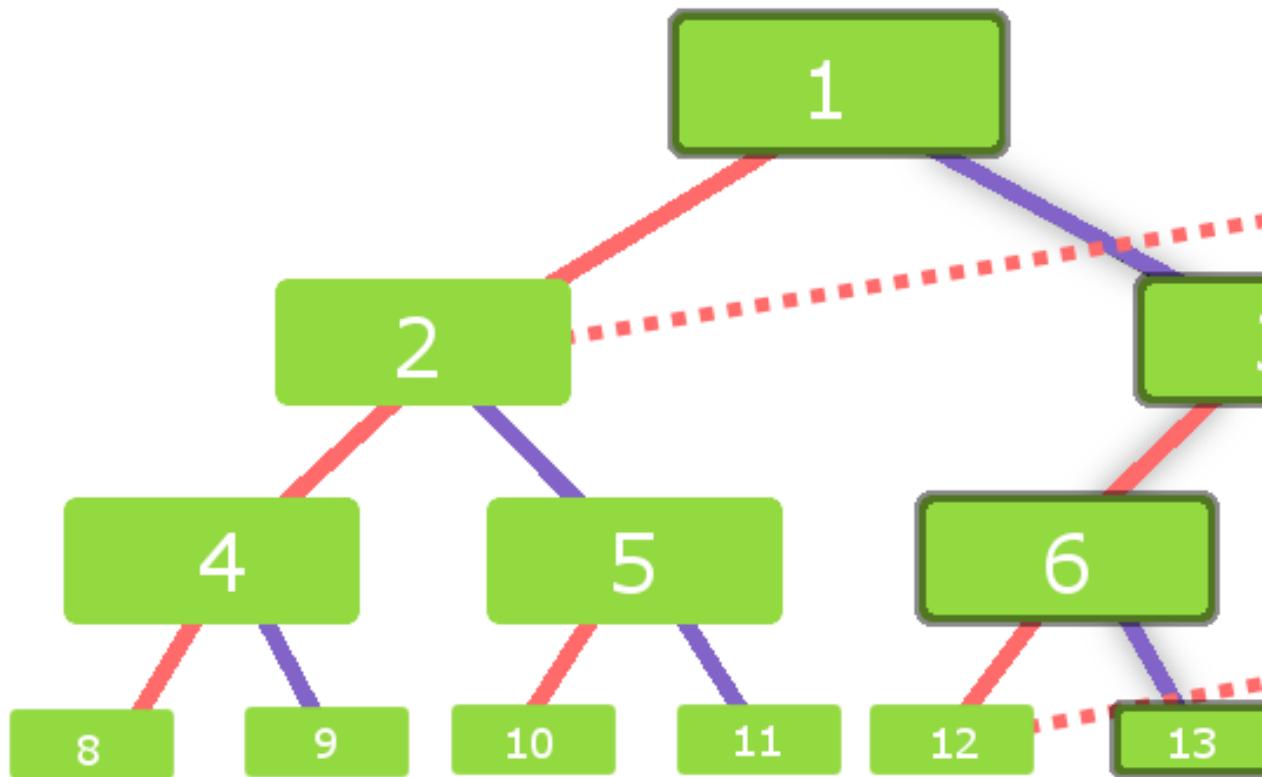
But creating the whole tree for every version will take $O(n \log n)$ extra space and $O(n \log n)$ time. So, this idea runs out of time and memory for large number of versions.

Let's exploit the fact that for each new update(say point update for simplicity) in segment tree, At max $\log n$ nodes will be modified. So, our new version will only contain these $\log n$ new nodes and rest nodes will be the same as previous version. Therefore, it is quite clear that for each new version we only need to create these $\log n$ new nodes whereas the rest of nodes can be shared from the previous version.

Consider the below figure for better visualization(click on the image for better view) :-

Persistency in

Version 0



Consider the segment tree with green nodes . Lets call this segment tree as **version-0**. The left child for each node is connected with solid red edge where as the right child for each node is connected with solid purple edge. Clearly, this segment tree consists of 15 nodes.

Now consider we need to make change in the leaf node 13 of version-0.

So, the affected nodes will be – **node 13 , node 6 , node 3 , node 1**.

Therefore, for the new version (**Version-1**) we need to create only these **4 new nodes**.

Now, lets construct version-1 for this change in segment tree. We need a new node 1 as it is affected by change done in node 13. So , we will first create a new **node 1** (yellow color) . The left child for node 1 will be the same for left child for node 1 in version-0. So, we connect the left child of node 1 with node 2 of version-0(red dashed line in figure). Let's now examine the right child for node 1 in version-1. We need to create a new node as it is affected . So we create a new node called node 3 and make it the right child for node 1 (solid purple edge connection).

In the similar fashion we will now examine for **node 3** . The left child is affected , So we create a new node called **node 6** and connect it with solid red edge with node 3 , where as the right child for node 3 will be the same as right child of node 3 in version-0. So, we will make the right child of node 3 in version-0 as the right child of node 3 in version-1(see the purple dash edge.)

Same procedure is done for node 6 and we see that the left child of node 6 will be the left child of node 6 in version-0(red dashed connection) and right child is newly created node called **node 13** (solid purple dashed edge).

Each **yellow color node** is a newly created node and dashed edges are the inter-connection between the different versions of the segment tree.

Now, the Question arises : **How to keep track of all the versions?**

– We only need to keep track the first root node for all the versions and this will serve the purpose to track all the newly created nodes in the different versions. For this purpose we can maintain an array of pointers to the first node of segment trees for all versions.

Let's consider a very basic problem to see how to implement persistence in segment tree

Problem : Given an array A[] and different point update operations.Considering each point operation to create a new version of the array. We need to answer the queries of type
Q v l r : output the sum of elements in range l to r just after the v-th update.

We will create all the versions of the segment tree and keep track of their root node.Then for each range sum query we will pass the required version's root node in our query function and output the required sum.

Below is the C++ implementation for the above problem:-

```
// C++ program to implement persistent segment
// tree.
#include <bits/stdc++.h>
using namespace std;

#define MAXN 100

/* data type for individual
```

```

 * node in the segment tree */
struct node
{
    // stores sum of the elements in node
    int val;

    // pointer to left and right children
    node* left, *right;

    // required constructors.....
    node() {}
    node(node* l, node* r, int v)
    {
        left = l;
        right = r;
        val = v;
    }
};

// input array
int arr[MAXN];

// root pointers for all versions
node* version[MAXN];

// Constructs Version-0
// Time Complexity : O(nlogn)
void build(node* n,int low,int high)
{
    if (low==high)
    {
        n->val = arr[low];
        return;
    }
    int mid = (low+high) / 2;
    n->left = new node(NULL, NULL, 0);
    n->right = new node(NULL, NULL, 0);
    build(n->left, low, mid);
    build(n->right, mid+1, high);
    n->val = n->left->val + n->right->val;
}

/**
 * Upgrades to new Version
 * @param prev : points to node of previous version
 * @param cur : points to node of current version
 * Time Complexity : O(logn)
 * Space Complexity : O(logn) */

```

```

void upgrade(node* prev, node* cur, int low, int high,
            int idx, int value)
{
    if (idx > high or idx < low or low > high)
        return;

    if (low == high)
    {
        // modification in new version
        cur->val = value;
        return;
    }
    int mid = (low+high) / 2;
    if (idx <= mid)
    {
        // link to right child of previous version
        cur->right = prev->right;

        // create new node in current version
        cur->left = new node(NULL, NULL, 0);

        upgrade(prev->left, cur->left, low, mid, idx, value);
    }
    else
    {
        // link to left child of previous version
        cur->left = prev->left;

        // create new node for current version
        cur->right = new node(NULL, NULL, 0);

        upgrade(prev->right, cur->right, mid+1, high, idx, value);
    }

    // calculating data for current version
    // by combining previous version and current
    // modification
    cur->val = cur->left->val + cur->right->val;
}

int query(node* n, int low, int high, int l, int r)
{
    if (l > high or r < low or low > high)
        return 0;
    if (l <= low and high <= r)
        return n->val;
    int mid = (low+high) / 2;
    int p1 = query(n->left, low, mid, l, r);

```

```

        int p2 = query(n->right,mid+1,high,l,r);
        return p1+p2;
    }

int main(int argc, char const *argv[])
{
    int A[] = {1,2,3,4,5};
    int n = sizeof(A)/sizeof(int);

    for (int i=0; i<n; i++)
        arr[i] = A[i];

    // creating Version-0
    node* root = new node(NULL, NULL, 0);
    build(root, 0, n-1);

    // storing root node for version-0
    version[0] = root;

    // upgrading to version-1
    version[1] = new node(NULL, NULL, 0);
    upgrade(version[0], version[1], 0, n-1, 4, 1);

    // upgrading to version-2
    version[2] = new node(NULL, NULL, 0);
    upgrade(version[1],version[2], 0, n-1, 2, 10);

    cout << "In version 1 , query(0,4) : ";
    cout << query(version[1], 0, n-1, 0, 4) << endl;

    cout << "In version 2 , query(3,4) : ";
    cout << query(version[2], 0, n-1, 3, 4) << endl;

    cout << "In version 0 , query(0,3) : ";
    cout << query(version[0], 0, n-1, 0, 3) << endl;
    return 0;
}

```

Output:

```

In version 1 , query(0,4) : 11
In version 2 , query(3,4) : 5
In version 0 , query(0,3) : 10

```

Note : The above problem can also be solved by processing the queries offline by sorting it with respect to the version and answering the queries just after the corresponding update.

Time Complexity : The time complexity will be the same as the query and point update operation in the segment tree as we can consider the extra node creation step to be done in $O(1)$. Hence, the overall Time Complexity per query for new version creation and range sum query will be $O(\log n)$.

Source

<https://www.geeksforgeeks.org/persistent-segment-tree-set-1-introduction/>

Chapter 127

Persistent data structures

Persistent data structures - GeeksforGeeks

All the data structures discussed here so far are non-persistent (or ephemeral). A persistent data structure is a data structure that always preserves the previous version of itself when it is modified. They can be considered as ‘immutable’ as updates are not in-place.

A data structure is **partially persistent** if all versions can be accessed but only the newest version can be modified. **Fully persistent** if every version can be both accessed and modified. Confluently persistent is when we merge two or more versions to get a new version. This induces a DAG on the version graph.

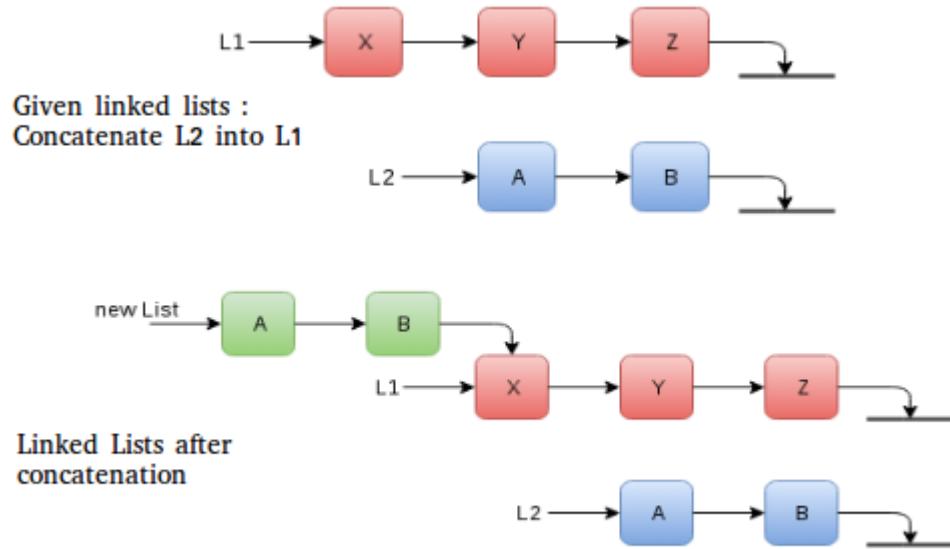
Persistence can be achieved by simply copying, but this is inefficient in CPU and RAM usage as most operations will make only a small change in the DS. Therefore, a better method is to exploit the similarity between the new and old versions to share structure between them.

Examples:

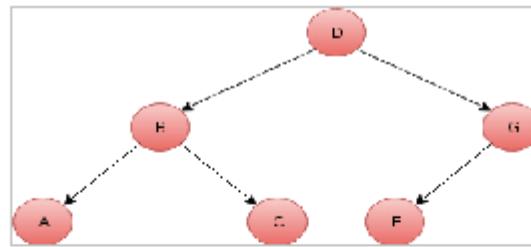
1. **Linked List Concatenation :** Consider the problem of concatenating two singly linked lists with n and m as the number of nodes in them. Say $n > m$. We need to keep the versions, i.e., we should be able to original list.

One way is to make a copy of every node and do the connections. $O(n + m)$ for traversal of lists and $O(1)$ each for adding $(n + m - 1)$ connections.

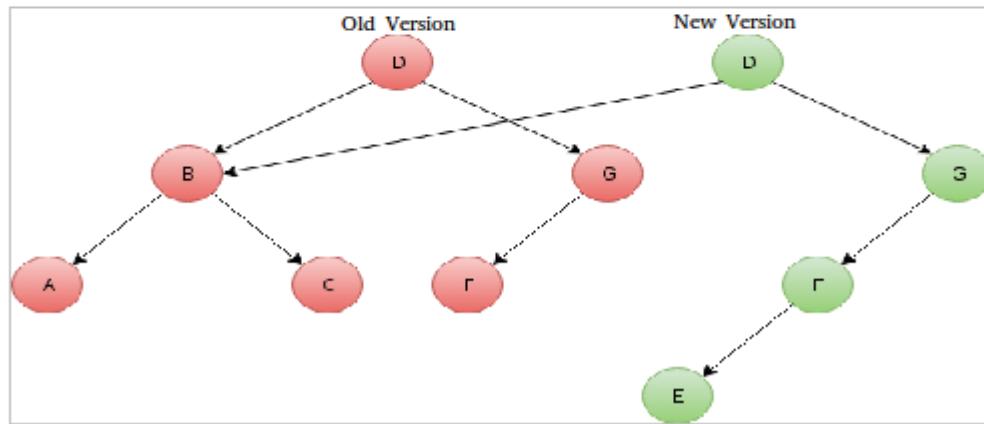
Other way, and a more efficient way in time and space, involves traversal of only one of the two lists and fewer new connections. Since we assume $m < n$, we can pick list with m nodes to be copied. This means $O(m)$ for traversal and $O(1)$ for each one of the (m) connections. We must copy it otherwise the original form of list wouldn’t have persisted.



2. **Binary Search Tree Insertion :** Consider the problem of insertion of a new node in a binary search tree. Being a binary search tree, there is a specific location where the new node will be placed. All the nodes in the path from the new node to the root of the BST will observe a change in structure (cascading). For example, the node for which the new node is the child will now have a new pointer. This change in structure induces change in the complete path up to the root. Consider tree below with value for node listed inside each one of them.



Let's add node with value 'E' to this tree.



Approaches to make data structures persistent

For the methods suggested below, updates and access time and space of versions vary with whether we are implementing full or partial persistence.

1. **Path copying:** Make a copy of the node we are about to update. Then proceed for update. And finally, cascade the change back through the data structure, something very similar to what we did in example two above. This causes a chain of updates, until you reach a node no other node points to — the root.
How to access the state at time t? Maintain an array of roots indexed by timestamp.
2. **Fat nodes:** As the name suggests, we make every node store its modification history, thereby making it 'fat'.
3. **Nodes with boxes:** Amortized O(1) time and space can be achieved for access and updates. This method was given by Sleator, Tarjan and their team. For a tree, it involves using a modification box that can hold:
 - a. one modification to the node (the modification could be of one of the pointers, or to the node's key or to some other node-specific data)
 - b. the time when the mod was applied
 The timestamp is crucial for reaching to the version of the node we care about. One can read more about it [here](#). With this algorithm, given any time t, at most one

modification box exists in the data structure with time t . Thus, a modification at time t splits the tree into three parts: one part contains the data from before time t , one part contains the data from after time t , and one part was unaffected by the modification (Source : [MIT OCW](#)). Non-tree data structures may require more than one modification box, but limited to in-degree of the node for amortized $O(1)$.

Useful Links:

1. [MIT OCW](#) (Erik Demaine)
2. [MIT OCW](#) (David Karger)
3. [Dann Toliver's talk](#)
4. [Wiki Page](#)

Persistent Segment Tree Set 1 (Introduction)

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

Source

<https://www.geeksforgeeks.org/persistent-data-structures/>

Chapter 128

Print Kth character in sorted concatenated substrings of a string

Print Kth character in sorted concatenated substrings of a string - GeeksforGeeks

Given a string of lower alphabetic characters, find K-th character in a string formed by substrings (of given string) when concatenated in sorted form.

Examples:

```
Input : str = "banana"
        K = 10
Output : n
All substring in sorted form are,
"a", "an", "ana", "anan", "anana",
"b", "ba", "ban", "bana", "banan",
"banana", "n", "na", "nan", "nana"
Concatenated string = "aananaanana
nanabbabanbanabanbananannanannana"
We can see a 10th character in the
above concatenated string is 'n'
which is our final answer.
```

A **simple solution** is to generate all substrings of a given string and store them in an array. Once substrings are generated, sort them and concatenate after sorting. Finally print K-th character in the concatenated string.

An **efficient solution** is based on [counting distinct substring of a string using suffix array](#). Same method is used in solving this problem also. After getting suffix array and lcp array, we loop over all lcp values and for each such value, we calculate characters to skip. We keep

subtracting these many characters from our K, when character to skip becomes more than K, we stop and loop over substrings corresponding to current lcp[i], in which we loop from lcp[i] till the maximum length of string and then print the Kth character.

```
// C/C++ program to print Kth character
// in sorted concatenated substrings
#include <bits/stdc++.h>
using namespace std;

// Structure to store information of a suffix
struct suffix
{
    int index; // To store original index
    int rank[2]; // To store ranks and next
                 // rank pair
};

// A comparison function used by sort() to compare
// two suffixes. Compares two pairs, returns 1 if
// first pair is smaller
int cmp(struct suffix a, struct suffix b)
{
    return (a.rank[0] == b.rank[0])?
           (a.rank[1] < b.rank[1] ?1: 0):
           (a.rank[0] < b.rank[0] ?1: 0);
}

// This is the main function that takes a string
// 'txt' of size n as an argument, builds and return
// the suffix array for the given string
vector<int> buildSuffixArray(string txt, int n)
{
    // A structure to store suffixes and their indexes
    struct suffix suffixes[n];

    // Store suffixes and their indexes in an array
    // of structures. The structure is needed to sort
    // the suffixes alphabetically and maintain their
    // old indexes while sorting
    for (int i = 0; i < n; i++)
    {
        suffixes[i].index = i;
        suffixes[i].rank[0] = txt[i] - 'a';
        suffixes[i].rank[1] = ((i+1) < n)?
                           (txt[i + 1] - 'a'): -1;
    }

    // Sort the suffixes using the comparison function
```

```
// defined above.
sort(suffixes, suffixes+n, cmp);

// At this point, all suffixes are sorted according
// to first 2 characters. Let us sort suffixes
// according to first 4 characters, then first
// 8 and so on
int ind[n]; // This array is needed to get the
             // index in suffixes[] from original
             // index. This mapping is needed to get
             // next suffix.

for (int k = 4; k < 2*n; k = k*2)
{
    // Assigning rank and index values to first suffix
    int rank = 0;
    int prev_rank = suffixes[0].rank[0];
    suffixes[0].rank[0] = rank;
    ind[suffixes[0].index] = 0;

    // Assigning rank to suffixes
    for (int i = 1; i < n; i++)
    {
        // If first rank and next ranks are same as
        // that of previous suffix in array, assign
        // the same new rank to this suffix
        if (suffixes[i].rank[0] == prev_rank &&
            suffixes[i].rank[1] == suffixes[i-1].rank[1])
        {
            prev_rank = suffixes[i].rank[0];
            suffixes[i].rank[0] = rank;
        }
        else // Otherwise increment rank and assign
        {
            prev_rank = suffixes[i].rank[0];
            suffixes[i].rank[0] = ++rank;
        }
        ind[suffixes[i].index] = i;
    }

    // Assign next rank to every suffix
    for (int i = 0; i < n; i++)
    {
        int nextindex = suffixes[i].index + k/2;
        suffixes[i].rank[1] = (nextindex < n)?
                           suffixes[ind[nextindex]].rank[0]: -1;
    }
}
```

```
// Sort the suffixes according to first k characters
sort(suffixes, suffixes+n, cmp);
}

// Store indexes of all sorted suffixes in the suffix
// array
vector<int>suffixArr;
for (int i = 0; i < n; i++)
    suffixArr.push_back(suffixes[i].index);

// Return the suffix array
return suffixArr;
}

/* To construct and return LCP */
vector<int> kasai(string txt, vector<int> suffixArr)
{
    int n = suffixArr.size();

    // To store LCP array
    vector<int> lcp(n, 0);

    // An auxiliary array to store inverse of suffix array
    // elements. For example if suffixArr[0] is 5, the
    // invSuff[5] would store 0. This is used to get next
    // suffix string from suffix array.
    vector<int> invSuff(n, 0);

    // Fill values in invSuff[]
    for (int i=0; i < n; i++)
        invSuff[suffixArr[i]] = i;

    // Initialize length of previous LCP
    int k = 0;

    // Process all suffixes one by one starting from
    // first suffix in txt[]
    for (int i=0; i<n; i++)
    {
        /* If the current suffix is at n-1, then we don't
           have next substring to consider. So lcp is not
           defined for this substring, we put zero. */
        if (invSuff[i] == n-1)
        {
            k = 0;
            continue;
        }
        ...
```

```

/* j contains index of the next substring to
   be considered to compare with the present
   substring, i.e., next string in suffix array */
int j = suffixArr[invSuff[i]+1];

// Directly start matching from k'th index as
// at-least k-1 characters will match
while (i+k<n && j+k<n && txt[i+k]==txt[j+k])
    k++;

lcp[invSuff[i]] = k; // lcp for the present suffix.

// Deleting the starting character from the string.
if (k>0)
    k--;
}

// return the constructed lcp array
return lcp;
}

// Utility method to get sum of first N numbers
int sumOfFirstN(int N)
{
    return (N * (N + 1)) / 2;
}

// Returns Kth character in sorted concatenated
// substrings of str
char printKthCharInConcatSubstring(string str, int K)
{
    int n = str.length();
    // calculating suffix array and lcp array
    vector<int> suffixArr = buildSuffixArray(str, n);
    vector<int> lcp = kasai(str, suffixArr);

    for (int i = 0; i < lcp.size(); i++)
    {
        // skipping characters common to substring
        // (n - suffixArr[i]) is length of current
        // maximum substring lcp[i] will length of
        // common substring
        int charToSkip = sumOfFirstN(n - suffixArr[i]) -
                        sumOfFirstN(lcp[i]);

        /* if characters are more than K, that means
           Kth character belongs to substring

```

```
corresponding to current lcp[i]*/
if (K <= charToSkip)
{
    // loop from current lcp value to current
    // string length
    for (int j = lcp[i] + 1; j <= (n-suffixArr[i]); j++)
    {
        int curSubstringLen = j;

        /* Again reduce K by current substring's
           length one by one and when it becomes less,
           print Kth character of current susbtring */
        if (K <= curSubstringLen)
            return str[(suffixArr[i] + K - 1)];
        else
            K -= curSubstringLen;

    }
    break;
}
else
    K -= charToSkip;
}

// Driver code to test above methods
int main()
{
    string str = "banana";
    int K = 10;
    cout << printKthCharInConcatSubstring(str, K);
    return 0;
}
```

Output:

n

Source

<https://www.geeksforgeeks.org/print-kth-character-sorted-concatenated-substrings-string/>

Chapter 129

Print all valid words that are possible using Characters of Array

Print all valid words that are possible using Characters of Array - GeeksforGeeks

Given a dictionary and a character array, print all valid words that are possible using characters from the array.

Examples:

```
Input : Dict - {"go","bat","me","eat","goal",
                 "boy", "run"}
        arr[] = {'e','o','b', 'a','m','g', 'l'}
Output : go, me, goal.
```

Asked In : Microsoft Interview

The idea is to use Trie data structure to store dictionary, then search words in Trie using characters of given array.

1. Create an empty Trie and insert all words of given dictionary into the Trie.
2. After that, we have pick only those characters in ‘Arr[]’ which are a child of the root of Trie.
3. To quickly find whether a character is present in array or not, we create a hash of character arrays.

Below is c++ implementation of above idea

C++

```
// C++ program to print all valid words that
// are possible using character of array
#include<bits/stdc++.h>
using namespace std;

// Converts key current character into index
// use only 'a' through 'z'
#define char_int(c) ((int)c - (int)'a')

// converts current integer into character
#define int_to_char(c) ((char)c + (char)'a')

// Alphabet size
#define SIZE (26)

// trie Node
struct TrieNode
{
    TrieNode *Child[SIZE];

    // isLeaf is true if the node represents
    // end of a word
    bool leaf;
};

// Returns new trie node (initialized to NULLs)
TrieNode *getNode()
{
    TrieNode *newNode = new TrieNode;
    newNode->leaf = false;
    for (int i = 0 ; i < SIZE ; i++)
        newNode->Child[i] = NULL;
    return newNode;
}

// If not present, inserts key into trie
// If the key is prefix of trie node, just
// marks leaf node
void insert(TrieNode *root, char *Key)
{
    int n = strlen(Key);
    TrieNode * pChild = root;

    for (int i=0; i < n; i++)
    {
        int index = char_int(Key[i]);

        if (pChild->Child[index] == NULL)
```

```
pChild->Child[index] = getNode();

pChild = pChild->Child[index];
}

// make last node as leaf node
pChild->leaf = true;
}

// A recursive function to print all possible valid
// words present in array
void searchWord(TrieNode *root, bool Hash[], string str)
{
    // if we found word in trie / dictionary
    if (root->leaf == true)
        cout << str << endl;

    // traverse all child's of current root
    for (int K =0; K < SIZE; K++)
    {
        if (Hash[K] == true && root->Child[K] != NULL )
        {
            // add current character
            char c = int_to_char(K);

            // Recursively search reaming character of word
            // in trie
            searchWord(root->Child[K], Hash, str + c);
        }
    }
}

// Prints all words present in dictionary.
void PrintAllWords(char Arr[], TrieNode *root, int n)
{
    // create a 'has' array that will store all present
    // character in Arr[]
    bool Hash[SIZE];

    for (int i = 0 ; i < n; i++)
        Hash[char_int(Arr[i])] = true;

    // temporary node
    TrieNode *pChild = root ;

    // string to hold output words
    string str = "";
```

```
// Traverse all matrix elements. There are only 26
// character possible in char array
for (int i = 0 ; i < SIZE ; i++)
{
    // we start searching for word in dictionary
    // if we found a character which is child
    // of Trie root
    if (Hash[i] == true && pChild->Child[i] )
    {
        str = str+(char)int_to_char(i);
        searchWord(pChild->Child[i], Hash, str);
        str = "";
    }
}
}

//Driver program to test above function
int main()
{
    // Let the given dictionary be following
    char Dict[] [20] = {"go", "bat", "me", "eat",
                        "goal", "boy", "run"} ;

    // Root Node of Trie
    TrieNode *root = getNode();

    // insert all words of dictionary into trie
    int n = sizeof(Dict)/sizeof(Dict[0]);
    for (int i=0; i<n; i++)
        insert(root, Dict[i]);

    char arr[] = {'e', 'o', 'b', 'a', 'm', 'g', 'l'} ;
    int N = sizeof(arr)/sizeof(arr[0]);

    PrintAllWords(arr, root, N);

    return 0;
}
```

Java

```
// Java program to print all valid words that
// are possible using character of array
public class SearchDict_charArray {

    // Alphabet size
    static final int SIZE = 26;
```

```
// trie Node
static class TrieNode
{
    TrieNode[] Child = new TrieNode[SIZE];

    // isLeaf is true if the node represents
    // end of a word
    boolean leaf;

    // Constructor
    public TrieNode() {
        leaf = false;
        for (int i = 0 ; i < SIZE ; i++)
            Child[i] = null;
    }
}

// If not present, inserts key into trie
// If the key is prefix of trie node, just
// marks leaf node
static void insert(TrieNode root, String Key)
{
    int n = Key.length();
    TrieNode pChild = root;

    for (int i=0; i<n; i++)
    {
        int index = Key.charAt(i) - 'a';

        if (pChild.Child[index] == null)
            pChild.Child[index] = new TrieNode();

        pChild = pChild.Child[index];
    }

    // make last node as leaf node
    pChild.leaf = true;
}

// A recursive function to print all possible valid
// words present in array
static void searchWord(TrieNode root, boolean Hash[],
                      String str)
{
    // if we found word in trie / dictionary
    if (root.leaf == true)
        System.out.println(str);
}
```

```
// traverse all child's of current root
for (int K =0; K < SIZE; K++)
{
    if (Hash[K] == true && root.Child[K] != null )
    {
        // add current character
        char c = (char) (K + 'a');

        // Recursively search reaming character
        // of word in trie
        searchWord(root.Child[K], Hash, str + c);
    }
}

// Prints all words present in dictionary.
static void PrintAllWords(char Arr[], TrieNode root,
                           int n)
{
    // create a 'has' array that will store all
    // present character in Arr[]
    boolean[] Hash = new boolean[SIZE];

    for (int i = 0 ; i < n; i++)
        Hash[Arr[i] - 'a'] = true;

    // temporary node
    TrieNode pChild = root ;

    // string to hold output words
    String str = "";

    // Traverse all matrix elements. There are only
    // 26 character possible in char array
    for (int i = 0 ; i < SIZE ; i++)
    {
        // we start searching for word in dictionary
        // if we found a character which is child
        // of Trie root
        if (Hash[i] == true && pChild.Child[i] != null )
        {
            str = str+(char)(i + 'a');
            searchWord(pChild.Child[i], Hash, str);
            str = "";
        }
    }
}
```

```
//Driver program to test above function
public static void main(String args[])
{
    // Let the given dictionary be following
    String Dict[] = {"go", "bat", "me", "eat",
                     "goal", "boy", "run"} ;

    // Root Node of Trie
    TrieNode root = new TrieNode();

    // insert all words of dictionary into trie
    int n = Dict.length;
    for (int i=0; i<n; i++)
        insert(root, Dict[i]);

    char arr[] = {'e', 'o', 'b', 'a', 'm', 'g', 'l'} ;
    int N = arr.length;

    PrintAllWords(arr, root, N);
}
}

// This code is contributed by Sumit Ghosh
```

Output:

```
go
goal
me
```

Source

<https://www.geeksforgeeks.org/print-valid-words-possible-using-characters-array/>

Chapter 130

Print all words matching a pattern in CamelCase Notation Dictionary

Print all words matching a pattern in CamelCase Notation Dictionary - GeeksforGeeks

Given a dictionary of words where each word follows CamelCase notation, print all words in the dictionary that match with a given pattern consisting of uppercase characters only.

CamelCase is the practice of writing compound words or phrases such that each word or abbreviation begins with a capital letter. Common examples include: “PowerPoint” and “WikiPedia”, “GeeksForGeeks”, “CodeBlocks”, etc.

Examples:

Input:

```
dict[] = ["Hi", "Hello", "HelloWorld", "HiTech", "HiGeek",
"HiTechWorld", "HiTechCity", "HiTechLab"]
```

For pattern "HT",

Output: ["HiTech", "HiTechWorld", "HiTechCity", "HiTechLab"]

For pattern "H",

Output: ["Hi", "Hello", "HelloWorld", "HiTech", "HiGeek",
"HiTechWorld", "HiTechCity", "HiTechLab"]

For pattern "HTC",

Output: ["HiTechCity"]

Input:

```
dict[] = ["WelcomeGeek", "WelcomeToGeeksForGeeks", "GeeksForGeeks"]
```

```
For pattern "WTG",
Output: ["WelcomeToGeeksForGeeks"]
```

```
For pattern "GFG",
Output: [GeeksForGeeks]
```

```
For pattern "GG",
Output: No match found
```

The idea is to insert all dictionary keys into the Trie one by one. Here key refers to only Uppercase characters in original word in CamelCase notation. If we encounter the key for the first time, we need to mark the last node as leaf node and insert the complete word for that key into the vector associated with the leaf node. If we encounter a key that is already in the trie, we update the vector associated with the leaf node with current word. After all dictionary words are processed, we search for the pattern in the trie and print all words that matches the pattern.

Below is C++ implementation of above idea –

C++

```
// C++ program to print all words in the CamelCase
// dictionary that matches with a given pattern
#include <bits/stdc++.h>
using namespace std;

// Alphabet size (# of upper-Case characters)
#define ALPHABET_SIZE 26

// A Trie node
struct TrieNode
{
    TrieNode* children[ALPHABET_SIZE];

    // isLeaf is true if the node represents
    // end of a word
    bool isLeaf;

    // vector to store list of complete words
    // in leaf node
    list<string> word;
};

// Returns new Trie node (initialized to NULLs)
TrieNode* getNewTrieNode(void)
{
    TrieNode* pNode = new TrieNode;
```

```
if (pNode)
{
    pNode->isLeaf = false;

    for (int i = 0; i < ALPHABET_SIZE; i++)
        pNode->children[i] = NULL;
}

return pNode;
}

// Function to insert word into the Trie
void insert(TrieNode* root, string word)
{
    int index;
    TrieNode* pCrawl = root;

    for (int level = 0; level < word.length(); level++)
    {
        // consider only uppercase characters
        if (islower(word[level]))
            continue;

        // get current character position
        index = int(word[level]) - 'A';
        if (!pCrawl->children[index])
            pCrawl->children[index] = getNewTrieNode();

        pCrawl = pCrawl->children[index];
    }

    // mark last node as leaf
    pCrawl->isLeaf = true;

    // push word into vector associated with leaf node
    (pCrawl->word).push_back(word);
}

// Function to print all children of Trie node root
void printAllWords(TrieNode* root)
{
    // if current node is leaf
    if (root->isLeaf)
    {
        for(string str : root->word)
            cout << str << endl;
    }
}
```

```
// recurse for all children of root node
for (int i = 0; i < ALPHABET_SIZE; i++)
{
    TrieNode* child = root->children[i];
    if (child)
        printAllWords(child);
}
}

// search for pattern in Trie and print all words
// matching that pattern
bool search(TrieNode* root, string pattern)
{
    int index;
    TrieNode* pCrawl = root;

    for (int level = 0; level < pattern.length(); level++)
    {
        index = int(pattern[level]) - 'A';
        // Invalid pattern
        if (!pCrawl->children[index])
            return false;

        pCrawl = pCrawl->children[index];
    }

    // print all words matching that pattern
    printAllWords(pCrawl);

    return true;
}

// Main function to print all words in the CamelCase
// dictionary that matches with a given pattern
void findAllWords(vector<string> dict, string pattern)
{
    // construct Trie root node
    TrieNode* root = getNewTrieNode();

    // Construct Trie from given dict
    for (string word : dict)
        insert(root, word);

    // search for pattern in Trie
    if (!search(root, pattern))
        cout << "No match found";
}
```

```
// Driver function
int main()
{
    // dictionary of words where each word follows
    // CamelCase notation
    vector<string> dict = {
        "Hi", "Hello", "HelloWorld", "HiTech", "HiGeek",
        "HiTechWorld", "HiTechCity", "HiTechLab"
    };

    // pattern consisting of uppercase characters only
    string pattern = "HT";

    findAllWords(dict, pattern);

    return 0;
}
```

Java

```
// Java program to print all words in the CamelCase
// dictionary that matches with a given pattern
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
public class CamelCase {

    // Alphabet size (# of upper-Case characters)
    static final int ALPHABET_SIZE = 26;

    // A Trie node
    static class TrieNode {
        TrieNode[] children = new TrieNode[ALPHABET_SIZE];

        // isLeaf is true if the node represents
        // end of a word
        boolean isLeaf;

        // vector to store list of complete words
        // in leaf node
        List<String> word;

        public TrieNode() {
            isLeaf = false;
            for (int i = 0; i < ALPHABET_SIZE; i++)
                children[i] = null;
        }
    }

    // Function to print all words in the dictionary
    // which matches with the given pattern
    void findAllWords(List<String> dict, String pattern) {
        TrieNode root = new TrieNode();
        for (String word : dict) {
            TrieNode node = root;
            for (int i = 0; i < word.length(); i++) {
                int index = word.charAt(i) - 'A';
                if (node.children[index] == null) {
                    node.children[index] = new TrieNode();
                }
                node = node.children[index];
            }
            node.isLeaf = true;
        }

        // Print words starting from root
        printWords(root, pattern);
    }

    // Prints words starting from current node
    void printWords(TrieNode node, String pattern) {
        if (node.isLeaf) {
            System.out.println(pattern);
        }
        for (int i = 0; i < ALPHABET_SIZE; i++) {
            if (node.children[i] != null) {
                printWords(node.children[i], pattern + (char)(i + 'A'));
            }
        }
    }
}
```

```
        word = new ArrayList<String>();
    }
}

static TrieNode root;

// Function to insert word into the Trie
static void insert(String word) {
    int index;
    TrieNode pCrawl = root;

    for (int level = 0; level < word.length(); level++) {

        // consider only uppercase characters
        if (Character.isLowerCase(word.charAt(level)))
            continue;

        // get current character position
        index = word.charAt(level) - 'A';
        if (pCrawl.children[index] == null)
            pCrawl.children[index] = new TrieNode();

        pCrawl = pCrawl.children[index];
    }

    // mark last node as leaf
    pCrawl.isLeaf = true;

    // push word into vector associated with leaf node
    (pCrawl.word).add(word);
}

// Function to print all children of Trie node root
static void printAllWords(TrieNode root) {

    // if current node is leaf
    if (root.isLeaf) {
        for (String str : root.word)
            System.out.println(str);
    }

    // recurse for all children of root node
    for (int i = 0; i < ALPHABET_SIZE; i++) {
        TrieNode child = root.children[i];
        if (child != null)
            printAllWords(child);
    }
}
```

```
// search for pattern in Trie and print all words
// matching that pattern
static boolean search(String pattern) {
    int index;
    TrieNode pCrawl = root;

    for (int level = 0; level < pattern.length(); level++) {
        index = pattern.charAt(level) - 'A';

        // Invalid pattern
        if (pCrawl.children[index] == null)
            return false;

        pCrawl = pCrawl.children[index];
    }

    // print all words matching that pattern
    printAllWords(pCrawl);

    return true;
}

// Main function to print all words in the CamelCase
// dictionary that matches with a given pattern
static void findAllWords(List<String> dict, String pattern)
{
    // construct Trie root node
    root = new TrieNode();

    // Construct Trie from given dict
    for (String word : dict)
        insert(word);

    // search for pattern in Trie
    if (!search(pattern))
        System.out.println("No match found");
}

// Driver function
public static void main(String args[]) {

    // dictionary of words where each word follows
    // CamelCase notation
    List<String> dict = Arrays.asList("Hi", "Hello",
                                      "HelloWorld", "HiTech", "HiGeek",
                                      "HiTechWorld", "HiTechCity",
                                      "HiTechCity", "HiTechWorld");
}
```

```
        "HiTechLab");  
  
    // pattern consisting of uppercase characters only  
    String pattern = "HT";  
  
    findAllWords(dict, pattern);  
}  
}  
// This code is contributed by Sumit Ghosh
```

Output:

```
HiTech  
HiTechCity  
HiTechLab  
HiTechWorld
```

Source

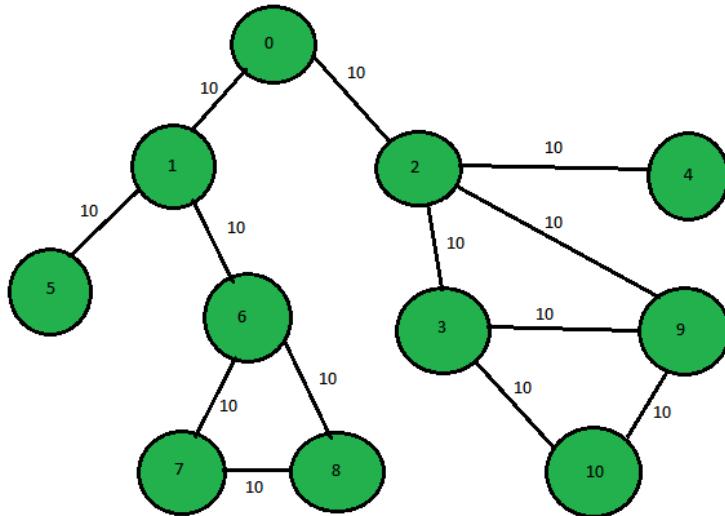
<https://www.geeksforgeeks.org/print-words-matching-pattern-camelcase-notation-dictionary/>

Chapter 131

Print the DFS traversal step-wise (Backtracking also)

Print the DFS traversal step-wise (Backtracking also) - GeeksforGeeks

Given a [graph](#), the task is to print the DFS traversal of a graph which includes the every step including the backtracking.

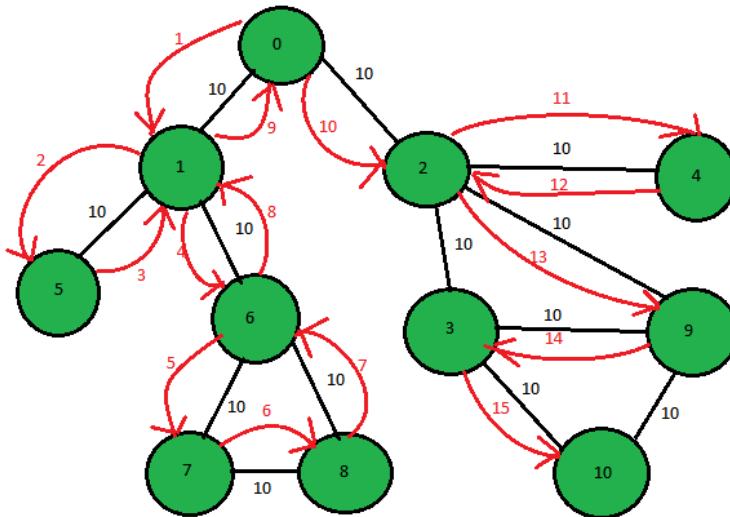


1st step:- 0 → 1
2nd step:- 1 → 5
3rd step:- 5 → 1 (backtracking step)
4th step:- 1 → 6...
and so on till all the nodes are visited.

Dfs step-wise(including backtracking) is:

0 1 5 1 6 7 8 7 6 1 0 2 4 2 9 3 10

Note: In this above diagram the weight between the edges has just been added, it does not have any role in DFS-traversal



Approach: **DFS with Backtracking** will be used here. First, visit every node using DFS simultaneously and keep track of the previously used edge and the parent node. If a node comes whose all the adjacent node has been visited, backtrack using the last used edge and print the nodes. Continue the steps and at every step, the parent node will become the present node. Continue the above steps to find the complete DFS traversal of the graph.

Below is the implementation of the above approach:

```
// C++ program to print the complete
// DFS-traversal of graph
// using back-tracking
#include <bits/stdc++.h>
using namespace std;
const int N = 1000;
vector<int> adj[N];

// Function to print the complete DFS-traversal
void dfsUtil(int u, int node, bool visited[],
             vector<pair<int, int> > road_used, int parent, int it)
{
    int c = 0;
```

```
// Check if all th node is visited or not
// and count unvisited nodes
for (int i = 0; i < node; i++)
    if (visited[i])
        c++;

// If all the node is visited return;
if (c == node)
    return;

// Mark not visited node as visited
visited[u] = true;

// Track the current edge
road_used.push_back({ parent, u });

// Print the node
cout << u << " ";

// Check for not visited node and proceed with it.
for (int x : adj[u]) {

    // call the DFs function if not visited
    if (!visited[x])
        dfsUtil(x, node, visited, road_used, u, it + 1);
}

// Backtrack through the last
// visited nodes
for (auto y : road_used)
    if (y.second == u)
        dfsUtil(y.first, node, visited,
                road_used, u, it + 1);
}

// Function to call the DFS function
// which prints the DFS-travesal stepwise
void dfs(int node)
{

    // Create a array of visited ndoe
    bool visited[node];

    // Vector to track last visited road
    vector<pair<int, int> > road_used;

    // Initialize all the node with false
    for (int i = 0; i < node; i++)
```

```
visited[i] = false;

// call the function
dfsUtil(0, node, visited, road_used, -1, 0);
}

// Function to insert edges in Graph
void insertEdge(int u, int v)
{
    adj[u].push_back(v);
    adj[v].push_back(u);
}

// Driver Code
int main()
{
    // number of nodes and edges in the graph
    int node = 11, edge = 13;

    // Function call to create the graph
    insertEdge(0, 1);
    insertEdge(0, 2);
    insertEdge(1, 5);
    insertEdge(1, 6);
    insertEdge(2, 4);
    insertEdge(2, 9);
    insertEdge(6, 7);
    insertEdge(6, 8);
    insertEdge(7, 8);
    insertEdge(2, 3);
    insertEdge(3, 9);
    insertEdge(3, 10);
    insertEdge(9, 10);

    // Call the function to print
    dfs(node);

    return 0;
}
```

Output:

```
0 1 5 1 6 7 8 7 6 1 0 2 4 2 9 3 10
```

Source

<https://www.geeksforgeeks.org/print-the-dfs-traversal-step-wise-backtracking-also/>

Chapter 132

Print unique rows in a given boolean matrix

Print unique rows in a given boolean matrix - GeeksforGeeks

Given a binary matrix, print all unique rows of the given matrix.

Input:

```
{0, 1, 0, 0, 1}  
{1, 0, 1, 1, 0}  
{0, 1, 0, 0, 1}  
{1, 1, 1, 0, 0}
```

Output:

```
0 1 0 0 1  
1 0 1 1 0  
1 1 1 0 0
```

Method 1 (Simple)

A simple approach is to check each row with all processed rows. Print the first row. Now, starting from the second row, for each row, compare the row with already processed rows. If the row matches with any of the processed rows, don't print it. If the current row doesn't match with any row, print it.

Time complexity: $O(\text{ROW}^2 \times \text{COL})$

Auxiliary Space: $O(1)$

Method 2 (Use Binary Search Tree)

Find the decimal equivalent of each row and insert it into BST. Each node of the BST will contain two fields, one field for the decimal value, other for row number. Do not insert a node if it is duplicated. Finally, traverse the BST and print the corresponding rows.

Time complexity: $O(\text{ROW} \times \text{COL} + \text{ROW} \times \log(\text{ROW}))$

Auxiliary Space: $O(\text{ROW})$

This method will lead to Integer Overflow if number of columns is large.

Method 3 (Use Trie data structure)

Since the matrix is boolean, a variant of Trie data structure can be used where each node will be having two children one for 0 and other for 1. Insert each row in the Trie. If the row is already there, don't print the row. If row is not there in Trie, insert it in Trie and print it.

Below is C implementation of method 3.

```
//Given a binary matrix of M X N of integers, you need to return only unique rows of binary array
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

#define ROW 4
#define COL 5

// A Trie node
typedef struct Node
{
    bool isEndOfCol;
    struct Node *child[2]; // Only two children needed for 0 and 1
} Node;

// A utility function to allocate memory for a new Trie node
Node* newNode()
{
    Node* temp = (Node *)malloc( sizeof( Node ) );
    temp->isEndOfCol = 0;
    temp->child[0] = temp->child[1] = NULL;
    return temp;
}

// Inserts a new matrix row to Trie. If row is already
// present, then returns 0, otherwise inserts the row and
// return 1
bool insert( Node** root, int (*M)[COL], int row, int col )
{
    // base case
    if ( *root == NULL )
        *root = newNode();

    // Recur if there are more entries in this row
    if ( col < COL )
        return insert( &(*root)->child[ M[row][col] ], M, row, col+1 );

    else // If all entries of this row are processed
```

```
{  
    // unique row found, return 1  
    if ( !( (*root)->isEndOfCol ) )  
        return (*root)->isEndOfCol = 1;  
  
    // duplicate row found, return 0  
    return 0;  
}  
}  
  
// A utility function to print a row  
void printRow( int (*M)[COL], int row )  
{  
    int i;  
    for( i = 0; i < COL; ++i )  
        printf( "%d ", M[row][i] );  
    printf("\n");  
}  
  
// The main function that prints all unique rows in a  
// given matrix.  
void findUniqueRows( int (*M)[COL] )  
{  
    Node* root = NULL; // create an empty Trie  
    int i;  
  
    // Iterate through all rows  
    for ( i = 0; i < ROW; ++i )  
        // insert row to TRIE  
        if ( insert(&root, M, i, 0) )  
            // unique row found, print it  
            printRow( M, i );  
}  
  
// Driver program to test above functions  
int main()  
{  
    int M[ROW][COL] = {{0, 1, 0, 0, 1},  
                      {1, 0, 1, 1, 0},  
                      {0, 1, 0, 0, 1},  
                      {1, 0, 1, 0, 0}};  
};  
  
findUniqueRows( M );  
  
return 0;  
}
```

Time complexity: O(ROW x COL)

Auxiliary Space: O(ROW x COL)

This method has better time complexity. Also, relative order of rows is maintained while printing.

Method 4 (Use HashSet data structure)

In this method convert the whole row into a single String and then check it is already present in HashSet or not. If row is present then we will leave it otherwise we will print unique row and add it to HashSet.

Thanks, Anshuman Kaushik for suggesting this method.

```
// Java code to print unique row in a
// given binary matrix
import java.util.HashSet;

public class GFG {

    public static void printArray(int arr[][],
                                 int row,int col)
    {

        HashSet<String> set = new HashSet<String>();

        for(int i = 0; i < row; i++)
        {
            String s = "";

            for(int j = 0; j < col; j++)
                s += String.valueOf(arr[i][j]);

            if(!set.contains(s)) {
                set.add(s);
                System.out.println(s);

            }
        }

        // Driver code
        public static void main(String[] args) {

            int arr[][] = { {0, 1, 0, 0, 1},
                           {1, 0, 1, 1, 0},
                           {0, 1, 0, 0, 1},
                           {1, 1, 1, 0, 0} };

            printArray(arr, 4, 5);
        }
    }
}
```

}

Time complexity: O(ROW x COL)

Auxiliary Space: O(ROW)

Improved By : [Anshuman Kaushik](#)

Source

<https://www.geeksforgeeks.org/print-unique-rows/>

Chapter 133

Program for assigning usernames using Trie

Program for assigning usernames using Trie - GeeksforGeeks

Suppose there is a queue of n users and your task is to assign a username to them. The system works in the following way. Every user has preferred login in the form of a string 's' s consists only of small case letters and numbers. User name is assigned in the following order s, s0, s1, s2....s11.... means first you check s if s is available assign it if it is already occupied check for s0 if it is free assign it or if it is occupied check s1 and so on... if a username is assigned to one user it becomes occupied for other users after him in the queue.

Examples:

Input: names[] = {abc, bcd}

Output: user_names[] = {abc bcd}

Input: names[] = {abc, bcd, abc}

Output: user_names[] = {abc bcd abc0}

Input : names[] = {geek, geek0, geek1, geek}

Output : user_names[] = {geek geek0 geek1 geek2}

For first user geek is free so it is assigned to him similarly for the second and third user but for fourth user geek is not free so we will check geek0 but it is also not free then we will go for geek1 but it is also not free then we will check geek2 it is free so it is assigned to him.

We solve this problem using [Trie](#). We do not use usual Trie which have 26 children but a Trie where nodes have 36 children 26 alphabets(a-z) and 10 numbers from (0-9). In addition to this each node of Trie will also have bool variable which will turn into true when a string ending at that node is inserted there will be a int variable as well lets call it add which will be initially -1 and suppose the string is geek and this int variable is equal to -1 then it means that we will directly return geek as it is asked for the first time but suppose it is 12 then it means that string geek, geek0.....geek12 are already present in the Trie.

Steps

Step 1: Maintain a Trie as discussed above.

Step 2: For every given name, check if the string given by user is not in the Trie then return the same string else start from i=add+1 (add is discussed above) and start checking if we concatenate i with the input string is present in the Trie or not if it is not present then return it and set add=i as well as insert it into Trie as well else increment i

Suppose string is geek and i=5 check if geek5 is in Trie or not if it is not present return geek5 set add for geek = 5 insert geek5 in Trie else if it is not present follow same steps for geek6 until you find a string that is not present in the Trie.

```
// C++ program to assign usernames to users
#include <bits/stdc++.h>
using namespace std;

#define MAX_CHAR 26

struct additional {

    // is checks if the current node is
    // leaf node or not
    bool is;

    // add counts number of concatenations
    // of string are present in Trie
    int add;
};

// represents Trie node
struct Trie {

    // MAX_CHAR character children
    Trie* character[MAX_CHAR];

    // 10 numbers (from 0 to 9)
    Trie* number[10];

    // one additional struct children
    additional a;
};

// function to get new node
Trie* getnew()
{
    // initialising the Trie node
    Trie* node = new Trie;
    node->a.is = false;
    node->a.add = -1;
    for (int i = 0; i < MAX_CHAR; i++)
```

```

        node->character[i] = NULL;
    for (int i = 0; i < 10; i++)
        node->number[i] = NULL;
    return node;
}

// inserting a string into Trie
void insert(Trie*& head, string s)
{
    Trie* curr = head;
    for (int i = 0; i < s.length(); i++) {
        if (s[i] - 'a' < 0) {
            if (curr->number[s[i] - '0'] == NULL) {
                curr->number[s[i] - '0'] = getnew();
            }
            curr = curr->number[s[i] - '0'];
        }
        else {
            if (curr->character[s[i] - 'a'] == NULL)
                curr->character[s[i] - 'a'] = getnew();
            curr = curr->character[s[i] - 'a'];
        }
    }
    curr->a.is = true;
}

// returns the structure additional
additional search(Trie* head, string s)
{
    additional x;
    x.is = false;
    x.add = -1;

    // if head is null directly return additional x
    if (head == NULL)
        return x;
    Trie* curr = head;

    // checking if string is present or not and
    // accordingly returning x
    for (int i = 0; i < s.size(); i++) {
        if (s[i] - 'a' < 0) {
            curr = curr->number[s[i] - '0'];
        }
        else
            curr = curr->character[s[i] - 'a'];
        if (curr == NULL)
            return x;
    }
}

```

```
}

x.is = curr->a.is;
x.add = curr->a.add;
return x;
}

// special function to update add variable to z
void update(Trie* head, string s, int z)
{
    Trie* curr = head;
    for (int i = 0; i < s.size(); i++) {
        if (s[i] - 'a' < 0)
            curr = curr->number[s[i] - '0'];
        else
            curr = curr->character[s[i] - 'a'];
    }
    curr->a.add = z;
}

void printUsernames(string username[], int n)
{
    // Initializing a Trie root
    Trie* head = getnew();

    // Assigning usernames one by one
    for (int i = 0; i < n; i++) {
        string s = username[i];
        additional x = search(head, s);

        // if string is not present directly return it
        if (x.is == false) {
            cout << s << endl;
            insert(head, s);
        }

        // to_string(x) converts integer x into string
        else if (x.is == true) {

            // start from x.add+1
            int y = x.add + 1;
            string x = s;

            // continuing searching the string
            // until a free username is found
            while (1 < 2) {

                // if free username is found
                if (search(head, x + to_string(y)).is == false) {
```

```
// print it inser it and update add
cout << x << y << endl;
insert(head, x + to_string(y));
update(head, s, y);
break;
}
// else increment y
else if (search(head, x + to_string(y)).is == true)
    y++;
}
}
}

// driver function
int main()
{
    string name[] = { "geek", "geek0", "geek1", "geek" };
    int n = sizeof(name) / sizeof(name[0]);
    printUsernames(name, n);
    return 0;
}
```

Output:

```
geek
geek0
geek1
geek2
```

Source

<https://www.geeksforgeeks.org/program-for-assigning-usernames-using-trie/>

Chapter 134

Quad Tree

Quad Tree - GeeksforGeeks

Quadtrees are trees used to efficiently store data of points on a two-dimensional space. In this tree, each node has at most four children.

We can construct a quadtree from a two-dimensional area using the following steps:

1. Divide the current two dimensional space into four boxes.
2. If a box contains one or more points in it, create a child object, storing in it the two dimensional space of the box
3. If a box does not contain any points, do not create a child for it
4. Recurse for each of the children.

Quadtrees are used in image compression, where each node contains the average colour of each of its children. The deeper you traverse in the tree, the more the detail of the image. Quadtrees are also used in searching for nodes in a two-dimensional area. For instance, if you wanted to find the closest point to given coordinates, you can do it using quadtrees.

Insert Function

The insert functions is used to insert a node into an existing Quad Tree. This function first checks whether the given node is within the boundaries of the current quad. If it is not, then we immediately cease the insertion. If it is within the boundaries, we select the appropriate child to contain this node based on its location.

This function is $O(\log N)$ where N is the size of distance.

Search Function

The search function is used to locate a node in the given quad. It can also be modified to return the closest node to the given point. This function is implemented by taking the given point, comparing with the boundaries of the child quads and recursing.

This function is $O(\log N)$ where N is size of distance.

The program given below demonstrates storage of nodes in a quadtree.

```
// C++ Implementation of Quad Tree
#include <iostream>
#include <cmath>
using namespace std;

// Used to hold details of a point
struct Point
{
    int x;
    int y;
    Point(int _x, int _y)
    {
        x = _x;
        y = _y;
    }
    Point()
    {
        x = 0;
        y = 0;
    }
};

// The objects that we want stored in the quadtree
struct Node
{
    Point pos;
    int data;
    Node(Point _pos, int _data)
    {
        pos = _pos;
        data = _data;
    }
    Node()
    {
        data = 0;
    }
};

// The main quadtree class
class Quad
{
    // Hold details of the boundary of this node
    Point topLeft;
    Point botRight;

    // Contains details of node
    Node *n;
```

```

// Children of this tree
Quad *topLeftTree;
Quad *topRightTree;
Quad *botLeftTree;
Quad *botRightTree;

public:
    Quad()
    {
        topLeft = Point(0, 0);
        botRight = Point(0, 0);
        n = NULL;
        topLeftTree = NULL;
        topRightTree = NULL;
        botLeftTree = NULL;
        botRightTree = NULL;
    }
    Quad(Point topL, Point botR)
    {
        n = NULL;
        topLeftTree = NULL;
        topRightTree = NULL;
        botLeftTree = NULL;
        botRightTree = NULL;
        topLeft = topL;
        botRight = botR;
    }
    void insert(Node*);
    Node* search(Point);
    bool inBoundary(Point);
};

// Insert a node into the quadtree
void Quad::insert(Node *node)
{
    if (node == NULL)
        return;

    // Current quad cannot contain it
    if (!inBoundary(node->pos))
        return;

    // We are at a quad of unit area
    // We cannot subdivide this quad further
    if (abs(topLeft.x - botRight.x) <= 1 &&
        abs(topLeft.y - botRight.y) <= 1)
    {
        if (n == NULL)

```

```

        n = node;
        return;
    }

    if ((topLeft.x + botRight.x) / 2 >= node->pos.x)
    {
        // Indicates topLeftTree
        if ((topLeft.y + botRight.y) / 2 >= node->pos.y)
        {
            if (topLeftTree == NULL)
                topLeftTree = new Quad(
                    Point(topLeft.x, topLeft.y),
                    Point((topLeft.x + botRight.x) / 2,
                          (topLeft.y + botRight.y) / 2));
            topLeftTree->insert(node);
        }

        // Indicates botLeftTree
        else
        {
            if (botLeftTree == NULL)
                botLeftTree = new Quad(
                    Point(topLeft.x,
                          (topLeft.y + botRight.y) / 2),
                    Point((topLeft.x + botRight.x) / 2,
                          botRight.y));
            botLeftTree->insert(node);
        }
    }
    else
    {
        // Indicates topRightTree
        if ((topLeft.y + botRight.y) / 2 >= node->pos.y)
        {
            if (topRightTree == NULL)
                topRightTree = new Quad(
                    Point((topLeft.x + botRight.x) / 2,
                          topLeft.y),
                    Point(botRight.x,
                          (topLeft.y + botRight.y) / 2));
            topRightTree->insert(node);
        }

        // Indicates botRightTree
        else
        {
            if (botRightTree == NULL)
                botRightTree = new Quad(

```

```

        Point((topLeft.x + botRight.x) / 2,
               (topLeft.y + botRight.y) / 2),
        Point(botRight.x, botRight.y));
    botRightTree->insert(node);
}
}

// Find a node in a quadtree
Node* Quad::search(Point p)
{
    // Current quad cannot contain it
    if (!inBoundary(p))
        return NULL;

    // We are at a quad of unit length
    // We cannot subdivide this quad further
    if (n != NULL)
        return n;

    if ((topLeft.x + botRight.x) / 2 >= p.x)
    {
        // Indicates topLeftTree
        if ((topLeft.y + botRight.y) / 2 >= p.y)
        {
            if (topLeftTree == NULL)
                return NULL;
            return topLeftTree->search(p);
        }

        // Indicates botLeftTree
        else
        {
            if (botLeftTree == NULL)
                return NULL;
            return botLeftTree->search(p);
        }
    }
    else
    {
        // Indicates topRightTree
        if ((topLeft.y + botRight.y) / 2 >= p.y)
        {
            if (topRightTree == NULL)
                return NULL;
            return topRightTree->search(p);
        }
    }
}

```

```

// Indicates botRightTree
else
{
    if (botRightTree == NULL)
        return NULL;
    return botRightTree->search(p);
}
}

};

// Check if current quadtree contains the point
bool Quad::inBoundary(Point p)
{
    return (p.x >= topLeft.x &&
            p.x <= botRight.x &&
            p.y >= topLeft.y &&
            p.y <= botRight.y);
}

// Driver program
int main()
{
    Quad center(Point(0, 0), Point(8, 8));
    Node a(Point(1, 1), 1);
    Node b(Point(2, 5), 2);
    Node c(Point(7, 6), 3);
    center.insert(&a);
    center.insert(&b);
    center.insert(&c);
    cout << "Node a: " <<
        center.search(Point(1, 1))->data << "\n";
    cout << "Node b: " <<
        center.search(Point(2, 5))->data << "\n";
    cout << "Node c: " <<
        center.search(Point(7, 6))->data << "\n";
    cout << "Non-existing node: "
        << center.search(Point(5, 5));
    return 0;
}

```

Output:

```

Node a: 1
Node b: 2
Node c: 3
Non-existing node: 0

```

Exercise:

Implement a Quad Tree which returns 4 closest nodes to a given point.

Further References:

<http://jimkang.com/quadtreetravis/>
<https://en.wikipedia.org/wiki/Quadtree>

Source

<https://www.geeksforgeeks.org/quad-tree/>

Chapter 135

Queries for number of distinct elements in a subarray

Queries for number of distinct elements in a subarray - GeeksforGeeks

Given a array ‘a[]’ of size n and number of queries q. Each query can be represented by two integers l and r. Your task is to print the number of distinct integers in the subarray l to r. Given $a[i] \leq 10^6$

Examples:

```
Input : a[] = {1, 1, 2, 1, 3}
        q = 3
        0 4
        1 3
        2 4
Output :3
        2
        3
In query 1, number of distinct integers
in a[0...4] is 3 (1, 2, 3)
In query 2, number of distinct integers
in a[1..3] is 2 (1, 2)
In query 3, number of distinct integers
in a[2..4] is 3 (1, 2, 3)
```

The idea is to use [Binary Indexed Tree](#)

1. **Step 1 :** Take an array last_visit of size 10^6 where $\text{last_visit}[i]$ holds the rightmost index of the number i in the array a. Initialize this array as -1.
2. **Step 2 :** Sort all the queries in ascending order of their right end r.

3. **Step 3 :** Create a [Binary Indexed Tree](#) in an array `bit[]`. Start traversing the array ‘`a`’ and queries simultaneously and check if `last_visit[a[i]]` is -1 or not. If it is not, update the bit array with value -1 at the idx `last_visit[a[i]]`.
4. **Step 4 :** Set `last_visit[a[i]] = i` and update the bit array `bit` array with value 1 at idx `i`.
5. **Step 5 :** Answer all the queries whose value of `r` is equal to `i` by querying the bit array. This can be easily done as queries are sorted.

```
// C++ code to find number of distinct numbers
// in a subarray
#include<bits/stdc++.h>
using namespace std;

const int MAX = 1000001;

// structure to store queries
struct Query
{
    int l, r, idx;
};

// cmp function to sort queries according to r
bool cmp(Query x, Query y)
{
    return x.r < y.r;
}

// updating the bit array
void update(int idx, int val, int bit[], int n)
{
    for (; idx <= n; idx += idx&~idx)
        bit[idx] += val;
}

// querying the bit array
int query(int idx, int bit[], int n)
{
    int sum = 0;
    for (; idx>0; idx-=idx&~idx)
        sum += bit[idx];
    return sum;
}

void answeringQueries(int arr[], int n, Query queries[], int q)
{
    // initialising bit array
    int bit[n+1];
```

```

        memset(bit, 0, sizeof(bit));

        // holds the rightmost index of any number
        // as numbers of a[i] are less than or equal to 10^6
        int last_visit[MAX];
        memset(last_visit, -1, sizeof(last_visit));

        // answer for each query
        int ans[q];
        int query_counter = 0;
        for (int i=0; i<n; i++)
        {
            // If last visit is not -1 update -1 at the
            // idx equal to last_visit[arr[i]]
            if (last_visit[arr[i]] !=-1)
                update (last_visit[arr[i]] + 1, -1, bit, n);

            // Setting last_visit[arr[i]] as i and updating
            // the bit array accordingly
            last_visit[arr[i]] = i;
            update(i + 1, 1, bit, n);

            // If i is equal to r of any query store answer
            // for that query in ans[]
            while (query_counter < q && queries[query_counter].r == i)
            {
                ans[queries[query_counter].idx] =
                    query(queries[query_counter].r + 1, bit, n)-
                    query(queries[query_counter].l, bit, n);
                query_counter++;
            }
        }

        // print answer for each query
        for (int i=0; i<q; i++)
            cout << ans[i] << endl;
    }

    // driver code
    int main()
    {
        int a[] = {1, 1, 2, 1, 3};
        int n = sizeof(a)/sizeof(a[0]);
        Query queries[3];
        queries[0].l = 0;
        queries[0].r = 4;
        queries[0].idx = 0;
        queries[1].l = 1;

```

```
queries[1].r = 3;
queries[1].idx = 1;
queries[2].l = 2;
queries[2].r = 4;
queries[2].idx = 2;
int q = sizeof(queries)/sizeof(queries[0]);
sort(queries, queries+q, cmp);
answeringQueries(a, n, queries, q);
return 0;
}
```

Output:

```
3
2
3
```

Source

<https://www.geeksforgeeks.org/queries-number-distinct-elements-subarray/>

Chapter 136

Queries on substring palindrome formation

Queries on substring palindrome formation - GeeksforGeeks

Given a string **S**, and two type of queries.

```
Type 1: 1 L x, Indicates update Lth index
          of string S by x character.
Type 2: 2 L R, Find if characters between position L and R
          of string S can form a palindrome string.
          If palindrome can be formed print "Yes",
          else print "No".
1 <= L, R <= |S|
```

Examples:

```
Input : S = "geeksforgeeks"
Query 1: 1 4 g
Query 2: 2 1 4
Query 3: 2 2 3
Query 4: 1 10 t
Query 5: 2 10 11
Output :
Yes
Yes
No
```

Query 1: update index 3 (position 4) of string S by character 'g'. So new string S = "geegsforgeeks".

Query 2: find if rearrangement between index 0 and 3 can form a palindrome. "geegs" is palindrome, print "Yes".

Query 3: find if rearrangement between index 1 and 2 can form a palindrome. "ee" is palindrome, print "Yes".

Query 4: update index 9 (position 10) of string S by character 't'. So new string S = "geegsforgteks".

Query 5: find if rearrangement between index 9 and 10 can form a palindrome. "te" is not palindrome, print "No".

Substring S[L...R] form a palindrome only if frequencies of all the character in S[L...R] are even, with one except allowed.

For query of type 1, simply update string S[L] by character x.

For each query of type 2, calculate the frequency of character and check if frequencies of all characters is even (with) one exception allowed.

Following are two different methods to find frequency of each character in S[L...R]:

Method 1: Use a frequency array to find the frequency of each element in S[L...R].

Below is C++ implementation of this approach:

```
// C++ program to Queries on substring palindrome
// formation.
#include<bits/stdc++.h>
using namespace std;

// Query type 1: update string position i with
// character x.
void qType1(int l, int x, char str[])
{
    str[l-1] = x;
}

// Print "Yes" if range [L..R] can form palindrome,
// else print "No".
void qType2(int l, int r, char str[])
{
```

```
int freq[27] = { 0 };

// Find the frequency of each character in
// S[L...R].
for (int i = l-1; i<=r-1; i++)
    freq[str[i] - 'a']++;

// Checking if more than one character have
// frequency greater than 1.
int count = 0;
for (int j = 0; j < 26; j++)
    if (freq[j] % 2)
        count++;

(count<=1)? (cout << "Yes" << endl):
            (cout << "No" << endl);
}

// Driven Program
int main()
{
    char str[] = "geeksforgeeks";
    int n = strlen(str);

    qType1(4, 'g', str);
    qType2(1, 4, str);
    qType2(2, 3, str);
    qType1(10, 't', str);
    qType2(10, 11, str);

    return 0;
}
```

Output:

```
Yes
Yes
No
```

Method 2 : Use Binary Indexed Tree

The efficient approach can be maintain 26 [Binary Index Tree](#) for each alphabet. Define a function `getFrequency(i,u)` which returns the frequency of ‘u’ in the i^{th} prefix. Frequency of character ‘u’ in range L...R can be find by `getFrequency(R, u) – getFrequency(L-1, u)`.

Whenever update(Query 1) comes to change S[i] from character ‘u’ to ‘v’. BIT[u] is updated with -1 at index i and BIT[v] is updated with +1 at index i.

Below is C++ implementation of this approach:

```
// C++ program to Queries on substring palindrome
// formation.
#include <bits/stdc++.h>
#define max 1000
using namespace std;

// Return the frequency of the character in the
// i-th prefix.
int getFrequency(int tree[max][27], int idx, int i)
{
    int sum = 0;

    while (idx > 0)
    {
        sum += tree[idx][i];
        idx -= (idx & -idx);
    }

    return sum;
}

// Updating the BIT
void update(int tree[max][27], int idx, int val, int i)
{
    while (idx <= max)
    {
        tree[idx][i] += val;
        idx += (idx & -idx);
    }
}

// Query to update the character in the string.
void qType1(int tree[max][27], int l, int x, char str[])
{
    // Adding -1 at L position
    update(tree, l, -1, str[l-1]-97+1);

    // Updating the character
    str[l-1] = x;

    // Adding +1 at R position
    update(tree, l, 1, str[l-1]-97+1);
}
```

```

// Query to find if rearrangement of character in range
// L...R can form palindrome
void qType2(int tree[max][27], int l, int r, char str[])
{
    int count = 0;

    for (int i = 1; i <= 26; i++)
    {
        // Checking on the first character of the string S.
        if (l == 1)
        {
            if (getFrequency(tree, r, i)%2 == 1)
                count++;
        }
        else
        {
            // Checking if frequency of character is even or odd.
            if ((getFrequency(tree, r, i) -
                  getFrequency(tree, l-1, i))%2 == 1)
                count++;
        }
    }

    (count<=1)?(cout << "Yes" << endl):(cout << "No" << endl);
}

// Creating the Binary Index Tree of all alphabet
void buildBIT(int tree[max][27], char str[], int n)
{
    memset(tree, 0, sizeof(tree));

    for (int i = 0; i < n; i++)
        update(tree, i+1, 1, str[i]-97+1);
}

// Driven Program
int main()
{
    char str[] = "geeksforgeeks";
    int n = strlen(str);

    int tree[max][27];
    buildBIT(tree, str, n);

    qType1(tree, 4, 'g', str);
    qType2(tree, 1, 4, str);
    qType2(tree, 2, 3, str);
    qType1(tree, 10, 't', str);
}

```

```
qType2(tree, 10, 11, str);  
  
    return 0;  
}
```

Output:

```
Yes  
Yes  
No
```

Source

<https://www.geeksforgeeks.org/queries-substring-palindrome-formation/>

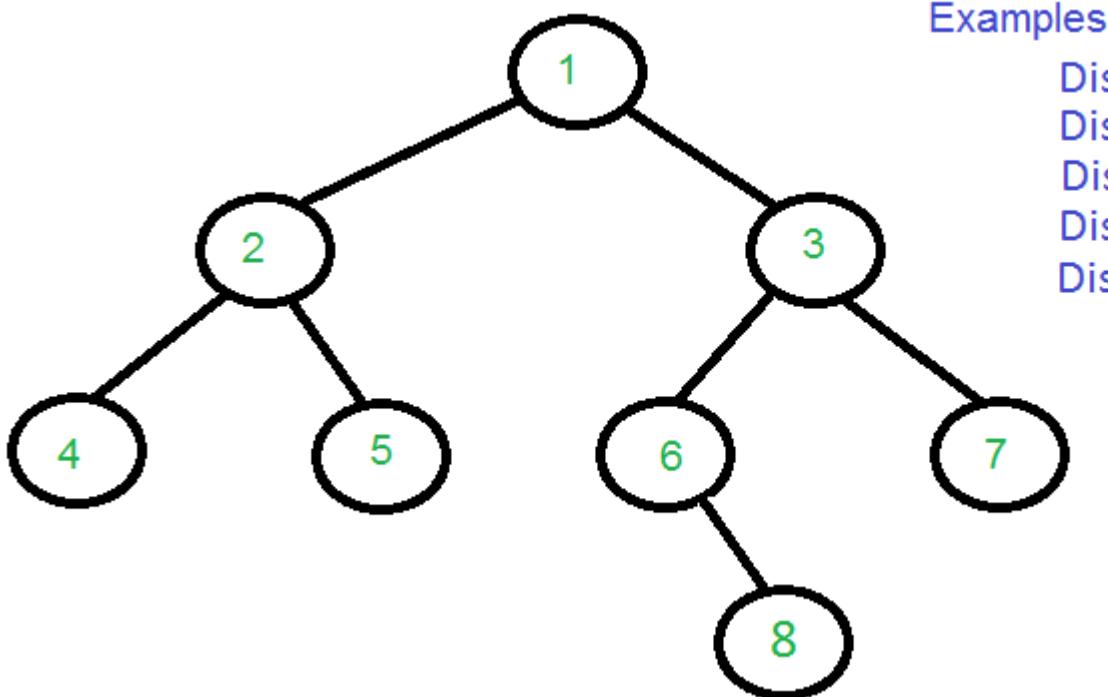
Chapter 137

Queries to find distance between two nodes of a Binary tree

Queries to find distance between two nodes of a Binary tree - GeeksforGeeks

Given a binary tree, the task is to find the distance between two keys in a binary tree, no parent pointers are given. The distance between two nodes is the minimum number of edges to be traversed to reach one node from other.

It has been already discussed in [this](#) for a single query in $O(\log n)$ time, here the task is to reduce multiple queries time to $O(1)$ by compromising with space complexity to $O(N \log n)$. In this post, we will use **Sparse table** instead of segment tree for finding the minimum in given range, which uses dynamic programming and bit manipulation to achieve $O(1)$ query time.



A sparse table will preprocess the minimum values given for an array in $N \log n$ space i.e. each node will contain chain of values of $\log(i)$ length where i is the index of the i th node in L array. Each entry in the sparse table says $M[i][j]$ will represent the index of the minimum value in the subarray starting at i having 2^j .

The distance between two nodes can be obtained in terms of lowest common ancestor.

$$\text{Dist}(n_1, n_2) = \text{Level}[n_1] + \text{Level}[n_2] - 2 * \text{Level}[\text{lca}]$$

This problem can be breakdown into **finding levels of each node**, **finding the Euler tour of binary tree** and **building sparse table** for LCA, these steps are explained below :

1. Find the levels of each node by applying [level order traversal](#).
2. Find the LCA of two nodes in the binary tree in $O(\log n)$ by Storing [Euler tour of tree](#) in array and computing two other arrays with the help of levels of each node and Euler tour.

These steps are shown below:

- (I) First, find [Euler Tour of binary tree](#).

Euler	1	2	4	2	5	2	1	3	6	8	6	3
	1	2	3	4	5	6	7	8	9	10	11	12
Euler tour of Binary Tree												

(II) Then, store levels of each node in Euler array.

L	0	1	2	1	2	1	0	1	2	3	2	1
	1	2	3	4	5	6	7	8	9	10	11	12
Levels of each node in Euler tour												

(III) Then, store First occurrences of all nodes of binary tree in Euler array.

H	1	2	8	3	5	9	13	10
	1	2	3	4	5	6	7	8
First occurrences of each node in Euler tree								

3. Then build sparse table on L array and find the minimum value say X in range ($H[A]$ to $H[B]$). Then use the index of value X as an index to Euler array to get LCA, i.e. $Euler[\text{index}(X)]$.

Let, A=8 and B=5.

(I) $H[8]=1$ and $H[5]=2$

(II) we get min value in L array between 1 and 2 as X=0, index=7

(III) Then, LCA= Euler[7], i.e LCA=1.

4. Finally, apply distance formula discussed above to get the distance between two nodes.

Source

<https://www.geeksforgeeks.org/queries-find-distance-two-nodes-binary-tree/>

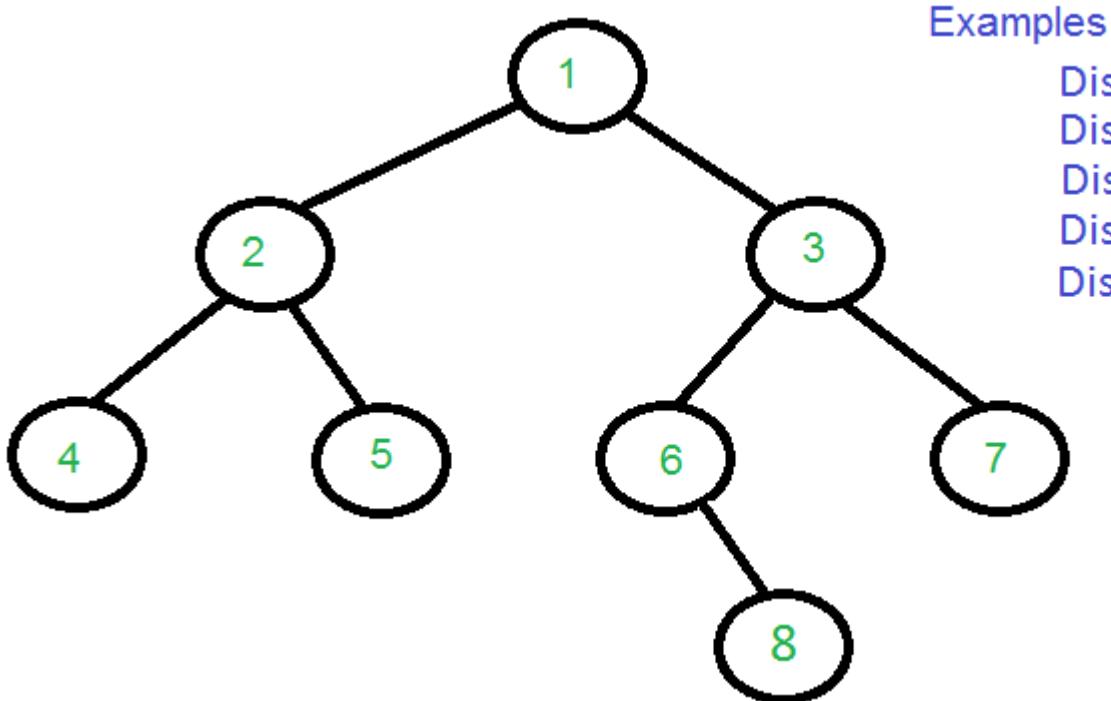
Chapter 138

Queries to find distance between two nodes of a Binary tree – O(logn) method

Queries to find distance between two nodes of a Binary tree - O(logn) method - GeeksforGeeks

Given a binary tree, the task is to find the distance between two keys in a binary tree, no parent pointers are given. Distance between two nodes is the minimum number of edges to be traversed to reach one node from other.

This problem has been already discussed in [previous post](#) but it uses **three traversals** of the Binary tree, one for finding Lowest Common Ancestor(LCA) of two nodes(let A and B) and then two traversals for finding distance between LCA and A and LCA and B which has $O(n)$ time complexity. In this post, a method will be discussed that requires the **$O(\log(n))$** time to find LCA of two nodes.



The distance between two nodes can be obtained in terms of lowest common ancestor. Following is the formula.

$\text{Dist}(n_1, n_2) = \text{Dist}(\text{root}, n_1) + \text{Dist}(\text{root}, n_2) - 2 * \text{Dist}(\text{root}, \text{lca})$
 'n1' and 'n2' are the two given keys
 'root' is root of given Binary Tree.
 'lca' is lowest common ancestor of n1 and n2.
 $\text{Dist}(n_1, n_2)$ is the distance between n1 and n2.

Above formula can also be written as:

$$\text{Dist}(n_1, n_2) = \text{Level}[n_1] + \text{Level}[n_2] - 2 * \text{Level}[\text{lca}]$$

This problem can be breakdown into:

1. Finding levels of each node
2. Finding the Euler tour of binary tree
3. Building segment tree for LCA,

These steps are explained below :

1. Find the levels of each node by applying [level order traversal](#).
 2. Find the LCA of two nodes in binary tree in $O(\log n)$ by Storing [Euler tour of Binary tree](#) in array and computing two other arrays with the help of levels of each node and Euler tour.
- These steps are shown below:

(I) First, find Euler Tour of binary tree.

Euler	1	2	4	2	5	2	1	3	6	8	6
	1	2	3	4	5	6	7	8	9	10	11
Euler tour of Binary Tree											

Euler tour of binary tree in example

(II) Then, store levels of each node in Euler array.

L	0	1	2	1	2	1	0	1	2	3	2
	1	2	3	4	5	6	7	8	9	10	11
Levels of each node in Euler tour											

(III) Then, store First occurrences of all nodes of binary tree in Euler array. H stores the indices of nodes from Euler array, so that range of query for finding minimum can be minimized and thereby further optimizing the query time.

H	1	2	8	3	5	9	13	10
	1	2	3	4	5	6	7	8
First occurrences of each node in Euler tree								

3. Then **build segment tree on L array** and take the low and high values from H array that will give us the first occurrences of say Two nodes(A and B). Then, *we query segment tree to find the minimum value* say X in range ($H[A]$ to $H[B]$). Then **we use the index of value X as index to Euler array to get LCA**, i.e. $Euler[\text{index}(X)]$.

Let, A = 8 and B = 5.

(I) $H[8] = 1$ and $H[5] = 2$

(II) Querying on Segment tree, we get min value in L array between 1 and 2 as X=0, index=7

(III) Then, LCA= $Euler[7]$, i.e LCA = 1.

4. Finally, we apply distance formula discussed above to get distance between two nodes.

```
// C++ program to find distance between
// two nodes for multiple queries
#include <bits/stdc++.h>
#define MAX 100001
using namespace std;

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

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

// Array to store level of each node
int level[MAX];

// Utility Function to store level of all nodes
void FindLevels(struct Node* root)
{
    if (!root)
        return;

    // queue to hold tree node with level
    queue<pair<struct Node*, int>> q;

    // let root node be at level 0
    q.push({root, 0});

    pair<struct Node*, int> p;

    // Do level Order Traversal of tree
    while (!q.empty()) {
        p = q.front();
        q.pop();

        // Node p.first is on level p.second
```

```
level[p.first->data] = p.second;

// If left child exists, put it in queue
// with current_level +1
if (p.first->left)
    q.push({ p.first->left, p.second + 1 });

// If right child exists, put it in queue
// with current_level +1
if (p.first->right)
    q.push({ p.first->right, p.second + 1 });
}

// Stores Euler Tour
int Euler[MAX];

// index in Euler array
int idx = 0;

// Find Euler Tour
void eulerTree(struct Node* root)
{
    // store current node's data
    Euler[++idx] = root->data;

    // If left node exists
    if (root->left) {

        // traverse left subtree
        eulerTree(root->left);

        // store parent node's data
        Euler[++idx] = root->data;
    }

    // If right node exists
    if (root->right) {
        // traverse right subtree
        eulerTree(root->right);

        // store parent node's data
        Euler[++idx] = root->data;
    }
}

// checks for visited nodes
```

```
int vis[MAX];

// Stores level of Euler Tour
int L[MAX];

// Stores indices of first occurrence
// of nodes in Euler tour
int H[MAX];

// Preprocessing Euler Tour for finding LCA
void preprocessEuler(int size)
{
    for (int i = 1; i <= size; i++) {
        L[i] = level[Euler[i]];

        // If node is not visited before
        if (vis[Euler[i]] == 0) {
            // Add to first occurrence
            H[Euler[i]] = i;

            // Mark it visited
            vis[Euler[i]] = 1;
        }
    }
}

// Stores values and positions
pair<int, int> seg[4 * MAX];

// Utility function to find minimum of
// pair type values
pair<int, int> min(pair<int, int> a,
                    pair<int, int> b)
{
    if (a.first <= b.first)
        return a;
    else
        return b;
}

// Utility function to build segment tree
pair<int, int> buildSegTree(int low, int high, int pos)
{
    if (low == high) {
        seg[pos].first = L[low];
        seg[pos].second = low;
        return seg[pos];
    }
}
```

```
int mid = low + (high - low) / 2;
buildSegTree(low, mid, 2 * pos);
buildSegTree(mid + 1, high, 2 * pos + 1);

seg[pos] = min(seg[2 * pos], seg[2 * pos + 1]);
}

// Utility function to find LCA
pair<int, int> LCA(int qlow, int qhigh, int low,
                     int high, int pos)
{
    if (qlow <= low && qhigh >= high)
        return {seg[pos], 0};

    if (qlow > high || qhigh < low)
        return {INT_MAX, 0};

    int mid = low + (high - low) / 2;

    return min(LCA(qlow, qhigh, low, mid, 2 * pos),
               LCA(qlow, qhigh, mid + 1, high, 2 * pos + 1));
}

// Function to return distance between
// two nodes n1 and n2
int findDistance(int n1, int n2, int size)
{
    // Maintain original Values
    int prevn1 = n1, prevn2 = n2;

    // Get First Occurrence of n1
    n1 = H[n1];

    // Get First Occurrence of n2
    n2 = H[n2];

    // Swap if low > high
    if (n2 < n1)
        swap(n1, n2);

    // Get position of minimum value
    int lca = LCA(n1, n2, 1, size, 1).second;

    // Extract value out of Euler tour
    lca = Euler[lca];

    // return calculated distance
    return level[prevn1] + level[prevn2] - 2 * level[lca];
}
```

```
}

void preProcessing(Node* root, int N)
{
    // Build Tree
    eulerTree(root);

    // Store Levels
    FindLevels(root);

    // Find L and H array
    preprocessEuler(2 * N - 1);

    // Build segment Tree
    buildSegTree(1, 2 * N - 1, 1);
}

/* Driver function to test above functions */
int main()
{
    int N = 8; // Number of nodes

    /* Constructing tree given in the above figure */
    Node* root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->left = newNode(6);
    root->right->right = newNode(7);
    root->right->left->right = newNode(8);

    // Function to do all preprocessing
    preProcessing(root, N);

    cout << "Dist(4, 5) = " << findDistance(4, 5, 2 * N - 1) << "\n";
    cout << "Dist(4, 6) = " << findDistance(4, 6, 2 * N - 1) << "\n";
    cout << "Dist(3, 4) = " << findDistance(3, 4, 2 * N - 1) << "\n";
    cout << "Dist(2, 4) = " << findDistance(2, 4, 2 * N - 1) << "\n";
    cout << "Dist(8, 5) = " << findDistance(8, 5, 2 * N - 1) << "\n";

    return 0;
}
```

Output:

```
Dist(4, 5) = 2
```

```
Dist(4, 6) = 4  
Dist(3, 4) = 3  
Dist(2, 4) = 1  
Dist(8, 5) = 5
```

Time Complexity: $O(\log N)$

Space Complexity: $O(N)$

[Queries to find distance between two nodes of a Binary tree – O\(1\) method](#)

Source

<https://www.geeksforgeeks.org/queries-find-distance-two-nodes-binary-tree-ologn-method/>

Chapter 139

Queries to find maximum product pair in range with updates

Queries to find maximum product pair in range with updates - GeeksforGeeks

Given an array of N positive integers. The task is to perform the following operations according to the type of query given.

1. Print the maximum pair product in a given range. [L-R]
2. Update A_i with some given value.

Examples:

Input: A={1, 3, 4, 2, 5}

Queries:

Type 1: L = 0, R = 2.

Type 2: i = 1, val = 6

Type 1: L = 0, R = 2.

Output:

12

24

For the query1, the maximum product in a range [0, 2] is $3*4 = 12$.

For the query2, after an update, the array becomes [1, 6, 4, 2, 5]

For the query3, the maximum product in a range [0, 2] is $6*4 = 24$.

Naive Solution: The brute force approach is to traverse from L to R and check for every pair and then find the maximum product pair among them.

Time Complexity: $O(N^2)$ for every query.

A better solution is to find the first and the second largest number in the range L to R by traversing and then returning their product.

Time Complexity: $O(N)$ for every query.

A **efficient solution** is to use a [segment tree](#) to store the largest and second largest number in the nodes and then return the product of them.

Below is the implementation of above approach.

```
// C++ program to find the maximum
// product in a range with updates
#include <bits/stdc++.h>
using namespace std;
#define ll long long

// structure defined
struct segment {
    // l for largest
    // sl for second largest
    ll l;
    ll sl;
};

// function to perform queries
segment query(segment* tree, ll index,
              ll s, ll e, ll qs, ll qe)
{
    segment res;
    res.l = -1;
    res.sl = -1;
    // no overlapping case
    if (qs > e || qe < s || s > e) {

        return res;
    }

    // complete overlap case
    if (s >= qs && e <= qe) {

        return tree[index];
    }

    // partial overlap case
    ll mid = (s + e) / 2;

    // calling left node and right node
    segment left = query(tree, 2 * index,
                         s, mid, qs, qe);
    segment right = query(tree, 2 * index + 1,
                          mid + 1, e, qs, qe);

    // largest of (left.l, right.l)
```

```
ll largest = max(left.l, right.l);

// compute second largest
// second largest will be minimum
// of maximum from left and right node
ll second_largest = min(max(left.l, right.sl),
                        max(right.l, left.sl));

// store largest and
// second_largest in res
res.l = largest;
res.sl = second_largest;

// return the resulting node
return res;
}

// function to update the query
void update(segment* tree, ll index,
           ll s, ll e, ll i, ll val)
{
    // no overlapping case
    if (i < s || i > e) {
        return;
    }

    // reached leaf node

    if (s == e) {
        tree[index].l = val;
        tree[index].sl = INT_MIN;
        return;
    }

    // partial overlap

    ll mid = (s + e) / 2;

    // left subtree call
    update(tree, 2 * index, s, mid, i, val);

    // right subtree call
    update(tree, 2 * index + 1, mid + 1, e, i, val);

    // largest of (left.l, right.l)
    tree[index].l = max(tree[2 * index].l, tree[2 * index + 1].l);

    // compute second largest
```

```
// second largest will be
// minimum of maximum from left and right node

tree[index].sl = min(max(tree[2 * index].l, tree[2 * index + 1].sl),
                     max(tree[2 * index + 1].l, tree[2 * index].sl));
}

// Function to build the tree
void buildtree(segment* tree, ll* a, ll index, ll s, ll e)
{
    // tree is build bottom to up
    if (s > e) {
        return;
    }

    // leaf node
    if (s == e) {
        tree[index].l = a[s];
        tree[index].sl = INT_MIN;
        return;
    }

    ll mid = (s + e) / 2;

    // calling the left node
    buildtree(tree, a, 2 * index, s, mid);

    // calling the right node
    buildtree(tree, a, 2 * index + 1, mid + 1, e);

    // largest of (left.l, right.l)
    ll largest = max(tree[2 * index].l, tree[2 * index + 1].l);

    // compute second largest
    // second largest will be minimum
    // of maximum from left and right node
    ll second_largest = min(max(tree[2 * index].l, tree[2 * index + 1].sl),
                           max(tree[2 * index + 1].l, tree[2 * index].sl));

    // storing the largest and
    // second_largest values in the current node
    tree[index].l = largest;
    tree[index].sl = second_largest;
}

// Driver Code
int main()
{
```

```
// your code goes here
ll n = 5;

ll a[5] = { 1, 3, 4, 2, 5 };

// allocating memory for segment tree
segment* tree = new segment[4 * n + 1];

// buildtree(tree, a, index, start, end)
buildtree(tree, a, 1, 0, n - 1);

// query section
// storing the resulting node
segment res = query(tree, 1, 0, n - 1, 0, 2);

cout << "Maximum product in the range "
     << "0 and 2 before update: " << (res.l * res.sl);

// update section
// update(tree, index, start, end, i, v)
update(tree, 1, 0, n - 1, 1, 6);

res = query(tree, 1, 0, n - 1, 0, 2);

cout << "\nMaximum product in the range "
     << "0 and 2 after update: " << (res.l * res.sl);

return 0;
}
```

Output:

```
Maximum product in the range 0 and 2 before update: 12
Maximum product in the range 0 and 2 after update: 24
```

Time Complexity: $O(\log N)$ per query and $O(N)$ for building the tree.

Source

<https://www.geeksforgeeks.org/queries-to-find-maximum-product-pair-in-range-with-updates/>

Chapter 140

Querying the number of distinct colors in a subtree of a colored tree using BIT

Querying the number of distinct colors in a subtree of a colored tree using BIT - Geeks-forGeeks

Prerequisites : [BIT](#), [DFS](#)

Given a rooted tree T, with ‘n’ nodes, each node has a color denoted by the array color[] (color[i] denotes the color of ith node in form of an integer). Respond to ‘Q’ queries of the following type:

- **distinct u** – Print the number of distinct colored nodes under the subtree rooted under ‘u’

Examples:

```
      1
     / \
    2   3
   / \ | \
  4 5 6 7 8
   |
   |
  9 10 11

color[] = {0, 2, 3, 3, 4, 1, 3, 4, 3, 2, 1, 1}
Indexes   NA 1 2 3 4 5 6 7 8 9 10 11
(Node Values and colors start from index 1)

distinct 3 -> output should be '4'.
```

There are six different nodes in subtree rooted with 3, nodes are 3, 7, 8, 9, 10 and 11. These nodes have four distinct colors (3, 4, 2 and 1)

distinct 2 -> output should be '3'.
 distinct 7 -> output should be '3'.

Building a solution in steps:

1. Flatten the tree using DFS; store the visiting time and ending time for every node in two arrays, **vis_time[i]** stores the visiting time of the ith node while **end_time[i]** stores the ending time.
2. In the same DFS call, store the value of color of every node in an array **flat_tree[]**, at indices: **vis_time[i]** and **end_time[i]** for ith node.
 Note: size of the array **flat_tree[]** will be $2n$.

Now the problem is reduced to finding the number of distinct elements in the range [**vis_time[u]**, **end_time[u]**] in the array **flat_tree[]** for each query of the specified type. To do so, we will process the queries off-line (processing the queries in an order different than the one provided in the question, and storing the results, and finally printing the result for each in the order specified in the question).

Steps:

1. First, we pre-process the array **flat_tree[]**; we maintain a **table[]** (an array of vectors), **table[i]** stores the vector containing all the indices in **flat_tree[]** that have value i. That is, if **flat_tree[j] = i**, then **table[i]** will have one of its element **j**.
2. In BIT, we update '1' at ith index if we want the ith element of **flat_tree[]** to be counted in **query()** method. We now maintain another array **traverser[]**; **traverser[i]** contains the pointer to the next element of **table[i]** that is not marked in BIT yet.
3. We now update our BIT and set '1' at first occurrence of every element in **flat_tree[]** and increment corresponding **traverser[]** by '1' (if **flat_tree[i]** is occurring for the first time then **traverser[flat_tree[i]]** is incremented by '1') to point to the next occurrence of that element.
4. Now our **query(R)** function for BIT would return the number of distinct elements in **flat_tree[]** in the range [1, R].
5. We sort all the queries in order of increasing **vis_time[]**, let l_i denote **vis_time[i]** and r_i denote the **end_time[i]**. Sorting the queries in increasing order of l_i gives us an edge, as when processing the ith query we won't see any query in future with its 'l' smaller than l_i . So we can remove all the elements' occurrences up to $l_i - 1$ from BIT and add their next occurrences using the **traverser[]** array. And then **query(R)** would return the number of distinct elements in the range $[l_i, r_i]$.

```
// A C++ program implementing the above design
#include<bits/stdc++.h>
#define max_color 1000005
#define maxn 100005
using namespace std;

// Note: All elements of global arrays are
// initially zero
// All the arrays have been described above
int bit[maxn], vis_time[maxn], end_time[maxn];
int flat_tree[2 * maxn];
vector<int> tree[maxn];
vector<int> table[max_color];
int traverser[max_color];

bool vis[maxn];
int tim = 0;

//li, ri and index are stored in queries vector
//in that order, as the sort function will use
//the value li for comparison
vector< pair< pair<int, int>, int > > queries;

//ans[i] stores answer to ith query
int ans[maxn];

//update function to add val to idx in BIT
void update(int idx, int val)
{
    while ( idx < maxn )
    {
        bit[idx] += val;
        idx += idx & -idx;
    }
}

//query function to find sum(1, idx) in BIT
int query(int idx)
{
    int res = 0;
    while ( idx > 0 )
    {
        res += bit[idx];
        idx -= idx & -idx;
    }
    return res;
}
```

```

void dfs(int v, int color[])
{
    //mark the node visited
    vis[v] = 1;

    //set visiting time of the node v
    vis_time[v] = ++tim;

    //use the color of node v to fill flat_tree[]
    flat_tree[tim] = color[v];

    vector<int>::iterator it;
    for (it=tree[v].begin(); it!=tree[v].end(); it++)
        if (!vis[*it])
            dfs(*it, color);

    // set ending time for node v
    end_time[v] = ++tim;

    // setting its color in flat_tree[] again
    flat_tree[tim] = color[v];
}

//function to add an edge(u, v) to the tree
void addEdge(int u, int v)
{
    tree[u].push_back(v);
    tree[v].push_back(u);
}

//function to build the table[] and also add
//first occurrences of elements to the BIT
void hashMarkFirstOccurrences(int n)
{
    for (int i = 1 ; i <= 2 * n ; i++)
    {
        table[flat_tree[i]].push_back(i);

        //if it is the first occurrence of the element
        //then add it to the BIT and increment traverser
        if (table[flat_tree[i]].size() == 1)
        {
            //add the occurrence to bit
            update(i, 1);

            //make traverser point to next occurrence
            traverser[flat_tree[i]]++;
        }
    }
}

```

```

        }
    }
}

//function to process all the queries and store thier answers
void processQueries()
{
    int j = 1;
    for (int i=0; i<queries.size(); i++)
    {
        //for each query remove all the ocurences before its li
        //li is the visiting time of the node
        //which is stored in first element of first pair
        for ( ; j < queries[i].first.first ; j++ )
        {
            int elem = flat_tree[j];

            //update(i, -1) removes an element at ith index
            //in the BIT
            update( table[elem] [traverser[elem] - 1], -1);

            //if there is another occurrence of the same element
            if ( traverser[elem] < table[elem].size() )
            {
                //add the occurrence to the BIT and
                //increment traverser
                update(table[elem] [ traverser[elem] ], 1);
                traverser[elem]++;
            }
        }

        //store the answer for the query, the index of the query
        //is the second element of the pair
        //And ri is stored in second element of the first pair
        ans[queries[i].second] = query(queries[i].first.second);
    }
}

// Count distinct colors in subtrees rooted with qVer[0],
// qVer[1], ...qVer[qn-1]
void countDistinctColors(int color[], int n, int qVer[], int qn)
{
    // build the flat_tree[], vis_time[] and end_time[]
    dfs(1, color);

    // add query for u = 3, 2 and 7
    for (int i=0; i<qn; i++)
        queries.push_back(make_pair(make_pair(vis_time[qVer[i]],


```

```

        end_time[qVer[i]]), i) );

// sort the queries in order of increasing vis_time
sort(queries.begin(), queries.end());

// make table[] and set '1' at first occurrences of elements
hashMarkFirstOccurrences(n);

// process queries
processQueries();

// print all the answers, in order asked
// in the question
for (int i=0; i<queries.size() ; i++)
{
    cout << "Distinct colors in the corresponding subtree"
    "is: " << ans[i] << endl;
}
}

//driver code
int main()
{
/*
     1
    / \
   2   3
  / \   | \
 4 5 6 7  8
    / \
   9 10 11
*/
int n = 11;
int color[] = {0, 2, 3, 3, 4, 1, 3, 4, 3, 2, 1, 1};

// add all the edges to the tree
addEdge(1, 2);
addEdge(1, 3);
addEdge(2, 4);
addEdge(2, 5);
addEdge(2, 6);
addEdge(3, 7);
addEdge(3, 8);
addEdge(7, 9);
addEdge(7, 10);
addEdge(7, 11);

int qVer[] = {3, 2, 7};

```

```
int qn = sizeof(qVer)/sizeof(qVer[0]);  
  
countDistinctColors(color, n, qVer, qn);  
  
return 0;  
}
```

Output:

```
Distinct colors in the corresponding subtree is:4  
Distinct colors in the corresponding subtree is:3  
Distinct colors in the corresponding subtree is:3
```

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

Source

<https://www.geeksforgeeks.org/querying-the-number-of-distinct-colors-in-a-subtree-of-a-colored-tree-using-bit/>

Chapter 141

Range LCM Queries

Range LCM Queries - GeeksforGeeks

Given an array of integers, evaluate queries of the form $\text{LCM}(l, r)$. There might be many queries, hence evaluate the queries efficiently.

$\text{LCM}(l, r)$ denotes the LCM of array elements
that lie between the index l and r
(inclusive of both indices)

Mathematically,

$\text{LCM}(l, r) = \text{LCM}(\text{arr}[l], \text{arr}[l+1], \dots, \text{arr}[r-1], \text{arr}[r])$

Examples:

```
Inputs : Array = {5, 7, 5, 2, 10, 12 ,11, 17, 14, 1, 44}
         Queries: LCM(2, 5), LCM(5, 10), LCM(0, 10)
Outputs: 60 15708 78540
Explanation : In the first query LCM(5, 2, 10, 12) = 60,
              similarly in other queries.
```

A naive solution would be to traverse the array for every query and calculate the answer by using,

$$\text{LCM}(a, b) = (a * b) / \text{GCD}(a, b)$$

However as the number of queries can be large, this solution would be impractical.

An efficient solution would be to use [segment tree](#). Recall that in this case, where no update is required, we can build the tree once and can use that repeatedly to answer the queries. Each node in the tree should store the LCM value for that particular segment and we can

use the same formula as above to combine the segments. Hence we can answer each query efficiently!

Below is a C++ solution for the same.

```
// LCM of given range queries using Segment Tree
#include <bits/stdc++.h>
using namespace std;

#define MAX 1000

// allocate space for tree
int tree[4*MAX];

// declaring the array globally
int arr[MAX];

// Function to return gcd of a and b
int gcd(int a, int b)
{
    if (a == 0)
        return b;
    return gcd(b%a, a);
}

//utility function to find lcm
int lcm(int a, int b)
{
    return a*b/gcd(a,b);
}

// Function to build the segment tree
// Node starts beginning index of current subtree.
// start and end are indexes in arr[] which is global
void build(int node, int start, int end)
{
    // If there is only one element in current subarray
    if (start==end)
    {
        tree[node] = arr[start];
        return;
    }

    int mid = (start+end)/2;

    // build left and right segments
    build(2*node, start, mid);
    build(2*node+1, mid+1, end);
```

```

// build the parent
int left_lcm = tree[2*node];
int right_lcm = tree[2*node+1];

tree[node] = lcm(left_lcm, right_lcm);
}

// Function to make queries for array range [l, r].
// Node is index of root of current segment in segment
// tree (Note that indexes in segment tree begin with 1
// for simplicity).
// start and end are indexes of subarray covered by root
// of current segment.
int query(int node, int start, int end, int l, int r)
{
    // Completely outside the segment, returning
    // 1 will not affect the lcm;
    if (end<l || start>r)
        return 1;

    // completely inside the segment
    if (l<=start && r>=end)
        return tree[node];

    // partially inside
    int mid = (start+end)/2;
    int left_lcm = query(2*node, start, mid, l, r);
    int right_lcm = query(2*node+1, mid+1, end, l, r);
    return lcm(left_lcm, right_lcm);
}

//driver function to check the above program
int main()
{
    //initialize the array
    arr[0] = 5;
    arr[1] = 7;
    arr[2] = 5;
    arr[3] = 2;
    arr[4] = 10;
    arr[5] = 12;
    arr[6] = 11;
    arr[7] = 17;
    arr[8] = 14;
    arr[9] = 1;
    arr[10] = 44;

    // build the segment tree
}

```

```
build(1, 0, 10);

// Now we can answer each query efficiently

// Print LCM of (2, 5)
cout << query(1, 0, 10, 2, 5) << endl;

// Print LCM of (5, 10)
cout << query(1, 0, 10, 5, 10) << endl;

// Print LCM of (0, 10)
cout << query(1, 0, 10, 0, 10) << endl;

return 0;
}
```

Output:

```
60
15708
78540
```

Source

<https://www.geeksforgeeks.org/range-lcm-queries/>

Chapter 142

Range Minimum Query (Square Root Decomposition and Sparse Table)

Range Minimum Query (Square Root Decomposition and Sparse Table) - GeeksforGeeks

We have an array $\text{arr}[0 \dots n-1]$. We should be able to efficiently find the minimum value from index L (query start) to R (query end) where $0 \leq L \leq R \leq n-1$. Consider a situation when there are many range queries.

Example:

```
Input: arr[] = {7, 2, 3, 0, 5, 10, 3, 12, 18};  
       query[] = [0, 4], [4, 7], [7, 8]
```

```
Output: Minimum of [0, 4] is 0  
        Minimum of [4, 7] is 3  
        Minimum of [7, 8] is 12
```

A **simple solution** is to run a loop from L to R and find minimum element in given range. This solution takes $O(n)$ time to query in worst case.

Another approach is to use **Segment tree**. With segment tree, preprocessing time is $O(n)$ and time to for range minimum query is $O(\log n)$. The extra space required is $O(n)$ to store the segment tree. Segment tree allows updates also in $O(\log n)$ time.

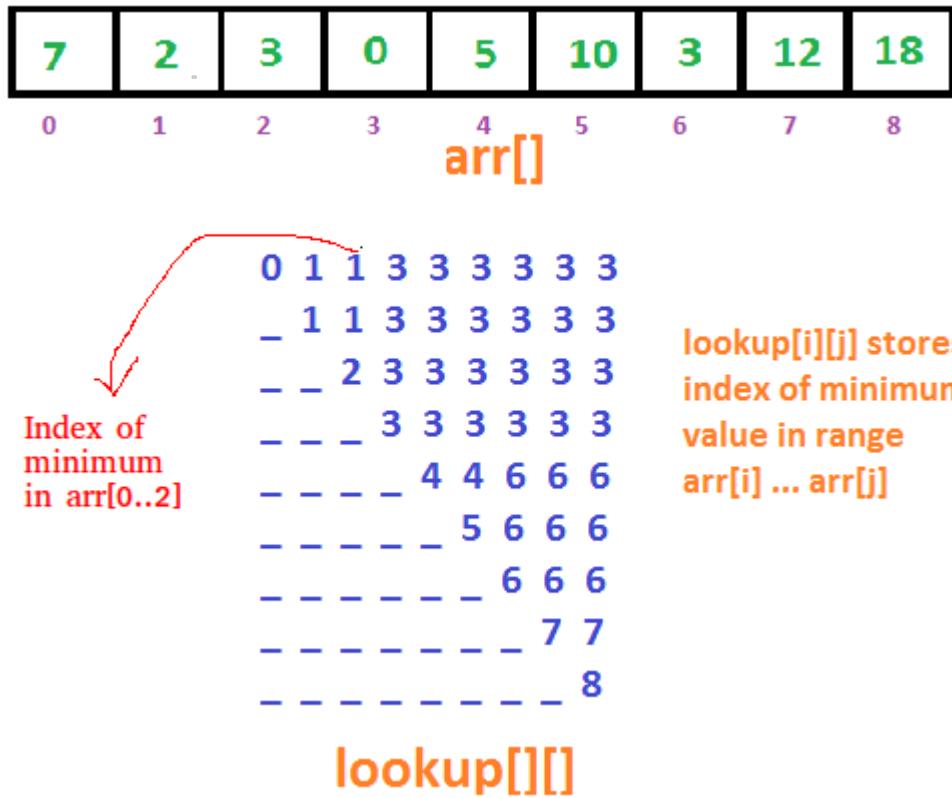
How to optimize query time when there are no update operations and there are many range minimum queries?

Below are different methods.

Method 1 (Simple Solution)

A Simple Solution is to create a 2D array $\text{lookup}[][]$ where an entry $\text{lookup}[i][j]$ stores the

minimum value in range $\text{arr}[i..j]$. Minimum of a given range can now be calculated in $O(1)$ time.



```
// C++ program to do range minimum query in O(1) time with O(n*n)
// extra space and O(n*n) preprocessing time.
#include<bits/stdc++.h>
using namespace std;
#define MAX 500

// lookup[i][j] is going to store index of minimum value in
// arr[i..j]
int lookup[MAX][MAX];

// Structure to represent a query range
struct Query
{
    int L, R;
};

// Fills lookup array lookup[n][n] for all possible values of
// query ranges
```

```

void preprocess(int arr[], int n)
{
    // Initialize lookup[][] for the intervals with length 1
    for (int i = 0; i < n; i++)
        lookup[i][i] = i;

    // Fill rest of the entries in bottom up manner
    for (int i=0; i<n; i++)
    {
        for (int j = i+1; j<n; j++)

            // To find minimum of [0,4], we compare minimum of
            // arr[lookup[0][3]] with arr[4].
            if (arr[lookup[i][j - 1]] < arr[j])
                lookup[i][j] = lookup[i][j - 1];
            else
                lookup[i][j] = j;
    }
}

// Prints minimum of given m query ranges in arr[0..n-1]
void RMQ(int arr[], int n, Query q[], int m)
{
    // Fill lookup table for all possible input queries
    preprocess(arr, n);

    // One by one compute sum of all queries
    for (int i=0; i<m; i++)
    {
        // Left and right boundaries of current range
        int L = q[i].L, R = q[i].R;

        // Print sum of current query range
        cout << "Minimum of [" << L << ", "
             << R << "] is " << arr[lookup[L][R]] << endl;
    }
}

// Driver program
int main()
{
    int a[] = {7, 2, 3, 0, 5, 10, 3, 12, 18};
    int n = sizeof(a)/sizeof(a[0]);
    Query q[] = {{0, 4}, {4, 7}, {7, 8}};
    int m = sizeof(q)/sizeof(q[0]);
    RMQ(a, n, q, m);
    return 0;
}

```

Output:

```
Minimum of [0, 4] is 0
Minimum of [4, 7] is 3
Minimum of [7, 8] is 12
```

This approach supports query in $O(1)$, but preprocessing takes $O(n^2)$ time. Also, this approach needs $O(n^2)$ extra space which may become huge for large input arrays.

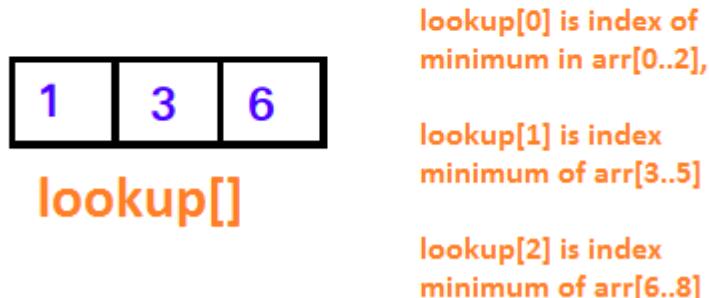
Method 2 (Square Root Decomposition)

We can use Square Root Decompositions to reduce space required in above method.

Preprocessing:

- 1) Divide the range $[0, n-1]$ into different blocks of \sqrt{n} each.
- 2) Compute minimum of every block of size \sqrt{n} and store the results.

Preprocessing takes $O(\sqrt{n} * \sqrt{n}) = O(n)$ time and $O(\sqrt{n})$ space.



Query:

- 1) To query a range $[L, R]$, we take minimum of all blocks that lie in this range. For left and right corner blocks which may partially overlap with given range, we linearly scan them to find minimum.

Time complexity of query is $O(\sqrt{n})$. Note that we have minimum of middle block directly accessible and there can be at most $O(\sqrt{n})$ middle blocks. There can be atmost two corner blocks that we may have to scan, so we may have to scan $2*O(\sqrt{n})$ elements of corner blocks. Therefore, overall time complexity is $O(\sqrt{n})$.

Refer [Sqrt \(or Square Root\) Decomposition Technique Set 1 \(Introduction\)](#) for details.

Method 3 (Sparse Table Algorithm)

The above solution requires only $O(\sqrt{n})$ space, but takes $O(\sqrt{n})$ time to query. Sparse table method supports query time **$O(1)$** with extra space **$O(n \log n)$** .

The idea is to precompute minimum of all subarrays of size 2^j where j varies from 0 to **Log n**. Like method 1, we make a lookup table. Here $\text{lookup}[i][j]$ contains minimum of range starting from i and of size 2^j . For example $\text{lookup}[0][3]$ contains minimum of range [0, 7] (starting with 0 and of size 2^3)

Preprocessing:

How to fill this lookup table? The idea is simple, fill in bottom up manner using previously computed values.

For example, to find minimum of range [0, 7], we can use minimum of following two.

- a) Minimum of range [0, 3]
- b) Minimum of range [4, 7]

Based on above example, below is formula,

```
// If arr[lookup[0][2]] <= arr[lookup[4][2]],  
// then lookup[0][3] = lookup[0][2]  
If arr[lookup[i][j-1]] <= arr[lookup[i+2j-1-1][j-1]]  
    lookup[i][j] = lookup[i][j-1]  
  
// If arr[lookup[0][2]] > arr[lookup[4][2]],  
// then lookup[0][3] = lookup[4][2]  
Else  
    lookup[i][j] = lookup[i+2j-1-1][j-1]
```



0 1 3 3	lookup[i][j] contains index of minimum in range from arr[i] to arr[i + 2^j - 1]
1 1 3 3	
2 3 3 _	
3 3 3 _	
4 4 6 _	
5 6 6 _	
6 6 _ _	
7 7 _ _	
8 _ _ _	

lookup[][]

Query:

For any arbitrary range $[l, R]$, we need to use ranges which are in powers of 2. The idea is to use closest power of 2. We always need to do at most one comparison (compare minimum of two ranges which are powers of 2). One range starts with L and ends with “L + closest-power-of-2”. The other range ends at R and starts with “R – same-closest-power-of-2 + 1”. For example, if given range is (2, 10), we compare minimum of two ranges (2, 9) and (3, 10).

Based on above example, below is formula,

```
// For (2,10), j = floor(Log2(10-2+1)) = 3
j = floor(Log(R-L+1))

// If arr[lookup[0][3]] <= arr[lookup[3][3]],
// then RMQ(2,10) = lookup[0][3]
If arr[lookup[L][j]] <= arr[lookup[R-(int)pow(2,j)+1][j]]
    RMQ(L, R) = lookup[L][j]

// If arr[lookup[0][3]] > arr[lookup[3][3]],
// then RMQ(2,10) = lookup[3][3]
Else
```

```
RMQ(L, R) = lookup[R-(int)pow(2,j)+1][j]
```

Since we do only one comparison, time complexity of query is O(1).

Below is C++ implementation of above idea.

```
// C++ program to do range minimum query in O(1) time with
// O(n Log n) extra space and O(n Log n) preprocessing time
#include<bits/stdc++.h>
using namespace std;
#define MAX 500

// lookup[i][j] is going to store index of minimum value in
// arr[i..j]. Ideally lookup table size should not be fixed and
// should be determined using n Log n. It is kept constant to
// keep code simple.
int lookup[MAX][MAX];

// Structure to represent a query range
struct Query
{
    int L, R;
};

// Fills lookup array lookup[][] in bottom up manner.
void preprocess(int arr[], int n)
{
    // Initialize M for the intervals with length 1
    for (int i = 0; i < n; i++)
        lookup[i][0] = i;

    // Compute values from smaller to bigger intervals
    for (int j=1; (1<<j)<=n; j++)
    {
        // Compute minimum value for all intervals with size 2^j
        for (int i=0; (i+(1<<j)-1) < n; i++)
        {
            // For arr[2][10], we compare arr[lookup[0][3]] and
            // arr[lookup[3][3]]
            if (arr[lookup[i][j-1]] < arr[lookup[i + (1<<(j-1))][j-1]])
                lookup[i][j] = lookup[i][j-1];
            else
                lookup[i][j] = lookup[i + (1 << (j-1))][j-1];
        }
    }
}

// Returns minimum of arr[L..R]
```

```

int query(int arr[], int L, int R)
{
    // For [2,10], j = 3
    int j = (int)log2(R-L+1);

    // For [2,10], we compare arr[lookup[0][3]] and
    // arr[lookup[3][3]],
    if (arr[lookup[L][j]] <= arr[lookup[R - (1<<j) + 1][j]])
        return arr[lookup[L][j]];

    else return arr[lookup[R - (1<<j) + 1][j]];
}

// Prints minimum of given m query ranges in arr[0..n-1]
void RMQ(int arr[], int n, Query q[], int m)
{
    // Fills table lookup[n][Log n]
    preprocess(arr, n);

    // One by one compute sum of all queries
    for (int i=0; i<m; i++)
    {
        // Left and right boundaries of current range
        int L = q[i].L, R = q[i].R;

        // Print sum of current query range
        cout << "Minimum of [" << L << ", "
              << R << "] is " << query(arr, L, R) << endl;
    }
}

// Driver program
int main()
{
    int a[] = {7, 2, 3, 0, 5, 10, 3, 12, 18};
    int n = sizeof(a)/sizeof(a[0]);
    Query q[] = {{0, 4}, {4, 7}, {7, 8}};
    int m = sizeof(q)/sizeof(q[0]);
    RMQ(a, n, q, m);
    return 0;
}

```

Output:

```

Minimum of [0, 4] is 0
Minimum of [4, 7] is 3
Minimum of [7, 8] is 12

```

So sparse table method supports query operation in $O(1)$ time with $O(n \log n)$ preprocessing time and $O(n \log n)$ space.

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

Improved By : [costom](#)

Source

<https://www.geeksforgeeks.org/range-minimum-query-for-static-array/>

Chapter 143

Range Queries for Longest Correct Bracket Subsequence

Range Queries for Longest Correct Bracket Subsequence - GeeksforGeeks

Given a bracket sequence or in other words a string S of length n, consisting of characters '(' and ')'. Find length of the maximum correct bracket subsequence of sequence for a given query range. *Note: A correct bracket sequence is the one that have matched bracket pairs or which contains another nested correct bracket sequence. For e.g (), (()), ()() are some correct bracket sequence.*

Examples:

```
Input : S = ()()()()
        Start Index of Range = 0,
        End Index of Range = 11
Output : 10
Explanation: Longest Correct Bracket Subsequence is ()()()
```

```
Input : S = ()()()()
        Start Index of Range = 1,
        End Index of Range = 2
Output : 0
```

Segment Trees can be used to solve this problem **efficiently**

At each node of the segment tree, we store the following:

- 1) a - Number of correctly matched pairs of brackets.
- 2) b - Number of unused open brackets.
- 3) c - Number of unused closed brackets.

(unused open bracket – means they can't be matched with any closing bracket, unused closed bracket – means they can't be matched with any opening bracket, for e.g S =)(contains an unused open and an unused closed bracket)

For each interval [L, R], we can match X number of unused open brackets '(' in interval [L, MID] with unused closed brackets ')' in interval [MID + 1, R] where

$$X = \min(\text{number of unused } '(' \text{ in } [L, \text{MID}], \text{number of unused } ')' \text{ in } [\text{MID} + 1, R])$$

Hence, X is also the number of correctly matched pairs built by combination.

So, for interval [L, R]

- 1) Total number of correctly matched pairs becomes the sum of correctly matched pairs in left child and correctly matched pairs in right child and number of combinations of unused '(' and unused ')' from left and right child respectively.

$$a[L, R] = a[L, \text{MID}] + a[\text{MID} + 1, R] + X$$

- 2) Total number of unused open brackets becomes the sum of unused open brackets in left child and unused open brackets in right child minus X (minus – because we used X unused '(' from left child to match with unused ')' from right child).

$$a[L, R] = b[L, \text{MID}] + b[\text{MID} + 1, R] - X$$

- 3) Similarly, for unused closed brackets, following relation holds.

$$a[L, R] = c[L, \text{MID}] + c[\text{MID} + 1, R] - X$$

where a, b and c are the representations described above for each node to be stored in.

Below is the implementation of above approach in C++.

```
/* CPP Program to find the longest correct
   bracket subsequence in a given range */
#include <bits/stdc++.h>
using namespace std;

/* Declaring Structure for storing
   three values in each segment tree node */
struct Node {
    int pairs;
    int open; // unused
    int closed; // unused

    Node()
    {
        pairs = open = closed = 0;
    }
};

Node;
```

```

// A utility function to get the middle index from corner indexes.
int getMid(int s, int e) { return s + (e - s) / 2; }

// Returns Parent Node after merging its left and right child
Node merge(Node leftChild, Node rightChild)
{
    Node parentNode;
    int minMatched = min(leftChild.open, rightChild.closed);
    parentNode.pairs = leftChild.pairs + rightChild.pairs + minMatched;
    parentNode.open = leftChild.open + rightChild.open - minMatched;
    parentNode.closed = leftChild.closed + rightChild.closed - minMatched;
    return parentNode;
}

// A recursive function that constructs Segment Tree
// for string[ss..se]. si is index of current node in
// segment tree st
void constructSTUtil(char str[], int ss, int se, Node* st,
                     int si)
{
    // If there is one element in string, store it in
    // current node of segment tree and return
    if (ss == se) {

        // since it contains one element, pairs
        // will be zero
        st[si].pairs = 0;

        // check whether that one element is opening
        // bracket or not
        st[si].open = (str[ss] == '(' ? 1 : 0);

        // check whether that one element is closing
        // bracket or not
        st[si].closed = (str[ss] == ')' ? 1 : 0);

        return;
    }

    // If there are more than one elements, then recur
    // for left and right subtrees and store the relation
    // of values in this node
    int mid = getMid(ss, se);
    constructSTUtil(str, ss, mid, st, si * 2 + 1);
    constructSTUtil(str, mid + 1, se, st, si * 2 + 2);

    // Merge left and right child into the Parent Node
    st[si] = merge(st[si * 2 + 1], st[si * 2 + 2]);
}

```

```

}

/* Function to construct segment tree from given
   string. This function allocates memory for segment
   tree and calls constructSTUtil() to fill the
   allocated memory */
Node* constructST(char str[], int n)
{
    // Allocate memory for segment tree

    // Height of segment tree
    int x = (int)(ceil(log2(n)));

    // Maximum size of segment tree
    int max_size = 2 * (int)pow(2, x) - 1;

    // Declaring array of structure Allocate memory
    Node* st = new Node[max_size];

    // Fill the allocated memory st
    constructSTUtil(str, 0, n - 1, st, 0);

    // Return the constructed segment tree
    return st;
}

/* A Recursive function to get the desired
   Maximum Sum Sub-Array,
The following are parameters of the function-

st      --> Pointer to segment tree
si --> Index of the segment tree Node
ss & se  --> Starting and ending indexes of the
               segment represented by
               current Node, i.e., tree[index]
qs & qe  --> Starting and ending indexes of query range */
Node queryUtil(Node* st, int ss, int se, int qs,
               int qe, int si)
{
    // No overlap
    if (ss > qe || se < qs) {

        // returns a Node for out of bounds condition
        Node nullNode;
        return nullNode;
    }

    // Complete overlap

```

```

if (ss >= qs && se <= qe) {
    return st[si];
}

// Partial Overlap Merge results of Left
// and Right subtrees
int mid = getMid(ss, se);
Node left = queryUtil(st, ss, mid, qs, qe, si * 2 + 1);
Node right = queryUtil(st, mid + 1, se, qs, qe, si * 2 + 2);

// merge left and right subtree query results
Node res = merge(left, right);
return res;
}

/* Returns the maximum length correct bracket
subsequence between start and end
It mainly uses queryUtil(). */
int query(Node* st, int qs, int qe, int n)
{
    Node res = queryUtil(st, 0, n - 1, qs, qe, 0);

    // since we are storing numbers pairs
    // and have to return maximum length, hence
    // multiply no of pairs by 2
    return 2 * res.pairs;
}

// Driver Code
int main()
{
    char str[] = "())(()))(";
    int n = strlen(str);

    // Build segment tree from given string
    Node* st = constructST(str, n);

    int startIndex = 0, endIndex = 11;
    cout << "Maximum Length Correct Bracket"
        " Subsequence between "
        << startIndex << " and " << endIndex << " = "
        << query(st, startIndex, endIndex, n) << endl;

    startIndex = 1, endIndex = 2;
    cout << "Maximum Length Correct Bracket"
        " Subsequence between "
        << startIndex << " and " << endIndex << " = "
        << query(st, startIndex, endIndex, n) << endl;
}

```

```
    return 0;  
}
```

Output:

```
Maximum Length Correct Bracket Subsequence between 0 and 11 = 10  
Maximum Length Correct Bracket Subsequence between 1 and 2 = 0
```

Time complexity for each query is **O(logN)**, where N is the size of string.

Source

<https://www.geeksforgeeks.org/range-queries-longest-correct-bracket-subsequence/>

Chapter 144

Range and Update Query for Chessboard Pieces

Range and Update Query for Chessboard Pieces - GeeksforGeeks

Given N pieces of chessboard all being ‘white’ and a number of queries Q. There are two types of queries :

1. **Update** : Given indices of a range [L, R]. Paint all the pieces with their respective opposite color between L and R (i.e. white pieces should be painted with black color and black pieces should be painted with white color).
2. **Get** : Given indices of a range [L, R]. Find out the number of black pieces between L and R.

Let us represent ‘white’ pieces with ‘0’ and ‘black’ pieces with ‘1’.

Prerequisites: Segment Trees Lazy Propagation

Examples :

```
Input : N = 4, Q = 5
        Get : L = 0, R = 3
        Update : L = 1, R = 2
        Get : L = 0, R = 1
        Update : L = 0, R = 3
        Get : L = 0, R = 3
Output : 0
         1
         2
```

Explanation :

Query1 : A[] = { 0, 0, 0, 0 } Since initially all pieces are white, number of black pieces will

be zero.

Query2 : $A[] = \{ 0, 1, 1, 0 \}$

Query3 : Number of black pieces in $[0, 1] = 1$

Query4 : Change the color to its opposite color in $[0, 3]$, $A[] = \{ 1, 0, 0, 1 \}$

Query5 : Number of black pieces in $[0, 3] = 2$

Naive Approach :

Update(L, R) : Iterate over the subarray from L to R and change the color of all the pieces (i.e. change 0 to 1 and 1 to 0)

Get(L, R) : To get the number of black pieces, simply count the number of ones in range $[L, R]$.

Both update and getBlackPieces() function will have $O(N)$ time complexity. The time complexity in worst case is $O(Q * N)$ where Q is number of queries and N is number of chessboard pieces.

Efficient Approach :

An efficient method to solve this problem is by using **Segment Trees** which can reduce the time complexity of update and getBlackPieces functions to $O(\log N)$.

Build Structure: Each leaf node of segment tree will contain either 0 or 1 depending upon the color of the piece (i.e. if the piece is black, node will contain 1 otherwise 0). Internal nodes will contain the sum of ones or number of black pieces of its left child and right child. Thus, the root node will give us the total number of black pieces in the whole array $[0..N-1]$

Update Structure : Point updates takes $O(\log(N))$ time but when there are range updates, optimize the updates using **Lazy Propagation**. Below is the modified update method.

```
UpdateRange(ss, se)
1. If current node's range lies completely in update query range.
...a) Value of current node becomes the difference of total count
    of black pieces in the subtree of current node and current
    value of node, i.e. tree[curNode] = (se - ss + 1) - tree[curNode]
...b) Provide the lazy value to its children by setting
    lazy[2*curNode] = 1 - lazy[2*curNode]
    lazy[2*curNode + 1] = 1 - lazy[2*curNode + 1]

2. If the current node's lazy value is not zero, first update
    it and provide lazy value to children.

3. Partial Overlap of current node's range with query range
...a) Recurse for left and right child
...b) Combine the results of step (a)
```

Query Structure : Query Structure will also change a bit in the same way as update structure by checking pending updates and updating them to get the correct query output.

Below is the implementation of above approach in C++.

```
// C code for queries on chessboard
#include <bits/stdc++.h>

using namespace std;

// A utility function to get the
// middle index from corner indexes.
int getMid(int s, int e)
{
    return s + (e - s) / 2;
}

/* A recursive function to get the
   sum of values in given range of
   the array. The following are
   parameters for this function.
si --> Index of current node in
      the segment tree. Initially
      0 is passed as root is always
      at index 0
ss & se --> Starting and ending
            indexes of the segment
            represented by current
            node, i.e., tree[si]
qs & qe --> Starting and ending
            indexes of query range */
int getSumUtil(int* tree, int* lazy, int ss,
               int se, int qs, int qe, int si)
{
    // If lazy flag is set for current node
    // of segment tree, then there are some
    // pending updates. So we need to make
    // sure that the pending updates are done
    // before processing the sub sum query
    if (lazy[si] != 0)
    {
        // Make pending updates to this node.
        // Note that this node represents
        // sum of elements in arr[ss..se]
        tree[si] = (se - ss + 1) - tree[si];

        // checking if it is not leaf node
        // because if it is leaf node then
        // we cannot go further
        if (ss != se)
        {
            // Since we are not yet updating
            // children os si, we need to set
```

```

// lazy values for the children
lazy[si * 2 + 1] =
    1 - lazy[si * 2 + 1];

lazy[si * 2 + 2] =
    1 - lazy[si * 2 + 2];
}

// unset the lazy value for current
// node as it has been updated
lazy[si] = 0;
}

// Out of range
if (ss > se || ss > qe || se < qs)
    return 0;

// At this point we are sure that pending
// lazy updates are done for current node.
// So we can return value (same as it was
// for query in our previous post)

// If this segment lies in range
if (ss >= qs && se <= qe)
    return tree[si];

// If a part of this segment overlaps
// with the given range
int mid = (ss + se) / 2;
return getSumUtil(tree, lazy, ss, mid,
                  qs, qe, 2 * si + 1) +
       getSumUtil(tree, lazy, mid + 1,
                  se, qs, qe, 2 * si + 2);
}

// Return sum of elements in range from index
// qs (query start) to qe (query end). It
// mainly uses getSumUtil()
int getSum(int* tree, int* lazy, int n,
           int qs, int qe)
{
    // Check for erroneous input values
    if (qs < 0 || qe > n - 1 || qs > qe)
    {
        printf("Invalid Input");
        return -1;
    }
}

```

```

        return getSumUtil(tree, lazy, 0, n - 1,
                           qs, qe, 0);
    }

/* si -> index of current node in segment tree
   ss and se -> Starting and ending indexes of
                  elements for which current
                  nodes stores sum.
   us and ue -> starting and ending indexes
                  of update query */
void updateRangeUtil(int* tree, int* lazy, int si,
                     int ss, int se, int us, int ue)
{
    // If lazy value is non-zero for current node
    // of segment tree, then there are some
    // pending updates. So we need to make sure that
    // the pending updates are done before making
    // new updates. Because this value may be used by
    // parent after recursive calls (See last line
    // of this function)
    if (lazy[si] != 0) {

        // Make pending updates using value stored
        // in lazy nodes
        tree[si] = (se - ss + 1) - tree[si];

        // checking if it is not leaf node because if
        // it is leaf node then we cannot go further
        if (ss != se)
        {
            // We can postpone updating children
            // we don't need their new values now.
            // Since we are not yet updating children
            // of si, we need to set lazy flags for
            // the children
            lazy[si * 2 + 1] = 1 - lazy[si * 2 + 1];
            lazy[si * 2 + 2] = 1 - lazy[si * 2 + 2];
        }

        // Set the lazy value for current node
        // as 0 as it has been updated
        lazy[si] = 0;
    }

    // out of range
    if (ss > se || ss > ue || se < us)
        return;
}

```

```

// Current segment is fully in range
if (ss >= us && se <= ue) {

    // Add the difference to current node
    tree[si] = (se - ss + 1) - tree[si];

    // same logic for checking leaf
    // node or not
    if (ss != se)
    {
        // This is where we store values in
        // lazy nodes, rather than updating
        // the segment tree itself. Since we
        // don't need these updated values now
        // we postpone updates by storing
        // values in lazy[]
        lazy[si * 2 + 1] = 1 - lazy[si * 2 + 1];
        lazy[si * 2 + 2] = 1 - lazy[si * 2 + 2];
    }
    return;
}

// If not completely in rang, but overlaps,
// recur for children
int mid = (ss + se) / 2;
updateRangeUtil(tree, lazy, si * 2 + 1,
                ss, mid, us, ue);
updateRangeUtil(tree, lazy, si * 2 + 2,
                mid + 1, se, us, ue);

// And use the result of children calls
// to update this node
tree[si] = tree[si * 2 + 1] + tree[si * 2 + 2];
}

// Function to update a range of values
// in segment tree
/* us and eu -> starting and ending indexes
   of update query ue -> ending index
   of update query, diff -> which we need
   to add in the range us to ue */
void updateRange(int* tree, int* lazy,
                 int n, int us, int ue)
{
    updateRangeUtil(tree, lazy, 0, 0, n - 1, us, ue);
}

// A recursive function that constructs

```

```

// Segment Tree for array[ss..se]. si is
// index of current node in segment tree st
int constructSTUtil(int arr[], int ss, int se,
                    int* tree, int si)
{
    // If there is one element in array, store
    // it in current node of segment tree and return
    if (ss == se)
    {
        tree[si] = arr[ss];
        return arr[ss];
    }

    // If there are more than one elements, then
    // recur for left and right subtrees and
    // store the sum of values in this node
    int mid = getMid(ss, se);
    tree[si] = constructSTUtil(arr, ss, mid,
                               tree, si * 2 + 1) +
               constructSTUtil(arr, mid + 1,
                               se, tree, si * 2 + 2);
    return tree[si];
}

/* Function to construct segment tree from
   given array. This function allocates
   memory for segment tree and calls
   constructSTUtil() to fill the
   allocated memory */
int* constructST(int arr[], int n)
{
    // Allocate memory for segment tree

    // Height of segment tree
    int x = (int)(ceil(log2(n)));

    // Maximum size of segment tree
    int max_size = 2 * (int)pow(2, x) - 1;

    // Allocate memory
    int* tree = new int[max_size];

    // Fill the allocated memory st
    constructSTUtil(arr, 0, n - 1, tree, 0);

    // Return the constructed segment tree
    return tree;
}

```

```
/* Function to construct lazy array for
   segment tree. This function allocates
   memory for lazy array */
int* constructLazy(int arr[], int n)
{
    // Allocate memory for lazy array

    // Height of lazy array
    int x = (int)(ceil(log2(n)));

    // Maximum size of lazy array
    int max_size = 2 * (int)pow(2, x) - 1;

    // Allocate memory
    int* lazy = new int[max_size];

    // Return the lazy array
    return lazy;
}

// Driver program to test above functions
int main()
{
    // Intialize the array to zero
    // since all pieces are white
    int arr[] = { 0, 0, 0, 0 };
    int n = sizeof(arr) / sizeof(arr[0]);

    // Build segment tree from given array
    int* tree = constructST(arr, n);

    // Allocate memory for Lazy array
    int* lazy = constructLazy(arr, n);

    // Print number of black pieces
    // from index 0 to 3
    cout << "Black Pieces in given range = "
        << getSum(tree, lazy, n, 0, 3) << endl;

    // UpdateRange: Change color of pieces
    // from index 1 to 2
    updateRange(tree, lazy, n, 1, 2);

    // Print number of black pieces
    // from index 0 to 1
    cout << "Black Pieces in given range = "
        << getSum(tree, lazy, n, 0, 1) << endl;
}
```

```
// UpdateRange: Change color of
// pieces from index 0 to 3
updateRange(tree, lazy, n, 0, 3);

// Print number of black pieces
// from index 0 to 3
cout << "Black Pieces in given range = "
    << getSum(tree, lazy, n, 0, 3) << endl;

return 0;
}
```

Output:

```
Black Pieces in given range = 0
Black Pieces in given range = 1
Black Pieces in given range = 2
```

Time Complexity : Each query and each update will take $O(\log(N))$ time, where N is the number of chessboard pieces. Hence for Q queries, worst case complexity will be $(Q * \log(N))$

Source

<https://www.geeksforgeeks.org/range-update-query-chessboard-pieces/>

Chapter 145

Range query for Largest Sum Contiguous Subarray

Range query for Largest Sum Contiguous Subarray - GeeksforGeeks

Given a number N, and Q queries of two types 1 and 2. Task is to write a code for the given query where, in type-1, given l and r, and task is to print the [Largest sum Contiguous Subarray](#) and for type 2, given type, index, and value, update value to A_{index}.

Examples :

```
Input : a = {-2, -3, 4, -1, -2, 1, 5, -3}
        1st query : 1 5 8
        2nd query : 2 1 10
        3rd query : 1 1 3
Output : Answer to 1st query : 6
Answer to 3rd query : 11
```

Explanation : In the first query, task is to print the largest sum of a contiguous subarray in range 5-8, which consists of {-2, 1, 5, -3}. The largest sum is 6, which is formed by the subarray {1, 5}. In the second query, an update operation is done, which updates a[1] to 10, hence the sequence is {10, -3, 4, -1, -2, 1, 5, -3}. In the third query, task is to print the largest sum of a contiguous subarray in range 1-3, which consists of {10, -3, 4}. The largest sum is 11, which is formed by the subarray {10, -3, 4}.

A **naive approach** is to use [Kadane's algorithm](#) for every type-1 query. The complexity of every type-1 query is O(n). The type-2 query is done in O(1).

Efficient Approach :

An efficient approach is to build a segment tree where each node stores four values(sum, prefixsum, suffixsum, maxsum), and do a range query on it to find the answer to every query. The nodes of segment tree store the four values as mentioned above. The parent will store the merging of left and right child. The parent node stores the value as mentioned below :

```

parent.sum = left.sum + right.sum
parent.prefixsum = max(left.prefixsum, left.sum + right.prefixsum)
parent.suffixsum = max(right.suffixsum, right.sum + left.suffixsum)
parent.maxsum = max(parent.prefixsum, parent.suffixsum, left.maxsum,
right.maxsum, left.suffixsum + right.prefixsum)

```

Parent node stores the following :

- Parent node's sum is the summation of left and right child sum.
- Parent node's prefix sum will be equivalent to maximum of left child's prefix sum or left child sum + right child prefix sum.
- Parent node's suffix sum will be equal to right child suffix sum or right child sum + suffix sum of left child
- Parent node's maxsum will be the maximum of prefixsum or suffix sum of parent or the left or right child's maxsum or the summation of suffixsum of left child and prefixsum of right child.

Representation of Segment trees :

1. Leaf Nodes are the elements of the input array.
2. Each internal node represents some merging of the leaf nodes. The merging may be different for different problems. For this problem, merging is done as given above.

An array representation of tree is used to represent Segment Trees. For each node at index i , the left child is at index $2 * i + 1$, right child at $2 * i + 2$ and the parent is at $(i - 1) / 2$.

Construction of Segment Tree from given array :

Start with a segment $\text{arr}[0 \dots n-1]$. and every time divide the current segment into two halves(if it has not yet become a segment of length 1), and then call the same procedure on both halves, and for each such segment, store the values in all the four variables as given in the formulae above.

Update a given value in array and segment Tree :

Start with the complete segment of the array provided to us. Every time divide the array into two halves, **ignore the half in which the index to be updated is not present**. Keep on ignoring halves at every step until reach the leaf node, where update the value to the given index. Now, merge the updated values according to the given formulae to all the nodes that are present in the path we have traversed.

Answering a query:

For every query, move to the left and right halves of the tree. Whenever the given range completely overlaps any halve of a tree, return the Node from that half without traversing further in that region. When a halve of the tree completely lies outside the given range, return INT_MIN. On partial overlapping of range, traverse in left and right halves and return accordingly.

Below is the implementation of the above idea :

```
// CPP program to find Largest Sum Contiguous
// Subarray in a given range with updates
#include <bits/stdc++.h>
using namespace std;

// Structure to store
// 4 values that are to be stored
// in the nodes
struct node {
    int sum, prefixsum, suffixsum, maxsum;
};

// array to store the segment tree
node tree[4 * 100];

// function to build the tree
void build(int arr[], int low, int high, int index)
{
    // the leaf node
    if (low == high) {
        tree[index].sum = arr[low];
        tree[index].prefixsum = arr[low];
        tree[index].suffixsum = arr[low];
        tree[index].maxsum = arr[low];
    }
    else {
        int mid = (low + high) / 2;

        // left subtree
        build(arr, low, mid, 2 * index + 1);

        // right subtree
        build(arr, mid + 1, high, 2 * index + 2);

        // parent node's sum is the summation
        // of left and right child
        tree[index].sum = tree[2 * index + 1].sum +
                         tree[2 * index + 2].sum;

        // parent node's prefix sum will be equivalent
        // to maximum of left child's prefix sum or left
        // child sum + right child prefix sum.
        tree[index].prefixsum =
            max(tree[2 * index + 1].prefixsum,
                tree[2 * index + 1].sum +
                tree[2 * index + 2].prefixsum);

        // parent node's suffix sum will be equal to right
    }
}
```

```

// child suffix sum or rigth child sum + suffix
// sum of left child
tree[index].suffixsum =
    max(tree[2 * index + 2].suffixsum,
        tree[2 * index + 2].sum +
        tree[2 * index + 1].suffixsum);

// maxum will be the maximum of prefix, suffix of
// parent or maximum of left child or right child
// and summation of left child's suffix and right
// child's prefix.
tree[index].maxsum =
    max(tree[index].prefixsum,
        max(tree[index].suffixsum,
            max(tree[2 * index + 1].maxsum,
                max(tree[2 * index + 2].maxsum,
                    tree[2 * index + 1].suffixsum +
                    tree[2 * index + 2].prefixsum))));

}

}

// function to update the tree
void update(int arr[], int index, int low, int high,
            int idx, int value)
{
    // the node to be updated
    if (low == high) {
        tree[index].sum = value;
        tree[index].prefixsum = value;
        tree[index].suffixsum = value;
        tree[index].maxsum = value;
    }
    else {

        int mid = (low + high) / 2;

        // if node to be updated is in left subtree
        if (idx <= mid)
            update(arr, 2 * index + 1, low, mid, idx, value);

        // if node to be updated is in right subtree
        else
            update(arr, 2 * index + 2, mid + 1,
                  high, idx, value);

        // parent node's sum is the summation of left
        // and rigth child
        tree[index].sum = tree[2 * index + 1].sum +
    }
}

```

```

        tree[2 * index + 2].sum;

    // parent node's prefix sum will be equivalent
    // to maximum of left child's prefix sum or left
    // child sum + right child prefix sum.
    tree[index].prefixsum =
        max(tree[2 * index + 1].prefixsum,
            tree[2 * index + 1].sum +
            tree[2 * index + 2].prefixsum);

    // parent node's suffix sum will be equal to right
    // child suffix sum or right child sum + suffix
    // sum of left child
    tree[index].suffixsum =
        max(tree[2 * index + 2].suffixsum,
            tree[2 * index + 2].sum +
            tree[2 * index + 1].suffixsum);

    // maxsum will be the maximum of prefix, suffix of
    // parent or maximum of left child or right child
    // and summation of left child's suffix and
    // right child's prefix.
    tree[index].maxsum =
        max(tree[index].prefixsum,
            max(tree[index].suffixsum,
                max(tree[2 * index + 1].maxsum,
                    max(tree[2 * index + 2].maxsum,
                        tree[2 * index + 1].suffixsum +
                        tree[2 * index + 2].prefixsum))));

    }

}

// function to return answer to every type-1 query
node query(int arr[], int index, int low,
           int high, int l, int r)
{
    // initially all the values are INT_MIN
    node result;
    result.sum = result.prefixsum =
    result.suffixsum =
    result.maxsum = INT_MIN;

    // range does not lies in this subtree
    if (r < low || high < l)
        return result;

    // complete overlap of range
    if (l <= low && high <= r)

```

```
    return tree[index];

    int mid = (low + high) / 2;

    // right subtree
    if (l > mid)
        return query(arr, 2 * index + 2,
                     mid + 1, high, l, r);

    // left subtree
    if (r <= mid)
        return query(arr, 2 * index + 1,
                     low, mid, l, r);

    node left = query(arr, 2 * index + 1,
                       low, mid, l, r);
    node right = query(arr, 2 * index + 2,
                        mid + 1, high, l, r);

    // finding the maximum and returning it
    result.sum = left.sum + right.sum;
    result.prefixsum = max(left.prefixsum, left.sum +
                           right.prefixsum);

    result.suffixsum = max(right.suffixsum,
                           right.sum + left.suffixsum);
    result.maxsum = max(result.prefixsum,
                         max(result.suffixsum,
                             max(left.maxsum,
                                 max(right.maxsum,
                                     left.suffixsum + right.prefixsum))));

    return result;
}

// Driver Code
int main()
{
    int a[] = { -2, -3, 4, -1, -2, 1, 5, -3 };
    int n = sizeof(a) / sizeof(a[0]);

    // build the tree
    build(a, 0, n - 1, 0);

    // 1st query type-1
    int l = 5, r = 8;
    cout << query(a, 0, 0, n - 1, l - 1, r - 1).maxsum;
    cout << endl;
```

```
// 2nd type-2 query
int index = 1;
int value = 10;
a[index - 1] = value;
update(a, 0, 0, n - 1, index - 1, value);

// 3rd type-1 query
l = 1, r = 3;
cout << query(a, 0, 0, n - 1, l - 1, r - 1).maxsum;

return 0;
}
```

Output:

```
6
11
```

Time Complexity : $O(n \log n)$ for building the tree, $O(\log n)$ for every type-1 query, $O(1)$ for type-2 query.

Improved By : [FelipeNoronha](#)

Source

<https://www.geeksforgeeks.org/range-query-largest-sum-contiguous-subarray/>

Chapter 146

Range sum query using Sparse Table

Range sum query using Sparse Table - GeeksforGeeks

We have an array arr[]. We need to find the sum of all the elements in the range L and R where $0 \leq L \leq R \leq n-1$. Consider a situation when there are many range queries.

Examples:

```
Input : 3 7 2 5 8 9
        query(0, 5)
        query(3, 5)
        query(2, 4)
Output : 34
         22
         15
```

Note : array is 0 based indexed
and queries too.

Since there are no updates/modifications, we use [Sparse table](#) to answer queries efficiently.
In sparse table, we break queries in powers of 2.

Suppose we are asked to compute sum of
elements from arr[i] to arr[i+12].
We do the following:

```
// Use sum of 8 (or 23) elements
table[i][3] = sum(arr[i], arr[i + 1], ...
                  arr[i + 7]).
```

```
// Use sum of 4 elements  
table[i+8][2] = sum(arr[i+8], arr[i+9], ..  
                     arr[i+11]).  
  
// Use sum of single element  
table[i + 12][0] = sum(arr[i + 12]).
```

Our result is sum of above values.

Notice that it took only 4 actions to compute result over subarray of size 13.

C++

```
// CPP program to find the sum in a given  
// range in an array using sparse table.  
#include <bits/stdc++.h>  
using namespace std;  
  
// Because 2^17 is larger than 10^5  
const int k = 16;  
  
// Maximum value of array  
const int N = 1e5;  
  
// k + 1 because we need to access  
// table[r][k]  
long long table[N][k + 1];  
  
// it builds sparse table.  
void buildSparseTable(int arr[], int n)  
{  
    for (int i = 0; i < n; i++)  
        table[i][0] = arr[i];  
  
    for (int j = 1; j <= k; j++)  
        for (int i = 0; i <= n - (1 << j); i++)  
            table[i][j] = table[i][j - 1] +  
                         table[i + (1 << (j - 1))][j - 1];  
}  
  
// Returns the sum of the elements in the range  
// L and R.  
long long query(int L, int R)  
{  
    // boundaries of next query, 0-indexed  
    long long answer = 0;  
    for (int j = k; j >= 0; j--) {  
        if (L + (1 << j) - 1 <= R) {
```

```
        answer = answer + table[L][j];

        // instead of having L', we
        // increment L directly
        L += 1 << j;
    }
}
return answer;
}

// Driver program.
int main()
{
    int arr[] = { 3, 7, 2, 5, 8, 9 };
    int n = sizeof(arr) / sizeof(arr[0]);

    buildSparseTable(arr, n);

    cout << query(0, 5) << endl;
    cout << query(3, 5) << endl;
    cout << query(2, 4) << endl;

    return 0;
}
```

Java

```
// Java program to find the sum
// in a given range in an array
// using sparse table.
class GFG
{

    // Because 2^17 is larger than 10^5
    static int k = 16;

    // Maximum value of array
    static int N = 100000;

    // k + 1 because we need
    // to access table[r][k]
    static long table[][] = new long[N][k + 1];

    // it builds sparse table.
    static void buildSparseTable(int arr[],
                                int n)
    {
        for (int i = 0; i < n; i++)
```

```

        table[i][0] = arr[i];

        for (int j = 1; j <= k; j++)
            for (int i = 0; i <= n - (1 << j); i++)
                table[i][j] = table[i][j - 1] +
                table[i + (1 << (j - 1))][j - 1];
    }

    // Returns the sum of the
    // elements in the range L and R.
    static long query(int L, int R)
    {
        // boundaries of next query,
        // 0-indexed
        long answer = 0;
        for (int j = k; j >= 0; j--)
        {
            if (L + (1 << j) - 1 <= R)
            {
                answer = answer + table[L][j];

                // instead of having L', we
                // increment L directly
                L += 1 << j;
            }
        }
        return answer;
    }

    // Driver Code
    public static void main(String args[])
    {
        int arr[] = { 3, 7, 2, 5, 8, 9 };
        int n = arr.length;

        buildSparseTable(arr, n);

        System.out.println(query(0, 5));
        System.out.println(query(3, 5));
        System.out.println(query(2, 4));
    }
}

// This code is contributed
// by Kirti_Mangal

```

C#

```
// C# program to find the
// sum in a given range
// in an array using
// sparse table.
using System;

class GFG
{
    // Because 2^17 is
    // larger than 10^5
    static int k = 16;

    // Maximum value
    // of array
    static int N = 100000;

    // k + 1 because we
    // need to access table[r,k]
    static long [,]table =
        new long[N, k + 1];

    // it builds sparse table.
    static void buildSparseTable(int []arr,
                                int n)
    {
        for (int i = 0; i < n; i++)
            table[i, 0] = arr[i];

        for (int j = 1; j <= k; j++)
            for (int i = 0;
                 i <= n - (1 << j); i++)
                table[i, j] = table[i, j - 1] +
                    table[i + (1 << (j - 1)), j - 1];
    }

    // Returns the sum of the
    // elements in the range
    // L and R.
    static long query(int L, int R)
    {
        // boundaries of next
        // query, 0-indexed
        long answer = 0;
        for (int j = k; j >= 0; j--)
        {
            if (L + (1 << j) - 1 <= R)
            {
                answer = answer +

```

```
        table[L, j];

        // instead of having
        // L', we increment
        // L directly
        L += 1 << j;
    }
}
return answer;
}

// Driver Code
static void Main()
{
    int []arr = new int[]{3, 7, 2,
                         5, 8, 9};
    int n = arr.Length;

    buildSparseTable(arr, n);

    Console.WriteLine(query(0, 5));
    Console.WriteLine(query(3, 5));
    Console.WriteLine(query(2, 4));
}
}

// This code is contributed by
// Manish Shaw(manishshaw1)
```

Output:

```
34
22
15
```

This algorithm for answering queries with Sparse Table works in $O(k)$, which is $O(\log(n))$, because we choose minimal k such that $2^k + 1 > n$.

Time complexity of sparse table construction : Outer loop runs in $O(k)$, inner loop runs in $O(n)$. Thus, in total we get $O(n * k) = O(n * \log(n))$

Improved By : [manishshaw1](#), [Kirti_Mangal](#)

Source

<https://www.geeksforgeeks.org/range-sum-query-using-sparse-table/>

Chapter 147

Reconstructing Segment Tree

Reconstructing Segment Tree - GeeksforGeeks

We are given $2^N - 1$ integers. We need to check whether it is possible to construct a [Range Minimum Query segment tree](#) for an array of N distinct integers from these integers. If so, we must output the segment tree array. N is given to be a power of 2.

An [RMQ segment tree](#) is a binary tree where each node is equal to the minimum value of its children. This type of tree is used to efficiently find the minimum value of elements in a given range.

```
Input : 1 1 1 1 2 2 3 3 3 4 4 5 6 7 8  
Output : 1 1 3 1 2 3 4 1 5 2 6 3 7 4 8  
The segment tree is shown below
```

```
Input : -381 -460 -381 95 -460 855 -242  
        405 -460 982 -381 -460 95 981 855  
Output : -460 -460 -381 -460 95 -381 855  
        -460 -242 95 405 -381 981 855 982  
By constructing a segment tree from the output,  
we can see that it a valid tree for RMQ and the  
leaves are all distinct integers.
```

What we first do is iterate through the given integers counting the number of occurrences of each number, then sorting them by value. In C++, we can use the data structure map, which stores elements in sorted order.

Now we maintain a queue for each possible level of the segment tree. We put the initial root of the tree (array index 0) into the queue for the max level. We then insert the smallest element into the leftmost nodes. We then detach these nodes from the main tree. As we detach a node, we create a new tree of height $h - 1$, where h is the height of the current node.

We can see this in figure 2. We insert the root node of this new tree into the appropriate queue based on its height.

We go through each element, getting a tree of appropriate height based on the number of occurrences of that element. If at any point such a tree does not exist, then it is not possible to create a segment tree.

```
// C++ Program to Create RMQ Segment Tree
#include <bits/stdc++.h>
using namespace std;

// Returns true if it is possible to construct
// a range minimum segment tree from given array.
bool createTree(int arr[], int N)
{
    // Store the height of the final tree
    const int height = log2(N) + 1;

    // Container to sort and store occurrences of elements
    map<int, int> multi;

    // Insert elements into the container
    for (int i = 0; i < 2 * N - 1; ++i)
        ++multi[arr[i]];

    // Used to store new subtrees created
    set<int> Q[height];

    // Insert root into set
    Q[height - 1].emplace(0);

    // Iterate through each unique element in set
    for (map<int, int>::iterator it = multi.begin();
         it != multi.end(); ++it)
    {
        // Number of occurrences is greater than height
        // Or, no subtree exists that can accomodate it
        if (it->second > height || Q[it->second - 1].empty())
            return false;

        // Get the appropriate subtree
        int node = *Q[it->second - 1].begin();

        // Delete the subtree we grabbed
        Q[it->second - 1].erase(Q[it->second - 1].begin());

        int level = 1;
        for (int i = node; i < 2 * N - 1;
             i = 2 * i + 1, ++level)
    }
}
```

```

    {
        // Insert new subtree created into root
        if (2 * i + 2 < 2 * N - 1)
            Q[it->second - level - 1].emplace(2 * i + 2);

        // Insert element into array at position
        arr[i] = it->first;
    }
}

return true;
}

// Driver program
int main()
{
    int N = 8;
    int arr[2 * N - 1] = {1, 1, 1, 1, 2, 2,
                          3, 3, 3, 4, 4, 5, 6, 7, 8};
    if (createTree(arr, N))
    {
        cout << "YES\n";
        for (int i = 0; i < 2 * N - 1; ++i)
            cout << arr[i] << " ";
    }
    else
        cout << "NO\n";
    return 0;
}

```

Output:

```

YES
1 1 3 1 2 3 4 1 5 2 6 3 7 4 8

```

Main time complexity is caused by sorting elements.

Time Complexity: $O(N \log N)$

Space Complexity: $O(N)$

Source

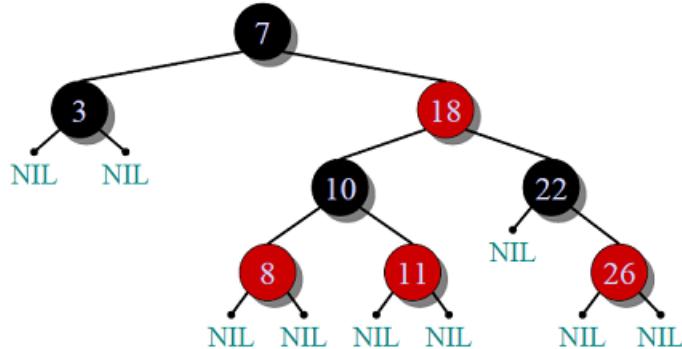
<https://www.geeksforgeeks.org/reconstructing-segment-tree/>

Chapter 148

Red-Black Tree Set 1 (Introduction)

Red-Black Tree Set 1 (Introduction) - GeeksforGeeks

Red-Black Tree is a self-balancing Binary Search Tree (BST) where every node follows following rules.



- 1) Every node has a color either red or black.
- 2) Root of tree is always black.
- 3) There are no two adjacent red nodes (A red node cannot have a red parent or red child).
- 4) Every path from root to a NULL node has same number of black nodes.

Why Red-Black Trees?

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

Comparison with [AVL Tree](#)

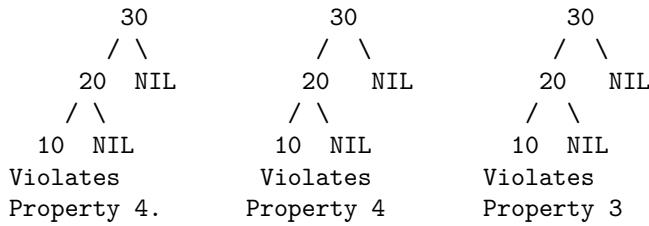
The AVL trees are more balanced compared to Red-Black Trees, but they may cause more rotations during insertion and deletion. So if your application involves many frequent insertions and deletions, then Red Black trees should be preferred. And if the insertions and deletions are less frequent and search is a more frequent operation, then AVL tree should be preferred over Red-Black Tree.

How does a Red-Black Tree ensure balance?

A simple example to understand balancing is, a chain of 3 nodes is not possible in the red-black tree. We can try any combination of colours and see all of them violate Red-Black tree property.

A chain of 3 nodes is nodes is not possible in Red-Black Trees.

Following are NOT Red-Black Trees



Following are different possible Red-Black Trees with above 3 keys



From the above examples, we get some idea how Red-Black trees ensure balance. Following is an important fact about balancing in Red-Black Trees.

Black Height of a Red-Black Tree :

Black height is number of black nodes on a path from a node to a leaf. Leaf nodes are also counted black nodes. From above properties 3 and 4, we can derive, *a node of height h has black-height >= h/2*.

Every Red Black Tree with n nodes has height <= 2Log₂(n+1)

This can be proved using following facts:

- 1) For a general Binary Tree, let k be the minimum number of nodes on all root to NULL paths, then $n \geq 2^k - 1$ (Ex. If k is 3, then n is atleast 7). This expression can also be written as $k \leq 2\log_2(n+1)$
- 2) From property 4 of Red-Black trees and above claim, we can say in a Red-Black Tree with n nodes, there is a root to leaf path with at-most $\log_2(n+1)$ black nodes.
- 3) From property 3 of Red-Black trees, we can claim that the number black nodes in a Red-Black tree is at least $n/2$ where n is the total number of nodes.

From above 2 points, we can conclude the fact that Red Black Tree with n nodes has height $\leq 2\log_2(n+1)$

In this post, we introduced Red-Black trees and discussed how balance is ensured. The hard part is to maintain balance when keys are added and removed. We will soon be discussing insertion and deletion operations in coming posts on the Red-Black tree.

Exercise:

- 1) Is it possible to have all black nodes in a Red-Black tree?
- 2) Draw a Red-Black Tree that is not an [AVL tree](#) structure wise?

Insertion and Deletion

[Red Black Tree Insertion](#)

[Red-Black Tree Deletion](#)

References:

Introduction to Algorithms 3rd Edition by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest

http://en.wikipedia.org/wiki/Red%20black_tree

[Video Lecture on Red-Black Tree by Tim Roughgarden](#)

[MIT Video Lecture on Red-Black Tree](#)

[MIT Lecture Notes on Red Black Tree](#)

Source

<https://www.geeksforgeeks.org/red-black-tree-set-1-introduction-2/>

Chapter 149

Red-Black Tree Set 2 (Insert)

Red-Black Tree Set 2 (Insert) - GeeksforGeeks

In the [previous post](#), we discussed introduction to Red-Black Trees. In this post, insertion is discussed.

In [AVL tree insertion](#), we used rotation as a tool to do balancing after insertion caused imbalance. In Red-Black tree, we use two tools to do balancing.

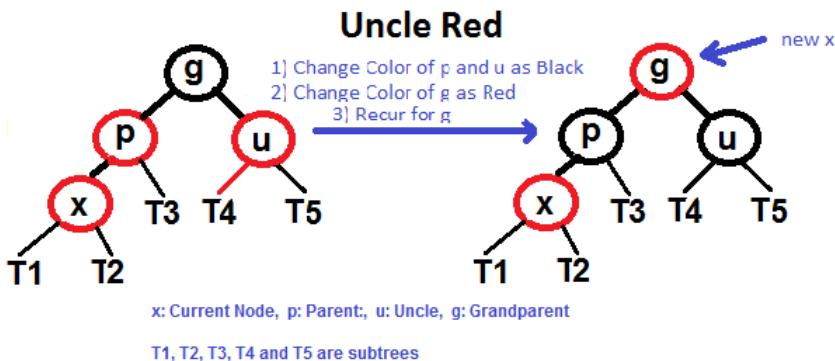
- 1) Recoloring
- 2) [Rotation](#)

We try recoloring first, if recoloring doesn't work, then we go for rotation. Following is detailed algorithm. The algorithms has mainly two cases depending upon the color of uncle. If uncle is red, we do recoloring. If uncle is black, we do rotations and/or recoloring.

Color of a NULL node is considered as BLACK.

Let x be the newly inserted node.

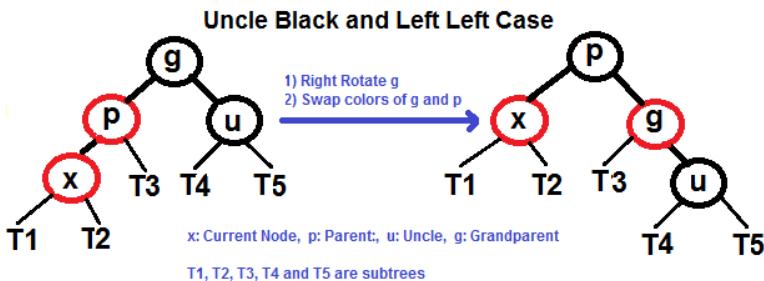
- 1) Perform [standard BST insertion](#) and make the color of newly inserted nodes as RED.
- 2) If x is root, change color of x as BLACK (Black height of complete tree increases by 1).
- 3) Do following if color of x's parent is not BLACK or x is not root.
 -a) **If x's uncle is RED** (Grand parent must have been black from [property 4](#))
 -(i) Change color of parent and uncle as BLACK.
 -(ii) color of grand parent as RED.
 -(iii) Change x = x's grandparent, repeat steps 2 and 3 for new x.



-b) If x's uncle is BLACK, then there can be four configurations for x, x's parent (p) and x's grandparent (g) (This is similar to AVL Tree)
-i) Left Left Case (p is left child of g and x is left child of p)
 -ii) Left Right Case (p is left child of g and x is right child of p)
 -iii) Right Right Case (Mirror of case a)
 -iv) Right Left Case (Mirror of case c)

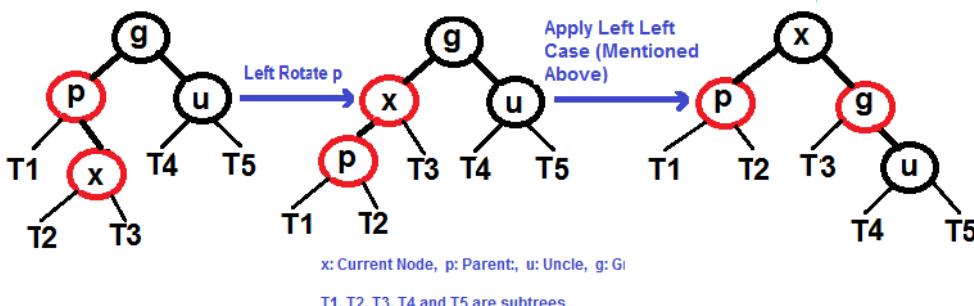
Following are operations to be performed in four subcases when uncle is BLACK.

Left Left Case (See g, p and x)

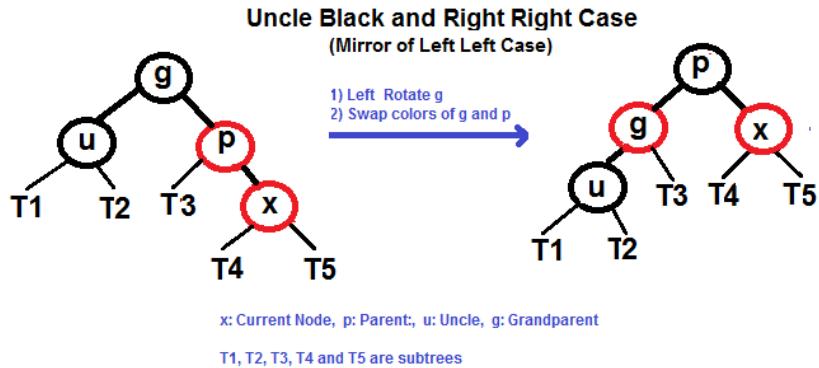


Left Right Case (See g, p and x)

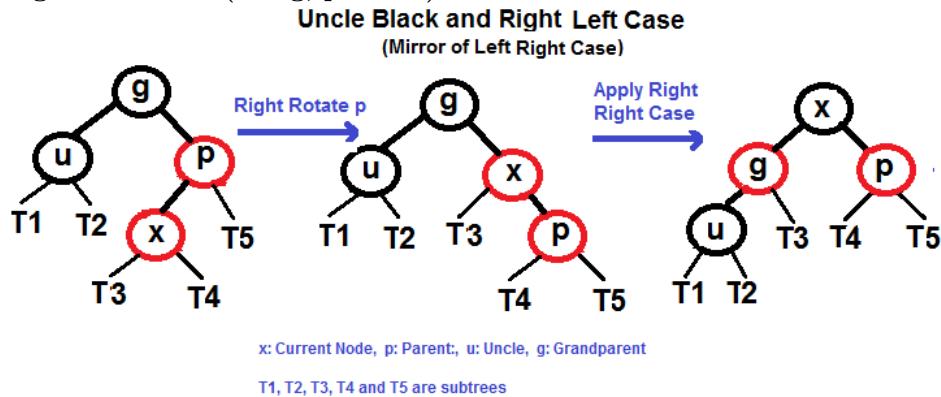
Uncle Black and Left Right Case



Right Right Case (See g, p and x)

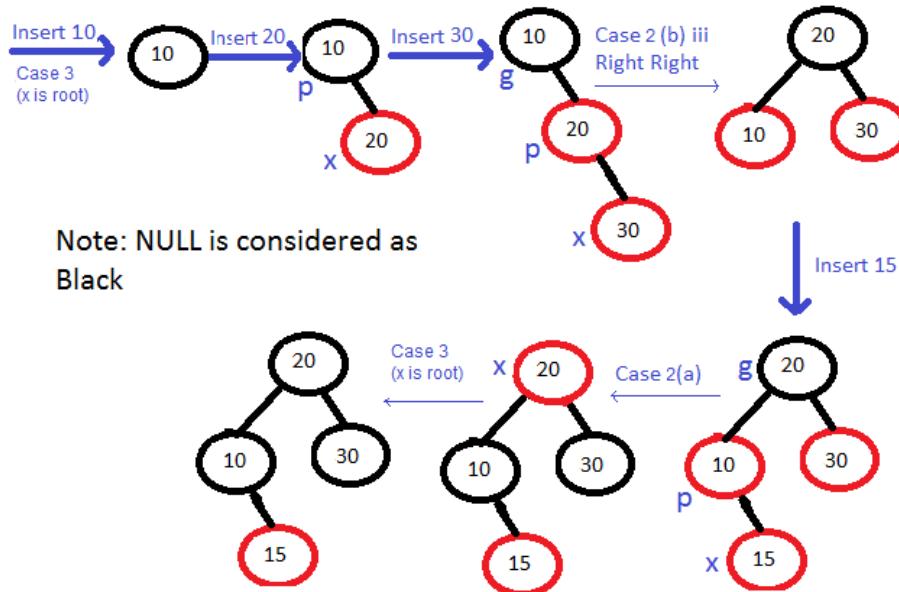


Right Left Case (See g, p and x)



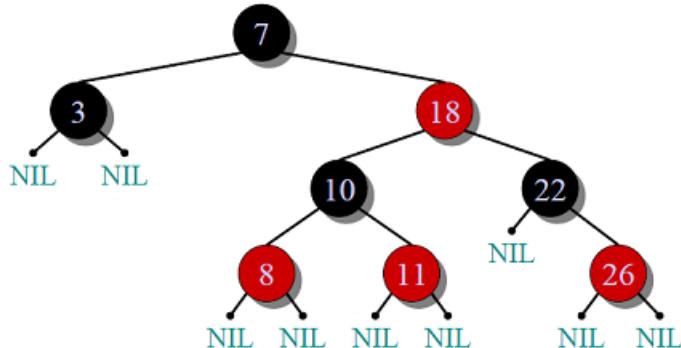
Examples of Insertion

Insert 10, 20, 30 and 15 in an empty tree



Exercise:

Insert 2, 6 and 13 in below tree.



Please refer [C Program for Red Black Tree Insertion](#) for complete implementation of above algorithm.

[Red-Black Tree Set 3 \(Delete\)](#)

Source

<https://www.geeksforgeeks.org/red-black-tree-set-2-insert/>

Chapter 150

Red-Black Tree Set 3 (Delete)

[Red-Black Tree Set 3 \(Delete\) - GeeksforGeeks](#)

We have discussed following topics on Red-Black tree in previous posts. We strongly recommend to refer following post as prerequisite of this post.

[Red-Black Tree Introduction](#)

[Red Black Tree Insert](#)

Insertion Vs Deletion:

Like Insertion, recoloring and rotations are used to maintain the Red-Black properties.

In insert operation, we check color of uncle to decide the appropriate case. In delete operation, *we check color of sibling* to decide the appropriate case.

The main property that violates after insertion is two consecutive reds. In delete, the main violated property is, change of black height in subtrees as deletion of a black node may cause reduced black height in one root to leaf path.

Deletion is fairly complex process. To understand deletion, notion of double black is used. When a black node is deleted and replaced by a black child, the child is marked as **double black**. The main task now becomes to convert this double black to single black.

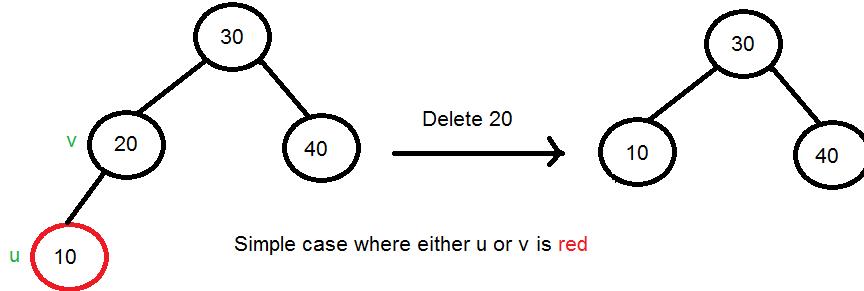
Deletion Steps

Following are detailed steps for deletion.

1) Perform standard BST delete. When we perform standard delete operation in BST, we always end up deleting a node which is either leaf or has only one child (For an internal node, we copy the successor and then recursively call delete for successor, successor is always a leaf node or a node with one child). So we only need to handle cases where a node is leaf or has one child. Let v be the node to be deleted and u be the child that replaces v (Note that u is NULL when v is a leaf and color of NULL is considered as Black).

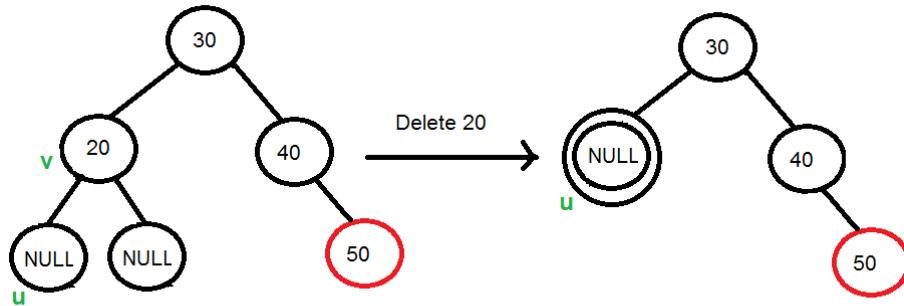
2) Simple Case: If either u or v is red, we mark the replaced child as black (No change in black height). Note that both u and v cannot be red as v is parent of u and two

consecutive reds are not allowed in red-black tree.



3) If Both u and v are Black.

3.1) Color u as double black. Now our task reduces to convert this double black to single black. Note that If v is leaf, then u is NULL and color of NULL is considered as black. So the deletion of a black leaf also causes a double black.



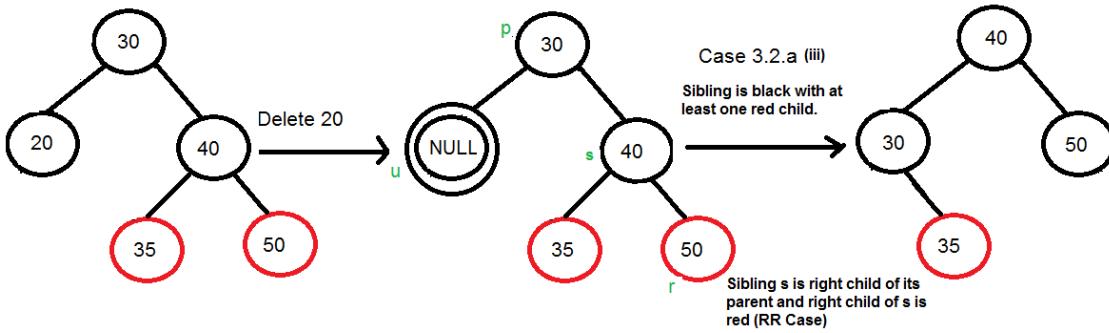
3.2) Do following while the current node u is double black and it is not root. Let sibling of node be s.

....(a): If sibling s is black and at least one of sibling's children is red, perform rotation(s). Let the red child of s be r. This case can be divided in four subcases depending upon positions of s and r.

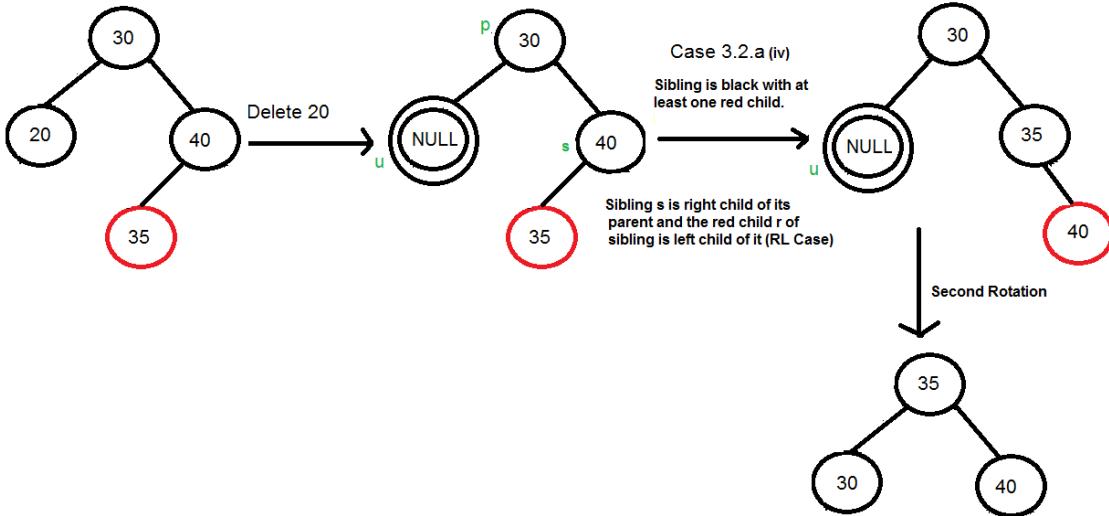
.....(i) Left Left Case (s is left child of its parent and r is left child of s or both children of s are red). This is mirror of right right case shown in below diagram.

.....(ii) Left Right Case (s is left child of its parent and r is right child). This is mirror of right left case shown in below diagram.

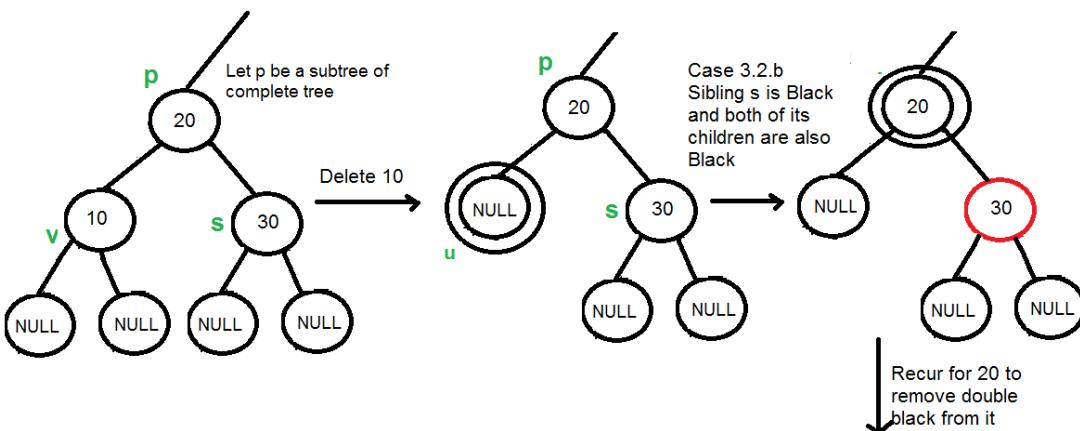
.....(iii) Right Right Case (s is right child of its parent and r is right child of s or both children of s are red)



.....(iv) Right Left Case (s is right child of its parent and r is left child of s)



....(b): If sibling is black and its both children are black, perform recoloring, and recur for the parent if parent is black.

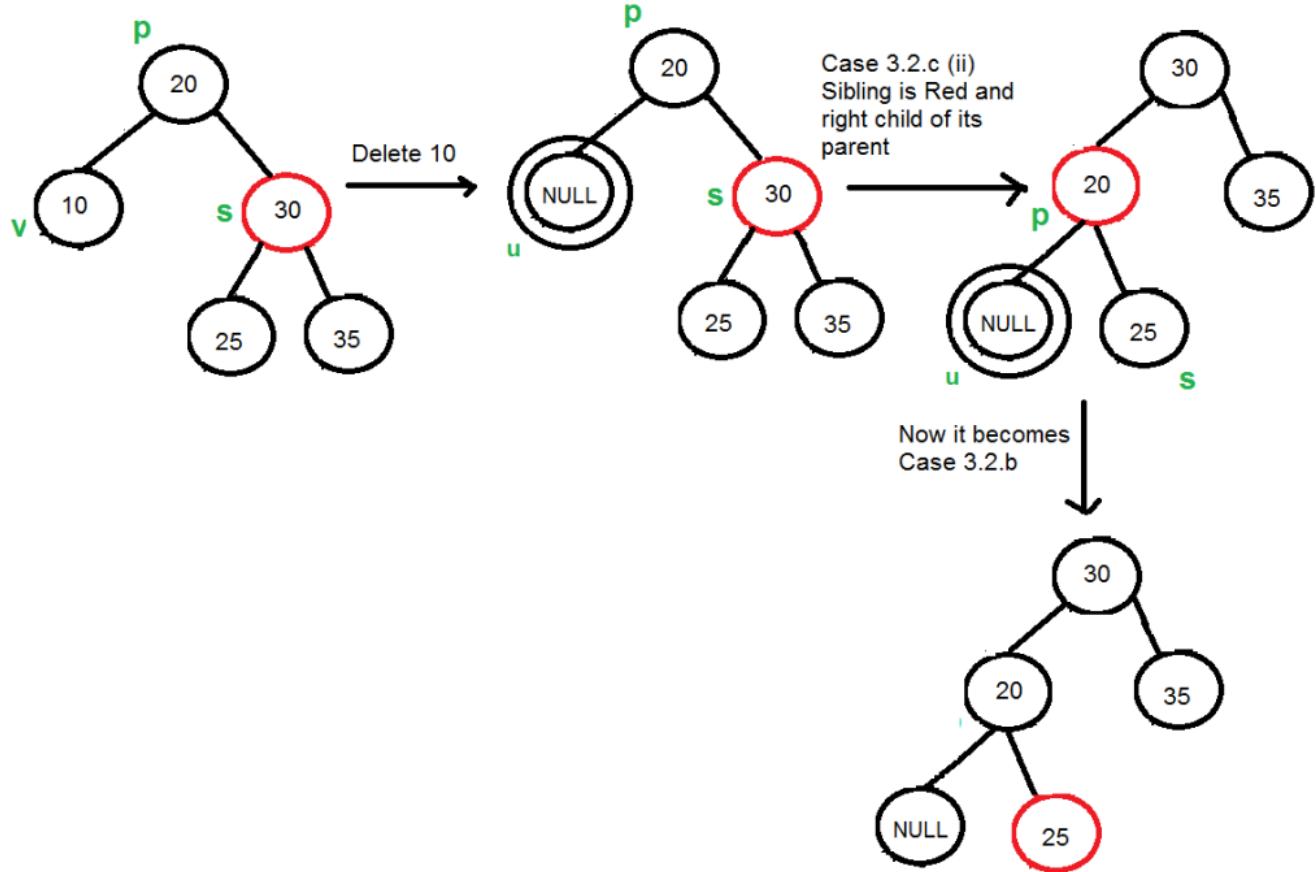


In this case, if parent was red, then we didn't need to recur for parent, we can simply make it black (red + double black = single black)

.....(c): If sibling is red, perform a rotation to move old sibling up, recolor the old sibling and parent. The new sibling is always black (See the below diagram). This mainly converts the tree to black sibling case (by rotation) and leads to case (a) or (b). This case can be divided in two subcases.

.....(i) Left Case (s is left child of its parent). This is mirror of right right case shown in below diagram. We right rotate the parent p.

.....(iii) Right Case (s is right child of its parent). We left rotate the parent p.



3.3) If u is root, make it single black and return (Black height of complete tree reduces by 1).

References:

<https://www.cs.purdue.edu/homes/ayg/CS251/slides/chap13c.pdf>

Introduction to Algorithms 3rd Edition by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest

Improved By : shwetanknaveen

Source

<https://www.geeksforgeeks.org/red-black-tree-set-3-delete-2/>

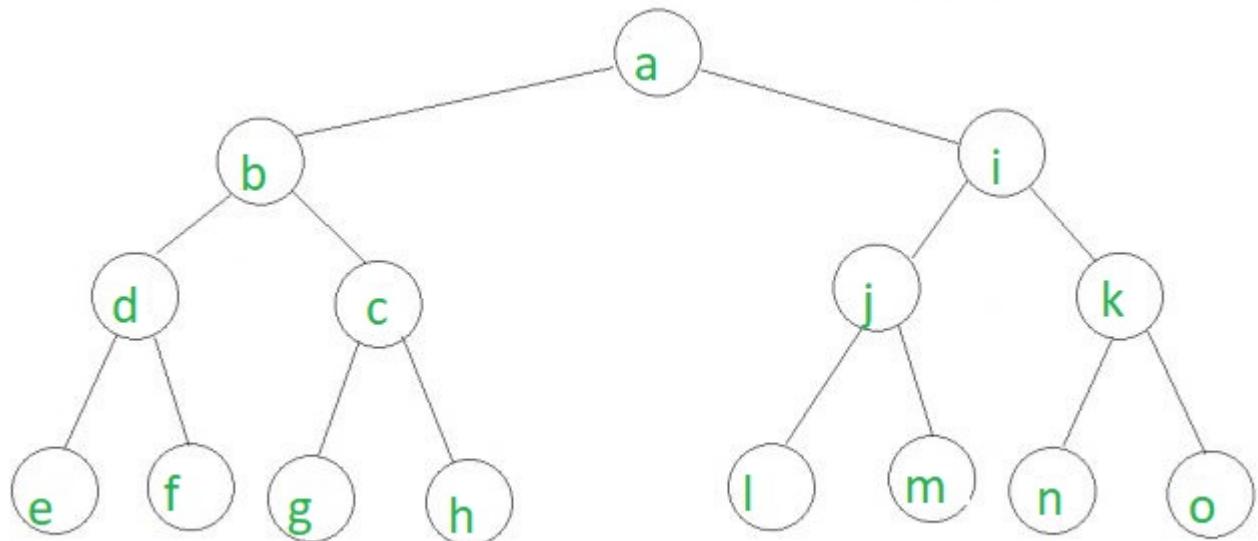
Chapter 151

Remove edges connected to a node such that the three given nodes are in different trees

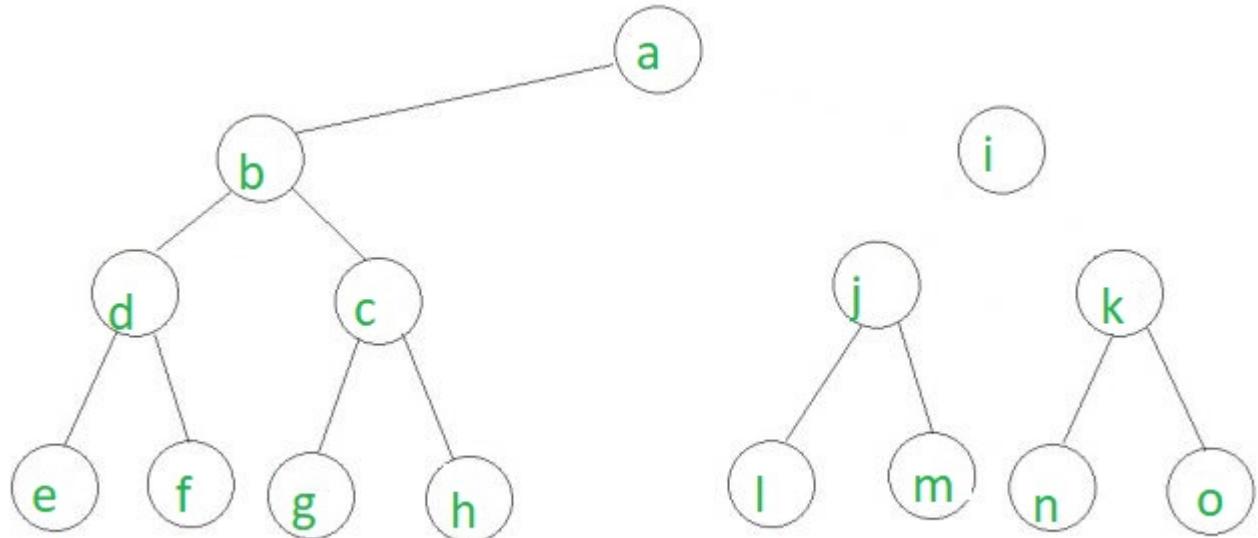
Remove edges connected to a node such that the three given nodes are in different trees - GeeksforGeeks

Given a binary tree and 3 nodes a, b and c, the task is to find a node in the tree such that after removing all the edge connected to that node, a, b and c are in three different trees.

Given below is a tree with input nodes as c, j and o.



In the above tree, if node i gets disconnected from the tree, then the given nodes c, j, and o will be in three different trees which have been shown below.



A simple **approach** is to find **LCA** of all possible pairs of nodes given.
Let,

- lca of (a, b) = x
- lca of (b, c) = y
- lca of (c, a) = z

In any case, either of (x, y), (y, z), (z, x) or (x, y, z) will always be the same. In the first three cases, return the node which is not the same. In the last case returning any node of x, y or z will give the answer.

Below is the implementation of the above approach:

```
// C++ program for disconnecting a
// node to result in three different tree
#include <bits/stdc++.h>
using namespace std;

// node class
struct Node {
    int key;
    struct Node *left, *right;
};
```

```
Node* newNode(int key)
{
    Node* temp = new Node;
    temp->key = key;
    temp->left = temp->right = NULL;
    return (temp);
}

// LCA function taken from the above link mentioned
// This function returns a pointer to LCA of two given
// values n1 and n2. This function assumes that n1 and n2
// are present in Binary Tree
struct Node* findLCA(struct Node* root, int n1, int n2)
{
    // Base case
    if (root == NULL)
        return NULL;

    // If either n1 or n2 matches with root's key, report
    // the presence by returning root (Note that if a key is
    // ancestor of other, then the ancestor key becomes LCA
    if (root->key == n1 || root->key == n2)
        return root;

    // Look for keys in left and right subtrees
    Node* left_lca = findLCA(root->left, n1, n2);
    Node* right_lca = findLCA(root->right, n1, n2);

    // If both of the above calls return Non-NULL, then one key
    // is present in once subtree and other is present in other,
    // So this node is the LCA
    if (left_lca && right_lca)
        return root;

    // Otherwise check if left subtree or right subtree is LCA
    return (left_lca != NULL) ? left_lca : right_lca;
}

// the function assumes a, b, c are present in the tree
// and returns a node disconnecting which
// results in all three nodes in different trees
Node* findNode(Node* root, int a, int b, int c)
{
    // lca of a, b
    Node* x = findLCA(root, a, b);

    // lca of b, c
    Node* y = findLCA(root, b, c);
```

```
// lca of c, a
Node* z = findLCA(root, c, a);

if (x->key == y->key)
    return z;
else if (x->key == z->key)
    return y;
else
    return x;
}

// Driver Code
int main()
{
    // Declare tree
    // Insert elements in the tree
    Node* root = newNode(1);

    root->left = newNode(2);
    root->right = newNode(3);

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

    root->left->left->left = newNode(8);
    root->left->left->right = newNode(9);

    root->left->right->left = newNode(10);
    root->left->right->right = newNode(11);

    root->right->left = newNode(6);
    root->right->right = newNode(7);
    root->right->left->left = newNode(12);
    root->right->left->right = newNode(13);
    root->right->right->left = newNode(14);
    root->right->right->right = newNode(15);

/*
          1
         /   \
        2     3
       / \   / \
      4   5   6   7
     / \ / \ / \
    8  9 10 11 12 13 14 15
*/
```

```
// update all the suitable_children
// keys of all the nodes in O( N )

cout << "Disconnect node "
     << findNode(root, 5, 6, 15)->key
     << " from the tree";

return 0;
}
```

Output:

```
Disconnect node 3 from the tree
```

Source

<https://www.geeksforgeeks.org/remove-edges-connected-to-a-node-such-that-the-three-given-nodes-are-in-different-trees/>

Chapter 152

Ropes Data Structure (Fast String Concatenation)

Ropes Data Structure (Fast String Concatenation) - GeeksforGeeks

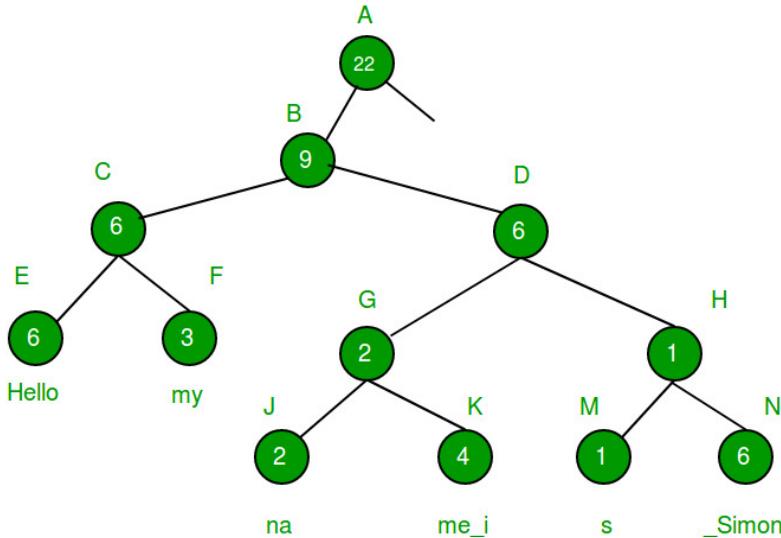
One of the most common operations on strings is appending or concatenation. Appending to the end of a string when the string is stored in the traditional manner (i.e. an array of characters) would take a minimum of $O(n)$ time (where n is the length of the original string).

We can reduce time taken by append using Ropes Data Structure.

Ropes Data Structure

A Rope is a binary tree structure where each node except the leaf nodes, *contains the number of characters present to the left of that node*. Leaf nodes contain the actual string broken into substrings (size of these substrings can be decided by the user).

Consider the image below.



The image shows how the string is stored in memory. Each leaf node contains substrings of the original string and all other nodes contain the number of characters present to the left of that node. The idea behind storing the number of characters to the left is to minimise the cost of finding the character present at i -th position.

Advantages

1. Ropes drastically cut down the cost of appending two strings.
2. Unlike arrays, ropes do not require large contiguous memory allocations.
3. Ropes do not require $O(n)$ additional memory to perform operations like insertion/deletion/searching.
4. In case a user wants to undo the last concatenation made, he can do so in $O(1)$ time by just removing the root node of the tree.

Disadvantages

1. The complexity of source code increases.
2. Greater chances of bugs.
3. Extra memory required to store parent nodes.
4. Time to access i -th character increases.

Now let's look at a situation that explains why Ropes are a good substitute to monolithic string arrays.

Given two strings $a[]$ and $b[]$. Concatenate them in a third string $c[]$.

Examples:

Input : $a[] = \text{"This is "}$, $b[] = \text{"an apple"}$
 Output : $\text{"This is an apple"}$

Input : $a[] = \text{"This is "}$, $b[] = \text{"geeksforgeeks"}$
 Output : $\text{"This is geeksforgeeks"}$

We create a string c[] to store concatenated string. We first traverse a[] and copy all characters of a[] to c[]. Then we copy all characters of b[] to c[].

```
// Simple C++ program to concatenate two strings
#include <iostream>
using namespace std;

// Function that concatenates strings a[0..n1-1]
// and b[0..n2-1] and stores the result in c[]
void concatenate(char a[], char b[], char c[],
                  int n1, int n2)
{
    // Copy characters of A[] to C[]
    int i;
    for (i=0; i<n1; i++)
        c[i] = a[i];

    // Copy characters of B[]
    for (int j=0; j<n2; j++)
        c[i+j] = b[j];

    c[i] = '\0';
}

// Driver code
int main()
{
    char a[] = "Hi This is geeksforgeeks. ";
    int n1 = sizeof(a)/sizeof(a[0]);

    char b[] = "You are welcome here.";
    int n2 = sizeof(b)/sizeof(b[0]);

    // Concatenate a[] and b[] and store result
    // in c[]
    char c[n1 + n2 - 1];
    concatenate(a, b, c, n1, n2);
    for (int i=0; i<n1+n2-1; i++)
        cout << c[i];

    return 0;
}
```

Output:

This is geeksforgeeks

Time complexity : **O(n)**

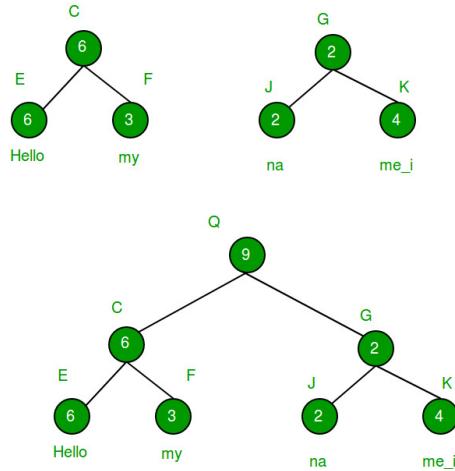
Now let's try to solve the same problem using Ropes.

This rope structure can be utilized to concatenate two strings in constant time.

1. Create a new root node (that stores the root of the new concatenated string)
2. Mark the left child of this node, the root of the string that appears first.
3. Mark the right child of this node, the root of the string that appears second.

And that's it. Since this method only requires to make a new node, it's complexity is **O(1)**.

Consider the image below (Image source : [https://en.wikipedia.org/wiki/Rope_\(data_structure\)](https://en.wikipedia.org/wiki/Rope_(data_structure)))



```
// C++ program to concatenate two strings using
// rope data structure.
#include <bits/stdc++.h>
using namespace std;

// Maximum no. of characters to be put in leaf nodes
const int LEAF_LEN = 2;

// Rope structure
class Rope
{
public:
    Rope *left, *right, *parent;
    char *str;
    int lCount;
};

// Function that creates a Rope structure.
// node --> Reference to pointer of current root node
// l --> Left index of current substring (initially 0)
// r --> Right index of current substring (initially n-1)
// par --> Parent of current node (Initially NULL)
```

```

void createRopeStructure(Rope *&node, Rope *par,
                        char a[], int l, int r)
{
    Rope *tmp = new Rope();
    tmp->left = tmp->right = NULL;

    // We put half nodes in left subtree
    tmp->parent = par;

    // If string length is more
    if ((r-l) > LEAF_LEN)
    {
        tmp->str = NULL;
        tmp->lCount = (r-l)/2;
        node = tmp;
        int m = (l + r)/2;
        createRopeStructure(node->left, node, a, l, m);
        createRopeStructure(node->right, node, a, m+1, r);
    }
    else
    {
        node = tmp;
        tmp->lCount = (r-l);
        int j = 0;
        tmp->str = new char[LEAF_LEN];
        for (int i=l; i<=r; i++)
            tmp->str[j++] = a[i];
    }
}

// Function that prints the string (leaf nodes)
void printstring(Rope *r)
{
    if (r==NULL)
        return;
    if (r->left==NULL && r->right==NULL)
        cout << r->str;
    printstring(r->left);
    printstring(r->right);
}

// Function that efficiently concatenates two strings
// with roots root1 and root2 respectively. n1 is size of
// string represented by root1.
// root3 is going to store root of concatenated Rope.
void concatenate(Rope *&root3, Rope *root1, Rope *root2, int n1)
{
    // Create a new Rope node, and make root1

```

```
// and root2 as children of tmp.  
Rope *tmp = new Rope();  
tmp->parent = NULL;  
tmp->left = root1;  
tmp->right = root2;  
root1->parent = root2->parent = tmp;  
tmp->lCount = n1;  
  
// Make string of tmp empty and update  
// reference r  
tmp->str = NULL;  
root3 = tmp;  
}  
  
// Driver code  
int main()  
{  
    // Create a Rope tree for first string  
    Rope *root1 = NULL;  
    char a[] = "Hi This is geeksforgeeks. ";  
    int n1 = sizeof(a)/sizeof(a[0]);  
    createRopeStructure(root1, NULL, a, 0, n1-1);  
  
    // Create a Rope tree for second string  
    Rope *root2 = NULL;  
    char b[] = "You are welcome here.";  
    int n2 = sizeof(b)/sizeof(b[0]);  
    createRopeStructure(root2, NULL, b, 0, n2-1);  
  
    // Concatenate the two strings in root3.  
    Rope *root3 = NULL;  
    concatenate(root3, root1, root2, n1);  
  
    // Print the new concatenated string  
    printstring(root3);  
    cout << endl;  
    return 0;  
}
```

Output:

Hi This is geeksforgeeks. You are welcome here.

Source

<https://www.geeksforgeeks.org/ropes-data-structure-fast-string-concatenation/>

Chapter 153

ScapeGoat Tree Set 1 (Introduction and Insertion)

ScapeGoat Tree Set 1 (Introduction and Insertion) - GeeksforGeeks

A ScapeGoat tree is a self-balancing Binary Search Tree like [AVL Tree](#), [Red-Black Tree](#), [Splay Tree](#), ..etc.

- Search time is $O(\log n)$ in worst case. Time taken by deletion and insertion is [amortized](#) $O(\log n)$
- The balancing idea is to make sure that nodes are size balanced. A size balanced means sizes of left and right subtrees are at most α (Size of node). The idea is based on the fact that *if a node is A weight balanced, then it is also height balanced: height $\leq \log_{1/\alpha}(size) + 1$*
- Unlike other self-balancing BSTs, ScapeGoat tree doesn't require extra space per node. For example, Red Black Tree nodes are required to have color. In below implementation of ScapeGoat Tree, we only have left, right and parent pointers in Node class. Use of parent is done for simplicity of implementation and can be avoided.

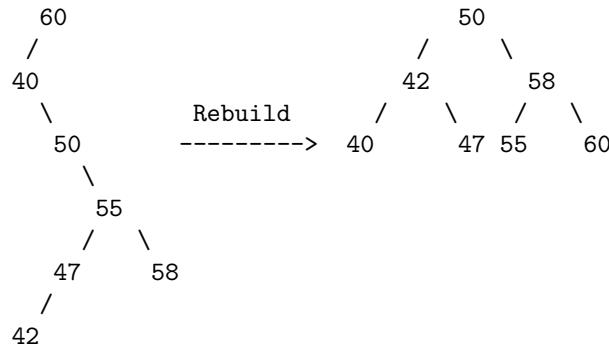
Insertion (Assuming $\alpha = 2/3$):

To insert value x in a Scapegoat Tree:

- Create a new node u and insert x using the [BST insert](#) algorithm.
- If the depth of u is greater than $\log_{3/2}n$ where n is number of nodes in tree then we need to make tree balanced. To make balanced, we use below step to find a scapegoat.
 - Walk up from u until we reach a node w with $\text{size}(w) > (2/3)*\text{size}(w.parent)$.
This node is scapegoat
 - Rebuild the subtree rooted at $w.parent$.

What does rebuilding the subtree mean?

In rebuilding, we simply convert the subtree to the most possible balanced BST. We first store inorder traversal of BST in an array, then we build a new BST from array by recursively dividing it into two halves.



Below is C++ implementation of insert operation on Scapegoat Tree.

```

// C++ program to implement insertion in
// ScapeGoat Tree
#include<bits/stdc++.h>
using namespace std;

// Utility function to get value of log32(n)
static int const log32(int n)
{
    double const log23 = 2.4663034623764317;
    return (int)ceil(log23 * log(n));
}

// A ScapeGoat Tree node
class Node
{
public:
    Node *left, *right, *parent;
    float value;
    Node()
    {
        value = 0;
        left = right = parent = NULL;
    }
    Node (float v)
    {
        value = v;
        left = right = parent = NULL;
    }
};

```

```
        }
    };

// This functions stores inorder traversal
// of tree rooted with ptr in an array arr[]
int storeInArray(Node *ptr, Node *arr[], int i)
{
    if (ptr == NULL)
        return i;

    i = storeInArray(ptr->left, arr, i);
    arr[i++] = ptr;
    return storeInArray(ptr->right, arr, i);
}

// Class to represent a ScapeGoat Tree
class SGTree
{
private:
    Node *root;
    int n; // Number of nodes in Tree
public:
    void preorder(Node *);
    int size(Node *);
    bool insert(float x);
    void rebuildTree(Node *u);
    SGTree() { root = NULL; n = 0; }
    void preorder() { preorder(root); }

    // Function to built tree with balanced nodes
    Node *buildBalancedFromArray(Node **a, int i, int n);

    // Height at which element is to be added
    int BSTInsertAndFindDepth(Node *u);
};

// Preorder traversal of the tree
void SGTree::preorder(Node *node)
{
    if (node != NULL)
    {
        cout << node->value << " ";
        preorder(node -> left);
        preorder(node -> right);
    }
}

// To count number of nodes in the tree
```

```
int SGTree::size(Node *node)
{
    if (node == NULL)
        return 0;
    return 1 + size(node->left) + size(node->right);
}

// To insert new element in the tree
bool SGTree::insert(float x)
{
    // Create a new node
    Node *node = new Node(x);

    // Perform BST insertion and find depth of
    // the inserted node.
    int h = BSTInsertAndFindDepth(node);

    // If tree becomes unbalanced
    if (h > log32(n))
    {
        // Find Scapegoat
        Node *p = node->parent;
        while (3*size(p) <= 2*size(p->parent))
            p = p->parent;

        // Rebuild tree rooted under scapegoat
        rebuildTree(p->parent);
    }

    return h >= 0;
}

// Function to rebuilt tree from new node. This
// function basically uses storeInArray() to
// first store inorder traversal of BST rooted
// with u in an array.
// Then it converts array to the most possible
// balanced BST using buildBalancedFromArray()
void SGTree::rebuildTree(Node *u)
{
    int n = size(u);
    Node *p = u->parent;
    Node **a = new Node* [n];
    storeInArray(u, a, 0);
    if (p == NULL)
    {
        root = buildBalancedFromArray(a, 0, n);
        root->parent = NULL;
    }
}
```

```
        }
        else if (p->right == u)
        {
            p->right = buildBalancedFromArray(a, 0, n);
            p->right->parent = p;
        }
        else
        {
            p->left = buildBalancedFromArray(a, 0, n);
            p->left->parent = p;
        }
    }

// Function to built tree with balanced nodes
Node * SGTree::buildBalancedFromArray(Node **a,
                                      int i, int n)
{
    if (n== 0)
        return NULL;
    int m = n / 2;

    // Now a[m] becomes the root of the new
    // subtree a[0],.....,a[m-1]
    a[i+m]->left = buildBalancedFromArray(a, i, m);

    // elements a[0],...a[m-1] gets stored
    // in the left subtree
    if (a[i+m]->left != NULL)
        a[i+m]->left->parent = a[i+m];

    // elements a[m+1],....a[n-1] gets stored
    // in the right subtree
    a[i+m]->right =
        buildBalancedFromArray(a, i+m+1, n-m-1);
    if (a[i+m]->right != NULL)
        a[i+m]->right->parent = a[i+m];

    return a[i+m];
}

// Performs standard BST insert and returns
// depth of the inserted node.
int SGTree::BSTInsertAndFindDepth(Node *u)
{
    // If tree is empty
    Node *w = root;
    if (w == NULL)
    {
```

```
    root = u;
    n++;
    return 0;
}

// While the node is not inserted
// or a node with same key exists.
bool done = false;
int d = 0;
do
{
    if (u->value < w->value)
    {
        if (w->left == NULL)
        {
            w->left = u;
            u->parent = w;
            done = true;
        }
        else
            w = w->left;
    }
    else if (u->value > w->value)
    {
        if (w->right == NULL)
        {
            w->right = u;
            u->parent = w;
            done = true;
        }
        else
            w = w->right;
    }
    else
        return -1;
    d++;
}
while (!done);

n++;
return d;
}

// Driver code
int main()
{
    SGTree sgt;
    sgt.insert(7);
```

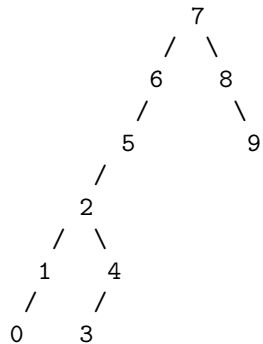
```
sgt.insert(6);
sgt.insert(3);
sgt.insert(1);
sgt.insert(0);
sgt.insert(8);
sgt.insert(9);
sgt.insert(4);
sgt.insert(5);
sgt.insert(2);
sgt.insert(3.5);
printf("Preorder traversal of the"
      " constructed ScapeGoat tree is \n");
sgt.preorder();
return 0;
}
```

Output:

```
Preorder traversal of the constructed ScapeGoat tree is
7 6 3 1 0 2 4 3.5 5 8 9
```

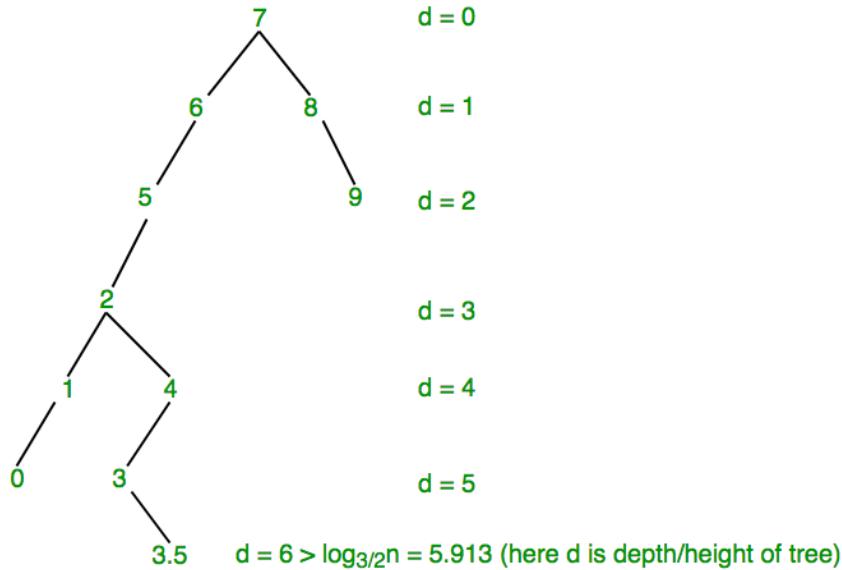
Example to illustrate insertion:

A scapegoat tree with 10 nodes and height 5.



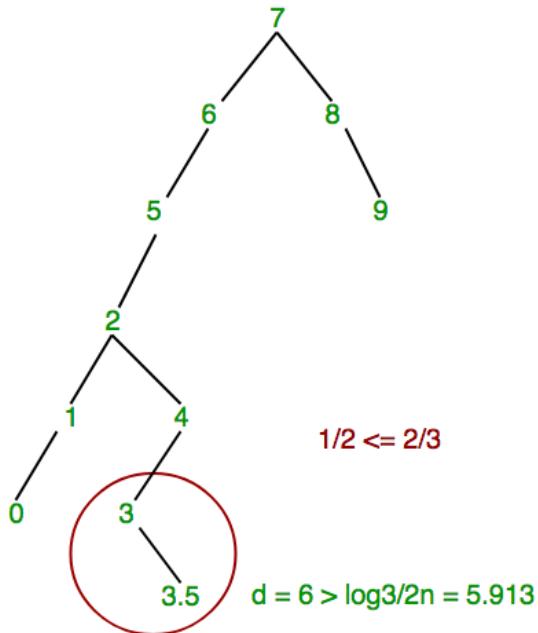
Let's insert 3.5 in the below scapegoat tree.

Initially $d = 5 < \log_{3/2} n$ where $n = 10$;

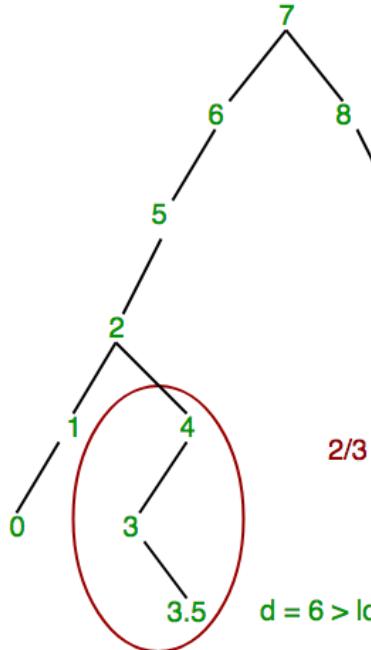


Since, $d > \log_{3/2}n$ i.e., $6 > \log_{3/2}n$, so we have to find the scapegoat in order to solve the problem of exceeding height.

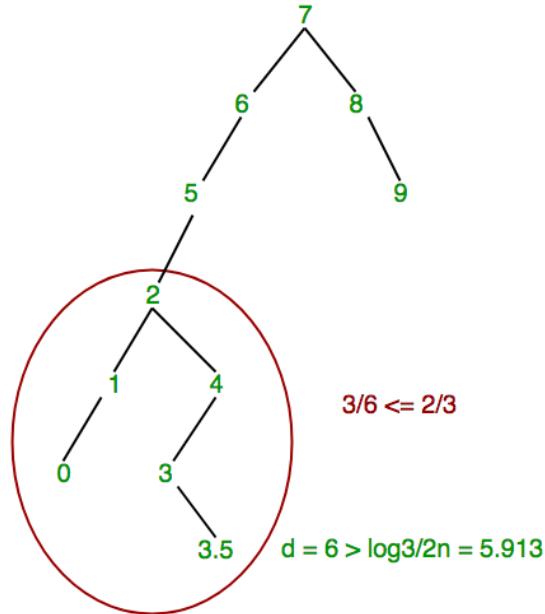
- Now we find a ScapeGoat. We start with newly added node 3.5 and check whether $\text{size}(3.5)/\text{size}(3) > 2/3$.
- Since, $\text{size}(3.5) = 1$ and $\text{size}(3) = 2$, so $\text{size}(3.5)/\text{size}(3) = 1/2$ which is less than $2/3$. So, this is not the scapegoat and we move up .



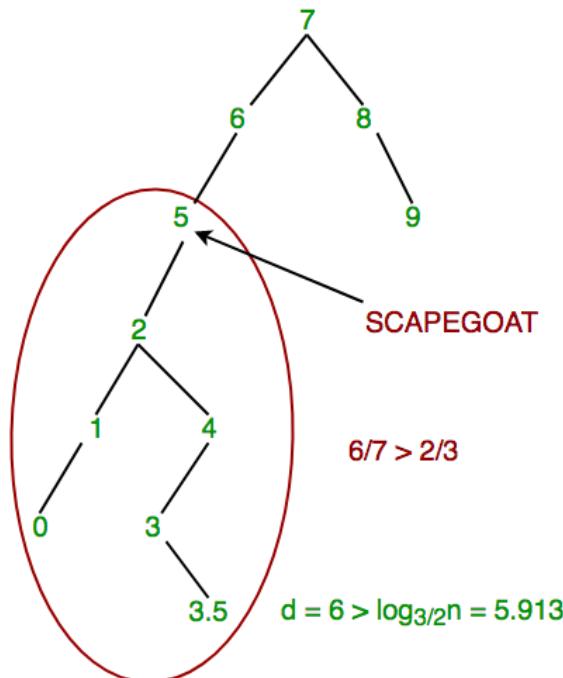
- Since 3 is not the scapegoat, we move and check the same condition for node 4. Since $\text{size}(3) = 2$ and $\text{size}(4) = 3$, so $\text{size}(3)/\text{size}(4) = 2/3$ which is not greater than $2/3$. So, this is not the scapegoat and we move up .



- Since 3 is not the scapegoat, we move and check the same condition for node 4. Since, $\text{size}(3) = 2$ and $\text{size}(4) = 3$, so $\text{size}(3)/\text{size}(4) = 2/3$ which is not greater than $2/3$. So, this is not the scapegoat and we move up .
- Now, $\text{size}(4)/\text{size}(2) = 3/6$. Since, $\text{size}(4)= 3$ and $\text{size}(2) = 6$ but $3/6$ is still less than $2/3$, which does not fulfill the condition of scapegoat so we again move up.

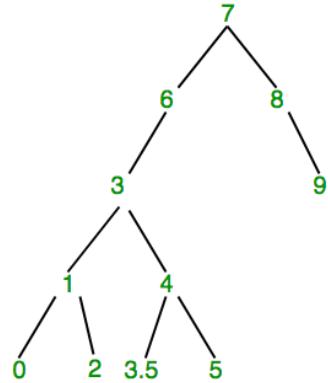


- Now, $\text{size}(2)/\text{size}(5) = 6/7$. Since, $\text{size}(2) = 6$ and $\text{size}(5) = 7$. $6/7 > 2/3$ which fulfills the condition of scapegoat, so we stop here and hence node 5 is a scapegoat



Finally, after finding the scapegoat, rebuilding will be taken at the subtree rooted at scapegoat i.e., at 5.

Final tree:



Comparison with other self-balancing BSTs

Red-Black and AVL : Time complexity of search, insert and delete is $O(\log n)$

Splay Tree : Worst case time complexities of search, insert and delete is $O(n)$. But amortized time complexity of these operations is $O(\log n)$.

ScapeGoat Tree: Like Splay Tree, it is easy to implement and has worst case time complexity of search as $O(\log n)$. Worst case and amortized time complexities of insert and delete are same as Splay Tree for Scapegoat tree.

References:

- https://en.wikipedia.org/wiki/Scapegoat_tree
- http://opendatastructures.org/ods-java/8_Scapegoat_Trees.html

Source

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

Chapter 154

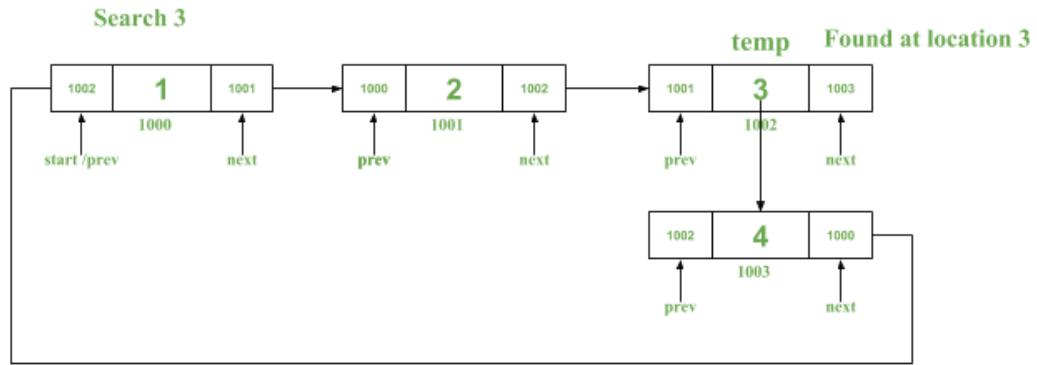
Search an Element in Doubly Circular Linked List

Search an Element in Doubly Circular Linked List - GeeksforGeeks

Pre-requisite: [Convert an Array to a Circular Doubly Linked List](#), [Doubly Circular Linked List](#)

Given a Doubly circular linked list. The task is to find the position of an *element* in the list.

Image Representation:



Algorithm:

- Declare a temp pointer, and initialize it to head of the list.
- Iterate the loop until temp reaches start address (last node in the list, as it is in a circular fashion), check for the *n* element, whether present or not.
- If it is present, raise a flag, increment count and break the loop.
- At the last, as the last node is not visited yet check for the *n* element if present do step 3.

Below program illustrate the above approach:

```
// 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 a node at the end
void insertNode(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 display the
// circular doubly linked list
void displayList(struct Node* start)
{
    struct Node *temp = start;

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

// Function to search the particular element
// from the list
int searchList(struct Node* start, int search)
{
    // Declare the temp variable
    struct Node *temp = start;

    // Declare other control
    // variable for the searching
    int count=0,flag=0,value;

    // If start is NULL return -1
    if(temp == NULL)
        return -1;
    else
    {
        // Move the temp pointer until,
        // temp->next doesn't move
        // start address (Circular Fashion)
        while(temp->next != start)
        {
            // Increment count for location
            count++;
            // If it is found raise the
            // flag and break the loop
            if(temp->data == search)
            {
                flag = 1;
                count--;
                break;
            }
        }
    }
}
```

```
        }
        // Increment temp pointer
        temp = temp->next;
    }
    // Check whether last element in the
    // list content the value if contain,
    // raise a flag and increment count
    if(temp->data == search)
    {
        count++;
        flag = 1;
    }

    // If flag is true, then element
    // found, else not
    if(flag == 1)
        cout<<"\n"<<search <<" found at location "<<
                                count<<endl;
    else
        cout<<"\n"<<search <<" not found"<<endl;
}
}

// Driver code
int main()
{
    /* Start with the empty list */
    struct Node* start = NULL;

    // Insert 4. So linked list becomes 4->NULL
    insertNode(&start, 4);

    // Insert 5. So linked list becomes 4->5
    insertNode(&start, 5);

    // Insert 7. So linked list
    // becomes 4->5->7
    insertNode(&start, 7);

    // Insert 8. So linked list
    // becomes 4->5->7->8
    insertNode(&start, 8);

    // Insert 6. So linked list
    // becomes 4->5->7->8->6
    insertNode(&start, 6);

    printf("Created circular doubly linked list is: ");
```

```
    displayList(start);  
  
    searchList(start, 5);  
  
    return 0;  
}
```

Output:

```
Created circular doubly linked list is: 4 5 7 8 6  
5 found at location 2
```

Time Complexity: As it uses linear search, so complexity is $O(n)$.

Source

<https://www.geeksforgeeks.org/search-an-element-in-doubly-circular-linked-list/>

Chapter 155

Second minimum element using minimum comparisons

Second minimum element using minimum comparisons - GeeksforGeeks

Given an array of integers, find the minimum (or maximum) element and the element just greater (or smaller) than that in less than $2n$ comparisons. The given array is not necessarily sorted. Extra space is allowed.

Examples:

Input: {3, 6, 100, 9, 10, 12, 7, -1, 10}

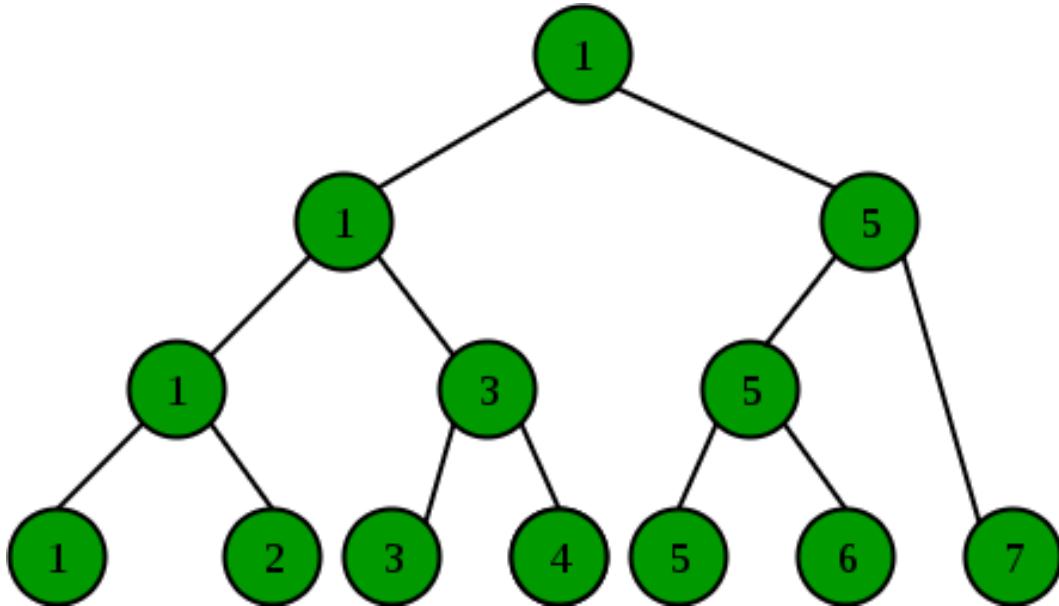
Output: Minimum: -1, Second minimum: 3

We have already discussed an approach in below post.

[Find the smallest and second smallest element in an array](#)

Comparisons of array elements can be costly if array elements are of large size, for example large strings. We can minimize the number of comparisons used in above approach.

The idea is based on [tournament tree](#). Consider each element in the given array as a leaf node.



First, we find the minimum element as explained by building a tournament tee. To build the tree, we compare all adjacent pairs of elements (leaf nodes) with each other. Now we have $n/2$ elements which are smaller than their counterparts (from each pair, the smaller element forms the level above that of the leaves). Again, find the smaller elements in each of the $n/4$ pairs. Continue this process until the root of the tree is created. The root is the minimum.

Next, we traverse the tree and while doing so, we discard the sub trees whose root is greater than the smallest element. Before discarding, we update the second smallest element, which will hold our result. The point to understand here is that the tree is height balanced and we have to spend only $\log n$ time to traverse all the elements that were ever compared to the minimum in step 1. Thus, the overall time complexity is $n + \log n$. Auxiliary space needed is $O(2n)$, as the number of leaf nodes will be approximately equal to the number of internal nodes.

```

// C++ program to find minimum and second minimum
// using minimum number of comparisons
#include <bits/stdc++.h>
using namespace std;

// Tournament Tree node
struct Node
{
    int idx;
    Node *left, *right;
};

// Utility function to create a tournament tree node
Node *createNode(int idx)
  
```

```
{  
    Node *t = new Node;  
    t->left = t->right = NULL;  
    t->idx = idx;  
    return t;  
}  
  
// This function traverses tree across height to  
// find second smallest element in tournament tree.  
// Note that root is smallest element of tournament  
// tree.  
void traverseHeight(Node *root, int arr[], int &res)  
{  
    // Base case  
    if (root == NULL || (root->left == NULL &&  
                           root->right == NULL))  
        return;  
  
    // If left child is smaller than current result,  
    // update result and recur for left subarray.  
    if (res > arr[root->left->idx] &&  
        root->left->idx != root->idx)  
    {  
        res = arr[root->left->idx];  
        traverseHeight(root->right, arr, res);  
    }  
  
    // If right child is smaller than current result,  
    // update result and recur for left subarray.  
    else if (res > arr[root->right->idx] &&  
             root->right->idx != root->idx)  
    {  
        res = arr[root->right->idx];  
        traverseHeight(root->left, arr, res);  
    }  
}  
  
// Prints minimum and second minimum in arr[0..n-1]  
void findSecondMin(int arr[], int n)  
{  
    // Create a list to store nodes of current  
    // level  
    list<Node *> li;  
  
    Node *root = NULL;  
    for (int i = 0; i < n; i += 2)  
    {  
        Node *t1 = createNode(i);  
    }
```

```
Node *t2 = NULL;
if (i + 1 < n)
{
    // Make a node for next element
    t2 = createNode(i + 1);

    // Make smaller of two as root
    root = (arr[i] < arr[i + 1])? createNode(i) :
                                                createNode(i + 1);

    // Make two nodes as children of smaller
    root->left = t1;
    root->right = t2;

    // Add root
    li.push_back(root);
}
else
    li.push_back(t1);
}

int lsize = li.size();

// Construct the complete tournament tree from above
// prepared list of winners in first round.
while (lsize != 1)
{
    // Find index of last pair
    int last = (lsize & 1)? (lsize - 2) : (lsize - 1);

    // Process current list items in pair
    for (int i = 0; i < last; i += 2)
    {
        // Extract two nodes from list, make a new
        // node for winner of two
        Node *f1 = li.front();
        li.pop_front();

        Node *f2 = li.front();
        li.pop_front();
        root = (arr[f1->idx] < arr[f2->idx])?
            createNode(f1->idx) : createNode(f2->idx);

        // Make winner as parent of two
        root->left = f1;
        root->right = f2;

        // Add winner to list of next level
    }
}
```

```
        li.push_back(root);
    }
    if (lsize & 1)
    {
        li.push_back(li.front());
        li.pop_front();
    }
    lsize = li.size();
}

// Traverse tree from root to find second minimum
// Note that minimum is already known and root of
// tournament tree.
int res = INT_MAX;
traverseHeight(root, arr, res);
cout << "Minimum: " << arr[root->idx]
     << ", Second minimum: " << res << endl;
}

// Driver code
int main()
{
    int arr[] = {61, 6, 100, 9, 10, 12, 17};
    int n = sizeof(arr)/sizeof(arr[0]);
    findSecondMin(arr, n);
    return 0;
}
```

Output:

```
Minimum: 6, Second minimum: 9
```

We can optimize above code by avoid creation of leaf nodes, as that information is stored in the array itself (and we know that the elements were compared in pairs).

Source

<https://www.geeksforgeeks.org/second-minimum-element-using-minimum-comparisons/>

Chapter 156

Segment Tree (XOR of a given range)

Segment Tree (XOR of a given range) - GeeksforGeeks

Let us consider the following problem to understand Segment Trees.

We have an array $\text{arr}[0 \dots n-1]$. We should be able to

1 Find the xor of elements from index l to r where $0 \leq l \leq r \leq n-1$.

2 Change value of a specified element of the array to a new value x. We need to do $\text{arr}[i] = x$ where $0 \leq i \leq n-1$.

Similar to [Sum of Given Range](#).

A simple solution is to run a loop from l to r and calculate xor of elements in given range. To update a value, simply do $\text{arr}[i] = x$. The first operation takes $O(n)$ time and second operation takes $O(1)$ time.

Efficient Approach

If the number of query and updates are equal, we can perform both the operations in $O(\log n)$ time. We can use a Segment Tree to do both operations in $O(\log n)$ time.

Representation of Segment trees

1. Leaf Nodes are the elements of the input array.
2. Each internal node represents some merging of the leaf nodes. The merging may be different for different problems. For this problem, merging is Xor of leaves under a node.

An array representation of tree is used to represent Segment Trees. For each node at index i, the left child is at index $2*i+1$, right child at $2*i+2$ and the parent is at $(i-1)/2$.

Query for Product of given range

Once the tree is constructed, how to get the Xor using the constructed segment tree. Following is algorithm to get the xor of elements.

```
int getXor(node, l, r)
{
    if range of node is within l and r
        return value in node
    else if range of node is completely outside l and r
        return 1
    else
        return getXor(node's left child, l, r) ^
               getXor(node's right child, l, r)
}
```

C++

```
// C++ program to show segment tree operations
// like construction, query and update
#include <bits/stdc++.h>
#include <math.h>
using namespace std;

// A utility function to get the middle
// index from corner indexes.
int getMid(int s, int e) {
    return s + (e - s)/2;
}

/* A recursive function to get the Xor of
values in given range of the array. The
following are parameters for this function.
st --> Pointer to segment tree
si --> Index of current node in the segment tree.
Initially 0 is passed as root is always
at index 0.
ss & se --> Starting and ending indexes of
the segment represented by current
node, i.e., st[si]
qs & qe --> Starting and ending indexes of
query range */
int getXorUtil(int *st, int ss, int se, int qs,
              int qe, int si)
{
    // If segment of this node is a part of given
    // range, then return the Xor of the segment
    if (qs <= ss && qe >= se)
        return st[si];

    // If segment of this node is outside
    // the given range
    if (se < qs || ss > qe)
```

```
        return 0;

        // If a part of this segment overlaps
        // with the given range
        int mid = getMid(ss, se);
        return getXorUtil(st, ss, mid, qs, qe, 2*si+1) ^
               getXorUtil(st, mid+1, se, qs, qe, 2*si+2);
    }

/* A recursive function to update the nodes
which have the given index in their range.
The following are parameters
st, si, ss and se are same as getXorUtil()
i --> index of the element to be updated.
This index is      in input array.*/
void updateValueUtil(int *st, int ss, int se, int i,
                     int prev_val, int new_val, int si)
{
    // Base Case: If the input index lies outside
    // the range of this segment
    if (i < ss || i > se)
        return;

    // If the input index is in range of this node,
    // then update the value of the node and its children
    st[si] = (st[si]^prev_val)^new_val;
    if (se != ss)
    {
        int mid = getMid(ss, se);
        updateValueUtil(st, ss, mid, i, prev_val,
                        new_val, 2*si + 1);
        updateValueUtil(st, mid+1, se, i, prev_val,
                        new_val, 2*si + 2);
    }
}

// The function to update a value in input
// array and segment tree. It uses updateValueUtil()
// to update the value in segment tree
void updateValue(int arr[], int *st, int n,
                 int i, int new_val)
{
    // Check for erroneous input index
    if (i < 0 || i > n-1)
    {
        printf("Invalid Input");
        return;
    }
}
```

```
int temp = arr[i];
// Update the value in array
arr[i] = new_val;

// Update the values of nodes in segment tree
updateValueUtil(st, 0, n-1, i, temp, new_val, 0);
}

// Return Xor of elements in range from index qs (quey start)
// to qe (query end). It mainly uses getXorUtil()
int getXor(int *st, int n, int qs, int qe)
{
    // Check for erroneous input values
    if (qs < 0 || qe > n-1 || qs > qe)
    {
        printf("Invalid Input");
        return 0;
    }

    return getXorUtil(st, 0, n-1, qs, qe, 0);
}

// A recursive function that constructs
// Segment Tree for array[ss..se]. si is
// index of current node in segment tree st
int constructSTUtil(int arr[], int ss, int se,
                    int *st, int si)
{
    // If there is one element in array,
    // store it in current node of segment
    // tree and return
    if (ss == se)
    {
        st[si] = arr[ss];
        return arr[ss];
    }

    // If there are more than one elements,
    // then recur for left and right subtrees
    // and store the Xor of values in this node
    int mid = getMid(ss, se);
    st[si] = constructSTUtil(arr, ss, mid, st, si*2+1) ^
             constructSTUtil(arr, mid+1, se, st, si*2+2);
    return st[si];
}

/* Function to construct segment tree from given array.
This function allocates memory for segment tree and
```

```
calls constructSTUtil() to fill the allocated memory */
int *constructST(int arr[], int n)
{
    // Allocate memory for segment tree

    // Height of segment tree
    int x = (int)(ceil(log2(n)));

    // Maximum size of segment tree
    int max_size = 2*(int)pow(2, x) - 1;

    // Allocate memory
    int *st = new int[max_size];

    // Fill the allocated memory st
    constructSTUtil(arr, 0, n-1, st, 0);

    // Return the constructed segment tree
    return st;
}

// Driver program to test above functions
int main()
{
    int arr[] = {1, 3, 5, 7, 9, 11};
    int n = sizeof(arr)/sizeof(arr[0]);

    // Build segment tree from given array
    int *st = constructST(arr, n);

    // Print Xor of values in array from index 1 to 3
    printf("Xor of values in given range = %d\n",
          getXor(st, n, 0, 2));

    // Update: set arr[1] = 10 and update corresponding
    // segment tree nodes
    updateValue(arr, st, n, 1, 10);

    // Find Xor after the value is updated
    printf("Updated Xor of values in given range = %d\n",
          getXor(st, n, 0, 2));
    return 0;
}
```

Source

<https://www.geeksforgeeks.org/segment-tree-xor-given-range/>

Chapter 157

Segment Tree Set 1 (Sum of given range)

Segment Tree Set 1 (Sum of given range) - GeeksforGeeks

Let us consider the following problem to understand Segment Trees.

We have an array $\text{arr}[0 \dots n-1]$. We should be able to

1 Find the sum of elements from index l to r where $0 \leq l \leq r \leq n-1$

2 Change value of a specified element of the array to a new value x. We need to do $\text{arr}[i] = x$ where $0 \leq i \leq n-1$.

A **simple solution** is to run a loop from l to r and calculate sum of elements in given range. To update a value, simply do $\text{arr}[i] = x$. The first operation takes $O(n)$ time and second operation takes $O(1)$ time.

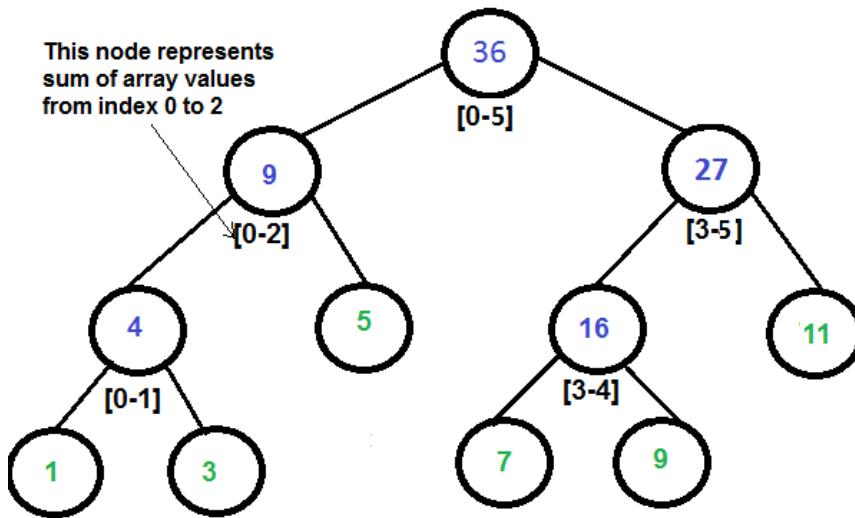
Another solution is to create another array and store sum from start to i at the ith index in this array. Sum of a given range can now be calculated in $O(1)$ time, but update operation takes $O(n)$ time now. This works well if the number of query operations are large and very few updates.

What if the number of query and updates are equal? **Can we perform both the operations in $O(\log n)$ time once given the array?** We can use a Segment Tree to do both operations in $O(\log n)$ time.

Representation of Segment trees

1. Leaf Nodes are the elements of the input array.
2. Each internal node represents some merging of the leaf nodes. The merging may be different for different problems. For this problem, merging is sum of leaves under a node.

An array representation of tree is used to represent Segment Trees. For each node at index i, the left child is at index $2*i+1$, right child at $2*i+2$ and the parent is at $\lfloor (i-1)/2 \rfloor$.



Segment Tree for input array {1, 3, 5, 7, 9, 11}

Construction of Segment Tree from given array

We start with a segment $\text{arr}[0 \dots n-1]$. and every time we divide the current segment into two halves(if it has not yet become a segment of length 1), and then call the same procedure on both halves, and for each such segment, we store the sum in the corresponding node. All levels of the constructed segment tree will be completely filled except the last level. Also, the tree will be a [Full Binary Tree](#) because we always divide segments in two halves at every level. Since the constructed tree is always a full binary tree with n leaves, there will be $n-1$ internal nodes. So total number of nodes will be $2*n - 1$.

Height of the segment tree will be $\lceil \log_2 n \rceil$. Since the tree is represented using array and relation between parent and child indexes must be maintained, size of memory allocated for segment tree will be $2 * 2^{\lceil \log_2 n \rceil} - 1$.

Query for Sum of given range

Once the tree is constructed, how to get the sum using the constructed segment tree. Following is the algorithm to get the sum of elements.

```

int getSum(node, l, r)
{
    if the range of the node is within l and r
        return value in the node
    else if the range of the node is completely outside l and r
        return 0
    else
        return getSum(node's left child, l, r) +
               getSum(node's right child, l, r)
}
  
```

Update a value

Like tree construction and query operations, the update can also be done recursively. We are given an index which needs to be updated. Let *diff* be the value to be added. We start from root of the segment tree and add *diff* to all nodes which have given index in their range. If a node doesn't have given index in its range, we don't make any changes to that node.

Implementation:

Following is the implementation of segment tree. The program implements construction of segment tree for any given array. It also implements query and update operations.

C

```
// C program to show segment tree operations like construction, query
// and update
#include <stdio.h>
#include <math.h>

// A utility function to get the middle index from corner indexes.
int getMid(int s, int e) { return s + (e - s)/2; }

/* A recursive function to get the sum of values in given range
of the array. The following are parameters for this function.

st    --> Pointer to segment tree
si    --> Index of current node in the segment tree. Initially
        0 is passed as root is always at index 0
ss & se  --> Starting and ending indexes of the segment represented
            by current node, i.e., st[si]
qs & qe  --> Starting and ending indexes of query range */
int getSumUtil(int *st, int ss, int se, int qs, int qe, int si)
{
    // If segment of this node is a part of given range, then return
    // the sum of the segment
    if (qs <= ss && qe >= se)
        return st[si];

    // If segment of this node is outside the given range
    if (se < qs || ss > qe)
        return 0;

    // If a part of this segment overlaps with the given range
    int mid = getMid(ss, se);
    return getSumUtil(st, ss, mid, qs, qe, 2*si+1) +
           getSumUtil(st, mid+1, se, qs, qe, 2*si+2);
}

/* A recursive function to update the nodes which have the given
index in their range. The following are parameters
```

```

st, si, ss and se are same as getSumUtil()
i    --> index of the element to be updated. This index is
        in the input array.
diff --> Value to be added to all nodes which have i in range */
void updateValueUtil(int *st, int ss, int se, int i, int diff, int si)
{
    // Base Case: If the input index lies outside the range of
    // this segment
    if (i < ss || i > se)
        return;

    // If the input index is in range of this node, then update
    // the value of the node and its children
    st[si] = st[si] + diff;
    if (se != ss)
    {
        int mid = getMid(ss, se);
        updateValueUtil(st, ss, mid, i, diff, 2*si + 1);
        updateValueUtil(st, mid+1, se, i, diff, 2*si + 2);
    }
}

// The function to update a value in input array and segment tree.
// It uses updateValueUtil() to update the value in segment tree
void updateValue(int arr[], int *st, int n, int i, int new_val)
{
    // Check for erroneous input index
    if (i < 0 || i > n-1)
    {
        printf("Invalid Input");
        return;
    }

    // Get the difference between new value and old value
    int diff = new_val - arr[i];

    // Update the value in array
    arr[i] = new_val;

    // Update the values of nodes in segment tree
    updateValueUtil(st, 0, n-1, i, diff, 0);
}

// Return sum of elements in range from index qs (query start)
// to qe (query end). It mainly uses getSumUtil()
int getSum(int *st, int n, int qs, int qe)
{
    // Check for erroneous input values

```

```
if (qs < 0 || qe > n-1 || qs > qe)
{
    printf("Invalid Input");
    return -1;
}

return getSumUtil(st, 0, n-1, qs, qe, 0);
}

// A recursive function that constructs Segment Tree for array[ss..se].
// si is index of current node in segment tree st
int constructSTUtil(int arr[], int ss, int se, int *st, int si)
{
    // If there is one element in array, store it in current node of
    // segment tree and return
    if (ss == se)
    {
        st[si] = arr[ss];
        return arr[ss];
    }

    // If there are more than one elements, then recur for left and
    // right subtrees and store the sum of values in this node
    int mid = getMid(ss, se);
    st[si] = constructSTUtil(arr, ss, mid, st, si*2+1) +
             constructSTUtil(arr, mid+1, se, st, si*2+2);
    return st[si];
}

/* Function to construct segment tree from given array. This function
   allocates memory for segment tree and calls constructSTUtil() to
   fill the allocated memory */
int *constructST(int arr[], int n)
{
    // Allocate memory for the segment tree

    //Height of segment tree
    int x = (int)(ceil(log2(n)));

    //Maximum size of segment tree
    int max_size = 2*(int)pow(2, x) - 1;

    // Allocate memory
    int *st = new int[max_size];

    // Fill the allocated memory st
    constructSTUtil(arr, 0, n-1, st, 0);
```

```
// Return the constructed segment tree
return st;
}

// Driver program to test above functions
int main()
{
    int arr[] = {1, 3, 5, 7, 9, 11};
    int n = sizeof(arr)/sizeof(arr[0]);

    // Build segment tree from given array
    int *st = constructST(arr, n);

    // Print sum of values in array from index 1 to 3
    printf("Sum of values in given range = %dn",
           getSum(st, n, 1, 3));

    // Update: set arr[1] = 10 and update corresponding
    // segment tree nodes
    updateValue(arr, st, n, 1, 10);

    // Find sum after the value is updated
    printf("Updated sum of values in given range = %dn",
           getSum(st, n, 1, 3));
    return 0;
}
```

Java

```
// Java Program to show segment tree operations like construction,
// query and update
class SegmentTree
{
    int st[]; // The array that stores segment tree nodes

    /* Constructor to construct segment tree from given array. This
       constructor allocates memory for segment tree and calls
       constructSTUtil() to fill the allocated memory */
    SegmentTree(int arr[], int n)
    {
        // Allocate memory for segment tree
        //Height of segment tree
        int x = (int) (Math.ceil(Math.log(n) / Math.log(2)));

        //Maximum size of segment tree
        int max_size = 2 * (int) Math.pow(2, x) - 1;

        st = new int[max_size]; // Memory allocation
    }
}
```

```

        constructSTUtil(arr, 0, n - 1, 0);
    }

    // A utility function to get the middle index from corner indexes.
    int getMid(int s, int e) {
        return s + (e - s) / 2;
    }

    /* A recursive function to get the sum of values in given range
       of the array. The following are parameters for this function.

       st      --> Pointer to segment tree
       si      --> Index of current node in the segment tree. Initially
                   0 is passed as root is always at index 0
       ss & se  --> Starting and ending indexes of the segment represented
                     by current node, i.e., st[si]
       qs & qe  --> Starting and ending indexes of query range */
    int getSumUtil(int ss, int se, int qs, int qe, int si)
    {
        // If segment of this node is a part of given range, then return
        // the sum of the segment
        if (qs <= ss && qe >= se)
            return st[si];

        // If segment of this node is outside the given range
        if (se < qs || ss > qe)
            return 0;

        // If a part of this segment overlaps with the given range
        int mid = getMid(ss, se);
        return getSumUtil(ss, mid, qs, qe, 2 * si + 1) +
               getSumUtil(mid + 1, se, qs, qe, 2 * si + 2);
    }

    /* A recursive function to update the nodes which have the given
       index in their range. The following are parameters
       st, si, ss and se are same as getSumUtil()
       i      --> index of the element to be updated. This index is in
                   input array.
       diff --> Value to be added to all nodes which have i in range */
    void updateValueUtil(int ss, int se, int i, int diff, int si)
    {
        // Base Case: If the input index lies outside the range of
        // this segment
        if (i < ss || i > se)
            return;

```

```
// If the input index is in range of this node, then update the
// value of the node and its children
st[si] = st[si] + diff;
if (se != ss) {
    int mid = getMid(ss, se);
    updateValueUtil(ss, mid, i, diff, 2 * si + 1);
    updateValueUtil(mid + 1, se, i, diff, 2 * si + 2);
}
}

// The function to update a value in input array and segment tree.
// It uses updateValueUtil() to update the value in segment tree
void updateValue(int arr[], int n, int i, int new_val)
{
    // Check for erroneous input index
    if (i < 0 || i > n - 1) {
        System.out.println("Invalid Input");
        return;
    }

    // Get the difference between new value and old value
    int diff = new_val - arr[i];

    // Update the value in array
    arr[i] = new_val;

    // Update the values of nodes in segment tree
    updateValueUtil(0, n - 1, i, diff, 0);
}

// Return sum of elements in range from index qs (quey start) to
// qe (query end). It mainly uses getSumUtil()
int getSum(int n, int qs, int qe)
{
    // Check for erroneous input values
    if (qs < 0 || qe > n - 1 || qs > qe) {
        System.out.println("Invalid Input");
        return -1;
    }
    return getSumUtil(0, n - 1, qs, qe, 0);
}

// A recursive function that constructs Segment Tree for array[ss..se].
// si is index of current node in segment tree st
int constructSTUtil(int arr[], int ss, int se, int si)
{
    // If there is one element in array, store it in current node of
    // segment tree and return
```

```
if (ss == se) {
    st[si] = arr[ss];
    return arr[ss];
}

// If there are more than one elements, then recur for left and
// right subtrees and store the sum of values in this node
int mid = getMid(ss, se);
st[si] = constructSTUtil(arr, ss, mid, si * 2 + 1) +
         constructSTUtil(arr, mid + 1, se, si * 2 + 2);
return st[si];
}

// Driver program to test above functions
public static void main(String args[])
{
    int arr[] = {1, 3, 5, 7, 9, 11};
    int n = arr.length;
    SegmentTree tree = new SegmentTree(arr, n);

    // Build segment tree from given array

    // Print sum of values in array from index 1 to 3
    System.out.println("Sum of values in given range = " +
                       tree.getSum(n, 1, 3));

    // Update: set arr[1] = 10 and update corresponding segment
    // tree nodes
    tree.updateValue(arr, n, 1, 10);

    // Find sum after the value is updated
    System.out.println("Updated sum of values in given range = " +
                       tree.getSum(n, 1, 3));
}
}

//This code is contributed by Ankur Narain Verma
```

Output:

```
Sum of values in given range = 15
Updated sum of values in given range = 22
```

Time Complexity:

Time Complexity for tree construction is $O(n)$. There are total $2n-1$ nodes, and value of every node is calculated only once in tree construction.

Time complexity to query is $O(\log n)$. To query a sum, we process at most four nodes at every level and number of levels is $O(\log n)$.

The time complexity of update is also $O(\log n)$. To update a leaf value, we process one node at every level and number of levels is $O(\log n)$.

Segment Tree Set 2 (Range Minimum Query)

References:

IIT Kanpur paper.

Source

<https://www.geeksforgeeks.org/segment-tree-set-1-sum-of-given-range/>

Chapter 158

Segment Tree Set 2 (Range Maximum Query with Node Update)

Segment Tree Set 2 (Range Maximum Query with Node Update) - GeeksforGeeks

Given an array $\text{arr}[0 \dots n-1]$. Find the maximum of elements from index l to r where $0 \leq l \leq r \leq n-1$. Also, change the value of a specified element of the array to a new value x. We need to do $\text{arr}[i] = x$ where $0 \leq i \leq n-1$ and then find the maximum element of given range with updated values.

Example :

```
Input : {1, 3, 5, 7, 9, 11}
Maximum Query : L = 1, R = 3
update : set arr[1] = 8
Output :
Max of values in given range = 7
Updated max of values in given range = 8
```

A **simple solution** is to run a loop from l to r and calculate the maximum of elements in given range. To update a value, simply do $\text{arr}[i] = x$. The first operation takes $O(n)$ time and the second operation takes $O(1)$ time.

Efficient Approach : Here, we need to perform operations in $O(\log n)$ time so we can use Segment Tree to do both operations in $O(\log n)$ time.

Representation of Segment trees

1. Leaf Nodes are the elements of the input array.
2. Each internal node represents the maximum of all of its child.

An array representation of tree is used to represent Segment Trees. For each node at index i , the **left child** is at index $2*i+1$, **right child** at index $2*i+2$ and the **parent** is at index $(i-1)/2$.

Construction of Segment Tree from given array :

We start with a segment $\text{arr}[0 \dots n-1]$, and every time we divide the current segment into two halves(if it has not yet become a segment of length 1), and then call the same procedure on both halves, and for each such segment, we store the maximum value in a segment tree node. All levels of the constructed segment tree will be completely filled except the last level. Also, the tree will be a full Binary Tree because we always divide segments into two halves at every level. Since the constructed tree is always a full binary tree with n leaves, there will be $n-1$ internal nodes. So **total nodes** will be $2*n - 1$. Height of the segment tree will be $\log_2 n$. Since the tree is represented using array and relation between parent and child indexes must be maintained, size of memory allocated for segment tree will be $2*(2^{\lceil \log_2 n \rceil}) - 1$.

Query for minimum value of given range : Once the tree is constructed, below is the algorithm to find maximum of given range.

```

node--> node number, l -->
query start index, r --> query end index;

int getMax(node, l, r)
{
    if range of node is within l and r
        return value of node
    else if range of node is completely outside l and r
        return -1
    else
        return max(getMax(node's left child, l, r),
                  getMax(node's right child, l, r))
}

```

Below is the implementation of above approach :

```

// CPP code for range maximum query and updates
#include <bits/stdc++.h>
using namespace std;

// A utility function to get the
// middle index of given range.
int getMid(int s, int e)
{
    return s + (e - s) / 2;
}

/* A recursive function to get the sum of

```

values in given range of the array.
The following are parameters for this function.

```

st      -> Pointer to segment tree
node    -> Index of current node in
           the segment tree .
ss & se -> Starting and ending indexes
           of the segment represented
           by current node, i.e., st[node]
l & r   -> Starting and ending indexes
           of range query */
int MaxUtil(int* st, int ss, int se, int l,
            int r, int node)
{
    // If segment of this node is completely
    // part of given range, then return
    // the max of segment
    if (l <= ss && r >= se)
        return st[node];

    // If segment of this node does not
    // belong to given range
    if (se < l || ss > r)
        return -1;

    // If segment of this node is partially
    // the part of given range
    int mid = getMid(ss, se);

    return max(MaxUtil(st, ss, mid, l, r,
                       2 * node + 1),
               MaxUtil(st, mid + 1, se, l,
                       r, 2 * node + 2));
}

/* A recursive function to update the nodes which
   have the given index in their range. The following
   are parameters st, ss and se are same as defined
   above index -> index of the element to be updated.*/
void updateValue(int arr[], int* st, int ss, int se,
                 int index, int value, int node)
{
    if (index < ss || index > se)
    {
        cout << "Invalid Input" << endl;
        return;
    }
}

```

```

if (ss == se)
{
    // update value in array and in segment tree
    arr[index] = value;
    st[node] = value;
}
else {
    int mid = getMid(ss, se);

    if (index >= ss && index <= mid)
        updateValue(arr, st, ss, mid, index,
                    value, 2 * node + 1);
    else
        updateValue(arr, st, mid + 1, se,
                    index, value, 2 * node + 2);

    st[node] = max(st[2 * node + 1],
                  st[2 * node + 2]);
}
return;
}

// Return max of elements in range from
// index l (query start) to r (query end).
int getMax(int* st, int n, int l, int r)
{
    // Check for erroneous input values
    if (l < 0 || r > n - 1 || l > r)
    {
        printf("Invalid Input");
        return -1;
    }

    return MaxUtil(st, 0, n - 1, l, r, 0);
}

// A recursive function that constructs Segment
// Tree for array[ss..se]. si is index of
// current node in segment tree st
int constructSTUtil(int arr[], int ss, int se,
                    int* st, int si)
{
    // If there is one element in array, store
    // it in current node of segment tree and return
    if (ss == se)
    {
        st[si] = arr[ss];
    }
}

```

```
        return arr[ss];
    }

    // If there are more than one elements, then
    // recur for left and right subtrees and
    // store the max of values in this node
    int mid = getMid(ss, se);

    st[si] = max(constructSTUtil(arr, ss, mid, st,
                                si * 2 + 1),
                 constructSTUtil(arr, mid + 1, se,
                                st, si * 2 + 2));

    return st[si];
}

/* Function to construct segment tree from given array.
   This function allocates memory for segment tree.*/
int* constructST(int arr[], int n)
{
    // Height of segment tree
    int x = (int)(ceil(log2(n)));

    // Maximum size of segment tree
    int max_size = 2 * (int)pow(2, x) - 1;

    // Allocate memory
    int* st = new int[max_size];

    // Fill the allocated memory st
    constructSTUtil(arr, 0, n - 1, st, 0);

    // Return the constructed segment tree
    return st;
}

// Driver code
int main()
{
    int arr[] = { 1, 3, 5, 7, 9, 11 };
    int n = sizeof(arr) / sizeof(arr[0]);

    // Build segment tree from given array
    int* st = constructST(arr, n);

    // Print max of values in array
    // from index 1 to 3
    cout << "Max of values in given range = "
```

```
<< getMax(st, n, 1, 3) << endl;

// Update: set arr[1] = 8 and update
// corresponding segment tree nodes.
updateValue(arr, st, 0, n - 1, 1, 8, 0);

// Find max after the value is updated
cout << "Updated max of values in given range = "
    << getMax(st, n, 1, 3) << endl;

return 0;
}
```

Output:

```
Max of values in given range = 7
Updated max of values in given range = 8
```

Source

<https://www.geeksforgeeks.org/segment-tree-set-2-range-maximum-query-node-update/>

Chapter 159

Segment Tree Set 2 (Range Minimum Query)

Segment Tree Set 2 (Range Minimum Query) - GeeksforGeeks

We have introduced [segment tree with a simple example](#) in the previous post. In this post, [Range Minimum Query](#) problem is discussed as another example where Segment Tree can be used. Following is problem statement.

We have an array $\text{arr}[0 \dots n-1]$. We should be able to efficiently find the minimum value from index qs (query start) to qe (query end) where $0 \leq qs \leq qe \leq n-1$.

A **simple solution** is to run a loop from qs to qe and find minimum element in given range. This solution takes $O(n)$ time in worst case.

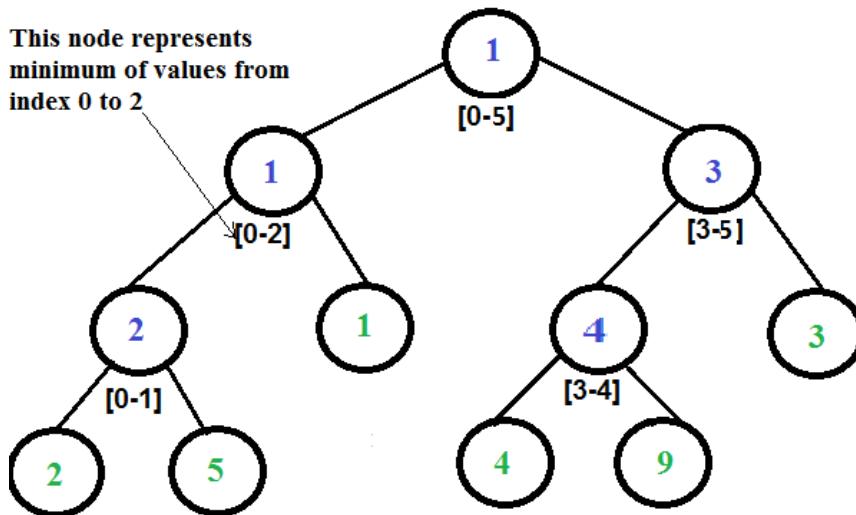
A **Another solution** is to create a 2D array where an entry $[\text{i}, \text{j}]$ stores the minimum value in range $\text{arr}[\text{i..j}]$. Minimum of a given range can now be calculated in $O(1)$ time, but preprocessing takes $O(n^2)$ time. Also, this approach needs $O(n^2)$ extra space which may become huge for large input arrays.

Segment tree can be used to do preprocessing and query in moderate time. With segment tree, preprocessing time is $O(n)$ and time to for range minimum query is $O(\log n)$. The extra space required is $O(n)$ to store the segment tree.

Representation of Segment trees

1. Leaf Nodes are the elements of the input array.
2. Each internal node represents minimum of all leaves under it.

An array representation of tree is used to represent Segment Trees. For each node at index i , the left child is at index $2*i+1$, right child at $2*i+2$ and the parent is at $\lfloor (i-1)/2 \rfloor$



Segment Tree for input array {2, 5, 1, 4, 9, 3 }

Construction of Segment Tree from given array

We start with a segment $\text{arr}[0 \dots n-1]$. and every time we divide the current segment into two halves(if it has not yet become a segment of length 1), and then call the same procedure on both halves, and for each such segment, we store the minimum value in a segment tree node.

All levels of the constructed segment tree will be completely filled except the last level. Also, the tree will be a [Full Binary Tree](#) because we always divide segments in two halves at every level. Since the constructed tree is always full binary tree with n leaves, there will be $n-1$ internal nodes. So total number of nodes will be $2*n - 1$.

Height of the segment tree will be $\lceil \log_2 n \rceil$. Since the tree is represented using array and relation between parent and child indexes must be maintained, size of memory allocated for segment tree will be $2 * 2^{\lceil \log_2 n \rceil} - 1$.

Query for minimum value of given range

Once the tree is constructed, how to do range minimum query using the constructed segment tree. Following is algorithm to get the minimum.

```
// qs --> query start index, qe --> query end index
int RMQ(node, qs, qe)
{
    if range of node is within qs and qe
        return value in node
    else if range of node is completely outside qs and qe
        return INFINITE
    else
        return min( RMQ(node's left child, qs, qe), RMQ(node's right child, qs, qe) )
```

}

Implementation:

C

```
// C program for range minimum query using segment tree
#include <stdio.h>
#include <math.h>
#include <limits.h>

// A utility function to get minimum of two numbers
int minVal(int x, int y) { return (x < y)? x: y; }

// A utility function to get the middle index from corner indexes.
int getMid(int s, int e) { return s + (e - s)/2; }

/* A recursive function to get the minimum value in a given range
   of array indexes. The following are parameters for this function.

   st    --> Pointer to segment tree
   index --> Index of current node in the segment tree. Initially
              0 is passed as root is always at index 0
   ss & se --> Starting and ending indexes of the segment represented
              by current node, i.e., st[index]
   qs & qe --> Starting and ending indexes of query range */
int RMQUtil(int *st, int ss, int se, int qs, int qe, int index)
{
    // If segment of this node is a part of given range, then return
    // the min of the segment
    if (qs <= ss && qe >= se)
        return st[index];

    // If segment of this node is outside the given range
    if (se < qs || ss > qe)
        return INT_MAX;

    // If a part of this segment overlaps with the given range
    int mid = getMid(ss, se);
    return minVal(RMQUtil(st, ss, mid, qs, qe, 2*index+1),
                  RMQUtil(st, mid+1, se, qs, qe, 2*index+2));
}

// Return minimum of elements in range from index qs (query start) to
// qe (query end). It mainly uses RMQUtil()
int RMQ(int *st, int n, int qs, int qe)
{
    // Check for erroneous input values
```

```

if (qs < 0 || qe > n-1 || qs > qe)
{
    printf("Invalid Input");
    return -1;
}

return RMQUtil(st, 0, n-1, qs, qe, 0);
}

// A recursive function that constructs Segment Tree for array[ss..se].
// si is index of current node in segment tree st
int constructSTUtil(int arr[], int ss, int se, int *st, int si)
{
    // If there is one element in array, store it in current node of
    // segment tree and return
    if (ss == se)
    {
        st[si] = arr[ss];
        return arr[ss];
    }

    // If there are more than one elements, then recur for left and
    // right subtrees and store the minimum of two values in this node
    int mid = getMid(ss, se);
    st[si] = minVal(constructSTUtil(arr, ss, mid, st, si*2+1),
                    constructSTUtil(arr, mid+1, se, st, si*2+2));
    return st[si];
}

/* Function to construct segment tree from given array. This function
   allocates memory for segment tree and calls constructSTUtil() to
   fill the allocated memory */
int *constructST(int arr[], int n)
{
    // Allocate memory for segment tree

    //Height of segment tree
    int x = (int)(ceil(log2(n)));

    // Maximum size of segment tree
    int max_size = 2*(int)pow(2, x) - 1;

    int *st = new int[max_size];

    // Fill the allocated memory st
    constructSTUtil(arr, 0, n-1, st, 0);

    // Return the constructed segment tree
}

```

```
    return st;
}

// Driver program to test above functions
int main()
{
    int arr[] = {1, 3, 2, 7, 9, 11};
    int n = sizeof(arr)/sizeof(arr[0]);

    // Build segment tree from given array
    int *st = constructST(arr, n);

    int qs = 1; // Starting index of query range
    int qe = 5; // Ending index of query range

    // Print minimum value in arr[qs..qe]
    printf("Minimum of values in range [%d, %d] is = %d\n",
           qs, qe, RMQ(st, n, qs, qe));

    return 0;
}
```

Java

```
// Program for range minimum query using segment tree
class SegmentTreeRMQ
{
    int st[]; //array to store segment tree

    // A utility function to get minimum of two numbers
    int minVal(int x, int y) {
        return (x < y) ? x : y;
    }

    // A utility function to get the middle index from corner
    // indexes.
    int getMid(int s, int e) {
        return s + (e - s) / 2;
    }

    /* A recursive function to get the minimum value in a given
     * range of array indexes. The following are parameters for
     * this function.

    st    --> Pointer to segment tree
    index --> Index of current node in the segment tree. Initially
              0 is passed as root is always at index 0
    ss & se --> Starting and ending indexes of the segment
```

```

        represented by current node, i.e., st[index]
        qs & qe --> Starting and ending indexes of query range */
int RMQUtil(int ss, int se, int qs, int qe, int index)
{
    // If segment of this node is a part of given range, then
    // return the min of the segment
    if (qs <= ss && qe >= se)
        return st[index];

    // If segment of this node is outside the given range
    if (se < qs || ss > qe)
        return Integer.MAX_VALUE;

    // If a part of this segment overlaps with the given range
    int mid = getMid(ss, se);
    return minVal(RMQUtil(ss, mid, qs, qe, 2 * index + 1),
                  RMQUtil(mid + 1, se, qs, qe, 2 * index + 2));
}

// Return minimum of elements in range from index qs (quey
// start) to qe (query end). It mainly uses RMQUtil()
int RMQ(int n, int qs, int qe)
{
    // Check for erroneous input values
    if (qs < 0 || qe > n - 1 || qs > qe) {
        System.out.println("Invalid Input");
        return -1;
    }

    return RMQUtil(0, n - 1, qs, qe, 0);
}

// A recursive function that constructs Segment Tree for
// array[ss..se]. si is index of current node in segment tree st
int constructSTUtil(int arr[], int ss, int se, int si)
{
    // If there is one element in array, store it in current
    // node of segment tree and return
    if (ss == se) {
        st[si] = arr[ss];
        return arr[ss];
    }

    // If there are more than one elements, then recur for left and
    // right subtrees and store the minimum of two values in this node
    int mid = getMid(ss, se);
    st[si] = minVal(constructSTUtil(arr, ss, mid, si * 2 + 1),
                    constructSTUtil(arr, mid + 1, se, si * 2 + 2));
}

```

```
        return st[si];
    }

/* Function to construct segment tree from given array. This function
allocates memory for segment tree and calls constructSTUtil() to
fill the allocated memory */
void constructST(int arr[], int n)
{
    // Allocate memory for segment tree

    //Height of segment tree
    int x = (int) (Math.ceil(Math.log(n) / Math.log(2)));

    //Maximum size of segment tree
    int max_size = 2 * (int) Math.pow(2, x) - 1;
    st = new int[max_size]; // allocate memory

    // Fill the allocated memory st
    constructSTUtil(arr, 0, n - 1, 0);
}

// Driver program to test above functions
public static void main(String args[])
{
    int arr[] = {1, 3, 2, 7, 9, 11};
    int n = arr.length;
    SegmentTreeRMQ tree = new SegmentTreeRMQ();

    // Build segment tree from given array
    tree.constructST(arr, n);

    int qs = 1; // Starting index of query range
    int qe = 5; // Ending index of query range

    // Print minimum value in arr[qs..qe]
    System.out.println("Minimum of values in range [" + qs + ", "
                      + qe + "] is = " + tree.RMQ(n, qs, qe));
}
}

// This code is contributed by Ankur Narain Verma
```

Output:

```
Minimum of values in range [1, 5] is = 2
```

Time Complexity:

Time Complexity for tree construction is $O(n)$. There are total $2n-1$ nodes, and value of every node is calculated only once in tree construction.

Time complexity to query is $O(\log n)$. To query a range minimum, we process at most two nodes at every level and number of levels is $O(\log n)$.

Please refer following links for more solutions to range minimum query problem.

<https://www.geeksforgeeks.org/range-minimum-query-for-static-array/>

[http://community.topcoder.com/tc?module=Static&d1=tutorials&d2=lowestCommonAncestor#Range_Minimum_Query_\(RMQ\)](http://community.topcoder.com/tc?module=Static&d1=tutorials&d2=lowestCommonAncestor#Range_Minimum_Query_(RMQ))

http://wcipeg.com/wiki/Range_minimum_query

Source

<https://www.geeksforgeeks.org/segment-tree-set-1-range-minimum-query/>

Chapter 160

Segment Tree Set 3 (XOR of given range)

Segment Tree Set 3 (XOR of given range) - GeeksforGeeks

We have an array $\text{arr}[0 \dots n-1]$. There are two type of queries

1. Find the XOR of elements from index l to r where $0 \leq l \leq r \leq n-1$
2. Change value of a specified element of the array to a new value x. We need to do $\text{arr}[i] = x$ where $0 \leq i \leq n-1$.

There will be total of q queries.

Input Constraint

$n \leq 10^5$, $q \leq 10^5$

Solution 1

A simple solution is to run a loop from l to r and calculate xor of elements in given range. To update a value, simply do $\text{arr}[i] = x$. The first operation takes $O(n)$ time and second operation takes $O(1)$ time. Worst case time complexity is $O(n*q)$ for q queries which will take huge time for $n \sim 10^5$ and $q \sim 10^5$. Hence this solution will exceed time limit.

Solution 2

Another solution is to store xor in all possible ranges but there are $O(n^2)$ possible ranges hence with $n \sim 10^5$ it wil exceed space complexity, hence without considering time complexity, we can state this solution will not work.

Solution 3 (Segment Tree)

Prerequisite : [Segment Tree](#)

We build a segment tree of given array such that array elements are at leaves and internal nodes store XOR of leaves covered under them.

```

// C program to show segment tree operations like construction,
// query and update
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

// A utility function to get the middle index from corner indexes.
int getMid(int s, int e) { return s + (e - s)/2; }

/* A recursive function to get the xor of values in given range
   of the array. The following are parameters for this function.

   st    --> Pointer to segment tree
   si    --> Index of current node in the segment tree. Initially
             0 is passed as root is always at index 0
   ss & se  --> Starting and ending indexes of the segment
                 represented by current node, i.e., st[si]
   qs & qe  --> Starting and ending indexes of query range */
int getXorUtil(int *st, int ss, int se, int qs, int qe, int si)
{
    // If segment of this node is a part of given range, then return
    // the xor of the segment
    if (qs <= ss && qe >= se)
        return st[si];

    // If segment of this node is outside the given range
    if (se < qs || ss > qe)
        return 0;

    // If a part of this segment overlaps with the given range
    int mid = getMid(ss, se);
    return getXorUtil(st, ss, mid, qs, qe, 2*si+1) ^
           getXorUtil(st, mid+1, se, qs, qe, 2*si+2);
}

/* A recursive function to update the nodes which have the given
   index in their range. The following are parameters
   st, si, ss and se are same as getXorUtil()
   i    --> index of the element to be updated. This index is
           in input array.
   diff --> Value to be added to all nodes which have i in range */
void updateValueUtil(int *st, int ss, int se, int i, int diff, int si)
{
    // Base Case: If the input index lies outside the range of
    // this segment
    if (i < ss || i > se)
        return;

```

```

// If the input index is in range of this node, then update
// the value of the node and its children
st[si] = st[si] + diff;
if (se != ss)
{
    int mid = getMid(ss, se);
    updateValueUtil(st, ss, mid, i, diff, 2*si + 1);
    updateValueUtil(st, mid+1, se, i, diff, 2*si + 2);
}
}

// The function to update a value in input array and segment tree.
// It uses updateValueUtil() to update the value in segment tree
void updateValue(int arr[], int *st, int n, int i, int new_val)
{
    // Check for erroneous input index
    if (i < 0 || i > n-1)
    {
        printf("Invalid Input");
        return;
    }

    // Get the difference between new value and old value
    int diff = new_val - arr[i];

    // Update the value in array
    arr[i] = new_val;

    // Update the values of nodes in segment tree
    updateValueUtil(st, 0, n-1, i, diff, 0);
}

// Return xor of elements in range from index qs (quey start)
// to qe (query end). It mainly uses getXorUtil()
int getXor(int *st, int n, int qs, int qe)
{
    // Check for erroneous input values
    if (qs < 0 || qe > n-1 || qs > qe)
    {
        printf("Invalid Input");
        return -1;
    }

    return getXorUtil(st, 0, n-1, qs, qe, 0);
}

// A recursive function that constructs Segment Tree for array[ss..se].
// si is index of current node in segment tree st

```

```

int constructSTUtil(int arr[], int ss, int se, int *st, int si)
{
    // If there is one element in array, store it in current node of
    // segment tree and return
    if (ss == se)
    {
        st[si] = arr[ss];
        return arr[ss];
    }

    // If there are more than one elements, then recur for left and
    // right subtrees and store the xor of values in this node
    int mid = getMid(ss, se);
    st[si] = constructSTUtil(arr, ss, mid, st, si*2+1) ^
              constructSTUtil(arr, mid+1, se, st, si*2+2);
    return st[si];
}

/* Function to construct segment tree from given array. This function
   allocates memory for segment tree and calls constructSTUtil() to
   fill the allocated memory */
int *constructST(int arr[], int n)
{
    // Allocate memory for segment tree

    //Height of segment tree
    int x = (int)(ceil(log2(n)));

    //Maximum size of segment tree
    int max_size = 2*(int)pow(2, x) - 1;

    // Allocate memory
    int *st = (int *)malloc(sizeof(int)*max_size);

    // Fill the allocated memory st
    constructSTUtil(arr, 0, n-1, st, 0);

    // Return the constructed segment tree
    return st;
}

// Driver program to test above functions
int main()
{
    int arr[] = {1, 3, 5, 7, 9, 11};
    int n = sizeof(arr)/sizeof(arr[0]);

    // Build segment tree from given array

```

```
int *st = constructST(arr, n);

// Print xor of values in array from index 1 to 3
printf("Xor of values in given range = %d\n",
       getXor(st, n, 1, 3));

// Update: set arr[1] = 10 and update corresponding
// segment tree nodes
updateValue(arr, st, n, 1, 10);

// Find xor after the value is updated
printf("Updated xor of values in given range = %d\n",
       getXor(st, n, 1, 3));
return 0;
}
```

Output:

```
Xor of values in given range = 1
Updated xor of values in given range = 8
```

Time and Space Complexity:

Time Complexity for tree construction is $O(n)$. There are total $2n-1$ nodes, and value of every node is calculated only once in tree construction.

Time complexity to query is $O(\log n)$.

The time complexity of update is also $O(\log n)$.

Total time Complexity is : $O(n)$ for construction + $O(\log n)$ for each query = $O(n) + O(n * \log n) = O(n * \log n)$

Time Complexity $O(n * \log n)$
Auxiliary Space $O(n)$

Source

<https://www.geeksforgeeks.org/segment-tree-set-3-xor-given-range/>

Chapter 161

Segment Trees (Product of given Range Modulo m)

Segment Trees (Product of given Range Modulo m) - GeeksforGeeks

Let us consider the following problem to understand Segment Trees.

We have an array $\text{arr}[0 \dots n-1]$. We should be able to

1 Find the product of elements from index l to r where $0 \leq l \leq r \leq n-1$ take its modulus by an integer m.

2 Change value of a specified element of the array to a new value x. We need to do $\text{arr}[i] = x$ where $0 \leq i \leq n-1$.

A simple solution is to run a loop from l to r and calculate product of elements in given range and modulo it by m. To update a value, simply do $\text{arr}[i] = x$. The first operation takes $O(n)$ time and second operation takes $O(1)$ time.

Another solution is to create two arrays and store the product modulo m from start to l-1 in first array and the product from r+1 to end of the array modulo m in another array. Product of a given range can now be calculated in $O(1)$ time, but update operation takes $O(n)$ time now.

Lets say the product of all the elements be P, then product P from a given range l to r can be calculated as :

P : Product of all the elements of the array modulo m.

A : Product of all the elements till l-1 modulo m.

B : Product of all the elements till r+1 modulo m.

$$\text{PDT} = P * (\text{modInverse}(A)) * (\text{modInverse}(B))$$

This works well if the number of query operations are large and very few updates.

Segment Tree Solution :

If the number of query and updates are equal, we can perform both the operations in $O(\log n)$ time. We can use a Segment Tree to do both operations in $O(\log n)$ time.

Representation of Segment trees

1. Leaf Nodes are the elements of the input array.
2. Each internal node represents some merging of the leaf nodes. The merging may be different for different problems. For this problem, merging is product of leaves under a node.

An array representation of tree is used to represent Segment Trees. For each node at index i , the left child is at index $2*i+1$, right child at $2*i+2$ and the parent is at $(i-1)/2$.

Query for Product of given range

Once the tree is constructed, how to get the product using the constructed segment tree. Following is algorithm to get the product of elements.

```
int getPdt(node, l, r)
{
    if range of node is within l and r
        return value in node
    else if range of node is completely outside l and r
        return 1
    else
        return (getPdt(node's left child, l, r)%mod *
                getPdt(node's right child, l, r)%mod)%mod
}
```

Update a value

Like tree construction and query operations, update can also be done recursively. We are given an index which needs to updated. We start from root of the segment tree, and multiply the range product with new value and divide the range product with previous value. If a node doesn't have given index in its range, we don't make any changes to that node.

Implementation:

Following is implementation of segment tree. The program implements construction of segment tree for any given array. It also implements query and update operations.

```
// C++ program to show segment tree operations like
// construction, query and update
#include <bits/stdc++.h>
#include <math.h>
using namespace std;
int mod = 1000000000;

// A utility function to get the middle index from
// corner indexes.
int getMid(int s, int e) { return s + (e - s)/2; }

/* A recursive function to get the Pdt of values
   in given range of the array. The following are
   parameters for this function.
```

```

st    --> Pointer to segment tree
si    --> Index of current node in the segment tree.
        Initially 0 is passed as root is always
        at index 0
ss & se --> Starting and ending indexes of the
            segment represented by current node,
            i.e., st[si]
qs & qe --> Starting and ending indexes of query
            range */

int getPdtUtil(int *st, int ss, int se, int qs, int qe,
               int si)
{
    // If segment of this node is a part of given
    // range, then return the Pdt of the segment
    if (qs <= ss && qe >= se)
        return st[si];

    // If segment of this node is outside the given range
    if (se < qs || ss > qe)
        return 1;

    // If a part of this segment overlaps with the
    // given range
    int mid = getMid(ss, se);
    return (getPdtUtil(st, ss, mid, qs, qe, 2*si+1)%mod *
            getPdtUtil(st, mid+1, se, qs, qe, 2*si+2)%mod)%mod;
}

/* A recursive function to update the nodes which have
   the given index in their range. The following are
   parameters
   st, si, ss and se are same as getPdtUtil()
   i    --> index of the element to be updated.
           This index is in input array.*/
void updateValueUtil(int *st, int ss, int se, int i,
                     int prev_val, int new_val, int si)
{
    // Base Case: If the input index lies outside
    // the range of this segment
    if (i < ss || i > se)
        return;

    // If the input index is in range of this node, then
    // update the value of the node and its children
    st[si] = (st[si]*new_val)/prev_val;
    if (se != ss)
    {
        int mid = getMid(ss, se);

```

```

        updateValueUtil(st, ss, mid, i, prev_val,
                        new_val, 2*si + 1);
        updateValueUtil(st, mid+1, se, i, prev_val,
                        new_val, 2*si + 2);
    }
}

// The function to update a value in input array
// and segment tree. It uses updateValueUtil() to
// update the value in segment tree
void updateValue(int arr[], int *st, int n, int i,
                 int new_val)
{
    // Check for erroneous input index
    if (i < 0 || i > n-1)
    {
        cout<<"Invalid Input";
        return;
    }
    int temp = arr[i];

    // Update the value in array
    arr[i] = new_val;

    // Update the values of nodes in segment tree
    updateValueUtil(st, 0, n-1, i, temp, new_val, 0);
}

// Return Pdt of elements in range from index qs
// (query start)to qe (query end). It mainly
// uses getPdtUtil()
int getPdt(int *st, int n, int qs, int qe)
{
    // Check for erroneous input values
    if (qs < 0 || qe > n-1 || qs > qe)
    {
        cout<<"Invalid Input";
        return -1;
    }

    return getPdtUtil(st, 0, n-1, qs, qe, 0);
}

// A recursive function that constructs Segment Tree
// for array[ss..se]. si is index of current node
// in segment tree st
int constructSTUtil(int arr[], int ss, int se,
                    int *st, int si)

```

```

{
    // If there is one element in array, store it
    // in current node of segment tree and return
    if (ss == se)
    {
        st[si] = arr[ss];
        return arr[ss];
    }

    // If there are more than one elements, then
    // recur for left and right subtrees and store
    // the Pdt of values in this node
    int mid = getMid(ss, se);
    st[si] = (constructSTUtil(arr, ss, mid, st, si*2+1)%mod *
              constructSTUtil(arr, mid+1, se, st, si*2+2)%mod)%mod;
    return st[si];
}

/* Function to construct segment tree from given array.
   This function allocates memory for segment tree and
   calls constructSTUtil() to fill the allocated memory */
int *constructST(int arr[], int n)
{
    // Allocate memory for segment tree

    // Height of segment tree
    int x = (int)(ceil(log2(n)));

    // Maximum size of segment tree
    int max_size = 2*(int)pow(2, x) - 1;

    // Allocate memory
    int *st = new int[max_size];

    // Fill the allocated memory st
    constructSTUtil(arr, 0, n-1, st, 0);

    // Return the constructed segment tree
    return st;
}

// Driver program to test above functions
int main()
{
    int arr[] = {1, 2, 3, 4, 5, 6};
    int n = sizeof(arr)/sizeof(arr[0]);

    // Build segment tree from given array

```

```
int *st = constructST(arr, n);

// Print Product of values in array from index 1 to 3
cout << "Product of values in given range = "
    << getPdt(st, n, 1, 3) << endl;

// Update: set arr[1] = 10 and update corresponding
// segment tree nodes
updateValue(arr, st, n, 1, 10);

// Find Product after the value is updated
cout << "Updated Product of values in given range = "
    << getPdt(st, n, 1, 3) << endl;
return 0;
}
```

Output:

```
Product of values in given range = 24
Updated Product of values in given range = 120
```

Improved By : [sahilkhoslaa](#)

Source

<https://www.geeksforgeeks.org/segment-trees-product-of-given-range-modulo-m/>

Chapter 162

Segment tree Efficient implementation

Segment tree Efficient implementation - GeeksforGeeks

Let us consider the following problem to understand Segment Trees without recursion.

We have an array $\text{arr}[0 \dots n-1]$. We should be able to,

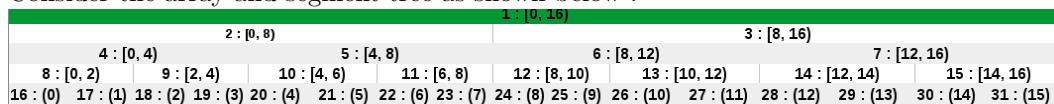
1. Find the sum of elements from index l to r where $0 \leq l \leq r \leq n-1$
2. Change value of a specified element of the array to a new value x. We need to do $\text{arr}[i] = x$ where $0 \leq i \leq n-1$.

A **simple solution** is to run a loop from l to r and calculate sum of elements in given range. To update a value, simply do $\text{arr}[i] = x$. The first operation takes **O(n)** time and second operation takes **O(1)** time.

Another solution is to create another array and store sum from start to i at the ith index in this array. Sum of a given range can now be calculated in **O(1)** time, but update operation takes **O(n)** time now. This works well if the number of query operations are large and very few updates.

What if the number of query and updates are equal? Can we perform both the operations in **O(log n)** time once given the array? We can use a **Segment Tree** to do both operations in **O(Logn)** time. We have discussed the complete implementation of segment trees in our [previous](#) post. In this post we will discuss about a more easy and yet efficient implementation of segment trees than the previous post.

Consider the array and segment tree as shown below :



You can see from the above image that the original array is at the bottom and is 0-indexed with 16 elements. The tree contains a total of 31 nodes where the leaf nodes or the elements

of original array starts from node 16. So, we can easily construct a segment tree for this array using a 2^N sized array where N is number of elements in original array. The leaf nodes will start from index N in this array and will go upto index $(2^N - 1)$. Therefore an element at index i in original array will be at index $(i + N)$ in the segment tree array. Now to calculate the parents, we will start from index $(N - 1)$ and move upward. For an index i , its left child will be at $(2 * i)$ and right child will be at $(2*i + 1)$ index. So the values at nodes at $(2 * i)$ and $(2*i + 1)$ is combined at i th node to construct the tree.

As you can see in the above figure, we can query in this tree in an interval $[L,R]$ with left index(L) included and right (R) excluded.

We will implement all of these multiplication and addition operations using bitwise operators.

Let us know have a look at the complete implementation:

C++

```
#include <bits/stdc++.h>
using namespace std;

// limit for array size
const int N = 100000;

int n; // array size

// Max size of tree
int tree[2 * N];

// function to build the tree
void build( int arr[])
{
    // insert leaf nodes in tree
    for (int i=0; i<n; i++)
        tree[n+i] = arr[i];

    // build the tree by calculating parents
    for (int i = n - 1; i > 0; --i)
        tree[i] = tree[i<<1] + tree[i<<1 | 1];
}

// function to update a tree node
void updateTreeNode(int p, int value)
{
    // set value at position p
    tree[p+n] = value;
    p = p+n;

    // move upward and update parents
    for (int i=p; i > 1; i >>= 1)
        tree[i>>1] = tree[i] + tree[i^1];
}
```

```
}  
  
// function to get sum on interval [l, r)  
int query(int l, int r)  
{  
    int res = 0;  
  
    // loop to find the sum in the range  
    for (l += n, r += n; l < r; l >>= 1, r >>= 1)  
    {  
        if (l&1)  
            res += tree[l++];  
  
        if (r&1)  
            res += tree[--r];  
    }  
  
    return res;  
}  
  
// driver program to test the above function  
int main()  
{  
    int a[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};  
  
    // n is global  
    n = sizeof(a)/sizeof(a[0]);  
  
    // build tree  
    build(a);  
  
    // print the sum in range(1,2) index-based  
    cout << query(1, 3)<<endl;  
  
    // modify element at 2nd index  
    updateTreeNode(2, 1);  
  
    // print the sum in range(1,2) index-based  
    cout << query(1, 3)<<endl;  
  
    return 0;  
}
```

Java

```
import java.io.*;  
  
public class GFG {
```

```
// limit for array size
static int N = 100000;

static int n; // array size

// Max size of tree
static int []tree = new int[2 * N];

// function to build the tree
static void build( int []arr)
{
    // insert leaf nodes in tree
    for (int i = 0; i < n; i++)
        tree[n + i] = arr[i];

    // build the tree by calculating
    // parents
    for (int i = n - 1; i > 0; --i)
        tree[i] = tree[i << 1] +
                   tree[i << 1 | 1];
}

// function to update a tree node
static void updateTreeNode(int p, int value)
{
    // set value at position p
    tree[p + n] = value;
    p = p + n;

    // move upward and update parents
    for (int i = p; i > 1; i >>= 1)
        tree[i >> 1] = tree[i] + tree[i^1];
}

// function to get sum on
// interval [l, r)
static int query(int l, int r)
{
    int res = 0;

    // loop to find the sum in the range
    for (l += n, r += n; l < r;
         l >>= 1, r >>= 1)
    {
        if ((l & 1) > 0)
```

```
        res += tree[l++];

        if ((r & 1) > 0)
            res += tree[--r];
    }

    return res;
}

// driver program to test the
// above function
static public void main (String[] args)
{
    int []a = {1, 2, 3, 4, 5, 6, 7, 8,
               9, 10, 11, 12};

    // n is global
    n = a.length;

    // build tree
    build(a);

    // print the sum in range(1,2)
    // index-based
    System.out.println(query(1, 3));

    // modify element at 2nd index
    updateTreeNode(2, 1);

    // print the sum in range(1,2)
    // index-based
    System.out.println(query(1, 3));
}
}

// This code is contributed by vt_m.
```

C#

```
using System;

public class GFG {

    // limit for array size
    static int N = 100000;

    static int n; // array size
```

```
// Max size of tree
static int []tree = new int[2 * N];

// function to build the tree
static void build( int []arr)
{
    // insert leaf nodes in tree
    for (int i = 0; i < n; i++)
        tree[n + i] = arr[i];

    // build the tree by calculating
    // parents
    for (int i = n - 1; i > 0; --i)
        tree[i] = tree[i << 1] +
                    tree[i << 1 | 1];
}

// function to update a tree node
static void updateTreeNode(int p, int value)
{
    // set value at position p
    tree[p + n] = value;
    p = p + n;

    // move upward and update parents
    for (int i = p; i > 1; i >>= 1)
        tree[i >> 1] = tree[i] + tree[i^1];
}

// function to get sum on
// interval [l, r)
static int query(int l, int r)
{
    int res = 0;

    // loop to find the sum in the range
    for (l += n, r += n; l < r;
                     l >>= 1, r >>= 1)
    {
        if (((l & 1) > 0)
            res += tree[l++];

        if (((r & 1) > 0)
            res += tree[--r];
    }

    return res;
}
```

```
}

// driver program to test the
// above function
static public void Main ()
{
    int []a = {1, 2, 3, 4, 5, 6, 7, 8,
               9, 10, 11, 12};

    // n is global
    n = a.Length;

    // build tree
    build(a);

    // print the sum in range(1,2)
    // index-based
    Console.WriteLine(query(1, 3));

    // modify element at 2nd index
    updateTreeNode(2, 1);

    // print the sum in range(1,2)
    // index-based
    Console.WriteLine(query(1, 3));
}
}

// This code is contributed by vt_m.
```

Output:

```
5
3
```

Yes! This is all. Complete implementation of segment tree including the query and update functions in such a less number of lines of code than the previous recursive one. Let us now understand about how each of the function is working:

1. The picture makes it clear that the leaf nodes are stored at $i+n$, so we can clearly insert all leaf nodes directly.
2. The next step is to build the tree and it takes $O(n)$ time. The parent has always its index less than its children so we just process all the nodes in decreasing order calculating the value of parent node. If the code inside the build function to calculate parents seems confusing then you can see this code, it is equivalent to that inside the build function.

```
tree[i]=tree[2*i]+tree[2*i+1]
```

3. Updating a value at any position is also simple and the time taken will be proportional to the height of the tree. We only update values in the parents of the given node which is being changed. so for getting the parent , we just go up to the parent node , which is $p/2$ or $p>>1$, for node p. p^1 turns $(2*i)$ to $(2*i + 1)$ and vice versa to get the second child of p.
4. Computing the sum also works in $O(\log(n))$ time .if we work through an interval of [3,11), we need to calculate only for nodes 19,26,12 and 5 in that order.

The idea behind the query function is that whether we should include an element in the sum or we should include its parent. Let's look at the image once again for proper understanding. Consider that L is the left border of an interval and R is the right border of the interval $[L,R)$. It is clear from the image that if L is odd then it means that it is the right child of it's parent and our interval includes only L and not it's parent. So we will simply include this node to sum and move to the parent of it's next node by doing $L = (L+1)/2$. Now, if L is even then it is the left child of it's parent and interval includes it's parent also unless the right borders interferes. Similar conditions is applied to the right border also for faster computation. We will stop this iteration once the left and right borders meet.

The theoretical time complexities of both previous implementation and this implementation is same but practically this is found to be much more efficient as there are no recursive calls. We simply iterate over the elements that we need. Also this is very easy to implement.

Time Complexities:

- Tree Construction : $O(n)$
- Query in Range : $O(\log n)$
- Updating an element : $O(\log n).$

References:

<http://codeforces.com/blog/entry/18051>

Improved By : [vt_m](#)

Source

<https://www.geeksforgeeks.org/segment-tree-efficient-implementation/>

Chapter 163

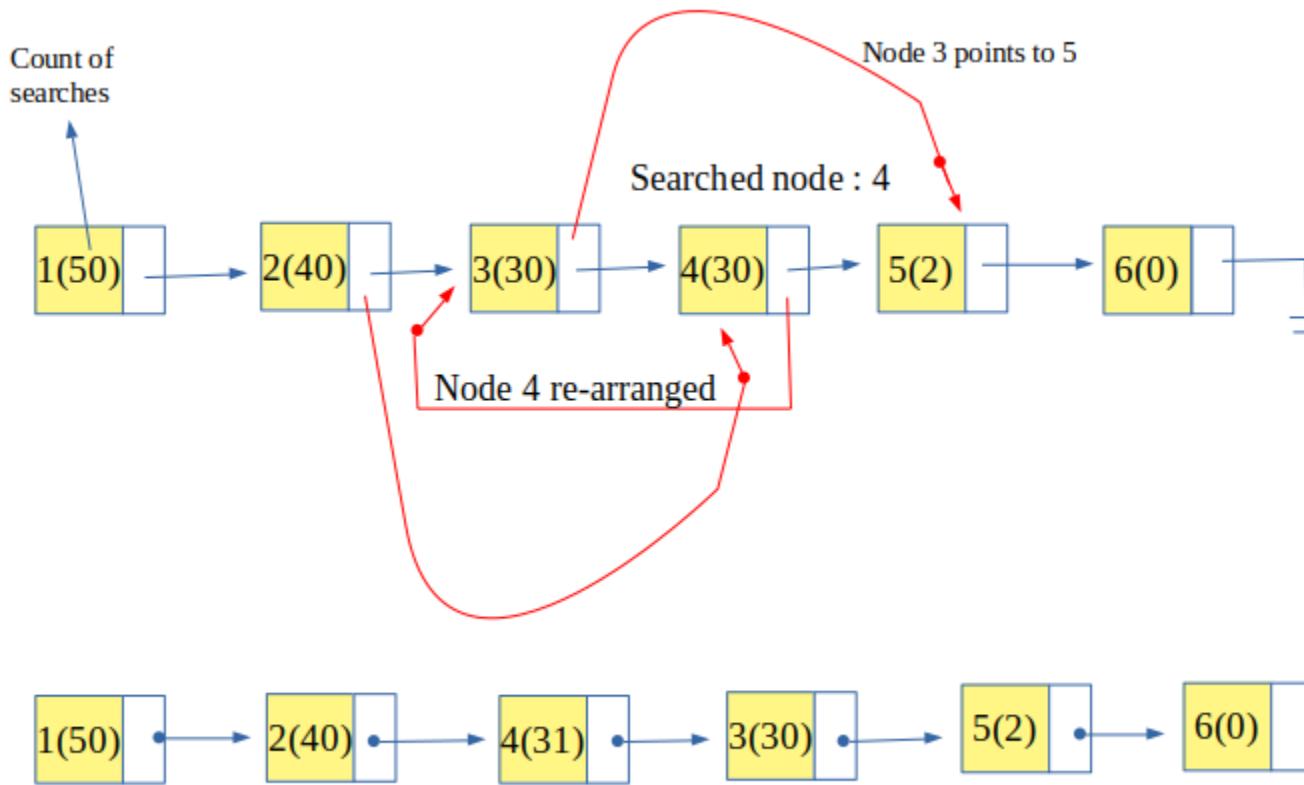
Self Organizing List : Count Method

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 164

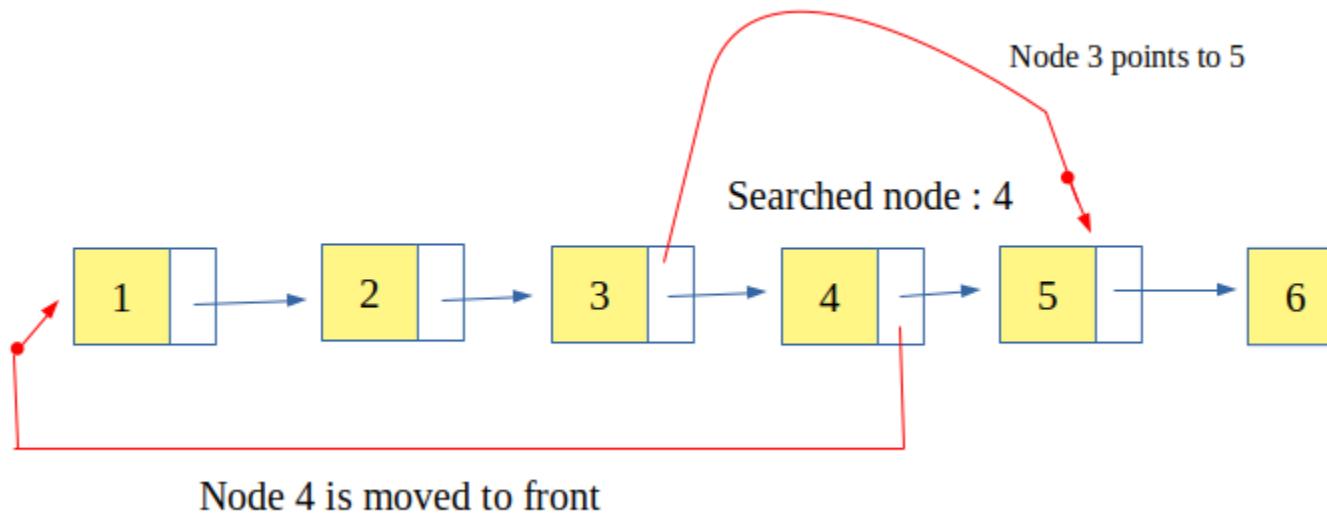
Self Organizing List : Move to Front Method

Self Organizing List : Move to Front Method - GeeksforGeeks

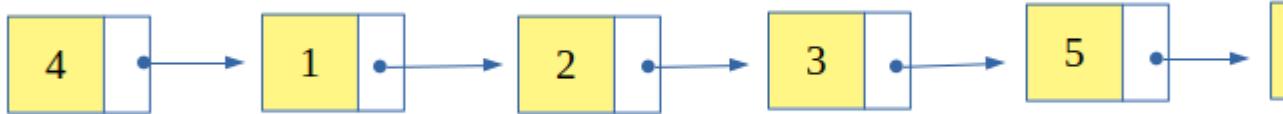
[Self Organizing list](#) is a list that re-organizes or re-arranges itself for better performance. In a simple list, an item to be searched is looked for in a sequential manner which gives the time complexity of $O(n)$. But in real scenario not all the items are searched frequently and most of the time 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 165

Self Organizing List Set 1 (Introduction)

Self Organizing List Set 1 (Introduction) - GeeksforGeeks

The worst case search time for a sorted linked list is $O(n)$. With a Balanced Binary Search Tree, we can skip almost half of the nodes after one comparison with root. For a sorted array, we have random access and we can apply Binary Search on arrays.

One idea to make search faster for Linked Lists is [Skip List](#). Another idea (which is discussed in this post) is to *place more frequently accessed items closer to head..* There can be two possibilities. offline (we know the complete search sequence in advance) and online (we don't know the search sequence).

In case of offline, we can put the nodes according to decreasing frequencies of search (The element having maximum search count is put first). For many practical applications, it may be difficult to obtain search sequence in advance. A [Self Organizing list](#) reorders its nodes based on searches which are done. The idea is to use locality of reference (In a typical database, 80% of the access are to 20% of the items). Following are different strategies used by Self Organizing Lists.

- 1) **Move-to-Front Method:** Any node searched is moved to the front. This strategy is easy to implement, but it may over-reward infrequently accessed items as it always move the item to front.
- 2) **Count Method:** Each node stores count of the number of times it was searched. Nodes are ordered by decreasing count. This strategy requires extra space for storing count.
- 3) **Transpose Method:** Any node searched is swapped with the preceding node. Unlike Move-to-front, this method does not adapt quickly to changing access patterns.

Competitive Analysis:

The worst case time complexity of all methods is $O(n)$. In worst case, the searched element is always the last element in list. For [average case analysis](#), we need probability distribution of search sequences which is not available many times.

For online strategies and algorithms like above, we have a totally different way of analyzing

them called *competitive analysis* where performance of an online algorithm is compared to the performance of an optimal offline algorithm (that can view the sequence of requests in advance). Competitive analysis is used in many practical algorithms like caching, disk paging, high performance computers. The best thing about competitive analysis is, we don't need to assume anything about probability distribution of input. The Move-to-front method is 4-competitive, means it never does more than a factor of 4 operations than offline algorithm (See [the MIT video lecture](#) for proof).

We will soon be discussing implementation and proof of the analysis given in the video lecture.

References:

- http://en.wikipedia.org/wiki/Self-organizing_list
- MIT Video Lecture
- http://www.eecs.yorku.ca/course_archive/2003-04/F/2011/2011A/DatStr_071_SOLists.pdf
- [http://en.wikipedia.org/wiki/Competitive_analysis_\(online_algorithm\)](http://en.wikipedia.org/wiki/Competitive_analysis_(online_algorithm))

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

Source

<https://www.geeksforgeeks.org/self-organizing-list-set-1-introduction/>

Chapter 166

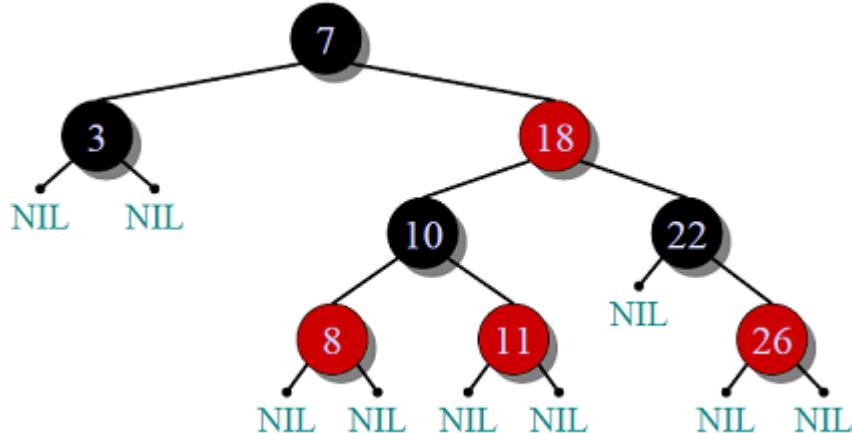
Self-Balancing-Binary-Search-Trees (Comparisons)

Self-Balancing-Binary-Search-Trees (Comparisons) - GeeksforGeeks

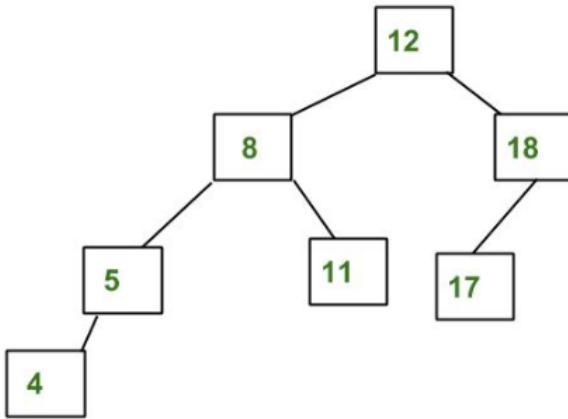
Self-Balancing Binary Search Trees are *height-balanced* binary search trees that automatically keeps height as small as possible when insertion and deletion operations are performed on tree. The height is typically maintained in order of $\log n$ so that all operations take $O(\log n)$ time on average.

Examples :

Red Black Tree



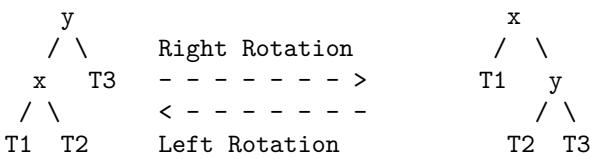
AVL Tree:



How do Self-Balancing-Tree maintain height?

A typical operation done by trees is rotation. Following are two basic operations that can be performed to re-balance a BST without violating the BST property ($\text{keys(left)} < \text{key(root)} < \text{keys(right)}$). 1) Left Rotation 2) Right Rotation

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



Keys in both of the above trees follow the following order

$\text{keys}(T_1) < \text{key}(x) < \text{keys}(T_2) < \text{key}(y) < \text{keys}(T_3)$
So BST property is not violated anywhere.

We have already discussed [AVL tree](#), [Red Black Tree](#) and [Splay Tree](#). In this article, we will compare the efficiency of these trees:

Metric	RB Tree	AVL Tree
worst case	$O(1)$	$O(\log n)$
of tree	$2 * \log(n)$	$1.44 * \log(n)$
worst case		
Moderate		
Faster		
Slower		

Metric	RB Tree	AVL Tree
Efficient Implementation requires node no extra information	Three pointers with color bit per node	
worst case	$O(\log n)$	$O(\log n)$
Mostly used retrieved again and again	As universal data structure	When frequent lookups
Real world Application	Multiset, Multimap, Map, Set, etc.	Database Transactions

Source

<https://www.geeksforgeeks.org/self-balancing-binary-search-trees-comparisons/>

Chapter 167

Skew Heap

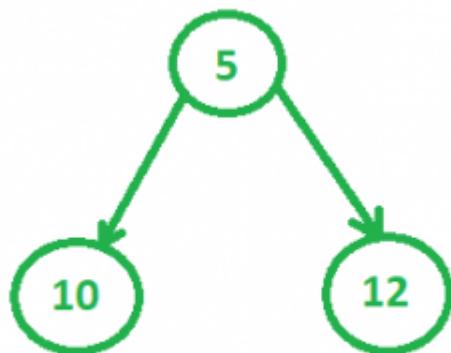
Skew Heap - GeeksforGeeks

A **skew heap** (or self – adjusting heap) is a heap data structure implemented as a **binary tree**. Skew heaps are advantageous because of their ability to **merge more quickly** than binary heaps. In contrast with **binary heaps**, there are no structural constraints, so there is no guarantee that the height of the tree is logarithmic. Only two conditions must be satisfied :

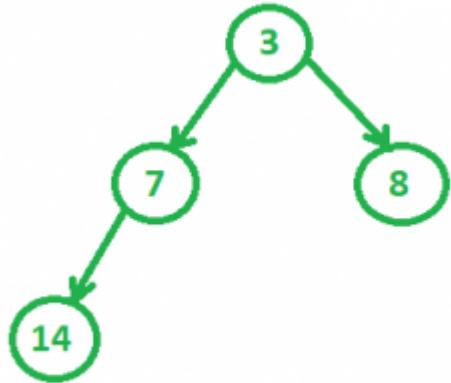
1. The general heap order must be there (root is minimum and same is recursively true for subtrees), but balanced property (all levels must be full except the last) is not required.
2. Main operation in Skew Heaps is Merge. We can implement other operations like insert, extractMin(), etc using Merge only.

Example :

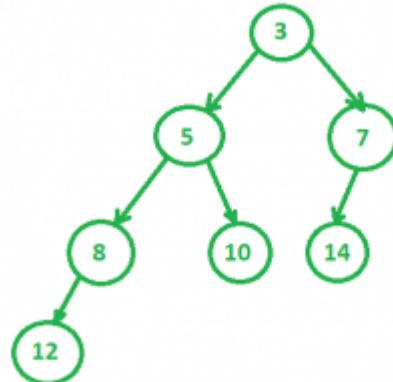
1. Consider the skew heap 1 to be



2. The second heap to be considered



4. And we obtain the final merged tree as



Recursive Merge Process :

merge(h1, h2)

1. Let h1 and h2 be the two min skew heaps to be merged. Let h1's root be smaller than h2's root (If not smaller, we can swap to get the same).
2. We swap h1->left and h1->right.
3. h1->left = merge(h2, h1->left)

Examples :

Let h1 be



```

      20      30
     /       /
    40      50
  
```

Let h2 be

```

      15
     /   \
    25   35
   /   \
  45   55
  
```

After swapping h1->left and h1->right, we get

```

      10
     /   \
    30   20
   /   /
  50   40
  
```

Now we recursively Merge

```

      30
     /   AND
    40
  
```

```

      15
     /   \
    25   35
   /   \
  45   55
  
```

After recursive merge, we get (Please do it using pen and paper).

```

      15
     /   \
    30   25
   / \   \
  35  40   45
  
```

We make this merged tree as left of original h1 and we get following result.

```

      10
     /   \
    15   20
   / \   /
  30  25  40
 / \   \
35  40  45
  
```

For visualization : <https://www.cs.usfca.edu/~galles/JavascriptVisual/LeftistHeap.html>

```
// CPP program to implement Skew Heap
// operations.
#include <bits/stdc++.h>
using namespace std;

struct SkewHeap
{
    int key;
    SkewHeap* right;
    SkewHeap* left;

    // constructor to make a new
    // node of heap
    SkewHeap()
    {
        key = 0;
        right = NULL;
        left = NULL;
    }

    // the special merge function that's
    // used in most of the other operations
    // also
    SkewHeap* merge(SkewHeap* h1, SkewHeap* h2)
    {
        // If one of the heaps is empty
        if (h1 == NULL)
            return h2;
        if (h2 == NULL)
            return h1;

        // Make sure that h1 has smaller
        // key.
        if (h1->key > h2->key)
            swap(h1, h2);

        // Swap h1->left and h1->right
        swap(h1->left, h1->right);

        // Merge h2 and h1->left and make
        // merged tree as left of h1.
        h1->left = merge(h2, h1->left);

        return h1;
    }

    // function to construct heap using
    // values in the array
```

```

SkewHeap* construct(SkewHeap* root,
                     int heap[], int n)
{
    SkewHeap* temp;
    for (int i = 0; i < n; i++) {
        temp = new SkewHeap;
        temp->key = heap[i];
        root = merge(root, temp);
    }
    return root;
}

// function to print the Skew Heap,
// as it is in form of a tree so we use
// tree traversal algorithms
void inorder(SkewHeap* root)
{
    if (root == NULL)
        return;
    else {
        inorder(root->left);
        cout << root->key << " ";
        inorder(root->right);
    }
    return;
};

// Driver Code
int main()
{
    // Construct two heaps
    SkewHeap heap, *temp1 = NULL,
              *temp2 = NULL;
    /*
      5
     / \
    /   \
   10   12  */
    int heap1[] = { 12, 5, 10 };
    /*
      3
     / \
    /   \
   7   8
  /
 /
14  */
}

```

```
int heap2[] = { 3, 7, 8, 14 };
int n1 = sizeof(heap1) / sizeof(heap1[0]);
int n2 = sizeof(heap2) / sizeof(heap2[0]);
temp1 = heap.construct(temp1, heap1, n1);
temp2 = heap.construct(temp2, heap2, n2);

// Merge two heaps
temp1 = heap.merge(temp1, temp2);
/*
      3
     / \
    /   \
   5     7
  / \   /
 8 10 14
/
12 */
cout << "Merged Heap is: " << endl;
heap.inorder(temp1);
}
```

Output:

The heap obtained after merging is:
12 8 5 10 3 14 7

Source

<https://www.geeksforgeeks.org/skew-heap/>

Chapter 168

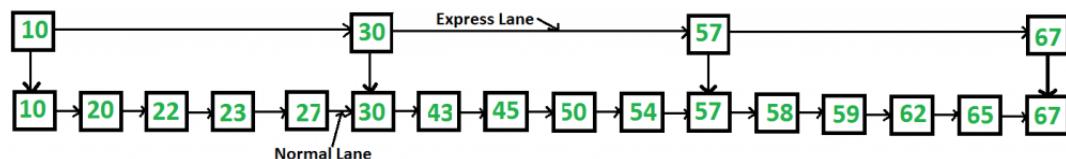
Skip List Set 1 (Introduction)

Skip List Set 1 (Introduction) - GeeksforGeeks

Can we search in a sorted linked list in better than O(n) time?

The worst case search time for a sorted linked list is O(n) as we can only linearly traverse the list and cannot skip nodes while searching. For a Balanced Binary Search Tree, we skip almost half of the nodes after one comparison with root. For a sorted array, we have random access and we can apply Binary Search on arrays.

Can we augment sorted linked lists to make the search faster? The answer is [Skip List](#). The idea is simple, we create multiple layers so that we can skip some nodes. See the following example list with 16 nodes and two layers. The upper layer works as an “express lane” which connects only main outer stations, and the lower layer works as a “normal lane” which connects every station. Suppose we want to search for 50, we start from first node of “express lane” and keep moving on “express lane” till we find a node whose next is greater than 50. Once we find such a node (30 is the node in following example) on “express lane”, we move to “normal lane” using pointer from this node, and linearly search for 50 on “normal lane”. In following example, we start from 30 on “normal lane” and with linear search, we find 50.



What is the time complexity with two layers? The worst case time complexity is number of nodes on “express lane” plus number of nodes in a segment (A segment is number of “normal lane” nodes between two “express lane” nodes) of “normal lane”. So if we have n nodes on “normal lane”, \sqrt{n} (square root of n) nodes on “express lane” and we equally divide the “normal lane”, then there will be \sqrt{n} nodes in every segment of “normal lane”. \sqrt{n} is actually optimal division with two layers. With this arrangement, the number of nodes traversed for a search will be $O(\sqrt{n})$. Therefore, with $O(\sqrt{n})$ extra space, we are able to reduce the time complexity to $O(\sqrt{n})$.

Can we do better?

The time complexity of skip lists can be reduced further by adding more layers. In fact, the time complexity of search, insert and delete can become $O(\log n)$ in average case with $O(n)$ extra space. We will soon be publishing more posts on Skip Lists.

References

MIT Video Lecture on Skip Lists
http://en.wikipedia.org/wiki/Skip_list

Improved By : [freeman1](#)

Source

<https://www.geeksforgeeks.org/skip-list/>

Chapter 169

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 –

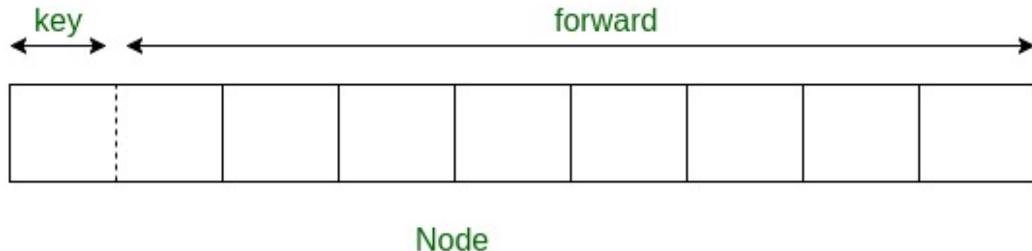
```
randomLevel()
lvl := 1
//random() that returns a random value in [0...1)
while random() < p and lvl < MaxLevel do
    lvl := lvl + 1
return lvl
```

MaxLevel is the upper bound on number of levels in the skip list. It can be determined as

$$\sum_{i=1}^{MaxLevel} \frac{1}{2^i} = \log p / \log 2$$
. 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.

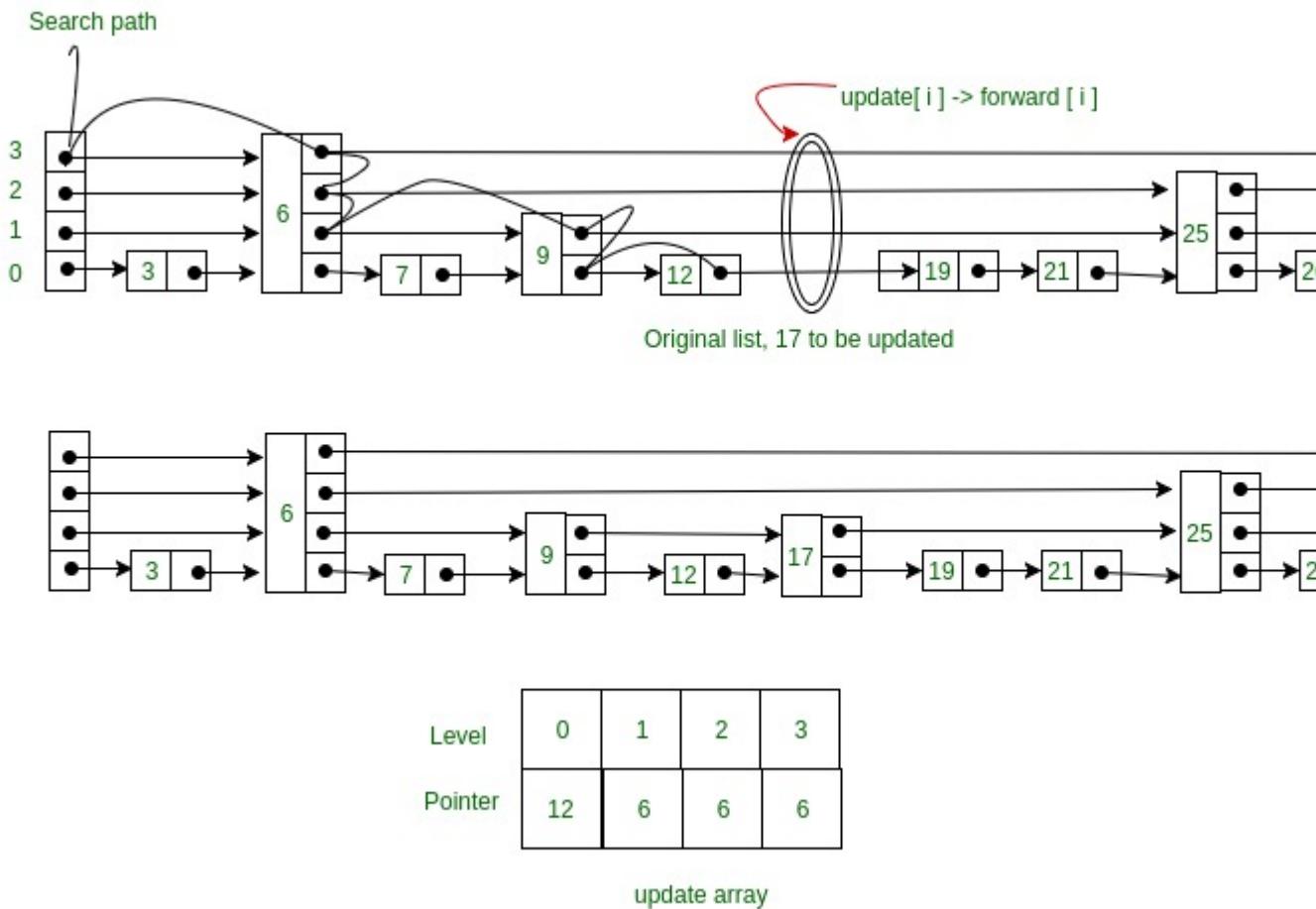
At the level 0, we will definitely find a position to insert given key. Following is the psuedo code for the insertion algorithm –

```

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()

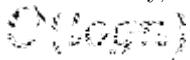
main()
```

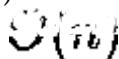
Output:

```
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 170

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 –

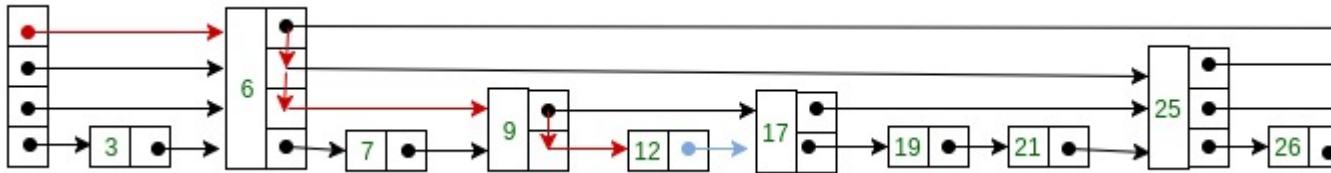
1. Key of next node is less than search key then we keep on moving forward on the same level.
2. Key of next node is greater than the key to be inserted then we store the pointer to current node **i** at **update[i]** and move one level down and continue our search.

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 .

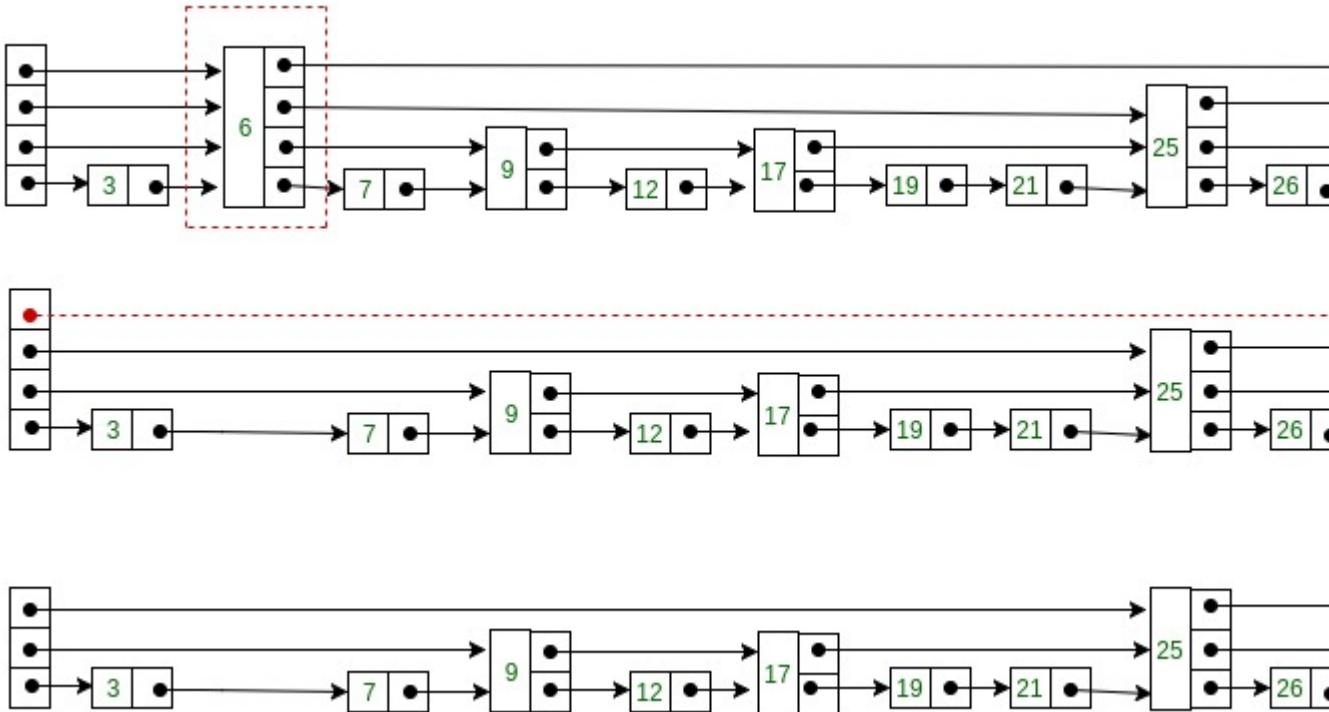
After deletion of element there could be levels with no elements, so we will remove these levels as well by decrementing the level of Skip list. Following is the pseudo code for deletion

```

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): $\mathcal{O}(\log n)$
Time complexity (Worst): $\mathcal{O}(n)$

References

<ftp://ftp.cs.umd.edu/pub/skipLists/skiplists.pdf>

Source

<https://www.geeksforgeeks.org/skip-list-set-3-searching-deletion/>

Chapter 171

Smallest Subarray with given GCD

Smallest Subarray with given GCD - GeeksforGeeks

Given an array arr[] of n numbers and an integer k, find length of the minimum sub-array with gcd equals to k.

Example:

```
Input: arr[] = {6, 9, 7, 10, 12,
                24, 36, 27},
           K = 3
Output: 2
Explanation: GCD of subarray {6,9} is 3.
GCD of subarray {24,36,27} is also 3,
but {6,9} is the smallest
```

Note: Time complexity analysis of below approaches assume that numbers are fixed size and finding GCD of two elements take constant time.

Method 1

Find GCD of all subarrays and keep track of the minimum length subarray with gcd k. Time Complexity of this is $O(n^3)$, $O(n^2)$ for each subarray and $O(n)$ for finding gcd of a subarray.

Method 2

Find GCD of all subarrays using segment tree based approach discussed [here](#). Time complexity of this solution is $O(n^2 \log n)$, $O(n^2)$ for each subarray and $O(\log n)$ for finding GCD of subarray using segment tree.

Method 3

The idea is to use [Segment Tree](#) and Binary Search to achieve time complexity $O(n (\log n)^2)$.

1. If we have any number equal to 'k' in the array then the answer is 1 as GCD of k is k. Return 1.
2. If there is no number which is divisible by k, then GCD doesn't exist. Return -1.
3. If none of the above cases is true, the length of minimum subarray is either greater than 1 or GCD doesn't exist. In this case, we follow following steps.
 - Build segment tree so that we can quickly find GCD of any subarray using the approach discussed [here](#)
 - After building Segment Tree, we consider every index as starting point and do binary search for ending point such that the subarray between these two points has GCD k

Following is C++ implementation of above idea.

```

// C++ Program to find GCD of a number in a given Range
// using segment Trees
#include <bits/stdc++.h>
using namespace std;

// To store segment tree
int *st;

// Function to find gcd of 2 numbers.
int gcd(int a, int b)
{
    if (a < b)
        swap(a, b);
    if (b==0)
        return a;
    return gcd(b, a%b);
}

/* A recursive function to get gcd of given
range of array indexes. The following are parameters for
this function.

st    --> Pointer to segment tree
si --> Index of current node in the segment tree. Initially
      0 is passed as root is always at index 0
ss & se  --> Starting and ending indexes of the segment
              represented by current node, i.e., st[index]
qs & qe  --> Starting and ending indexes of query range */
int findGcd(int ss, int se, int qs, int qe, int si)
{
    if (ss>qe || se < qs)
        return 0;

```

```

        if (qs<=ss && qe>=se)
            return st[si];
        int mid = ss+(se-ss)/2;
        return gcd(findGcd(ss, mid, qs, qe, si*2+1),
                   findGcd(mid+1, se, qs, qe, si*2+2));
    }

//Finding The gcd of given Range
int findRangeGcd(int ss, int se, int arr[], int n)
{
    if (ss<0 || se > n-1 || ss>se)
    {
        cout << "Invalid Arguments" << "\n";
        return -1;
    }
    return findGcd(0, n-1, ss, se, 0);
}

// A recursive function that constructs Segment Tree for
// array[ss..se]. si is index of current node in segment
// tree st
int constructST(int arr[], int ss, int se, int si)
{
    if (ss==se)
    {
        st[si] = arr[ss];
        return st[si];
    }
    int mid = ss+(se-ss)/2;
    st[si] = gcd(constructST(arr, ss, mid, si*2+1),
                  constructST(arr, mid+1, se, si*2+2));
    return st[si];
}

/* Function to construct segment tree from given array.
   This function allocates memory for segment tree and
   calls constructSTUtil() to fill the allocated memory */
int *constructSegmentTree(int arr[], int n)
{
    int height = (int)(ceil(log2(n)));
    int size = 2*(int)pow(2, height)-1;
    st = new int[size];
    constructST(arr, 0, n-1, 0);
    return st;
}

// Returns size of smallest subarray of arr[0..n-1]
// with GCD equal to k.

```

```
int findSmallestSubarr(int arr[], int n, int k)
{
    // To check if a multiple of k exists.
    bool found = false;

    // Find if k or its multiple is present
    for (int i=0; i<n; i++)
    {
        // If k is present, then subarray size is 1.
        if (arr[i] == k)
            return 1;

        // Break the loop to indicate presence of a
        // multiple of k.
        if (arr[i] % k == 0)
            found = true;
    }

    // If there was no multiple of k in arr[], then
    // we can't get k as GCD.
    if (found == false)
        return -1;

    // If there is a multiple of k in arr[], build
    // segment tree from given array
    constructSegmentTree(arr, n);

    // Initialize result
    int res = n+1;

    // Now consider every element as starting point
    // and search for ending point using Binary Search
    for (int i=0; i<n-1; i++)
    {
        // a[i] cannot be a starting point, if it is
        // not a multiple of k.
        if (arr[i] % k != 0)
            continue;

        // Initialize indexes for binary search of closest
        // ending point to i with GCD of subarray as k.
        int low = i+1;
        int high = n-1;

        // Initialize closest ending point for i.
        int closest = 0;

        // Binary Search for closest ending point
        while (low <= high)
        {
            int mid = low + (high - low) / 2;
            if (arr[mid] % k == 0)
            {
                closest = mid;
                high = mid - 1;
            }
            else
                low = mid + 1;
        }

        res = min(res, closest - i);
    }
}
```

```
// with GCD equal to k.
while (true)
{
    // Find middle point and GCD of subarray
    // arr[i..mid]
    int mid = low + (high-low)/2;
    int gcd = findRangeGcd(i, mid, arr, n);

    // If GCD is more than k, look further
    if (gcd > k)
        low = mid;

    // If GCD is k, store this point and look for
    // a closer point
    else if (gcd == k)
    {
        high = mid;
        closest = mid;
        break;
    }

    // If GCD is less than, look closer
    else
        high = mid;

    // If termination condition reached, set
    // closest
    if (abs(high-low) <= 1)
    {
        if (findRangeGcd(i, low, arr, n) == k)
            closest = low;
        else if (findRangeGcd(i, high, arr, n)==k)
            closest = high;
        break;
    }
}

if (closest != 0)
    res = min(res, closest - i + 1);
}

// If res was not changed by loop, return -1,
// else return its value.
return (res == n+1) ? -1 : res;
}

// Driver program to test above functions
int main()
```

```
{  
    int n = 8;  
    int k = 3;  
    int arr[] = {6, 9, 7, 10, 12, 24, 36, 27};  
    cout << "Size of smallest sub-array with given"  
         << " size is " << findSmallestSubarr(arr, n, k);  
    return 0;  
}
```

Output:

2

Example:

arr[] = {6, 9, 7, 10, 12, 24, 36, 27}, K = 3

```
// Initial value of minLen is equal to size  
// of array  
minLen = 8
```

No element is equal to k so result is either greater than 1 or doesn't exist.

First index

- GCD of subarray from 1 to 5 is 1.
- GCD < k
- GCD of subarray from 1 to 3 is 1.
- GCD < k
- GCD of subarray from 1 to 2 is 3
- minLen = minimum(8, 2) = 2

Second Index

- GCD of subarray from 2 to 5 is 1
- GCD < k
- GCD of subarray from 2 to 4 is 1
- GCD < k

- GCD of subarray from 6 to 8 is 3
- minLen = minimum(2, 3) = 2.

.

.

.

Sixth Index

- GCD of subarray from 6 to 7 is 12
- GCD > k
- GCD of subarray from 6 to 8 is 3
- minLen = minimum(2, 3) = 2

Time Complexity: $O(n (\log n)^2)$, $O(n)$ for traversing to each index, $O(\log n)$ for each subarray and $O(\log n)$ for GCD of each subarray.

Source

<https://www.geeksforgeeks.org/smallest-subarray-with-given-gcd/>

Chapter 172

Sort numbers stored on different machines

Sort numbers stored on different machines - GeeksforGeeks

Given N machines. Each machine contains some numbers in sorted form. But the amount of numbers, each machine has is not fixed. Output the numbers from all the machine in sorted non-decreasing form.

Example:

```
Machine M1 contains 3 numbers: {30, 40, 50}
Machine M2 contains 2 numbers: {35, 45}
Machine M3 contains 5 numbers: {10, 60, 70, 80, 100}

Output: {10, 30, 35, 40, 45, 50, 60, 70, 80, 100}
```

Representation of stream of numbers on each machine is considered as linked list. A Min Heap can be used to print all numbers in sorted order.

Following is the detailed process

1. Store the head pointers of the linked lists in a minHeap of size N where N is number of machines.
2. Extract the minimum item from the minHeap. Update the minHeap by replacing the head of the minHeap with the next number from the linked list or by replacing the head of the minHeap with the last number in the minHeap followed by decreasing the size of heap by 1.
3. Repeat the above step 2 until heap is not empty.

Below is C++ implementation of the above approach.

```
// A program to take numbers from different machines and print them in sorted order
```

```
#include <stdio.h>

// A Linked List node
struct ListNode
{
    int data;
    struct ListNode* next;
};

// A Min Heap Node
struct MinHeapNode
{
    ListNode* head;
};

// A Min Heao (Collection of Min Heap nodes)
struct MinHeap
{
    int count;
    int capacity;
    MinHeapNode* array;
};

// A function to create a Min Heap of given capacity
MinHeap* createMinHeap( int capacity )
{
    MinHeap* minHeap = new MinHeap;
    minHeap->capacity = capacity;
    minHeap->count = 0;
    minHeap->array = new MinHeapNode [minHeap->capacity];
    return minHeap;
}

/* A utility function to insert a new node at the begining
   of linked list */
void push (ListNode** head_ref, int new_data)
{
    /* allocate node */
    ListNode* new_node = new ListNode;

    /* 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 swap two min heap nodes. This function
// is needed in minHeapify
void swap( MinHeapNode* a, MinHeapNode* b )
{
    MinHeapNode temp = *a;
    *a = *b;
    *b = temp;
}

// The standard minHeapify function.
void minHeapify( MinHeap* minHeap, int idx )
{
    int left, right, smallest;
    left = 2 * idx + 1;
    right = 2 * idx + 2;
    smallest = idx;

    if ( left < minHeap->count &&
        minHeap->array[left].head->data <
        minHeap->array[smallest].head->data
    )
        smallest = left;

    if ( right < minHeap->count &&
        minHeap->array[right].head->data <
        minHeap->array[smallest].head->data
    )
        smallest = right;

    if( smallest != idx )
    {
        swap( &minHeap->array[smallest], &minHeap->array[idx] );
        minHeapify( minHeap, smallest );
    }
}

// A utility function to check whether a Min Heap is empty or not
int isEmpty( MinHeap* minHeap )
{
    return (minHeap->count == 0);
}

// A standard function to build a heap
void buildMinHeap( MinHeap* minHeap )
{
    int i, n;
```

```
n = minHeap->count - 1;
for( i = (n - 1) / 2; i >= 0; --i )
    minHeapify( minHeap, i );
}

// This function inserts array elements to heap and then calls
// buildHeap for heap property among nodes
void populateMinHeap( MinHeap* minHeap, ListNode* *array, int n )
{
    for( int i = 0; i < n; ++i )
        minHeap->array[ minHeap->count++ ].head = array[i];

    buildMinHeap( minHeap );
}

// Return minimum element from all linked lists
ListNode* extractMin( MinHeap* minHeap )
{
    if( isEmpty( minHeap ) )
        return NULL;

    // The root of heap will have minimum value
    MinHeapNode temp = minHeap->array[0];

    // Replace root either with next node of the same list.
    if( temp.head->next )
        minHeap->array[0].head = temp.head->next;
    else // If list empty, then reduce heap size
    {
        minHeap->array[0] = minHeap->array[ minHeap->count - 1 ];
        --minHeap->count;
    }

    minHeapify( minHeap, 0 );
    return temp.head;
}

// The main function that takes an array of lists from N machines
// and generates the sorted output
void externalSort( ListNode *array[], int N )
{
    // Create a min heap of size equal to number of machines
    MinHeap* minHeap = createMinHeap( N );

    // populate first item from all machines
    populateMinHeap( minHeap, array, N );

    while ( !isEmpty( minHeap ) )
```

```
{  
    ListNode* temp = extractMin( minHeap );  
    printf( "%d ",temp->data );  
}  
}  
  
// Driver program to test above functions  
int main()  
{  
    int N = 3; // Number of machines  
  
    // an array of pointers storing the head nodes of the linked lists  
    ListNode *array[N];  
  
    // Create a Linked List 30->40->50 for first machine  
    array[0] = NULL;  
    push (&array[0], 50);  
    push (&array[0], 40);  
    push (&array[0], 30);  
  
    // Create a Linked List 35->45 for second machine  
    array[1] = NULL;  
    push (&array[1], 45);  
    push (&array[1], 35);  
  
    // Create Linked List 10->60->70->80 for third machine  
    array[2] = NULL;  
    push (&array[2], 100);  
    push (&array[2], 80);  
    push (&array[2], 70);  
    push (&array[2], 60);  
    push (&array[2], 10);  
  
    // Sort all elements  
    externalSort( array, N );  
  
    return 0;  
}
```

Output:

10 30 35 40 45 50 60 70 80 100

Source

<https://www.geeksforgeeks.org/sort-numbers-stored-on-different-machines/>

Chapter 173

Sorting array of strings (or words) using Trie Set-2 (Handling Duplicates)

Sorting array of strings (or words) using Trie Set-2 (Handling Duplicates) - GeeksforGeeks

Given an array of strings, print them in alphabetical (dictionary) order. If there are duplicates in input array, we need to print all the occurrences.

Examples:

```
Input : arr[] = { "abc", "xyz", "abcd", "bcd", "abc" }  
Output : abc abc abcd bcd xyz
```

```
Input : arr[] = { "geeks", "for", "geeks", "a", "portal",  
                 "to", "learn" }  
Output : a for geeks geeks learn portal to
```

Prerequisite: [Trie \(Insert and Search\).](#)

Approach: In the [previous post](#) array of strings is being sorted, printing only single occurrence of duplicate strings. In this post all occurrences of duplicate strings are printed in lexicographic order. To print the strings in alphabetical order we have to first insert them in the trie and then perform preorder traversal to print in alphabetical order. The nodes of trie contain an **index[]** array which stores the index position of all the strings of **arr[]** ending at that node. Except for trie's leaf node all the other nodes have size 0 for the **index[]** array.

Below is the implementation of the above approach.

C++

```
// C++ implementation to sort an array
```

```
// of strings using Trie
#include <bits/stdc++.h>
using namespace std;

const int MAX_CHAR = 26;

struct Trie {

    // 'index' vectors size is greater than
    // 0 when node/ is a leaf node, otherwise
    // size is 0;
    vector<int> index;

    Trie* child[MAX_CHAR];

    /*to make new trie*/
    Trie()
    {
        // initializing fields
        for (int i = 0; i < MAX_CHAR; i++)
            child[i] = NULL;
    }
};

// function to insert a string in trie
void insert(Trie* root, string str, int index)
{
    Trie* node = root;

    for (int i = 0; i < str.size(); i++) {

        // taking ascii value to find index of
        // child node
        char ind = str[i] - 'a';

        // making a new path if not already
        if (!node->child[ind])
            node->child[ind] = new Trie();

        // go to next node
        node = node->child[ind];
    }

    // Mark leaf (end of string) and store
    // index of 'str' in index[]
    (node->index).push_back(index);
}
```

```
// function for preorder traversal of trie
void preorder(Trie* node, string arr[])
{
    // if node is empty
    if (node == NULL)
        return;

    for (int i = 0; i < MAX_CHAR; i++) {
        if (node->child[i] != NULL) {

            // if leaf node then print all the strings
            // for (node->child[i]->index).size() > 0)
            for (int j = 0; j < (node->child[i]->index).size(); j++)
                cout << arr[node->child[i]->index[j]] << " ";

            preorder(node->child[i], arr);
        }
    }
}

// function to sort an array
// of strings using Trie
void printSorted(string arr[], int n)
{
    Trie* root = new Trie();

    // insert all strings of dictionary into trie
    for (int i = 0; i < n; i++)
        insert(root, arr[i], i);

    // print strings in lexicographic order
    preorder(root, arr);
}

// Driver program to test above
int main()
{
    string arr[] = { "abc", "xyz", "abcd", "bcd", "abc" };
    int n = sizeof(arr) / sizeof(arr[0]);
    printSorted(arr, n);
    return 0;
}
```

Java

```
// Java implementation
// to sort an array of
// strings using Trie
```

```
import java.util.*;

class Trie {

    private Node rootNode;

    /*to make new trie*/
    Trie()
    {
        rootNode = null;
    }

    // function to insert
    // a string in trie
    void insert(String key, int index)
    {
        // making a new path
        // if not already
        if (rootNode == null)
        {
            rootNode = new Node();
        }

        Node currentNode = rootNode;

        for (int i = 0;i < key.length();i++)
        {
            char keyChar = key.charAt(i);

            if (currentNode.getChild(keyChar) == null)
            {
                currentNode.addChild(keyChar);
            }

            // go to next node
            currentNode = currentNode.getChild(keyChar);
        }

        // Mark leaf (end of string)
        // and store index of 'str'
        // in index[]
        currentNode.addIndex(index);
    }

    void traversePreorder(String[] array)
    {
        traversePreorder(rootNode, array);
    }
}
```

```
// function for preorder
// traversal of trie
private void traversePreorder(Node node,
                               String[] array)
{
    if (node == null)
    {
        return;
    }

    if (node.getIndices().size() > 0)
    {
        for (int index : node.getIndices())
        {
            System.out.print(array[index] + " ");
        }
    }

    for (char index = 'a';index <= 'z';index++)
    {
        traversePreorder(node.getChild(index), array);
    }
}

private static class Node {

    private Node[] children;
    private List<Integer> indices;

    Node()
    {
        children = new Node[26];
        indices = new ArrayList<>(0);
    }

    Node getChild(char index)
    {
        if (index < 'a' || index > 'z')
        {
            return null;
        }

        return children[index - 'a'];
    }

    void addChild(char index)
    {

```

```
if (index < 'a' || index > 'z')
{
    return;
}

Node node = new Node();
children[index - 'a'] = node;
}

List<Integer> getIndices()
{
    return indices;
}

void addIndex(int index)
{
    indices.add(index);
}
}

class SortStrings {

    // Driver program
    public static void main(String[] args)
    {
        String[] array = { "abc", "xyz",
                          "abcd", "bcd", "abc" };
        printInSortedOrder(array);
    }

    // function to sort an array
    // of strings using Trie
    private static void printInSortedOrder(String[] array)
    {
        Trie trie = new Trie();

        for (int i = 0;i < array.length;i++)
        {
            trie.insert(array[i], i);
        }

        trie.traversePreorder(array);
    }
}

// Contributed by Harikrishnan Rajan
```

Chapter 173. Sorting array of strings (or words) using Trie Set-2 (Handling Duplicates)

Output:

abc abc abcd bcd xyz

Source

<https://www.geeksforgeeks.org/sorting-array-strings-words-using-trie-set-2-handling-duplicates/>

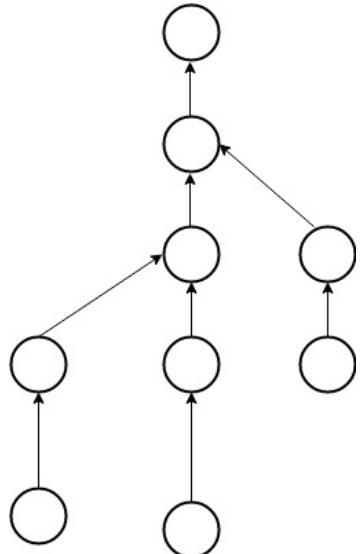
Chapter 174

Spaghetti Stack

Spaghetti Stack - GeeksforGeeks

Spaghetti stack

A spaghetti stack is an N-ary tree data structure in which child nodes have pointers to the parent nodes (but not vice-versa)



Spaghetti stack structure is used in situations when records are dynamically pushed and popped onto a stack as execution progresses, but references to the popped records remain in use. Following are some applications of Spaghetti Stack.

Compilers for languages such as C create a spaghetti stack as it opens and closes symbol tables representing block scopes. When a new block scope is opened, a symbol table is pushed onto a stack. When the closing curly brace is encountered, the scope is closed and the symbol table is popped. But that symbol table is remembered, rather than destroyed. And of course it remembers its higher level “parent” symbol table and so on.

Spaghetti Stacks are also used to implement [Disjoint-set data structure](#).

Sources:

http://en.wikipedia.org/wiki/Spaghetti_stack

Source

<https://www.geeksforgeeks.org/g-fact-87/>

Chapter 175

Sparse Set

Sparse Set - GeeksforGeeks

How to do the following operations efficiently if there are large number of queries for them.

1. Insertion
2. Deletion
3. Searching
4. Clearing/Removing all the elements.

One solution is to use a Self-Balancing Binary Search Tree like Red-Black Tree, AVL Tree, etc. Time complexity of this solution for insertion, deletion and searching is $O(\log n)$.

We can also use Hashing. With hashing, time complexity of first three operations is $O(1)$. But time complexity of the fourth operation is $O(n)$.

We can also use bit-vector (or direct access table), but bit-vector also requires $O(n)$ time for clearing.

Sparse Set outperforms all BST, Hashing and bit vector. We assume that we are given range of data (or maximum value an element can have) and maximum number of elements that can be stored in set. The idea is to maintain two arrays: `sparse[]` and `dense[]`.

```
dense[]  ==> Stores the actual elements
sparse[] ==> This is like bit-vector where
              we use elements as index. Here
              values are not binary, but
              indexes of dense array.
maxVal   ==> Maximum value this set can
              store. Size of sparse[] is
              equal to maxVal + 1.
```

```

capacity ==> Capacity of Set. Size of sparse
           is equal to capacity.
n         ==> Current number of elements in
           Set.

```

insert(x): Let x be the element to be inserted. If x is greater than **maxVal** or n (current number of elements) is greater than equal to capacity, we return.

If none of the above conditions is true, we insert x in `dense[]` at index n (position after last element in a 0 based indexed array), increment n by one (Current number of elements) and store n (index of x in `dense[]`) at `sparse[x]`.

search(x): To search an element x , we use x as index in `sparse[]`. The value `sparse[x]` is used as index in `dense[]`. And if value of `dense[sparse[x]]` is equal to x , we return `dense[x]`. Else we return -1.

delete(x): To delete an element x , we replace it with last element in `dense[]` and update index of last element in `sparse[]`. Finally decrement n by 1.

clear(): Set $n = 0$.

print(): We can print all elements by simply traversing `dense[]`.

Illustration:

Let there be a set with two elements {3, 5}, maximum value as 10 and capacity as 4. The set would be represented as below.

Initially:

```

maxVal = 10 // Size of sparse
capacity = 4 // Size of dense
n = 2        // Current number of elements in set

// dense[] Stores actual elements
dense[] = {3, 5, _, _}

// Uses actual elements as index and stores
// indexes of dense[]
sparse[] = {_, _, _, 0, _, 1, _, _, _, _}

'_' means it can be any value and not used in
sparse set

```

Insert 7:

```

n      = 3
dense[] = {3, 5, 7, _}
sparse[] = {_, _, _, 0, _, 1, _, 2, _, _}

```

```

Insert 4:
n      = 4
dense[] = {3, 5, 7, 4}
sparse[] = {_, _, _, 0, 3, 1, _, 2, _, _}

Delete 3:
n      = 3
dense[] = {4, 5, 7, _}
sparse[] = {_, _, _, _, 0, 1, _, 2, _, _}

Clear (Remove All):
n      = 0
dense[] = {_, _, _, _}
sparse[] = {_, _, _, _, _, _, _, _, _, _}

```

Below is C++ implementation of above functions.

```

/* A C program to implement Sparse Set and its operations */
#include<bits/stdc++.h>
using namespace std;

// A structure to hold the three parameters required to
// represent a sparse set.
class SSet
{
    int *sparse; // To store indexes of actual elements
    int *dense; // To store actual set elements
    int n; // Current number of elements
    int capacity; // Capacity of set or size of dense[]
    int maxValue; /* Maximum value in set or size of
                    sparse[] */

public:
    // Constructor
    SSet(int maxV, int cap)
    {
        sparse = new int[maxV+1];
        dense = new int[cap];
        capacity = cap;
        maxValue = maxV;
        n = 0; // No elements initially
    }

    // Destructor
    ~SSet()
    {
        delete[] sparse;
        delete[] dense;
    }
}

```

```
}

// If element is present, returns index of
// element in dense[]. Else returns -1.
int search(int x);

// Inserts a new element into set
void insert(int x);

// Deletes an element
void deletion(int x);

// Prints contents of set
void print();

// Removes all elements from set
void clear() { n = 0; }

// Finds intersection of this set with s
// and returns pointer to result.
SSet* intersection(SSet &s);

// A function to find union of two sets
// Time Complexity-O(n1+n2)
SSet *setUnion(SSet &s);
};

// If x is present in set, then returns index
// of it in dense[], else returns -1.
int SSet::search(int x)
{
    // Searched element must be in range
    if (x > maxValue)
        return -1;

    // The first condition verifies that 'x' is
    // within 'n' in this set and the second
    // condition tells us that it is present in
    // the data structure.
    if (sparse[x] < n && dense[sparse[x]] == x)
        return (sparse[x]);

    // Not found
    return -1;
}

// Inserts a new element into set
void SSet::insert(int x)
```

```

{
    // Corner cases, x must not be out of
    // range, dense[] should not be full and
    // x should not already be present
    if (x > maxValue)
        return;
    if (n >= capacity)
        return;
    if (search(x) != -1)
        return;

    // Inserting into array-dense[] at index 'n'.
    dense[n] = x;

    // Mapping it to sparse[] array.
    sparse[x] = n;

    // Increment count of elements in set
    n++;
}

// A function that deletes 'x' if present in this data
// structure, else it does nothing (just returns).
// By deleting 'x', we unset 'x' from this set.
void SSet::deletion(int x)
{
    // If x is not present
    if (search(x) == -1)
        return;

    int temp = dense[n-1]; // Take an element from end
    dense[sparse[x]] = temp; // Overwrite.
    sparse[temp] = sparse[x]; // Overwrite.

    // Since one element has been deleted, we
    // decrement 'n' by 1.
    n--;
}

// prints contents of set which are also content
// of dense[]
void SSet::print()
{
    for (int i=0; i<n; i++)
        printf("%d ", dense[i]);
    printf("\n");
}

```

```

// A function to find intersection of two sets
SSet* SSet::intersection(SSet &s)
{
    // Capacity and max value of result set
    int iCap      = min(n, s.n);
    int iMaxVal  = max(s.MaxValue, maxValue);

    // Create result set
    SSet *result = new SSet(iMaxVal, iCap);

    // Find the smaller of two sets
    // If this set is smaller
    if (n < s.n)
    {
        // Search every element of this set in 's'.
        // If found, add it to result
        for (int i = 0; i < n; i++)
            if (s.search(dense[i]) != -1)
                result->insert(dense[i]);
    }
    else
    {
        // Search every element of 's' in this set.
        // If found, add it to result
        for (int i = 0; i < s.n; i++)
            if (search(s.dense[i]) != -1)
                result->insert(s.dense[i]);
    }

    return result;
}

// A function to find union of two sets
// Time Complexity-O(n1+n2)
SSet* SSet::setUnion(SSet &s)
{
    // Find capacity and maximum value for result
    // set.
    int uCap      = s.n + n;
    int uMaxVal  = max(s.MaxValue, maxValue);

    // Create result set
    SSet *result = new SSet(uMaxVal, uCap);

    // Traverse the first set and insert all
    // elements of it in result.
    for (int i = 0; i < n; i++)
        result->insert(dense[i]);
}

```

```
// Traverse the second set and insert all
// elements of it in result (Note that sparse
// set doesn't insert an entry if it is already
// present)
for (int i = 0; i < s.n; i++)
    result->insert(s.dense[i]);

return result;
}

// Driver program
int main()
{
    // Create a set set1 with capacity 5 and max
    // value 100
    SSet s1(100, 5);

    // Insert elements into the set set1
    s1.insert(5);
    s1.insert(3);
    s1.insert(9);
    s1.insert(10);

    // Printing the elements in the data structure.
    printf("The elements in set1 are\n");
    s1.print();

    int index = s1.search(3);

    // 'index' variable stores the index of the number to
    // be searched.
    if (index != -1) // 3 exists
        printf("\n3 is found at index %d in set1\n",index);
    else // 3 doesn't exist
        printf("\n3 doesn't exists in set1\n");

    // Delete 9 and print set1
    s1.deletion(9);
    s1.print();

    // Create a set with capacity 6 and max value
    // 1000
    SSet s2(1000, 6);

    // Insert elements into the set
    s2.insert(4);
```

```

s2.insert(3);
s2.insert(7);
s2.insert(200);

// Printing set 2.
printf("\nThe elements in set2 are\n");
s2.print();

// Printing the intersection of the two sets
SSet *intersect = s2.intersection(s1);
printf("\nIntersection of set1 and set2\n");
intersect->print();

// Printing the union of the two sets
SSet *unionset = s1.setUnion(s2);
printf("\nUnion of set1 and set2\n");
unionset->print();

return 0;
}

```

Output :

```
The elements in set1 are
5 3 9 10
```

```
3 is found at index 1 in set1
5 3 10
```

```
The elements in set2 are-
4 3 7 200
```

```
Intersection of set1 and set2
3
```

```
Union of set1 and set2
5 3 10 4 7 200
```

Additional Operations:

The following are operations are also efficiently implemented using sparse set. It outperforms all the solutions discussed [here](#) and bit vector based solution, under the assumptions that range and maximum number of elements are known.

union():

- 1) Create an empty sparse set, say result.
- 2) Traverse the first set and insert all elements of it in result.
- 3) Traverse the second set and insert all elements of it in result (Note that sparse set doesn't

insert an entry if it is already present)

4) Return result.

intersection():

1) Create an empty sparse set, say result.

2) Let the smaller of two given sets be first set and larger be second.

3) Consider the smaller set and search every element of it in second. If element is found, add it to result.

4) Return result.

A common use of this data structure is with register allocation algorithms in compilers, which have a fixed universe(the number of registers in the machine) and are updated and cleared frequently (just like- Q queries) during a single processing run.

References:

<http://research.swtch.com/sparse>

<http://codingplayground.blogspot.in/2009/03/sparse-sets-with-o1-insert-delete.html>

This article is contributed by **Rachit Belwariar**. 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/sparse-set/>

Chapter 176

Sparse Table

Sparse Table - GeeksforGeeks

We have briefly discussed sparse table in [Range Minimum Query \(Square Root Decomposition and Sparse Table\)](#)

Sparse table concept is used for fast queries on a set of static data (elements do not change). It does preprocessing so that the queries can be answered efficiently.

Example Problem 1 : Range Minimum Query

We have an array $\text{arr}[0 \dots n-1]$. We need to efficiently find the minimum value from index L (query start) to R (query end) where $0 \leq L \leq R \leq n-1$. Consider a situation when there are many range queries.

Example:

```
Input: arr[] = {7, 2, 3, 0, 5, 10, 3, 12, 18};  
       query[] = [0, 4], [4, 7], [7, 8]
```

```
Output: Minimum of [0, 4] is 0  
        Minimum of [4, 7] is 3  
        Minimum of [7, 8] is 12
```

The idea is to precompute minimum of all subarrays of size 2^j where j varies from 0 to $\log n$. We make a table $\text{lookup}[i][j]$ such that $\text{lookup}[i][j]$ contains minimum of range starting from i and of size 2^j . For example $\text{lookup}[0][3]$ contains minimum of range $[0, 7]$ (starting with 0 and of size 2^3)

How to fill this lookup or sparse table?

The idea is simple, fill in bottom up manner using previously computed values. We compute

ranges with current power of 2 using values of lower power of two. For example, to find minimum of range [0, 7] (Range size is power of 3), we can use minimum of following two.

- Minimum of range [0, 3] (Range size is power of 2)
- Minimum of range [4, 7] (Range size is power of 2)

Based on above example, below is formula,

```
// Minimum of single element subarrays is same
// as the only element.
lookup[i][0] = arr[i]

// If lookup[0][2] <= lookup[4][2],
// then lookup[0][3] = lookup[0][2]
If lookup[i][j-1] <= lookup[i+2j-1-1][j-1]
    lookup[i][j] = lookup[i][j-1]

// If lookup[0][2] > lookup[4][2],
// then lookup[0][3] = lookup[4][2]
Else
    lookup[i][j] = lookup[i+2j-1-1][j-1]
```

7	2	3	0	5	10	3	12	18
0	1	2	3	4	5	6	7	8
arr[]								
j	0	1	2	3				
i	0	1	2	3				
	0	7	2	0	0			
	1	2	2	0	0	lookup[i][j] contains minimum		
	2	3	0	0	-	in range from arr[i] to		
	3	0	0	0	-	arr[i + 2^j - 1]		
	4	5	5	3	-			
	5	10	3	3	-			
	6	3	3	-	-			
	7	12	12	-	-			
	8	18	-	-	-			

For any arbitrary range $[l, R]$, we need to use ranges which are in powers of 2. The idea is to use closest power of 2. We always need to do at most one comparison (compare minimum of two ranges which are powers of 2). One range starts with L and ends with " $L + \text{closest-power-of-2}$ ". The other range ends at R and starts with " $R - \text{same-closest-power-of-2} + 1$ ". For example, if given range is (2, 10), we compare minimum of two ranges (2, 9) and (3, 10).

Based on above example, below is formula,

```

// For (2, 10), j = floor(Log2(10-2+1)) = 3
j = floor(Log(R-L+1))

// If lookup[0][3] <= lookup[3][3],
// then min(2, 10) = lookup[0][3]
If lookup[L][j] <= lookup[R-(int)pow(2, j)+1][j]
    min(L, R) = lookup[L][j]

// If lookup[0][3] > arr[lookup[3][3]],
// then min(2, 10) = lookup[3][3]
Else
    min(L, R) = lookup[i+2j-1-1][j-1]

```

Since we do only one comparison, time complexity of query is O(1).

Below is the implementation of above idea.

C++

```

// C++ program to do range minimum query
// using sparse table
#include <bits/stdc++.h>
using namespace std;
#define MAX 500

// lookup[i][j] is going to store minimum
// value in arr[i..j]. Ideally lookup table
// size should not be fixed and should be
// determined using n Log n. It is kept
// constant to keep code simple.
int lookup[MAX][MAX];

// Fills lookup array lookup[][] in bottom up manner.
void buildSparseTable(int arr[], int n)
{
    // Initialize M for the intervals with length 1
    for (int i = 0; i < n; i++)
        lookup[i][0] = arr[i];

    // Compute values from smaller to bigger intervals
    for (int j = 1; (1 << j) <= n; j++) {

        // Compute minimum value for all intervals with
        // size 2^j
        for (int i = 0; (i + (1 << j) - 1) < n; i++) {

            // For arr[2][10], we compare arr[lookup[0][7]]
            // and arr[lookup[3][10]]

```

```

        if (lookup[i][j - 1] <
            lookup[i + (1 << (j - 1))][j - 1])
            lookup[i][j] = lookup[i][j - 1];
        else
            lookup[i][j] =
                lookup[i + (1 << (j - 1))][j - 1];
    }
}
}

// Returns minimum of arr[L..R]
int query(int L, int R)
{
    // Find highest power of 2 that is smaller
    // than or equal to count of elements in given
    // range. For [2, 10], j = 3
    int j = (int)log2(R - L + 1);

    // Compute minimum of last 2^j elements with first
    // 2^j elements in range.
    // For [2, 10], we compare arr[lookup[0][3]] and
    // arr[lookup[3][3]],
    if (lookup[L][j] <= lookup[R - (1 << j) + 1][j])
        return lookup[L][j];

    else
        return lookup[R - (1 << j) + 1][j];
}

// Driver program
int main()
{
    int a[] = { 7, 2, 3, 0, 5, 10, 3, 12, 18 };
    int n = sizeof(a) / sizeof(a[0]);
    buildSparseTable(a, n);
    cout << query(0, 4) << endl;
    cout << query(4, 7) << endl;
    cout << query(7, 8) << endl;
    return 0;
}

```

Java

```

// Java program to do range minimum query
// using sparse table
import java.io.*;

class GFG {

```

```

static int MAX =500;

// lookup[i][j] is going to store minimum
// value in arr[i..j]. Ideally lookup table
// size should not be fixed and should be
// determined using n Log n. It is kept
// constant to keep code simple.
static int [][]lookup = new int[MAX][MAX];

// Fills lookup array lookup[][] in bottom up manner.
static void buildSparseTable(int arr[], int n)
{

    // Initialize M for the intervals with length 1
    for (int i = 0; i < n; i++)
        lookup[i][0] = arr[i];

    // Compute values from smaller to bigger intervals
    for (int j = 1; (1 << j) <= n; j++) {

        // Compute minimum value for all intervals with
        // size 2^j
        for (int i = 0; (i + (1 << j) - 1) < n; i++) {

            // For arr[2][10], we compare arr[lookup[0][7]]
            // and arr[lookup[3][10]]
            if (lookup[i][j - 1] <
                lookup[i + (1 << (j - 1))][j - 1])
                lookup[i][j] = lookup[i][j - 1];
            else
                lookup[i][j] =
                    lookup[i + (1 << (j - 1))][j - 1];
        }
    }
}

// Returns minimum of arr[L..R]
static int query(int L, int R)
{

    // Find highest power of 2 that is smaller
    // than or equal to count of elements in given
    // range. For [2, 10], j = 3
    int j = (int)Math.log(R - L + 1);

    // Compute minimum of last 2^j elements with first
    // 2^j elements in range.
}

```

```

// For [2, 10], we compare arr[lookup[0][3]] and
// arr[lookup[3][3]],
if (lookup[L][j] <= lookup[R - (1 << j) + 1][j])
    return lookup[L][j];

else
    return lookup[R - (1 << j) + 1][j];
}

// Driver program
public static void main (String[] args)
{
    int a[] = { 7, 2, 3, 0, 5, 10, 3, 12, 18 };
    int n = a.length;

    buildSparseTable(a, n);

    System.out.println(query(0, 4));
    System.out.println(query(4, 7));
    System.out.println(query(7, 8));

}
}

//This code is contributed by vt_m.

```

C#

```

// C# program to do range minimum query
// using sparse table
using System;

public class GFG {

    static int MAX= 500;

    // lookup[i][j] is going to store minimum
    // value in arr[i..j]. Ideally lookup table
    // size should not be fixed and should be
    // determined using n Log n. It is kept
    // constant to keep code simple.
    static int [,]lookup = new int[MAX, MAX];

    // Fills lookup array lookup[][] in bottom up manner.
    static void buildSparseTable(int []arr, int n)
    {
        // Initialize M for the intervals with length 1
        for (int i = 0; i < n; i++)

```

```

        lookup[i, 0] = arr[i];

    // Compute values from smaller to bigger intervals
    for (int j = 1; (1 << j) <= n; j++) {

        // Compute minimum value for all intervals with
        // size 2^j
        for (int i = 0; (i + (1 << j) - 1) < n; i++) {

            // For arr[2][10], we compare arr[lookup[0][7]]
            // and arr[lookup[3][10]]
            if (lookup[i, j - 1] <
                lookup[i + (1 << (j - 1)), j - 1])
                lookup[i, j] = lookup[i, j - 1];
            else
                lookup[i, j] =
                    lookup[i + (1 << (j - 1)), j - 1];
        }
    }

    // Returns minimum of arr[L..R]
    static int query(int L, int R)
    {

        // Find highest power of 2 that is smaller
        // than or equal to count of elements in given
        // range. For [2, 10], j = 3
        int j = (int)Math.Log(R - L + 1);

        // Compute minimum of last 2^j elements with first
        // 2^j elements in range.
        // For [2, 10], we compare arr[lookup[0][3]] and
        // arr[lookup[3][3]],
        if (lookup[L, j] <= lookup[R - (1 << j) + 1, j])
            return lookup[L, j];

        else
            return lookup[R - (1 << j) + 1, j];
    }

    // Driver program
    static public void Main ()
    {
        int []a = { 7, 2, 3, 0, 5, 10, 3, 12, 18 };
        int n = a.Length;

        buildSparseTable(a, n);
    }
}

```

```
        Console.WriteLine(query(0, 4));
        Console.WriteLine(query(4, 7));
        Console.WriteLine(query(7, 8));
    }
}

// This code is contributed by vt_m.
```

Output:

```
Minimum of [0, 4] is 0
Minimum of [4, 7] is 3
Minimum of [7, 8] is 12
```

So sparse table method supports query operation in $O(1)$ time with $O(n \log n)$ preprocessing time and $O(n \log n)$ space.

Example Problem 2 : Range GCD Query

We have an array $\text{arr}[0 \dots n-1]$. We need to find the [greatest common divisor](#) in the range L and R where $0 \leq L \leq R \leq n-1$. Consider a situation when there are many range queries

Examples:

```
Input : arr[] = {2, 3, 5, 4, 6, 8}
        queries[] = {(0, 2), (3, 5), (2, 3)}
Output : 1
        2
        1
```

We use below properties of GCD:

- GCD function is associative [$\text{GCD}(a, b, c) = \text{GCD}(\text{GCD}(a, b), c) = \text{GCD}(a, \text{GCD}(b, c))$], we can compute GCD of a range using GCDs of subranges.
- If we take GCD of an overlapping range more than once, then it does not change answer. For example $\text{GCD}(a, b, c) = \text{GCD}(\text{GCD}(a, b), \text{GCD}(b, c))$. Therefore like minimum range query problem, we need to do only one comparison to find GCD of given range.

We build sparse table using same logic as above. After building sparse table, we can find all GCDs by breaking given range in powers of 2 and add GCD of every piece to current answer.

```

// C++ program to do range minimum query
// using sparse table
#include <bits/stdc++.h>
using namespace std;
#define MAX 500

// lookup[i][j] is going to store GCD of
// arr[i..j]. Ideally lookup table
// size should not be fixed and should be
// determined using n Log n. It is kept
// constant to keep code simple.
int table[MAX][MAX];

// it builds sparse table.
void buildSparseTable(int arr[], int n)
{
    // GCD of single element is element itself
    for (int i = 0; i < n; i++)
        table[i][0] = arr[i];

    // Build sparse table
    for (int j = 1; j <= k; j++)
        for (int i = 0; i <= n - (1 << j); i++)
            table[i][j] = __gcd(table[i][j - 1],
                                table[i + (1 << (j - 1))][j - 1]);
}

// Returns GCD of arr[L..R]
int query(int L, int R)
{
    // Find highest power of 2 that is smaller
    // than or equal to count of elements in given
    // range. For [2, 10], j = 3
    int j = (int)log2(R - L + 1);

    // Compute GCD of last 2^j elements with first
    // 2^j elements in range.
    // For [2, 10], we find GCD of arr[lookup[0][3]] and
    // arr[lookup[3][3]],
    return __gcd(table[L][j], table[R - (1 << j) + 1][j]);
}

// Driver program
int main()
{
    int a[] = { 7, 2, 3, 0, 5, 10, 3, 12, 18 };
    int n = sizeof(a) / sizeof(a[0]);
    buildSparseTable(a, n);
}

```

```
cout << query(0, 2) << endl;
cout << query(1, 3) << endl;
cout << query(4, 5) << endl;
return 0;
}
```

Output:

```
1
1
5
```

Improved By : [kaushalkumarroy](#)

Source

<https://www.geeksforgeeks.org/sparse-table/>

Chapter 177

Splay Tree Set 1 (Search)

Splay Tree Set 1 (Search) - GeeksforGeeks

The worst case time complexity of Binary Search Tree (BST) operations like search, delete, insert is $O(n)$. The worst case occurs when the tree is skewed. We can get the worst case time complexity as $O(\log n)$ with [AVL](#) and Red-Black Trees.

Can we do better than AVL or Red-Black trees in practical situations?

Like [AVL](#) and Red-Black Trees, Splay tree is also [self-balancing BST](#). The main idea of splay tree is to bring the recently accessed item to root of the tree, this makes the recently searched item to be accessible in $O(1)$ time if accessed again. The idea is to use locality of reference (In a typical application, 80% of the access are to 20% of the items). Imagine a situation where we have millions or billions of keys and only few of them are accessed frequently, which is very likely in many practical applications.

All splay tree operations run in $O(\log n)$ time on average, where n is the number of entries in the tree. Any single operation can take $\Theta(n)$ time in the worst case.

Search Operation

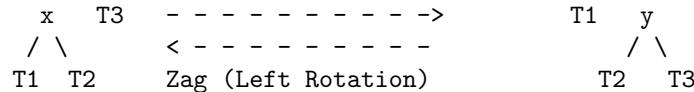
The search operation in Splay tree does the standard BST search, in addition to search, it also splays (move a node to the root). If the search is successful, then the node that is found is splayed and becomes the new root. Else the last node accessed prior to reaching the NULL is splayed and becomes the new root.

There are following cases for the node being accessed.

1) Node is root We simply return the root, don't do anything else as the accessed node is already root.

2) Zig: Node is child of root (the node has no grandparent). Node is either a left child of root (we do a right rotation) or node is a right child of its parent (we do a left rotation). T1, T2 and T3 are subtrees of the tree rooted with y (on left side) or x (on right side)

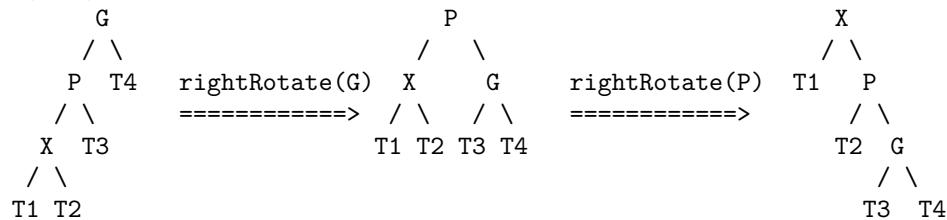




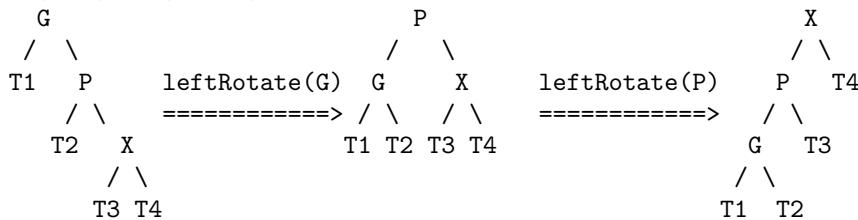
3) **Node has both parent and grandparent.** There can be following subcases.

.....3.a) **Zig-Zig and Zag-Zag** Node is left child of parent and parent is also left child of grand parent (Two right rotations) OR node is right child of its parent and parent is also right child of grand parent (Two Left Rotations).

Zig-Zig (Left Left Case):

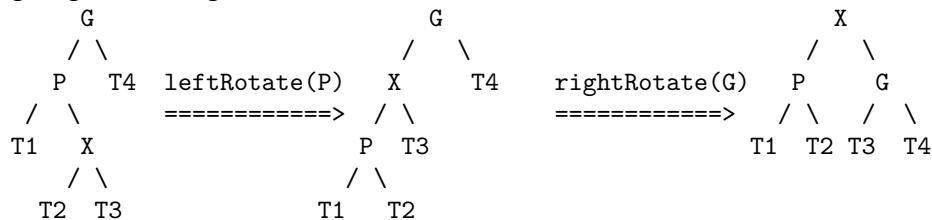


Zag-Zag (Right Right Case):

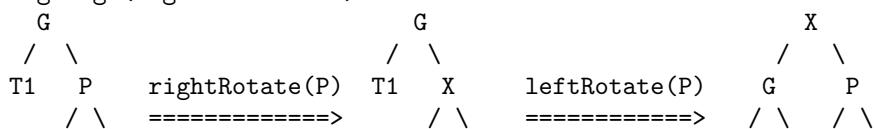


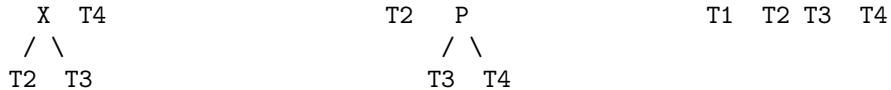
.....3.b) **Zig-Zag and Zag-Zig** Node is left child of parent and parent is right child of grand parent (Left Rotation followed by right rotation) OR node is right child of its parent and parent is left child of grand parent (Right Rotation followed by left rotation).

Zag-Zig (Left Right Case):

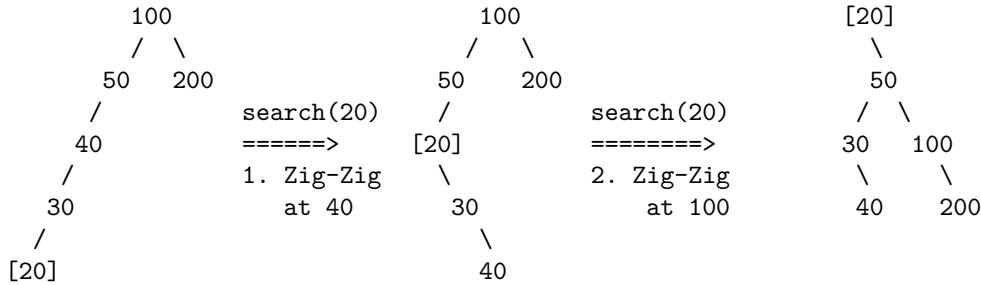


Zig-Zag (Right Left Case):





Example:



The important thing to note is, the search or splay operation not only brings the searched key to root, but also balances the BST. For example in above case, height of BST is reduced by 1.

Implementation:

```

// The code is adopted from http://goo.gl/SDH9hH
#include<stdio.h>
#include<stdlib.h>

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

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

// A utility function to right rotate subtree rooted with y
// See the diagram given above.
struct node *rightRotate(struct node *x)
{
    ...
}
    
```

```

    struct node *y = x->left;
    x->left = y->right;
    y->right = x;
    return y;
}

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

// This function brings the key at root if key is present in tree.
// If key is not present, then it brings the last accessed item at
// root. This function modifies the tree and returns the new root
struct node *splay(struct node *root, int key)
{
    // Base cases: root is NULL or key is present at root
    if (root == NULL || root->key == key)
        return root;

    // Key lies in left subtree
    if (root->key > key)
    {
        // Key is not in tree, we are done
        if (root->left == NULL) return root;

        // Zig-Zig (Left Left)
        if (root->left->key > key)
        {
            // First recursively bring the key as root of left-left
            root->left->left = splay(root->left->left, key);

            // Do first rotation for root, second rotation is done after else
            root = rightRotate(root);
        }
        else if (root->left->key < key) // Zig-Zag (Left Right)
        {
            // First recursively bring the key as root of left-right
            root->left->right = splay(root->left->right, key);

            // Do first rotation for root->left
            if (root->left->right != NULL)
                root->left = leftRotate(root->left);
        }
    }
}

```

```

    }

    // Do second rotation for root
    return (root->left == NULL)? root: rightRotate(root);
}

else // Key lies in right subtree
{
    // Key is not in tree, we are done
    if (root->right == NULL) return root;

    // Zag-Zig (Right Left)
    if (root->right->key > key)
    {
        // Bring the key as root of right-left
        root->right->left = splay(root->right->left, key);

        // Do first rotation for root->right
        if (root->right->left != NULL)
            root->right = rightRotate(root->right);
    }
    else if (root->right->key < key)// Zag-Zag (Right Right)
    {
        // Bring the key as root of right-right and do first rotation
        root->right->right = splay(root->right->right, key);
        root = leftRotate(root);
    }

    // Do second rotation for root
    return (root->right == NULL)? root: leftRotate(root);
}
}

// The search function for Splay tree. Note that this function
// returns the new root of Splay Tree. If key is present in tree
// then, it is moved to root.
struct node *search(struct node *root, int key)
{
    return splay(root, key);
}

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

```

```
        preOrder(root->right);
    }
}

/* Drier program to test above function*/
int main()
{
    struct node *root = newNode(100);
    root->left = newNode(50);
    root->right = newNode(200);
    root->left->left = newNode(40);
    root->left->left->left = newNode(30);
    root->left->left->left->left = newNode(20);

    root = search(root, 20);
    printf("Preorder traversal of the modified Splay tree is \n");
    preOrder(root);
    return 0;
}
```

Output:

```
Preorder traversal of the modified Splay tree is
20 50 30 40 100 200
```

Summary

- 1) Splay trees have excellent locality properties. Frequently accessed items are easy to find. Infrequent items are out of way.
- 2) All splay tree operations take $O(\log n)$ time on average. Splay trees can be rigorously shown to run in $O(\log n)$ average time per operation, over any sequence of operations (assuming we start from an empty tree)
- 3) Splay trees are simpler compared to [AVL](#) and Red-Black Trees as no extra field is required in every tree node.
- 4) Unlike [AVL tree](#), a splay tree can change even with read-only operations like search.

Applications of Splay Trees

Splay trees have become the most widely used basic data structure invented in the last 30 years, because they're the fastest type of balanced search tree for many applications. Splay trees are used in Windows NT (in the virtual memory, networking, and file system code), the gcc compiler and GNU C++ library, the sed string editor, Fore Systems network routers, the most popular implementation of Unix malloc, Linux loadable kernel modules, and in much other software (Source: <http://www.cs.berkeley.edu/~jrs/61b/lec/36>)

See [Splay Tree Set 2 \(Insert\)](#) for splay tree insertion.

References:

<http://www.cs.berkeley.edu/~jrs/61b/lec/36>

<http://www.cs.cornell.edu/courses/cs3110/2009fa/recitations/rec-splay.html>
<http://courses.cs.washington.edu/courses/cse326/01au/lectures/SplayTrees.ppt>

Improved By : [Ujjwal Mishra](#)

Source

<https://www.geeksforgeeks.org/splay-tree-set-1-insert/>

Chapter 178

Splay Tree Set 2 (Insert)

Splay Tree Set 2 (Insert) - GeeksforGeeks

It is recommended to refer following post as prerequisite of this post.

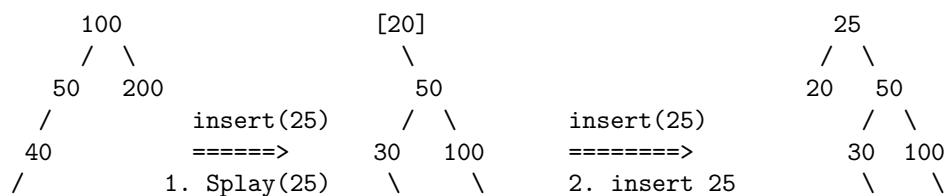
[Splay Tree Set 1 \(Search\)](#)

As discussed in the [previous post](#), Splay tree is a self-balancing data structure where the last accessed key is always at root. The insert operation is similar to Binary Search Tree insert with additional steps to make sure that the newly inserted key becomes the new root.

Following are different cases to insert a key k in splay tree.

- 1) Root is NULL: We simply allocate a new node and return it as root.
- 2) Splay the given key k. If k is already present, then it becomes the new root. If not present, then last accessed leaf node becomes the new root.
- 3) If new root's key is same as k, don't do anything as k is already present.
- 4) Else allocate memory for new node and compare root's key with k.
 -4.a) If k is smaller than root's key, make root as right child of new node, copy left child of root as left child of new node and make left child of root as NULL.
 -4.b) If k is greater than root's key, make root as left child of new node, copy right child of root as right child of new node and make right child of root as NULL.
- 5) Return new node as new root of tree.

Example:



```
30
/
[20]
```

```
40    200
```

```
40    200
```

```
// This code is adopted from http://algs4.cs.princeton.edu/33balanced/SplayBST.java.html
#include<stdio.h>
#include<stdlib.h>

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

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

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

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

// This function brings the key at root if key is present in tree.
// If key is not present, then it brings the last accessed item at
// root. This function modifies the tree and returns the new root
```

```

struct node *splay(struct node *root, int key)
{
    // Base cases: root is NULL or key is present at root
    if (root == NULL || root->key == key)
        return root;

    // Key lies in left subtree
    if (root->key > key)
    {
        // Key is not in tree, we are done
        if (root->left == NULL) return root;

        // Zig-Zig (Left Left)
        if (root->left->key > key)
        {
            // First recursively bring the key as root of left-left
            root->left->left = splay(root->left->left, key);

            // Do first rotation for root, second rotation is done after else
            root = rightRotate(root);
        }
        else if (root->left->key < key) // Zig-Zag (Left Right)
        {
            // First recursively bring the key as root of left-right
            root->left->right = splay(root->left->right, key);

            // Do first rotation for root->left
            if (root->left->right != NULL)
                root->left = leftRotate(root->left);
        }
    }

    // Do second rotation for root
    return (root->left == NULL)? root: rightRotate(root);
}

else // Key lies in right subtree
{
    // Key is not in tree, we are done
    if (root->right == NULL) return root;

    // Zig-Zag (Right Left)
    if (root->right->key > key)
    {
        // Bring the key as root of right-left
        root->right->left = splay(root->right->left, key);

        // Do first rotation for root->right
        if (root->right->left != NULL)
            root->right = rightRotate(root->right);
    }
}

```

```

    }

    else if (root->right->key < key)// Zag-Zag (Right Right)
    {
        // Bring the key as root of right-right and do first rotation
        root->right->right = splay(root->right->right, key);
        root = leftRotate(root);
    }

    // Do second rotation for root
    return (root->right == NULL)? root: leftRotate(root);
}

// Function to insert a new key k in splay tree with given root
struct node *insert(struct node *root, int k)
{
    // Simple Case: If tree is empty
    if (root == NULL) return newNode(k);

    // Bring the closest leaf node to root
    root = splay(root, k);

    // If key is already present, then return
    if (root->key == k) return root;

    // Otherwise allocate memory for new node
    struct node *newnode = newNode(k);

    // If root's key is greater, make root as right child
    // of newnode and copy the left child of root to newnode
    if (root->key > k)
    {
        newnode->right = root;
        newnode->left = root->left;
        root->left = NULL;
    }

    // If root's key is smaller, make root as left child
    // of newnode and copy the right child of root to newnode
    else
    {
        newnode->left = root;
        newnode->right = root->right;
        root->right = NULL;
    }

    return newnode; // newnode becomes new root
}

```

```
// A utility function to print preorder traversal of the tree.  
// The function also prints height of every node  
void preOrder(struct node *root)  
{  
    if (root != NULL)  
    {  
        printf("%d ", root->key);  
        preOrder(root->left);  
        preOrder(root->right);  
    }  
}  
  
/* Drier program to test above function*/  
int main()  
{  
    struct node *root = newNode(100);  
    root->left = newNode(50);  
    root->right = newNode(200);  
    root->left->left = newNode(40);  
    root->left->left->left = newNode(30);  
    root->left->left->left->left = newNode(20);  
    root = insert(root, 25);  
    printf("Preorder traversal of the modified Splay tree is \n");  
    preOrder(root);  
    return 0;  
}
```

Output:

```
Preorder traversal of the modified Splay tree is  
25 20 50 30 40 100 200
```

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

Source

<https://www.geeksforgeeks.org/splay-tree-set-2-insert-delete/>

Chapter 179

Splay Tree Set 3 (Delete)

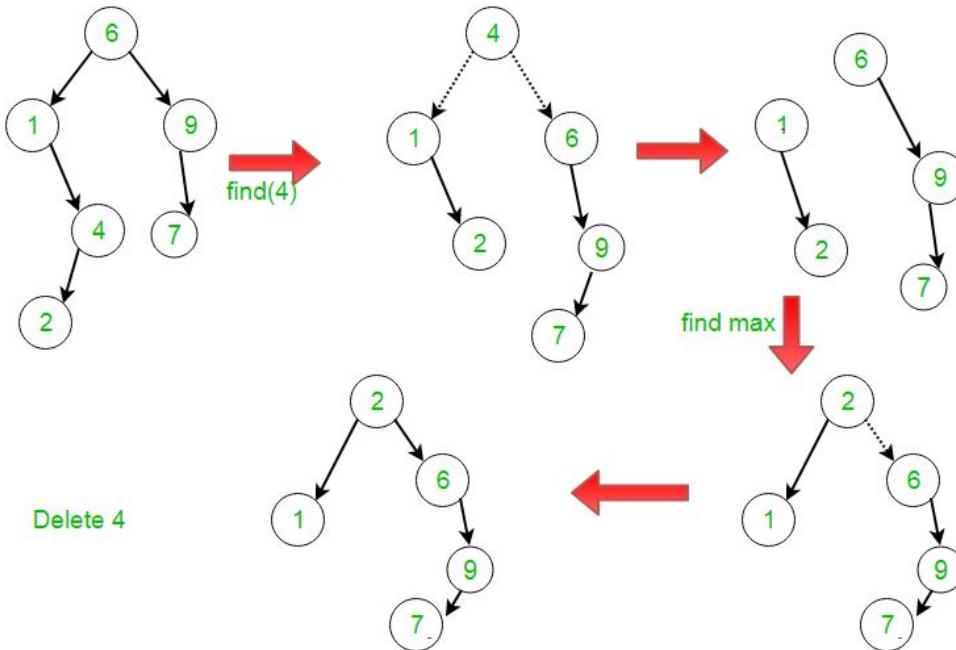
Splay Tree Set 3 (Delete) - GeeksforGeeks

It is recommended to refer following post as prerequisite of this post.

[Splay Tree Set 1 \(Search\)](#)

Following are the different cases to delete a key **k** from splay tree.

1. If **Root** is **NULL**: We simply return the root.
2. Else [Splay](#) the given key **k**. If **k** is present, then it becomes the new root. If not present, then last accessed leaf node becomes the new root.
3. If new root's key is not same as **k**, then return the root as **k** is not present.
4. Else the key **k** is present.
 - Split the tree into two trees **Tree1** = root's left subtree and **Tree2** = root's right subtree and delete the root node.
 - Let the root's of **Tree1** and **Tree2** be **Root1** and **Root2** respectively.
 - If **Root1** is **NULL**: Return **Root2**.
 - Else, Splay the maximum node (node having the maximum value) of **Tree1**.
 - After the Splay procedure, make **Root2** as the right child of **Root1** and return **Root1**.



```

// C implementation to delete a node from Splay Tree
#include<stdio.h>
#include<stdlib.h>

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

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

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

```

```

x->left = y->right;
y->right = x;
return y;
}

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

// This function brings the key at root if key is present in tree.
// If key is not present, then it brings the last accessed item at
// root. This function modifies the tree and returns the new root
struct node *splay(struct node *root, int key)
{
    // Base cases: root is NULL or key is present at root
    if (root == NULL || root->key == key)
        return root;

    // Key lies in left subtree
    if (root->key > key)
    {
        // Key is not in tree, we are done
        if (root->left == NULL) return root;

        // Zig-Zig (Left Left)
        if (root->left->key > key)
        {
            // First recursively bring the key as root of left-left
            root->left->left = splay(root->left->left, key);

            // Do first rotation for root, second rotation is
            // done after else
            root = rightRotate(root);
        }
        else if (root->left->key < key) // Zig-Zag (Left Right)
        {
            // First recursively bring the key as root of left-right
            root->left->right = splay(root->left->right, key);

            // Do first rotation for root->left
            if (root->left->right != NULL)
                root->left = leftRotate(root->left);
        }
    }
}

```

```

        }

        // Do second rotation for root
        return (root->left == NULL)? root: rightRotate(root);
    }
    else // Key lies in right subtree
    {
        // Key is not in tree, we are done
        if (root->right == NULL) return root;

        // Zag-Zig (Right Left)
        if (root->right->key > key)
        {
            // Bring the key as root of right-left
            root->right->left = splay(root->right->left, key);

            // Do first rotation for root->right
            if (root->right->left != NULL)
                root->right = rightRotate(root->right);
        }
        else if (root->right->key < key)// Zag-Zag (Right Right)
        {
            // Bring the key as root of right-right and do
            // first rotation
            root->right->right = splay(root->right->right, key);
            root = leftRotate(root);
        }

        // Do second rotation for root
        return (root->right == NULL)? root: leftRotate(root);
    }
}

// The delete function for Splay tree. Note that this function
// returns the new root of Splay Tree after removing the key
struct node* delete_key(struct node *root, int key)
{
    struct node *temp;
    if (!root)
        return NULL;

    // Splay the given key
    root = splay(root, key);

    // If key is not present, then
    // return root
    if (key != root->key)
        return root;
}

```

```
// If key is present
// If left child of root does not exist
// make root->right as root
if (!root->left)
{
    temp = root;
    root = root->right;
}

// Else if left child exists
else
{
    temp = root;

    /*Note: Since key == root->key,
    so after Splay(key, root->lchild),
    the tree we get will have no right child tree
    and maximum node in left subtree will get splayed*/
    // New root
    root = splay(root->left, key);

    // Make right child of previous root as
    // new root's right child
    root->right = temp->right;
}

// free the previous root node, that is,
// the node containing the key
free(temp);

// return root of the new Splay Tree
return root;

}

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

```
/* Drier program to test above function*/
int main()
{
    // Splay Tree Formation
    struct node *root = newNode(6);
    root->left = newNode(1);
    root->right = newNode(9);
    root->left->right = newNode(4);
    root->left->right->left = newNode(2);
    root->right->left = newNode(7);

    int key = 4;

    root = delete_key(root, key);
    printf("Preorder traversal of the modified Splay tree is \n");
    preOrder(root);
    return 0;
}
```

Output:

```
Preorder traversal of the modified Splay tree is
2 1 6 9 7
```

References:

<https://www.geeksforgeeks.org/splay-tree-set-1-insert/>
<http://courses.cs.washington.edu/courses/cse326/01au/lectures/SplayTrees.ppt>

Source

<https://www.geeksforgeeks.org/splay-tree-set-3-delete/>

Chapter 180

Sqrt (or Square Root) Decomposition Set 2 (LCA of Tree in O(sqrt(height)) time)

Sqrt (or Square Root) Decomposition Set 2 (LCA of Tree in O(sqrt(height)) time) - Geeks-forGeeks

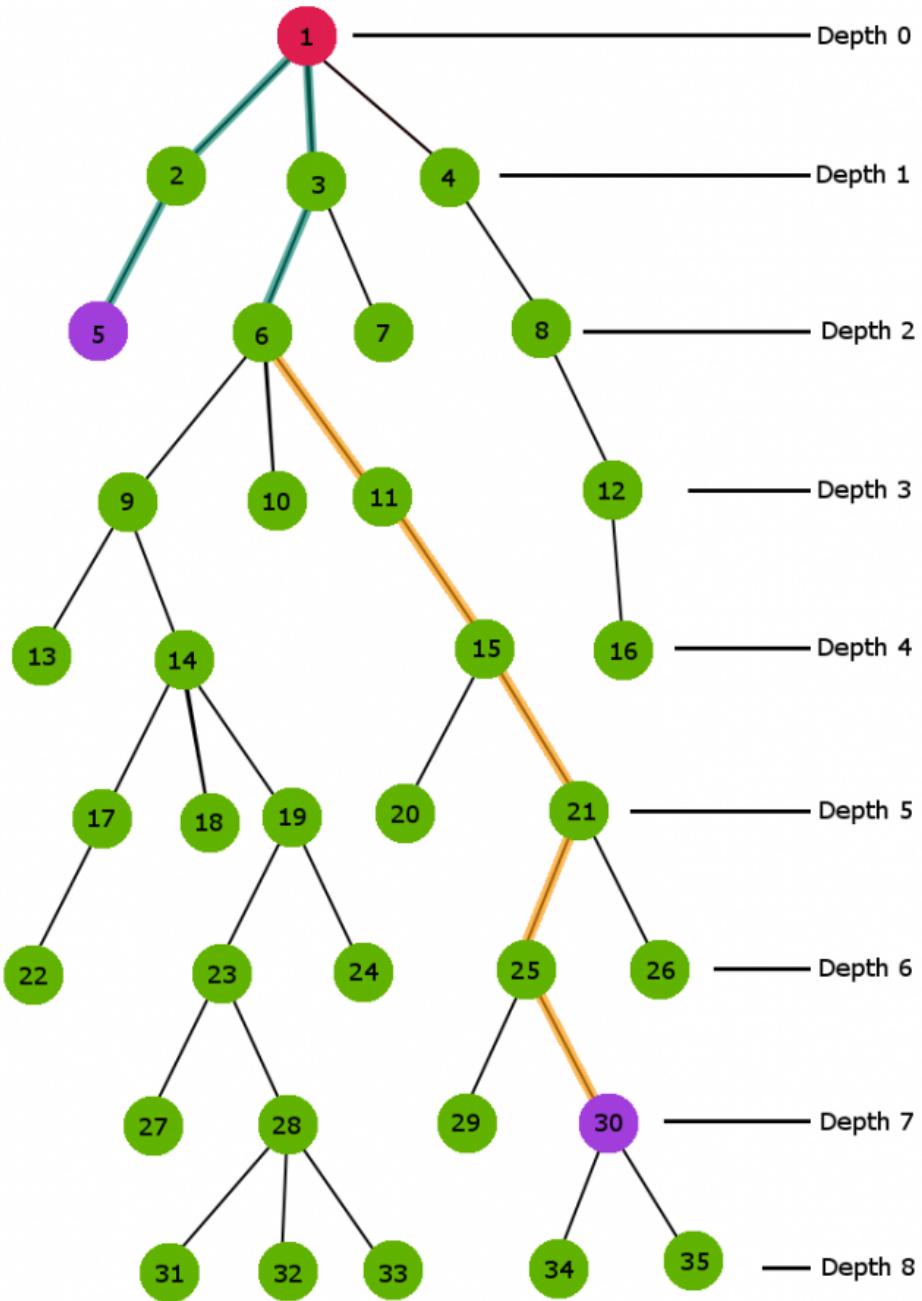
Prerequisite : [Introduction](#) and [DFS](#)

The task is to find LCA of two given nodes in a tree (not necessarily a Binary Tree). In previous posts, we have seen how to calculate [LCA using Sparse Matrix DP approach](#). In this post, we will see an optimization done on Naive method by sqrt decomposition technique that works well over the Naive Approach.

Naive Approach

To calculate the LCA of two nodes first of all we will bring both the nodes to same height by making the node with greater depth jump one parent up the tree till both the nodes are at same height. Once, both the nodes are at same height we can then start jumping one parent up for both the nodes simultaneously till both the nodes become equal and that node will be the LCA of the two originally given nodes.

Consider the below n-ary Tree with depth 9 and lets examine how naive approach works for this sample tree.



Here in the above Tree we need to calculate the LCA of node 6 and node 30

Clearly node 30 has greater depth than node 6. So first of all we start jumping one parent above for node 30 till we reach the depth value of node 6 i.e at depth 2.

The **orange colored path** in the above figure demonstrates the jumping sequence to reach the depth 2. In this procedure we just simply jump one parent above the current node.

Now both nodes are at same depth 2. Therefore, now both the nodes will jump one parent up till both the nodes become equal. This end node at which both the nodes become equal for the first time is our LCA.

The **blue color path** in the above figure shows the jumping route for both the nodes

C++ code for the above implementation:-

```
// Naive C++ implementation to find LCA in a tree
#include <iostream>
#include <vector>
#include <math.h>
using namespace std;
#define MAXN 1001

int depth[MAXN];           // stores depth for each node
int parent[MAXN];          // stores first parent for each node

vector < int > adj[MAXN];

void addEdge(int u,int v)
{
    adj[u].push_back(v);
    adj[v].push_back(u);
}

void dfs(int cur, int prev)
{
    // marking parent for each node
    parent[cur] = prev;

    // marking depth for each node
    depth[cur] = depth[prev] + 1;

    // propogating marking down the tree
    for (int i=0; i<adj[cur].size(); i++)
        if (adj[cur][i] != prev)
            dfs(adj[cur][i],cur);
}

void preprocess()
{
    // a dummy node
```

```
depth[0] = -1;

// precalclating 1)depth. 2)parent.
// for each node
dfs(1,0);
}

// Time Complexity : O(Height of tree)
// recursively jumps one node above
// till both the nodes become equal
int LCANaive(int u,int v)
{
    if (u == v)  return u;
    if (depth[u] > depth[v])
        swap(u, v);
    v = parent[v];
    return LCANaive(u,v);
}

// Driver function to call the above functions
int main(int argc, char const *argv[])
{
    // adding edges to the tree
    addEdge(1,2);
    addEdge(1,3);
    addEdge(1,4);
    addEdge(2,5);
    addEdge(2,6);
    addEdge(3,7);
    addEdge(4,8);
    addEdge(4,9);
    addEdge(9,10);
    addEdge(9,11);
    addEdge(7,12);
    addEdge(7,13);

    preprocess();

    cout << "LCA(11,8) : " << LCANaive(11,8) << endl;
    cout << "LCA(3,13) : " << LCANaive(3,13) << endl;

    return 0;
}
```

Output:

```
LCA(11,8) : 4
```

LCA(3,13) : 3

Time Complexity : We pre-calculate the depth for each node using one **DFS traversal in $O(n)$** . Now in worst case, the two nodes will be two bottom most node on the tree in different child branches of the root node. Therefore, in this case the root will be the LCA of both the nodes. Hence, both the nodes will have to jump exactly h height above, where h is the height of the tree. So, to answer each **LCA query Time Complexity will be $O(h)$** .

The Sqrt Decomposition Trick :

We categorize nodes of the tree into different groups according to their depth. Assuming the depth of the tree h is a perfect square. So once again like the [general sqrt decomposition approach](#) we will be having \sqrt{h} blocks or groups. Nodes from depth 0 to depth $\sqrt{h} - 1$ lie in first group; then nodes having depth \sqrt{h} to $2\sqrt{h} - 1$ lie in second group and so on till last node.

We keep track of the corresponding group number for every node and also depth of every node. This can be done by one single dfs on the tree (see the code for better understanding).

Sqrt trick :- In naive approach we were jumping one parent up the tree till both nodes aren't on the same depth. But here we perform group wise jump. To perform this group wise jump, we need two parameter associated with each node : 1) parent and 2) jump parent. Here **parent** for each node is defined as the first node above the current node that is directly connected to it, whereas **jump_parent** for each node is the node that is the first ancestor of the current node in the group just above the current node.

So, now we need to maintain 3 parameters for each node :

- 1) **depth**
- 2) **parent**
- 3) **jump_parent**

All these three parameters can be maintained in one dfs(refer to the code for better understanding)

Pseudo code for optimization process

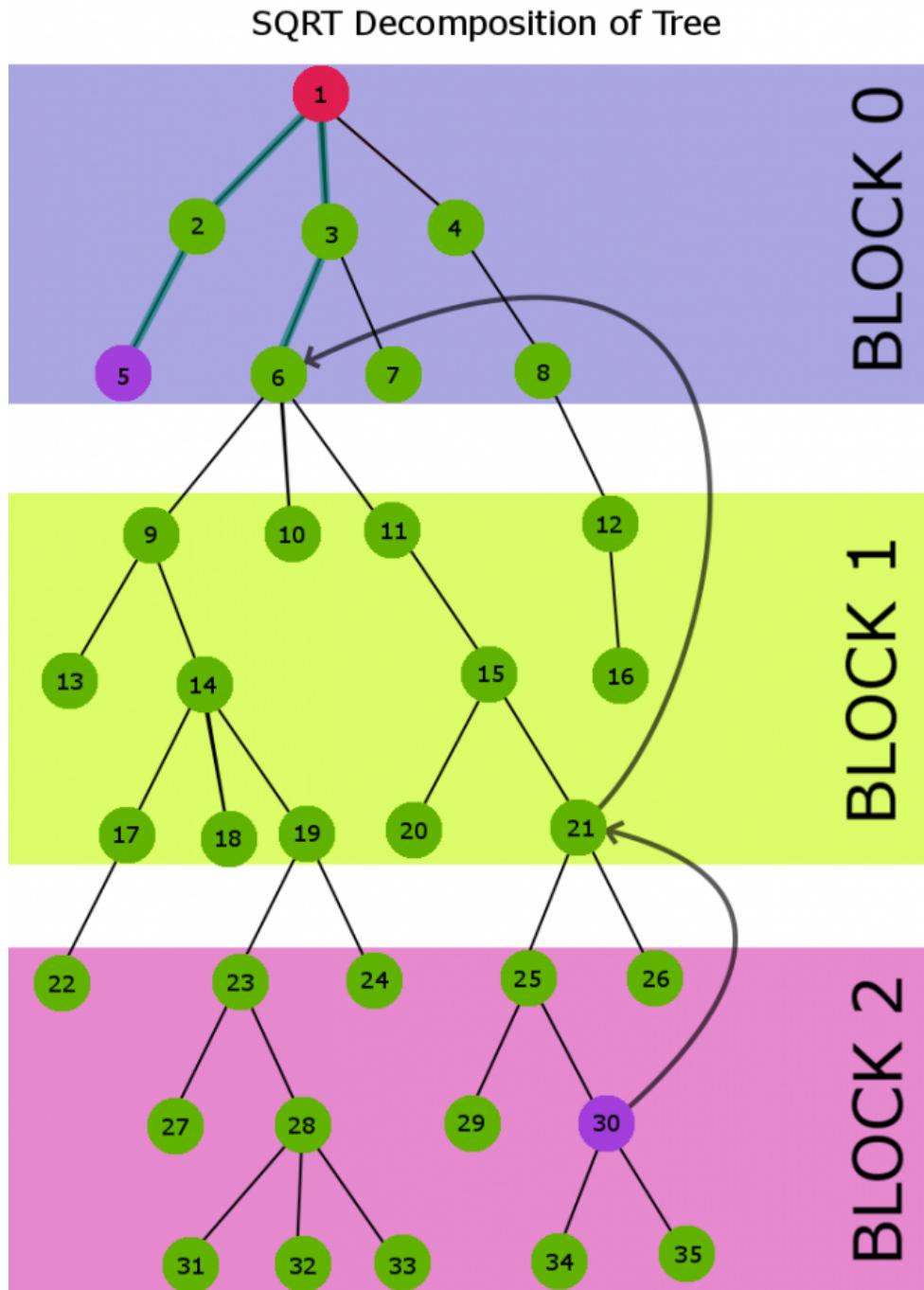
```
LCAsqrt(u, v){  
  
    // assuming v is at greater depth  
    while (jump_parent[u] != jump_parent[v]) {  
        v = jump_parent[v];  
    }  
  
    // now both nodes are in same group  
    // and have same jump_parent  
    return LCAnaive(u,v);  
}
```

The key concept here is that first we bring both the nodes in same group and having same **jump_parent** by climbing decomposed blocks above the tree one by one and then when both

the nodes are in same group and have same jump_parent we use our naive approach to find LCA of the nodes.

This optimized group jumping technique reduces the iterating space by a factor of **sqrt(h)** and hence reduces the Time Complexity(refer below for better time complexity analysis)

Lets decompose the above tree in $\text{sqrt}(h)$ groups ($h = 9$) and calculate LCA for node 6 and 30.



In the above decomposed tree

Jump_parent[6] = 0	parent[6] = 3
Jump_parent[5] = 0	parent[5] = 2
Jump_parent[1] = 0	parent[1] = 0
Jump_parent[11] = 6	parent[11] = 6
Jump_parent[15] = 6	parent[15] = 11
Jump_parent[21] = 6	parent[21] = 15
Jump_parent[25] = 21	parent[25] = 21
Jump_parent[26] = 21	parent[26] = 21
Jump_parent[30] = 21	parent[30] = 25

Now at this stage Jump_parent for node 30 is 21 and Jump_parent for node 5 is 0, So we will climb to jump_parent[30] i.e to node 21

Now once again Jump_parent of node 21 is not equal to Jump_parent of node 5, So once again we will climb to jump_parent[21] i.e node 6

At this stage jump_parent[6] == jump_parent[5], So now we will use our naive climbing approach and climb one parent above for both the nodes till it reach node 1 and that will be the required LCA .

Blue path in the above figure describes jumping path sequence for node 6 and node 5.

The C++ code for the above description is given below:-

```
// C++ program to find LCA using Sqrt decomposition
#include <iostream>
#include <vector>
#include <math.h>
using namespace std;
#define MAXN 1001

int block_sz;           // block size = sqrt(height)
int depth[MAXN];        // stores depth for each node
int parent[MAXN];       // stores first parent for
                        // each node
int jump_parent[MAXN]; // stores first ancestor in
                        // previous block

vector < int > adj[MAXN];

void addEdge(int u,int v)
{
    adj[u].push_back(v);
    adj[v].push_back(u);
}
```

```
int LCANaive(int u,int v)
{
    if (u == v)  return u;
    if (depth[u] > depth[v])
        swap(u,v);
    v = parent[v];
    return LCANaive(u,v);
}

// precalculating the required parameters
// associated with every node
void dfs(int cur, int prev)
{
    // marking depth of cur node
    depth[cur] = depth[prev] + 1;

    // marking parent of cur node
    parent[cur] = prev;

    // making jump_parent of cur node
    if (depth[cur] % block_sz == 0)

        /* if it is first node of the block
           then its jump_parent is its cur parent */
        jump_parent[cur] = parent[cur];

    else

        /* if it is not the first node of this block
           then its jump_parent is jump_parent of
           its parent */
        jump_parent[cur] = jump_parent[prev];

    // propogating the marking down the subtree
    for (int i = 0; i<adj[cur].size(); ++i)
        if (adj[cur][i] != prev)
            dfs(adj[cur][i], cur);
}

// using sqrt decomposition trick
int LCASQRT(int u, int v)
{
    while (jump_parent[u] != jump_parent[v])
    {
        if (depth[u] > depth[v])
```

```
// maintaining depth[v] > depth[u]
swap(u,v);

// climb to its jump parent
v = jump_parent[v];
}

// u and v have same jump_parent
return LCANaive(u,v);
}

void preprocess(int height)
{
    block_sz = sqrt(height);
    depth[0] = -1;

    // precalclating 1)depth. 2)parent. 3)jump_parent
    // for each node
    dfs(1, 0);
}

// Driver function to call the above functions
int main(int argc, char const *argv[])
{
    // adding edges to the tree
    addEdge(1,2);
    addEdge(1,3);
    addEdge(1,4);
    addEdge(2,5);
    addEdge(2,6);
    addEdge(3,7);
    addEdge(4,8);
    addEdge(4,9);
    addEdge(9,10);
    addEdge(9,11);
    addEdge(7,12);
    addEdge(7,13);

    // here we are directly taking height = 4
    // according to the given tree but we can
    // pre-calculate height = max depth
    // in one more dfs
    int height = 4;
    preprocess(height);

    cout << "LCA(11,8) : " << LCASQRT(11,8) << endl;
    cout << "LCA(3,13) : " << LCASQRT(3,13) << endl;
}
```

```
    return 0;
}
```

Output:

```
LCA(11,8) : 4
LCA(3,13) : 3
```

Note : The above code works even if height is not perfect square.

Now Lets see how the Time Complexity is changed by this simple grouping technique :

Time Complexity Analysis:

We have divided the tree into \sqrt{h} groups according to their depth and each group contain nodes having max difference in their depth equal to \sqrt{h} . Now once again take an example of worst case, let's say the first node 'u' is in first group and the node 'v' is in \sqrt{h} th group(last group). So, first we will make group jumps(single group jumps) till we reach group 1 from last group; This will take exactly $\sqrt{h} - 1$ iterations or jumps. So, till this step the Time Complexity is **$O(\sqrt{h})$** .

Now once we are in same group, we call the LCAnaive function. The Time complexity for LCA_Naive is $O(\sqrt{h})$, where h' is the height of the tree. Now, in our case value of h' will be \sqrt{h} , because each group has a subtree of at max \sqrt{h} height. So the complexity for this step is also $O(\sqrt{h})$.

Hence, the total Time Complexity will be **$O(\sqrt{h}) + \sqrt{h}) \sim O(\sqrt{h})$** .

Source

<https://www.geeksforgeeks.org/sqrt-square-root-decomposition-set-2-lca-tree-osqrth-time/>

Chapter 181

Substring with highest frequency length product

Substring with highest frequency length product - GeeksforGeeks

Given a string which contains lower alphabetic characters, we need to find out such a substring of this string whose product of length and frequency in string is maximum among all possible choices of substrings.

Examples:

```
Input : String str = "abddab"
Output : 6
All unique substring with product of their
frequency and length are,
Val["a"] = 2 * 1 = 2
Val["ab"] = 2 * 2 = 4
Val["abd"] = 1 * 3 = 3
Val["abdd"] = 1 * 4 = 4
Val["abdda"] = 1 * 5 = 5
Val["abddab"] = 1 * 6 = 6
Val["b"] = 2 * 1 = 2
Val["bd"] = 1 * 2 = 2
Val["bdd"] = 1 * 3 = 3
Val["bdda"] = 1 * 4 = 4
Val["bddab"] = 1 * 5 = 5
Val["d"] = 2 * 1 = 2
Val["da"] = 1 * 2 = 2
Val["dab"] = 1 * 3 = 3
Val["dd"] = 1 * 2 = 2
Val["dda"] = 1 * 3 = 3
Val["ddab"] = 1 * 4 = 4
```

```

Input : String str = "zzzzzz"
Output : 12
In above string maximum value 12 can
be obtained with substring "zzzz"

```

A **simple solution** is to consider all substrings one by one. For every substring, count number of occurrences of it in whole string.

An **efficient solution** to solve this problem by first constructing longest common prefix array, now suppose value of $lcp[i]$ is K then we can say that i -th and $(i+1)$ -th suffix has K length prefix in common i.e. there is a substring of length K which is repeating twice. In the same way, let three consecutive values of lcp are $(K, K-2, K+1)$ then we can say that there is a substring of length $(K-2)$ which is repeating three times in the string.

Now after above observation, we can see that our result will be such a range of lcp array whose smallest element times number of elements in the range is maximum because range will correspond to the frequency of string and smallest element of range will correspond to length of repeating string now this reformed problem can be solved similar to [largest rectangle in histogram problem](#).

In below code lcp array is constructed by [Kasai's algorithm](#).

```

// C++ program to find substring with highest
// frequency length product
#include <bits/stdc++.h>
using namespace std;

// Structure to store information of a suffix
struct suffix
{
    int index; // To store original index
    int rank[2]; // To store ranks and next rank pair
};

// A comparison function used by sort() to compare
// two suffixes. Compares two pairs, returns 1 if
// first pair is smaller
int cmp(struct suffix a, struct suffix b)
{
    return (a.rank[0] == b.rank[0])?
        (a.rank[1] < b.rank[1] ? 1: 0):
        (a.rank[0] < b.rank[0] ? 1: 0);
}

// This is the main function that takes a string
// 'txt' of size n as an argument, builds and
// return the suffix array for the given string
vector<int> buildSuffixArray(string txt, int n)
{
    // A structure to store suffixes and their indexes

```

```
struct suffix suffixes[n];

// Store suffixes and their indexes in an array
// of structures. The structure is needed to sort
// the suffixes alphabetically and maintain their
// old indexes while sorting
for (int i = 0; i < n; i++)
{
    suffixes[i].index = i;
    suffixes[i].rank[0] = txt[i] - 'a';
    suffixes[i].rank[1] = ((i+1) < n)? (txt[i + 1] - 'a'): -1;
}

// Sort the suffixes using the comparison function
// defined above.
sort(suffixes, suffixes+n, cmp);

// At this point, all suffixes are sorted according to first
// 2 characters. Let us sort suffixes according to first 4
// characters, then first 8 and so on
// This array is needed to get the index in suffixes[]
// from original index. This mapping is needed to get
// next suffix.
int ind[n];
for (int k = 4; k < 2*n; k = k*2)
{
    // Assigning rank and index values to first suffix
    int rank = 0;
    int prev_rank = suffixes[0].rank[0];
    suffixes[0].rank[0] = rank;
    ind[suffixes[0].index] = 0;

    // Assigning rank to suffixes
    for (int i = 1; i < n; i++)
    {
        // If first rank and next ranks are same as
        // that of previous suffix in array, assign
        // the same new rank to this suffix
        if (suffixes[i].rank[0] == prev_rank &&
            suffixes[i].rank[1] == suffixes[i-1].rank[1])
        {
            prev_rank = suffixes[i].rank[0];
            suffixes[i].rank[0] = rank;
        }
        else // Otherwise increment rank and assign
        {
            prev_rank = suffixes[i].rank[0];
            suffixes[i].rank[0] = ++rank;
        }
    }
}
```

```

        }
        ind[suffixes[i].index] = i;
    }

    // Assign next rank to every suffix
    for (int i = 0; i < n; i++)
    {
        int nextindex = suffixes[i].index + k/2;
        suffixes[i].rank[1] = (nextindex < n)?
            suffixes[ind[nextindex]].rank[0]: -1;
    }

    // Sort the suffixes according to first k characters
    sort(suffixes, suffixes+n, cmp);
}

// Store indexes of all sorted suffixes in the suffix array
vector<int>suffixArr;
for (int i = 0; i < n; i++)
    suffixArr.push_back(suffixes[i].index);

// Return the suffix array
return suffixArr;
}

/* To construct and return LCP */
vector<int> kasai(string txt, vector<int> suffixArr)
{
    int n = suffixArr.size();

    // To store LCP array
    vector<int> lcp(n, 0);

    // An auxiliary array to store inverse of suffix array
    // elements. For example if suffixArr[0] is 5, the
    // invSuff[5] would store 0. This is used to get next
    // suffix string from suffix array.
    vector<int> invSuff(n, 0);

    // Fill values in invSuff[]
    for (int i=0; i < n; i++)
        invSuff[suffixArr[i]] = i;

    // Initialize length of previous LCP
    int k = 0;

    // Process all suffixes one by one starting from
    // first suffix in txt[]
}

```

```

for (int i=0; i<n; i++)
{
    /* If the current suffix is at n-1, then we don't
       have next substring to consider. So lcp is not
       defined for this substring, we put zero. */
    if (invSuff[i] == n-1)
    {
        k = 0;
        continue;
    }

    /* j contains index of the next substring to
       be considered to compare with the present
       substring, i.e., next string in suffix array */
    int j = suffixArr[invSuff[i]+1];

    // Directly start matching from k'th index as
    // at-least k-1 characters will match
    while (i+k<n && j+k<n && txt[i+k]==txt[j+k])
        k++;

    lcp[invSuff[i]] = k; // lcp for the present suffix.

    // Deleting the starting character from the string.
    if (k>0)
        k--;
}

// return the constructed lcp array
return lcp;
}

// method to get LCP array
vector<int> getLCPArray(string str)
{
    vector<int>suffixArr = buildSuffixArray(str, str.length());
    return kasai(str, suffixArr);
}

// The main function to find the maximum rectangular
// area under given histogram with n bars
int getMaxArea(int hist[], int n)
{
    // Create an empty stack. The stack holds indexes
    // of hist[] array. The bars stored in stack are
    // always in increasing order of their heights.
    stack<int> s;
}

```

```

int max_area = 0; // Initialize max area
int tp; // To store top of stack

// To store area with top bar as the smallest bar
int area_with_top;

// Run through all bars of given histogram
int i = 0;
while (i < n)
{
    // If this bar is higher than the bar on
    // top stack, push it to stack
    if (s.empty() || hist[s.top()] <= hist[i])
        s.push(i++);

    // If this bar is lower than top of stack,
    // then calculate area of rectangle with
    // stack top as the smallest (or minimum
    // height) bar. 'i' is 'right index' for
    // the top and element before top in stack
    // is 'left index'
    else
    {
        tp = s.top(); // store the top index
        s.pop(); // pop the top

        // Calculate the area with hist[tp]
        // stack as smallest bar
        area_with_top = hist[tp] * (s.empty() ?
                                      (i + 1) : i - s.top());

        // update max area, if needed
        if (max_area < area_with_top)
            max_area = area_with_top;
    }
}

// Now pop the remaining bars from stack
// and calculate area with every
// popped bar as the smallest bar
while (s.empty() == false)
{
    tp = s.top();
    s.pop();
    area_with_top = hist[tp] * (s.empty() ?
                                (i + 1) : i - s.top());

    if (max_area < area_with_top)

```

```
        max_area = area_with_top;
    }

    return max_area;
}

// Returns maximum product of frequency and length
// of a substring.
int maxProductOfFreqLength(string str)
{
    // get LCP array by Kasai's algorithm
    vector<int> lcp = getLCPArray(str);

    int hist[lcp.size()];

    // copy lcp array into hist array
    for (int i = 0; i < lcp.size(); i++)
        hist[i] = lcp[i];

    // get the maximum area under lcp histogram
    int substrMaxValue = getMaxArea(hist, lcp.size());

    // if string length itself is greater than
    // histogram area, then return that
    if (str.length() > substrMaxValue)
        return str.length();
    else
        return substrMaxValue;
}

// Driver code to test above methods
int main()
{
    string str = "abddab";
    cout << maxProductOfFreqLength(str) << endl;
    return 0;
}
```

Output:

6

Source

<https://www.geeksforgeeks.org/substring-highest-frequency-length-product/>

Chapter 182

Suffix Array Set 1 (Introduction)

Suffix Array Set 1 (Introduction) - GeeksforGeeks

We strongly recommend to read following post on suffix trees as a pre-requisite for this post.

[Pattern Searching Set 8 \(Suffix Tree Introduction\)](#)

A suffix array is a sorted array of all suffixes of a given string. The definition is similar to [Suffix Tree](#) which is compressed trie of all suffixes of the given text. Any suffix tree based algorithm can be replaced with an algorithm that uses a suffix array enhanced with additional information and solves the same problem in the same time complexity (Source [Wiki](#)).

A suffix array can be constructed from Suffix tree by doing a DFS traversal of the suffix tree. In fact Suffix array and suffix tree both can be constructed from each other in linear time.

Advantages of suffix arrays over suffix trees include improved space requirements, simpler linear time construction algorithms (e.g., compared to Ukkonen's algorithm) and improved cache locality (Source: [Wiki](#))

Example:

Let the given string be "banana".

0 banana	Sort the Suffixes	5 a
1 anana	----->	3 ana
2 nana	alphabetically	1 anana
3 ana		0 banana
4 na		4 na
5 a		2 nana

So the suffix array for "banana" is {5, 3, 1, 0, 4, 2}

Naive method to build Suffix Array

A simple method to construct suffix array is to make an array of all suffixes and then sort the array. Following is implementation of simple method.

```

// Naive algorithm for building suffix array of a given text
#include <iostream>
#include <cstring>
#include <algorithm>
using namespace std;

// Structure to store information of a suffix
struct suffix
{
    int index;
    char *suff;
};

// A comparison function used by sort() to compare two suffixes
int cmp(struct suffix a, struct suffix b)
{
    return strcmp(a.suff, b.suff) < 0? 1 : 0;
}

// This is the main function that takes a string 'txt' of size n as an
// argument, builds and return the suffix array for the given string
int *buildSuffixArray(char *txt, int n)
{
    // A structure to store suffixes and their indexes
    struct suffix suffixes[n];

    // Store suffixes and their indexes in an array of structures.
    // The structure is needed to sort the suffixes alphabetically
    // and maintain their old indexes while sorting
    for (int i = 0; i < n; i++)
    {
        suffixes[i].index = i;
        suffixes[i].suff = (txt+i);
    }

    // Sort the suffixes using the comparison function
    // defined above.
    sort(suffixes, suffixes+n, cmp);

    // Store indexes of all sorted suffixes in the suffix array
    int *suffixArr = new int[n];
    for (int i = 0; i < n; i++)
        suffixArr[i] = suffixes[i].index;
}

```

```

        // Return the suffix array
        return suffixArr;
    }

    // A utility function to print an array of given size
    void printArr(int arr[], int n)
    {
        for(int i = 0; i < n; i++)
            cout << arr[i] << " ";
        cout << endl;
    }

    // Driver program to test above functions
    int main()
    {
        char txt[] = "banana";
        int n = strlen(txt);
        int *suffixArr = buildSuffixArray(txt, n);
        cout << "Following is suffix array for " << txt << endl;
        printArr(suffixArr, n);
        return 0;
    }
}

```

Output:

```

Following is suffix array for banana
5 3 1 0 4 2

```

The time complexity of above method to build suffix array is $O(n^2 \log n)$ if we consider a $O(n \log n)$ algorithm used for sorting. The sorting step itself takes $O(n^2 \log n)$ time as every comparison is a comparison of two strings and the comparison takes $O(n)$ time.

There are many efficient algorithms to build suffix array. We will soon be covering them as separate posts.

Search a pattern using the built Suffix Array

To search a pattern in a text, we preprocess the text and build a suffix array of the text. Since we have a sorted array of all suffixes, [Binary Search](#) can be used to search. Following is the search function. Note that the function doesn't report all occurrences of pattern, it only report one of them.

```

// This code only contains search() and main. To make it a complete running
// above code or see https://ide.geeksforgeeks.org/oY70kD

// A suffix array based search function to search a given pattern
// 'pat' in given text 'txt' using suffix array suffArr[]
void search(char *pat, char *txt, int *suffArr, int n)
{
    int m = strlen(pat); // get length of pattern, needed for strncmp()

```

```
// Do simple binary search for the pat in txt using the
// built suffix array
int l = 0, r = n-1; // Initialize left and right indexes
while (l <= r)
{
    // See if 'pat' is prefix of middle suffix in suffix array
    int mid = l + (r - l)/2;
    int res = strncmp(pat, txt+suffArr[mid], m);

    // If match found at the middle, print it and return
    if (res == 0)
    {
        cout << "Pattern found at index " << suffArr[mid];
        return;
    }

    // Move to left half if pattern is alphabetically less than
    // the mid suffix
    if (res < 0) r = mid - 1;

    // Otherwise move to right half
    else l = mid + 1;
}

// We reach here if return statement in loop is not executed
cout << "Pattern not found";
}

// Driver program to test above function
int main()
{
    char txt[] = "banana"; // text
    char pat[] = "nan"; // pattern to be searched in text

    // Build suffix array
    int n = strlen(txt);
    int *suffArr = buildSuffixArray(txt, n);

    // search pat in txt using the built suffix array
    search(pat, txt, suffArr, n);

    return 0;
}
```

Output:

```
Pattern found at index 2
```

The time complexity of the above search function is $O(m\log n)$. There are more efficient algorithms to search pattern once the suffix array is built. In fact there is a $O(m)$ suffix array based algorithm to search a pattern. We will soon be discussing efficient algorithm for search.

Applications of Suffix Array

Suffix array is an extremely useful data structure, it can be used for a wide range of problems. Following are some famous problems where Suffix array can be used.

- 1) Pattern Searching
- 2) [Finding the longest repeated substring](#)
- 3) [Finding the longest common substring](#)
- 4) [Finding the longest palindrome in a string](#)

See [this](#) for more problems where Suffix arrays can be used.

This post is a simple introduction. There is a lot to cover in Suffix arrays. We have discussed a [\$O\(n\log n\)\$ algorithm for Suffix Array construction here](#). We will soon be discussing more efficient suffix array algorithms.

References:

- <http://www.stanford.edu/class/cs97si/suffix-array.pdf>
- http://en.wikipedia.org/wiki/Suffix_array

Source

<https://www.geeksforgeeks.org/suffix-array-set-1-introduction/>

Chapter 183

Suffix Array Set 2 (nLogn Algorithm)

Suffix Array Set 2 (nLogn Algorithm) - GeeksforGeeks

A suffix array is a sorted array of all suffixes of a given string. The definition is similar to [Suffix Tree](#) which is compressed trie of all suffixes of the given text.

Let the given string be "banana".

0 banana	5 a	
1 anana	Sort the Suffixes	3 ana
2 nana	----->	1 anana
3 ana	alphabetically	0 banana
4 na		4 na
5 a		2 nana

The suffix array for "banana" is {5, 3, 1, 0, 4, 2}

We have discussed [Naive algorithm](#) for construction of suffix array. The Naive algorithm is to consider all suffixes, sort them using a $O(n \log n)$ sorting algorithm and while sorting, maintain original indexes. Time complexity of the Naive algorithm is $O(n^2 \log n)$ where n is the number of characters in the input string.

In this post, a **$O(n \log n)$ algorithm** for suffix array construction is discussed. Let us first discuss a $O(n * \log n * \log n)$ algorithm for simplicity. The idea is to use the fact that strings that are to be sorted are suffixes of a single string.

We first sort all suffixes according to first character, then according to first 2 characters, then first 4 characters and so on while the number of characters to be considered is smaller than $2n$. The important point is, if we have sorted suffixes according to first 2^i characters, then we can sort suffixes according to first 2^{i+1} characters in $O(n \log n)$ time using a $n \log n$ sorting algorithm like Merge Sort. This is possible as two suffixes can be compared in $O(1)$

time (we need to compare only two values, see the below example and code).

The sort function is called $O(\log n)$ times (Note that we increase number of characters to be considered in powers of 2). Therefore overall time complexity becomes $O(n \log n \log \log n)$. See <http://www.stanford.edu/class/cs97si/suffix-array.pdf> for more details.

Let us build suffix array the example string “banana” using above algorithm.

Sort according to first two characters Assign a rank to all suffixes using ASCII value of first character. A simple way to assign rank is to do “ $\text{str}[i] - 'a'$ ” for ith suffix of $\text{str}[]$

Index	Suffix	Rank
0	banana	1
1	anana	0
2	nana	13
3	ana	0
4	na	13
5	a	0

For every character, we also store rank of next adjacent character, i.e., the rank of character at $\text{str}[i + 1]$ (This is needed to sort the suffixes according to first 2 characters). If a character is last character, we store next rank as -1

Index	Suffix	Rank	Next Rank
0	banana	1	0
1	anana	0	13
2	nana	13	0
3	ana	0	13
4	na	13	0
5	a	0	-1

Sort all Suffixes according to rank and adjacent rank. Rank is considered as first digit or MSD, and adjacent rank is considered as second digit.

Index	Suffix	Rank	Next Rank
5	a	0	-1
1	anana	0	13
3	ana	0	13
0	banana	1	0
2	nana	13	0
4	na	13	0

Sort according to first four character

Assign new ranks to all suffixes. To assign new ranks, we consider the sorted suffixes one by one. Assign 0 as new rank to first suffix. For assigning ranks to remaining suffixes, we

consider rank pair of suffix just before the current suffix. If previous rank pair of a suffix is same as previous rank of suffix just before it, then assign it same rank. Otherwise assign rank of previous suffix plus one.

Index	Suffix	Rank	
5	a	0	[Assign 0 to first]
1	anana	1	(0, 13) is different from previous
3	ana	1	(0, 13) is same as previous
0	banana	2	(1, 0) is different from previous
2	nana	3	(13, 0) is different from previous
4	na	3	(13, 0) is same as previous

For every suffix str[i], also store rank of next suffix at str[i + 2]. If there is no next suffix at i + 2, we store next rank as -1

Index	Suffix	Rank	Next Rank
5	a	0	-1
1	anana	1	1
3	ana	1	0
0	banana	2	3
2	nana	3	3
4	na	3	-1

Sort all Suffixes according to rank and next rank.

Index	Suffix	Rank	Next Rank
5	a	0	-1
3	ana	1	0
1	anana	1	1
0	banana	2	3
4	na	3	-1
2	nana	3	3

```
// C++ program for building suffix array of a given text
#include <iostream>
#include <cstring>
#include <algorithm>
using namespace std;

// Structure to store information of a suffix
struct suffix
{
    int index; // To store original index
```

```

        int rank[2]; // To store ranks and next rank pair
    };

    // A comparison function used by sort() to compare two suffixes
    // Compares two pairs, returns 1 if first pair is smaller
    int cmp(struct suffix a, struct suffix b)
    {
        return (a.rank[0] == b.rank[0])? (a.rank[1] < b.rank[1] ?1: 0):
            (a.rank[0] < b.rank[0] ?1: 0);
    }

    // This is the main function that takes a string 'txt' of size n as an
    // argument, builds and return the suffix array for the given string
    int *buildSuffixArray(char *txt, int n)
    {
        // A structure to store suffixes and their indexes
        struct suffix suffixes[n];

        // Store suffixes and their indexes in an array of structures.
        // The structure is needed to sort the suffixes alphabetically
        // and maintain their old indexes while sorting
        for (int i = 0; i < n; i++)
        {
            suffixes[i].index = i;
            suffixes[i].rank[0] = txt[i] - 'a';
            suffixes[i].rank[1] = ((i+1) < n)? (txt[i + 1] - 'a'): -1;
        }

        // Sort the suffixes using the comparison function
        // defined above.
        sort(suffixes, suffixes+n, cmp);

        // At this point, all suffixes are sorted according to first
        // 2 characters. Let us sort suffixes according to first 4
        // characters, then first 8 and so on
        int ind[n]; // This array is needed to get the index in suffixes[]
                    // from original index. This mapping is needed to get
                    // next suffix.
        for (int k = 4; k < 2*n; k = k*2)
        {
            // Assigning rank and index values to first suffix
            int rank = 0;
            int prev_rank = suffixes[0].rank[0];
            suffixes[0].rank[0] = rank;
            ind[suffixes[0].index] = 0;

            // Assigning rank to suffixes
            for (int i = 1; i < n; i++)

```

```

{
    // If first rank and next ranks are same as that of previous
    // suffix in array, assign the same new rank to this suffix
    if (suffixes[i].rank[0] == prev_rank &&
        suffixes[i].rank[1] == suffixes[i-1].rank[1])
    {
        prev_rank = suffixes[i].rank[0];
        suffixes[i].rank[0] = rank;
    }
    else // Otherwise increment rank and assign
    {
        prev_rank = suffixes[i].rank[0];
        suffixes[i].rank[0] = ++rank;
    }
    ind[suffixes[i].index] = i;
}

// Assign next rank to every suffix
for (int i = 0; i < n; i++)
{
    int nextindex = suffixes[i].index + k/2;
    suffixes[i].rank[1] = (nextindex < n)?
                           suffixes[ind[nextindex]].rank[0]: -1;
}

// Sort the suffixes according to first k characters
sort(suffixes, suffixes+n, cmp);
}

// Store indexes of all sorted suffixes in the suffix array
int *suffixArr = new int[n];
for (int i = 0; i < n; i++)
    suffixArr[i] = suffixes[i].index;

// Return the suffix array
return suffixArr;
}

// A utility function to print an array of given size
void printArr(int arr[], int n)
{
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";
    cout << endl;
}

// Driver program to test above functions
int main()

```

```
{  
    char txt[] = "banana";  
    int n = strlen(txt);  
    int *suffixArr = buildSuffixArray(txt, n);  
    cout << "Following is suffix array for " << txt << endl;  
    printArr(suffixArr, n);  
    return 0;  
}
```

Output:

```
Following is suffix array for banana  
5 3 1 0 4 2
```

Note that the above algorithm uses standard sort function and therefore time complexity is $O(n\log n \log n)$. We can use [Radix Sort](#) here to reduce the time complexity to $O(n\log n)$.

Please note that suffix arrays can be constructed in $O(n)$ time also. We will soon be discussing $O(n)$ algorithms.

References:

<http://www.stanford.edu/class/cs97si/suffix-array.pdf>
<http://www.cse.brown.edu/~civan/courses/Fall2012/lec14b.pdf>

Improved By : [Akash Kumar 31](#)

Source

<https://www.geeksforgeeks.org/suffix-array-set-2-a-nlognlogn-algorithm/>

Chapter 184

Suffix Tree Application 1 – Substring Check

Suffix Tree Application 1 - Substring Check - GeeksforGeeks

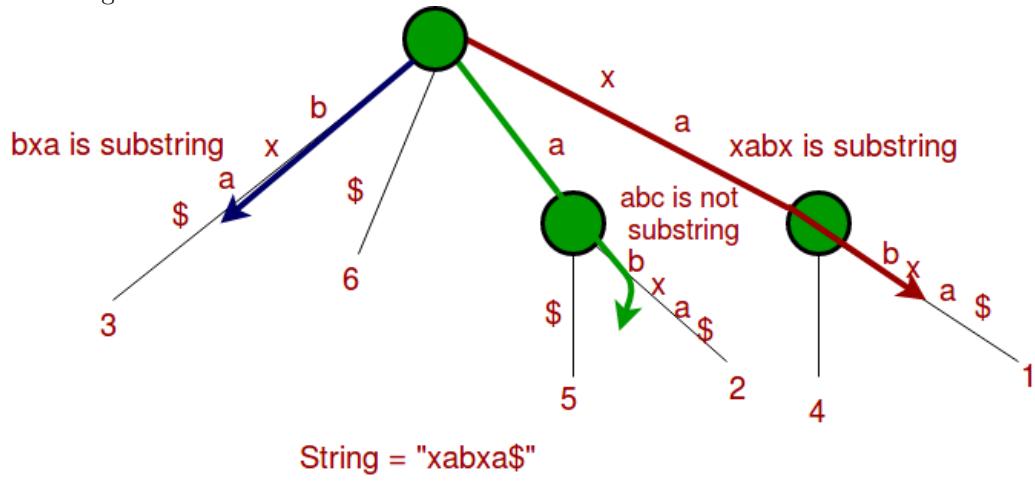
Given a text string and a pattern string, check if pattern exists in text or not.

Few pattern searching algorithms ([KMP](#), [Rabin-Karp](#), [Naive Algorithm](#), [Finite Automata](#)) are already discussed, which can be used for this check.

Here we will discuss suffix tree based algorithm.

As a prerequisite, we must know how to build a suffix tree in one or the other way.

Once we have a suffix tree built for given text, we need to traverse the tree from root to leaf against the characters in pattern. If we do not fall off the tree (i.e. there is a path from root to leaf or somewhere in middle) while traversal, then pattern exists in text as a substring.



Here we will build suffix tree using Ukkonen's Algorithm, discussed already as below:

- [Ukkonen's Suffix Tree Construction – Part 1](#)
- [Ukkonen's Suffix Tree Construction – Part 2](#)
- [Ukkonen's Suffix Tree Construction – Part 3](#)
- [Ukkonen's Suffix Tree Construction – Part 4](#)
- [Ukkonen's Suffix Tree Construction – Part 5](#)
- [Ukkonen's Suffix Tree Construction – Part 6](#)

The core traversal implementation for substring check, can be modified accordingly for suffix trees built by other algorithms.

```
// A C program for substring check using Ukkonen's Suffix Tree Construction
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define MAX_CHAR 256

struct SuffixTreeNode {
    struct SuffixTreeNode *children[MAX_CHAR];
    //pointer to other node via suffix link
    struct SuffixTreeNode *suffixLink;

    /*(start, end) interval specifies the edge, by which the
     node is connected to its parent node. Each edge will
     connect two nodes, one parent and one child, and
     (start, end) interval of a given edge will be stored
     in the child node. Lets say there are two nodes A and B
     connected by an edge with indices (5, 8) then this
     indices (5, 8) will be stored in node B. */
    int start;
    int *end;

    /*for leaf nodes, it stores the index of suffix for
     the path from root to leaf*/
    int suffixIndex;
};

typedef struct SuffixTreeNode Node;

char text[100]; //Input string
Node *root = NULL; //Pointer to root node

/*lastNewNode will point to newly created internal node,
 waiting for it's suffix link to be set, which might get
 a new suffix link (other than root) in next extension of
 same phase. lastNewNode will be set to NULL when last
 newly created internal node (if there is any) got it's
 suffix link reset to new internal node created in next
```

```

extension of same phase. */
Node *lastNewNode = NULL;
Node *activeNode = NULL;

/*activeEdge is represented as input string character
 index (not the character itself)*/
int activeEdge = -1;
int activeLength = 0;

// remainingSuffixCount tells how many suffixes yet to
// be added in tree
int remainingSuffixCount = 0;
int leafEnd = -1;
int *rootEnd = NULL;
int *splitEnd = NULL;
int size = -1; //Length of input string

Node *newNode(int start, int *end)
{
    Node *node =(Node*) malloc(sizeof(Node));
    int i;
    for (i = 0; i < MAX_CHAR; i++)
        node->children[i] = NULL;

    /*For root node, suffixLink will be set to NULL
    For internal nodes, suffixLink will be set to root
    by default in current extension and may change in
    next extension*/
    node->suffixLink = root;
    node->start = start;
    node->end = end;

    /*suffixIndex will be set to -1 by default and
     actual suffix index will be set later for leaves
     at the end of all phases*/
    node->suffixIndex = -1;
    return node;
}

int edgeLength(Node *n) {
    if(n == root)
        return 0;
    return *(n->end) - (n->start) + 1;
}

int walkDown(Node *currNode)
{
    /*activePoint change for walk down (APCFWD) using

```

```

Skip/Count Trick (Trick 1). If activeLength is greater
than current edge length, set next internal node as
activeNode and adjust activeEdge and activeLength
accordingly to represent same activePoint*/
if (activeLength >= edgeLength(currNode))
{
    activeEdge += edgeLength(currNode);
    activeLength -= edgeLength(currNode);
    activeNode = currNode;
    return 1;
}
return 0;
}

void extendSuffixTree(int pos)
{
    /*Extension Rule 1, this takes care of extending all
     leaves created so far in tree*/
    leafEnd = pos;

    /*Increment remainingSuffixCount indicating that a
     new suffix added to the list of suffixes yet to be
     added in tree*/
    remainingSuffixCount++;

    /*set lastNewNode to NULL while starting a new phase,
     indicating there is no internal node waiting for
     it's suffix link reset in current phase*/
    lastNewNode = NULL;

    //Add all suffixes (yet to be added) one by one in tree
    while(remainingSuffixCount > 0) {

        if (activeLength == 0)
            activeEdge = pos; //APCFALZ

        // There is no outgoing edge starting with
        // activeEdge from activeNode
        if (activeNode->children] == NULL)
        {
            //Extension Rule 2 (A new leaf edge gets created)
            activeNode->children] =
                newNode(pos, &leafEnd);

            /*A new leaf edge is created in above line starting
             from an existng node (the current activeNode), and
             if there is any internal node waiting for it's suffix
             link get reset, point the suffix link from that last

```

```

internal node to current activeNode. Then set lastNewNode
to NULL indicating no more node waiting for suffix link
reset.*/
if (lastNewNode != NULL)
{
    lastNewNode->suffixLink = activeNode;
    lastNewNode = NULL;
}
}
// There is an outgoing edge starting with activeEdge
// from activeNode
else
{
    // Get the next node at the end of edge starting
    // with activeEdge
    Node *next = activeNode->children];
    if (walkDown(next))//Do walkdown
    {
        //Start from next node (the new activeNode)
        continue;
    }
    /*Extension Rule 3 (current character being processed
       is already on the edge)*/
    if (text[next->start + activeLength] == text[pos])
    {
        //If a newly created node waiting for it's
        //suffix link to be set, then set suffix link
        //of that waiting node to current active node
        if(lastNewNode != NULL && activeNode != root)
        {
            lastNewNode->suffixLink = activeNode;
            lastNewNode = NULL;
        }

        //APCFER3
        activeLength++;
        /*STOP all further processing in this phase
        and move on to next phase*/
        break;
    }

    /*We will be here when activePoint is in middle of
       the edge being traversed and current character
       being processed is not on the edge (we fall off
       the tree). In this case, we add a new internal node
       and a new leaf edge going out of that new node. This
       is Extension Rule 2, where a new leaf edge and a new
       internal node get created*/
}

```



```
void print(int i, int j)
{
    int k;
    for (k=i; k<=j; k++)
        printf("%c", text[k]);
}

//Print the suffix tree as well along with setting suffix index
//So tree will be printed in DFS manner
//Each edge along with it's suffix index will be printed
void setSuffixIndexByDFS(Node *n, int labelHeight)
{
    if (n == NULL)    return;

    if (n->start != -1) //A non-root node
    {
        //Print the label on edge from parent to current node
        //Uncomment below line to print suffix tree
        // printf(n->start, *(n->end));
    }
    int leaf = 1;
    int i;
    for (i = 0; i < MAX_CHAR; i++)
    {
        if (n->children[i] != NULL)
        {
            //Uncomment below two lines to print suffix index
            // if (leaf == 1 && n->start != -1)
            //     printf(" [%d]\n", n->suffixIndex);

            //Current node is not a leaf as it has outgoing
            //edges from it.
            leaf = 0;
            setSuffixIndexByDFS(n->children[i], labelHeight +
                                edgeLength(n->children[i]));
        }
    }
    if (leaf == 1)
    {
        n->suffixIndex = size - labelHeight;
        //Uncomment below line to print suffix index
        //printf(" [%d]\n", n->suffixIndex);
    }
}

void freeSuffixTreeByPostOrder(Node *n)
{
```

```

if (n == NULL)
    return;
int i;
for (i = 0; i < MAX_CHAR; i++)
{
    if (n->children[i] != NULL)
    {
        freeSuffixTreeByPostOrder(n->children[i]);
    }
}
if (n->suffixIndex == -1)
    free(n->end);
free(n);
}

/*Build the suffix tree and print the edge labels along with
suffixIndex. suffixIndex for leaf edges will be >= 0 and
for non-leaf edges will be -1*/
void buildSuffixTree()
{
    size = strlen(text);
    int i;
    rootEnd = (int*) malloc(sizeof(int));
    *rootEnd = - 1;

    /*Root is a special node with start and end indices as -1,
    as it has no parent from where an edge comes to root*/
    root = newNode(-1, rootEnd);

    activeNode = root; //First activeNode will be root
    for (i=0; i<size; i++)
        extendSuffixTree(i);
    int labelHeight = 0;
    setSuffixIndexByDFS(root, labelHeight);
}

int traverseEdge(char *str, int idx, int start, int end)
{
    int k = 0;
    //Traverse the edge with character by character matching
    for(k=start; k<=end && str[idx] != '\0'; k++, idx++)
    {
        if(text[k] != str[idx])
            return -1; // no match
    }
    if(str[idx] == '\0')
        return 1; // match
    return 0; // more characters yet to match
}

```

```
}  
  
int doTraversal(Node *n, char* str, int idx)  
{  
    if(n == NULL)  
    {  
        return -1; // no match  
    }  
    int res = -1;  
    //If node n is not root node, then traverse edge  
    //from node n's parent to node n.  
    if(n->start != -1)  
    {  
        res = traverseEdge(str, idx, n->start, *(n->end));  
        if(res != 0)  
            return res; // match (res = 1) or no match (res = -1)  
    }  
    //Get the character index to search  
    idx = idx + edgeLength(n);  
    //If there is an edge from node n going out  
    //with current character str[idx], travrse that edge  
    if(n->children[str[idx]] != NULL)  
        return doTraversal(n->children[str[idx]], str, idx);  
    else  
        return -1; // no match  
}  
  
void checkForSubString(char* str)  
{  
    int res = doTraversal(root, str, 0);  
    if(res == 1)  
        printf("Pattern <%s> is a Substring\n", str);  
    else  
        printf("Pattern <%s> is NOT a Substring\n", str);  
}  
  
// driver program to test above functions  
int main(int argc, char *argv[])  
{  
    strcpy(text, "THIS IS A TEST TEXT$");  
    buildSuffixTree();  
  
    checkForSubString("TEST");  
    checkForSubString("A");  
    checkForSubString(" ");  
    checkForSubString("IS A");  
    checkForSubString(" IS A ");  
    checkForSubString("TEST1");
```

```
checkForSubString("THIS IS GOOD");
checkForSubString("TES");
checkForSubString("TESA");
checkForSubString("ISB");

//Free the dynamically allocated memory
freeSuffixTreeByPostOrder(root);

return 0;
}
```

Output:

```
Pattern <TEST> is a Substring
Pattern <A> is a Substring
Pattern <> is a Substring
Pattern <IS A> is a Substring
Pattern < IS A > is a Substring
Pattern <TEST1> is NOT a Substring
Pattern <THIS IS GOOD> is NOT a Substring
Pattern <TES> is a Substring
Pattern <TESA> is NOT a Substring
Pattern <ISB> is NOT a Substring
```

Ukkonen's Suffix Tree Construction takes $O(N)$ time and space to build suffix tree for a string of length N and after that, traversal for substring check takes $O(M)$ for a pattern of length M .

With slight modification in traversal algorithm discussed here, we can answer following:

1. Find all occurrences of a given pattern P present in text T .
2. How to check if a pattern is prefix of a text?
3. How to check if a pattern is suffix of a text?

We have published following more articles on suffix tree applications:

- [Suffix Tree Application 2 – Searching All Patterns](#)
- [Suffix Tree Application 3 – Longest Repeated Substring](#)
- [Suffix Tree Application 4 – Build Linear Time Suffix Array](#)
- [Generalized Suffix Tree 1](#)
- [Suffix Tree Application 5 – Longest Common Substring](#)
- [Suffix Tree Application 6 – Longest Palindromic Substring](#)

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

Source

<https://www.geeksforgeeks.org/suffix-tree-application-1-substring-check/>

Chapter 185

Suffix Tree Application 2 – Searching All Patterns

Suffix Tree Application 2 - Searching All Patterns - GeeksforGeeks

Given a text string and a pattern string, find all occurrences of the pattern in string.

Few pattern searching algorithms ([KMP](#), [Rabin-Karp](#), [Naive Algorithm](#), [Finite Automata](#)) are already discussed, which can be used for this check.

Here we will discuss suffix tree based algorithm.

In the 1st Suffix Tree Application ([Substring Check](#)), we saw how to check whether a given pattern is substring of a text or not. It is advised to go through [Substring Check 1st](#).

In this article, we will go a bit further on same problem. If a pattern is substring of a text, then we will find all the positions on pattern in the text.

As a prerequisite, we must know how to build a suffix tree in one or the other way.

Here we will build suffix tree using Ukkonen's Algorithm, discussed already as below:

[Ukkonen's Suffix Tree Construction – Part 1](#)

[Ukkonen's Suffix Tree Construction – Part 2](#)

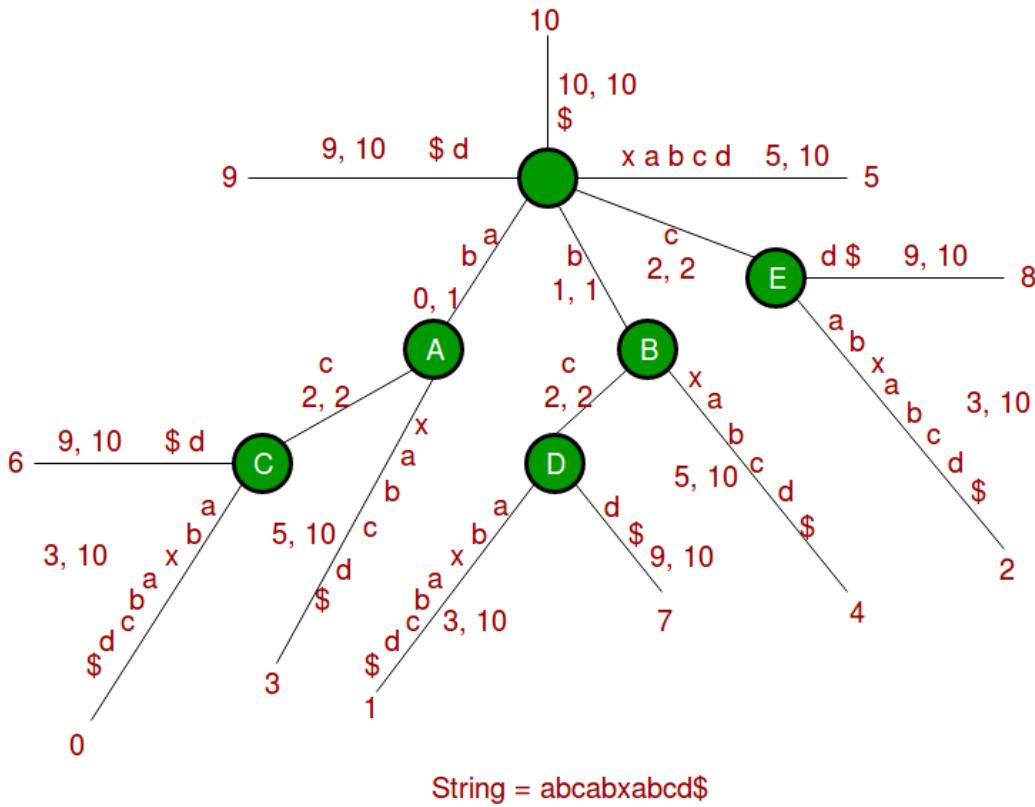
[Ukkonen's Suffix Tree Construction – Part 3](#)

[Ukkonen's Suffix Tree Construction – Part 4](#)

[Ukkonen's Suffix Tree Construction – Part 5](#)

[Ukkonen's Suffix Tree Construction – Part 6](#)

Lets look at following figure:



Suffix Tree

This is suffix tree for String “`abcabxabcd$`”, showing suffix indices and edge label indices (start, end). The (sub)string value on edges are shown only for explanatory purpose. We never store path label string in the tree.

Suffix Index of a path tells the index of a substring (starting from root) on that path.

Consider a path “bcd\$” in above tree with suffix index 7. It tells that substrings b, bc, bcd, bcd\$ are at index 7 in string.

Similarly path “`bxabcd$`” with suffix index 4 tells that substrings `b`, `bx`, `bxa`, `bxab`, `bxabc`, `bxabcd`, `bxabcd$` are at index 4.

Similarly path “bcabxabcd\$” with suffix index 1 tells that substrings b, bc, bca, bcab, bcabx, bcabxa, bcabxb, bcabxbc, bcabxbcd, bcabxbcd\$ are at index 1.

If we see all the above three paths together, we can see that:

- Substring “b” is at indices 1, 4 and 7
 - Substring “bc” is at indices 1 and 7

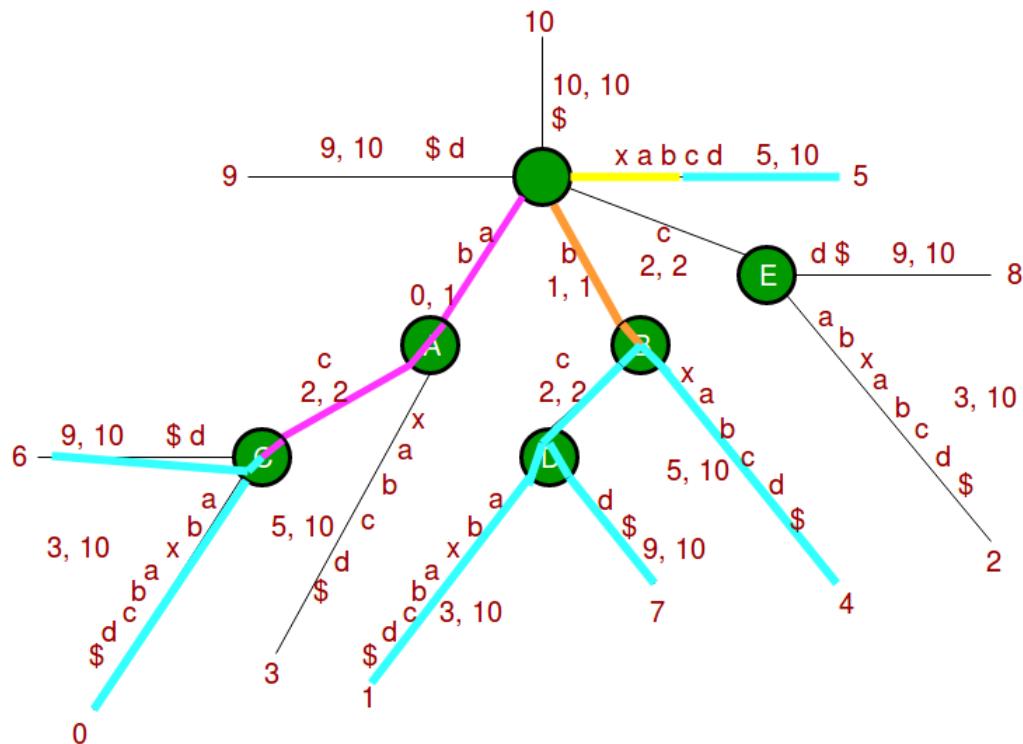
With above explanation, we should be able to see following:

- Substring “ab” is at indices 0, 3 and 6

- Substring “abc” is at indices 0 and 6
- Substring “c” is at indices 2 and 8
- Substring “xab” is at index 5
- Substring “d” is at index 9
- Substring “cd” is at index 8

Can you see how to find all the occurrences of a pattern in a string ?

1. 1st of all, check if the given pattern really exists in string or not (As we did in [Substring Check](#)). For this, traverse the suffix tree against the pattern.
2. If you find pattern in suffix tree (don’t fall off the tree), then traverse the subtree below that point and find all suffix indices on leaf nodes. All those suffix indices will be pattern indices in string



Substring abc is found , subtree traversal shows that it is at indices 0 and 5
 Substring xab is found , subtree traversal shows that it is at index 5

Substring b is found, subtree traversal shows that it is at indices 1, 4, and 7

```
// A C program to implement Ukkonen's Suffix Tree Construction
// And find all locations of a pattern in string
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define MAX_CHAR 256

struct SuffixTreeNode {
    struct SuffixTreeNode *children[MAX_CHAR];

    //pointer to other node via suffix link
    struct SuffixTreeNode *suffixLink;

    /*(start, end) interval specifies the edge, by which the
     node is connected to its parent node. Each edge will
     connect two nodes, one parent and one child, and
     (start, end) interval of a given edge will be stored
     in the child node. Lets say there are two nodes A and B
     connected by an edge with indices (5, 8) then this
     indices (5, 8) will be stored in node B. */
    int start;
    int *end;

    /*for leaf nodes, it stores the index of suffix for
     the path from root to leaf*/
    int suffixIndex;
};

typedef struct SuffixTreeNode Node;

char text[100]; //Input string
Node *root = NULL; //Pointer to root node

/*lastNewNode will point to newly created internal node,
 waiting for it's suffix link to be set, which might get
 a new suffix link (other than root) in next extension of
 same phase. lastNewNode will be set to NULL when last
 newly created internal node (if there is any) got it's
 suffix link reset to new internal node created in next
 extension of same phase. */
Node *lastNewNode = NULL;
Node *activeNode = NULL;

/*activeEdge is represented as input string character
 index (not the character itself)*/
int activeEdge = -1;
int activeLength = 0;
```

```

// remainingSuffixCount tells how many suffixes yet to
// be added in tree
int remainingSuffixCount = 0;
int leafEnd = -1;
int *rootEnd = NULL;
int *splitEnd = NULL;
int size = -1; //Length of input string

Node *newNode(int start, int *end)
{
    Node *node =(Node*) malloc(sizeof(Node));
    int i;
    for (i = 0; i < MAX_CHAR; i++)
        node->children[i] = NULL;

    /*For root node, suffixLink will be set to NULL
    For internal nodes, suffixLink will be set to root
    by default in current extension and may change in
    next extension*/
    node->suffixLink = root;
    node->start = start;
    node->end = end;

    /*suffixIndex will be set to -1 by default and
       actual suffix index will be set later for leaves
       at the end of all phases*/
    node->suffixIndex = -1;
    return node;
}

int edgeLength(Node *n) {
    if(n == root)
        return 0;
    return *(n->end) - (n->start) + 1;
}

int walkDown(Node *currNode)
{
    /*activePoint change for walk down (APCFWD) using
    Skip/Count Trick (Trick 1). If activeLength is greater
    than current edge length, set next internal node as
    activeNode and adjust activeEdge and activeLength
    accordingly to represent same activePoint*/
    if (activeLength >= edgeLength(currNode))
    {
        activeEdge += edgeLength(currNode);
        activeLength -= edgeLength(currNode);
        activeNode = currNode;
    }
}

```

```
        return 1;
    }
    return 0;
}

void extendSuffixTree(int pos)
{
    /*Extension Rule 1, this takes care of extending all
     leaves created so far in tree*/
    leafEnd = pos;

    /*Increment remainingSuffixCount indicating that a
     new suffix added to the list of suffixes yet to be
     added in tree*/
    remainingSuffixCount++;

    /*set lastNewNode to NULL while starting a new phase,
     indicating there is no internal node waiting for
     it's suffix link reset in current phase*/
    lastNewNode = NULL;

    //Add all suffixes (yet to be added) one by one in tree
    while(remainingSuffixCount > 0) {

        if (activeLength == 0)
            activeEdge = pos; //APCFALZ

        // There is no outgoing edge starting with
        // activeEdge from activeNode
        if (activeNode->children] == NULL)
        {
            //Extension Rule 2 (A new leaf edge gets created)
            activeNode->children] =
                newNode(pos, &leafEnd);

            /*A new leaf edge is created in above line starting
             from an existng node (the current activeNode), and
             if there is any internal node waiting for it's suffix
             link get reset, point the suffix link from that last
             internal node to current activeNode. Then set lastNewNode
             to NULL indicating no more node waiting for suffix link
             reset.*/
            if (lastNewNode != NULL)
            {
                lastNewNode->suffixLink = activeNode;
                lastNewNode = NULL;
            }
        }
    }
}
```

```

// There is an outgoing edge starting with activeEdge
// from activeNode
else
{
    // Get the next node at the end of edge starting
    // with activeEdge
    Node *next = activeNode->children];
    if (walkDown(next))//Do walkdown
    {
        //Start from next node (the new activeNode)
        continue;
    }
    /*Extension Rule 3 (current character being processed
     is already on the edge)*/
    if (text[next->start + activeLength] == text[pos])
    {
        //If a newly created node waiting for it's
        //suffix link to be set, then set suffix link
        //of that waiting node to current active node
        if(lastNewNode != NULL && activeNode != root)
        {
            lastNewNode->suffixLink = activeNode;
            lastNewNode = NULL;
        }

        //APCFER3
        activeLength++;
        /*STOP all further processing in this phase
         and move on to next phase*/
        break;
    }

    /*We will be here when activePoint is in middle of
     the edge being traversed and current character
     being processed is not on the edge (we fall off
     the tree). In this case, we add a new internal node
     and a new leaf edge going out of that new node. This
     is Extension Rule 2, where a new leaf edge and a new
     internal node get created*/
    splitEnd = (int*) malloc(sizeof(int));
    *splitEnd = next->start + activeLength - 1;

    //New internal node
    Node *split = newNode(next->start, splitEnd);
    activeNode->children] = split;

    //New leaf coming out of new internal node
    split->children] = newNode(pos, &leafEnd);
}

```

```
next->start += activeLength;
split->children] = next;

/*We got a new internal node here. If there is any
internal node created in last extensions of same
phase which is still waiting for it's suffix link
reset, do it now.*/
if (lastNewNode != NULL)
{
/*suffixLink of lastNewNode points to current newly
created internal node*/
    lastNewNode->suffixLink = split;
}

/*Make the current newly created internal node waiting
for it's suffix link reset (which is pointing to root
at present). If we come across any other internal node
(existing or newly created) in next extension of same
phase, when a new leaf edge gets added (i.e. when
Extension Rule 2 applies is any of the next extension
of same phase) at that point, suffixLink of this node
will point to that internal node.*/
    lastNewNode = split;
}

/* One suffix got added in tree, decrement the count of
suffixes yet to be added.*/
remainingSuffixCount--;
if (activeNode == root && activeLength > 0) //APCFER2C1
{
    activeLength--;
    activeEdge = pos - remainingSuffixCount + 1;
}
else if (activeNode != root) //APCFER2C2
{
    activeNode = activeNode->suffixLink;
}
}

void print(int i, int j)
{
    int k;
    for (k=i; k<=j; k++)
        printf("%c", text[k]);
}

//Print the suffix tree as well along with setting suffix index
```

```
//So tree will be printed in DFS manner
//Each edge along with it's suffix index will be printed
void setSuffixIndexByDFS(Node *n, int labelHeight)
{
    if (n == NULL)    return;

    if (n->start != -1) //A non-root node
    {
        //Print the label on edge from parent to current node
        //Uncomment below line to print suffix tree
        // print(n->start, *(n->end));
    }
    int leaf = 1;
    int i;
    for (i = 0; i < MAX_CHAR; i++)
    {
        if (n->children[i] != NULL)
        {
            //Uncomment below two lines to print suffix index
            // if (leaf == 1 && n->start != -1)
            //     printf(" [%d]\n", n->suffixIndex);

            //Current node is not a leaf as it has outgoing
            //edges from it.
            leaf = 0;
            setSuffixIndexByDFS(n->children[i], labelHeight +
                                edgeLength(n->children[i]));
        }
    }
    if (leaf == 1)
    {
        n->suffixIndex = size - labelHeight;
        //Uncomment below line to print suffix index
        //printf(" [%d]\n", n->suffixIndex);
    }
}

void freeSuffixTreeByPostOrder(Node *n)
{
    if (n == NULL)
        return;
    int i;
    for (i = 0; i < MAX_CHAR; i++)
    {
        if (n->children[i] != NULL)
        {
            freeSuffixTreeByPostOrder(n->children[i]);
        }
    }
}
```

```

        }
        if (n->suffixIndex == -1)
            free(n->end);
        free(n);
    }

/*Build the suffix tree and print the edge labels along with
suffixIndex. suffixIndex for leaf edges will be >= 0 and
for non-leaf edges will be -1*/
void buildSuffixTree()
{
    size = strlen(text);
    int i;
    rootEnd = (int*) malloc(sizeof(int));
    *rootEnd = - 1;

    /*Root is a special node with start and end indices as -1,
    as it has no parent from where an edge comes to root*/
    root = newNode(-1, rootEnd);

    activeNode = root; //First activeNode will be root
    for (i=0; i<size; i++)
        extendSuffixTree(i);
    int labelHeight = 0;
    setSuffixIndexByDFS(root, labelHeight);
}

int traverseEdge(char *str, int idx, int start, int end)
{
    int k = 0;
    //Traverse the edge with character by character matching
    for(k=start; k<=end && str[idx] != '\0'; k++, idx++)
    {
        if(text[k] != str[idx])
            return -1; // no match
    }
    if(str[idx] == '\0')
        return 1; // match
    return 0; // more characters yet to match
}

int doTraversalToCountLeaf(Node *n)
{
    if(n == NULL)
        return 0;
    if(n->suffixIndex > -1)
    {
        printf("\nFound at position: %d", n->suffixIndex);
    }
}

```

```

        return 1;
    }
    int count = 0;
    int i = 0;
    for (i = 0; i < MAX_CHAR; i++)
    {
        if(n->children[i] != NULL)
        {
            count += doTraversalToCountLeaf(n->children[i]);
        }
    }
    return count;
}

int countLeaf(Node *n)
{
    if(n == NULL)
        return 0;
    return doTraversalToCountLeaf(n);
}

int doTraversal(Node *n, char* str, int idx)
{
    if(n == NULL)
    {
        return -1; // no match
    }
    int res = -1;
    //If node n is not root node, then traverse edge
    //from node n's parent to node n.
    if(n->start != -1)
    {
        res = traverseEdge(str, idx, n->start, *(n->end));
        if(res == -1) //no match
            return -1;
        if(res == 1) //match
        {
            if(n->suffixIndex > -1)
                printf("\nsubstring count: 1 and position: %d",
                       n->suffixIndex);
            else
                printf("\nsubstring count: %d", countLeaf(n));
            return 1;
        }
    }
    //Get the character index to search
    idx = idx + edgeLength(n);
    //If there is an edge from node n going out
}

```

```

//with current character str[idx], travrse that edge
if(n->children[str[idx]] != NULL)
    return doTraversal(n->children[str[idx]], str, idx);
else
    return -1; // no match
}

void checkForSubString(char* str)
{
    int res = doTraversal(root, str, 0);
    if(res == 1)
        printf("\nPattern <%s> is a Substring\n", str);
    else
        printf("\nPattern <%s> is NOT a Substring\n", str);
}

// driver program to test above functions
int main(int argc, char *argv[])
{
    strcpy(text, "GEEKSFORGEEKS$");
    buildSuffixTree();
    printf("Text: GEEKSFORGEEKS, Pattern to search: GEEKS");
    checkForSubString("GEEKS");
    printf("\n\nText: GEEKSFORGEEKS, Pattern to search: GEEK1");
    checkForSubString("GEEK1");
    printf("\n\nText: GEEKSFORGEEKS, Pattern to search: FOR");
    checkForSubString("FOR");
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    strcpy(text, "AABAACAAADAABAAABAA$");
    buildSuffixTree();
    printf("\n\nText: AABAACAAADAABAAABAA, Pattern to search: AABA");
    checkForSubString("AABA");
    printf("\n\nText: AABAACAAADAABAAABAA, Pattern to search: AA");
    checkForSubString("AA");
    printf("\n\nText: AABAACAAADAABAAABAA, Pattern to search: AAE");
    checkForSubString("AAE");
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    strcpy(text, "AAAAAAA$");
    buildSuffixTree();
    printf("\n\nText: AAAAAAAA, Pattern to search: AAAA");
    checkForSubString("AAAA");
    printf("\n\nText: AAAAAAAA, Pattern to search: AA");
    checkForSubString("AA");
    printf("\n\nText: AAAAAAAA, Pattern to search: A");
}

```

```
checkForSubString("A");
printf("\n\nText: AAAAAAAA, Pattern to search: AB");
checkForSubString("AB");
//Free the dynamically allocated memory
freeSuffixTreeByPostOrder(root);

return 0;
}
```

Output:

```
Text: GEEKSFORGEEKS, Pattern to search: GEEKS
Found at position: 8
Found at position: 0
substring count: 2
Pattern <GEEKS> is a Substring
```

```
Text: GEEKSFORGEEKS, Pattern to search: GEEK1
Pattern <GEEK1> is NOT a Substring
```

```
Text: GEEKSFORGEEKS, Pattern to search: FOR
substring count: 1 and position: 5
Pattern <FOR> is a Substring
```

```
Text: AABAACAAADAABAAABAA, Pattern to search: AABA
Found at position: 13
Found at position: 9
Found at position: 0
substring count: 3
Pattern <AABA> is a Substring
```

```
Text: AABAACAAADAABAAABAA, Pattern to search: AA
Found at position: 16
Found at position: 12
Found at position: 13
Found at position: 9
Found at position: 0
Found at position: 3
Found at position: 6
substring count: 7
Pattern <AA> is a Substring
```

Text: AABAACAAADAABAAABAA, Pattern to search: AAE
Pattern <AAE> is NOT a Substring

Text: AAAAAAAA, Pattern to search: AAAA
Found at position: 5
Found at position: 4
Found at position: 3
Found at position: 2
Found at position: 1
Found at position: 0
substring count: 6
Pattern <AAAA> is a Substring

Text: AAAAAAAA, Pattern to search: AA
Found at position: 7
Found at position: 6
Found at position: 5
Found at position: 4
Found at position: 3
Found at position: 2
Found at position: 1
Found at position: 0
substring count: 8
Pattern <AA> is a Substring

Text: AAAAAAAA, Pattern to search: A
Found at position: 8
Found at position: 7
Found at position: 6
Found at position: 5
Found at position: 4
Found at position: 3
Found at position: 2
Found at position: 1
Found at position: 0
substring count: 9
Pattern <A> is a Substring

Text: AAAAAAAA, Pattern to search: AB
Pattern <AB> is NOT a Substring

Ukkonen's Suffix Tree Construction takes $O(N)$ time and space to build suffix tree for a string of length N and after that, traversal for substring check takes $O(M)$ for a pattern of length M and then if there are Z occurrences of the pattern, it will take $O(Z)$ to find indices

of all those Z occurrences.

Overall pattern complexity is linear: $O(M + Z)$.

A bit more detailed analysis

How many internal nodes will there be in a suffix tree of string of length N ??

Answer: $N-1$ (Why ??)

There will be N suffixes in a string of length N .

Each suffix will have one leaf.

So a suffix tree of string of length N will have N leaves.

As each internal node has at least 2 children, an N -leaf suffix tree has at most $N-1$ internal nodes.

If a pattern occurs Z times in string, means it will be part of Z suffixes, so there will be Z leaves below in point (internal node and in between edge) where pattern match ends in tree and so subtree with Z leaves below that point will have $Z-1$ internal nodes. A tree with Z leaves can be traversed in $O(Z)$ time.

Overall pattern complexity is linear: $O(M + Z)$.

For a given pattern, Z (the number of occurrences) can be atmost N .

So worst case complexity can be: $O(M + N)$ if Z is close/equal to N (A tree traversal with N nodes take $O(N)$ time).

Followup questions:

1. Check if a pattern is prefix of a text?
2. Check if a pattern is suffix of a text?

We have published following more articles on suffix tree applications:

- [Suffix Tree Application 1 – Substring Check](#)
- [Suffix Tree Application 3 – Longest Repeated Substring](#)
- [Suffix Tree Application 4 – Build Linear Time Suffix Array](#)
- [Generalized Suffix Tree 1](#)
- [Suffix Tree Application 5 – Longest Common Substring](#)
- [Suffix Tree Application 6 – Longest Palindromic Substring](#)

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

Source

<https://www.geeksforgeeks.org/suffix-tree-application-2-searching-all-patterns/>

Chapter 186

Suffix Tree Application 3 – Longest Repeated Substring

Suffix Tree Application 3 - Longest Repeated Substring - GeeksforGeeks

Given a text string, find [Longest Repeated Substring](#) in the text. If there are more than one Longest Repeated Substrings, get any one of them.

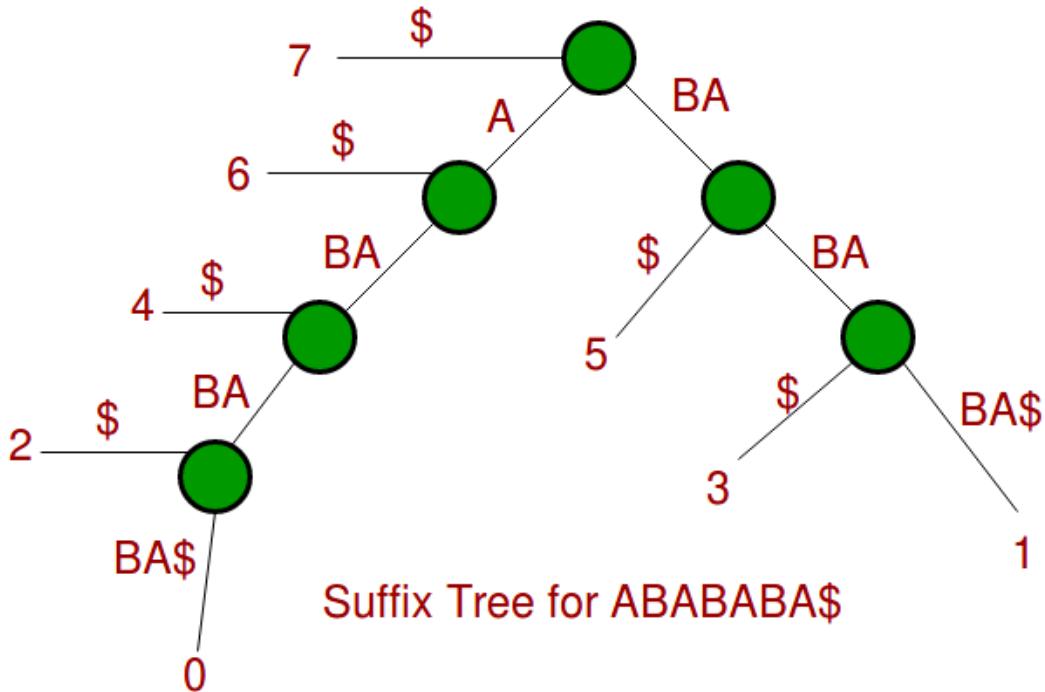
Longest Repeated Substring in GEEKSFORGEEEKS is: GEEKS
Longest Repeated Substring in AAAAAAAA is: AAAAAAAA
Longest Repeated Substring in ABCDEFG is: No repeated substring
Longest Repeated Substring in ABABABA is: ABABA
Longest Repeated Substring in ATCGATCGA is: ATCGA
Longest Repeated Substring in banana is: ana
Longest Repeated Substring in abcpqrabpq is: ab (pq is another LRS here)

This problem can be solved by different approaches with varying time and space complexities. Here we will discuss Suffix Tree approach (3rd Suffix Tree Application). Other approaches will be discussed soon.

As a prerequisite, we must know how to build a suffix tree in one or the other way. Here we will build suffix tree using Ukkonen's Algorithm, discussed already as below:

- [Ukkonen's Suffix Tree Construction – Part 1](#)
- [Ukkonen's Suffix Tree Construction – Part 2](#)
- [Ukkonen's Suffix Tree Construction – Part 3](#)
- [Ukkonen's Suffix Tree Construction – Part 4](#)
- [Ukkonen's Suffix Tree Construction – Part 5](#)
- [Ukkonen's Suffix Tree Construction – Part 6](#)

Lets look at following figure:



This is suffix tree for string “ABABABA\$”.

In this string, following substrings are repeated:

A, B, AB, BA, ABA, BAB, ABAB, BABA, ABABA

And Longest Repeated Substring is ABABA.

In a suffix tree, one node can't have more than one outgoing edge starting with same character, and so if there are repeated substring in the text, they will share on same path and that path in suffix tree will go through one or more internal node(s) down the tree (below the point where substring ends on that path).

In above figure, we can see that

- Path with Substring “A” has three internal nodes down the tree
- Path with Substring “AB” has two internal nodes down the tree
- Path with Substring “ABA” has two internal nodes down the tree
- Path with Substring “ABAB” has one internal node down the tree
- Path with Substring “ABABA” has one internal node down the tree
- Path with Substring “B” has two internal nodes down the tree
- Path with Substring “BA” has two internal nodes down the tree
- Path with Substring “BAB” has one internal node down the tree
- Path with Substring “BABA” has one internal node down the tree

All above substrings are repeated.

Substrings ABABAB, ABABABA, BABAB, BABABA have no internal node down the tree (after the point where substring end on the path), and so these are not repeated.

Can you see how to find longest repeated substring ??

We can see in figure that, longest repeated substring will end at the internal node which is farthest from the root (i.e. deepest node in the tree), because length of substring is the path label length from root to that internal node.

So finding longest repeated substring boils down to finding the deepest node in suffix tree and then get the path label from root to that deepest internal node.

```
// A C program to implement Ukkonen's Suffix Tree Construction
// And then find Longest Repeated Substring
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define MAX_CHAR 256

struct SuffixTreeNode {
    struct SuffixTreeNode *children[MAX_CHAR];

    //pointer to other node via suffix link
    struct SuffixTreeNode *suffixLink;

    /*(start, end) interval specifies the edge, by which the
     node is connected to its parent node. Each edge will
     connect two nodes, one parent and one child, and
     (start, end) interval of a given edge will be stored
     in the child node. Lets say there are two nodes A and B
     connected by an edge with indices (5, 8) then this
     indices (5, 8) will be stored in node B. */
    int start;
    int *end;

    /*for leaf nodes, it stores the index of suffix for
     the path from root to leaf*/
    int suffixIndex;
};

typedef struct SuffixTreeNode Node;

char text[100]; //Input string
Node *root = NULL; //Pointer to root node

/*lastNewNode will point to newly created internal node,
 waiting for it's suffix link to be set, which might get
 a new suffix link (other than root) in next extension of
```

```

same phase. lastNewNode will be set to NULL when last
newly created internal node (if there is any) got it's
suffix link reset to new internal node created in next
extension of same phase. */
Node *lastNewNode = NULL;
Node *activeNode = NULL;

/*activeEdge is represeted as input string character
 index (not the character itself)*/
int activeEdge = -1;
int activeLength = 0;

// remainingSuffixCount tells how many suffixes yet to
// be added in tree
int remainingSuffixCount = 0;
int leafEnd = -1;
int *rootEnd = NULL;
int *splitEnd = NULL;
int size = -1; //Length of input string

Node *newNode(int start, int *end)
{
    Node *node =(Node*) malloc(sizeof(Node));
    int i;
    for (i = 0; i < MAX_CHAR; i++)
        node->children[i] = NULL;

    /*For root node, suffixLink will be set to NULL
    For internal nodes, suffixLink will be set to root
    by default in current extension and may change in
    next extension*/
    node->suffixLink = root;
    node->start = start;
    node->end = end;

    /*suffixIndex will be set to -1 by default and
    actual suffix index will be set later for leaves
    at the end of all phases*/
    node->suffixIndex = -1;
    return node;
}

int edgeLength(Node *n) {
    if(n == root)
        return 0;
    return *(n->end) - (n->start) + 1;
}

```

```

int walkDown(Node *currNode)
{
    /*activePoint change for walk down (APCFWD) using
     Skip/Count Trick (Trick 1). If activeLength is greater
     than current edge length, set next internal node as
     activeNode and adjust activeEdge and activeLength
     accordingly to represent same activePoint*/
    if (activeLength >= edgeLength(currNode))
    {
        activeEdge += edgeLength(currNode);
        activeLength -= edgeLength(currNode);
        activeNode = currNode;
        return 1;
    }
    return 0;
}

void extendSuffixTree(int pos)
{
    /*Extension Rule 1, this takes care of extending all
     leaves created so far in tree*/
    leafEnd = pos;

    /*Increment remainingSuffixCount indicating that a
     new suffix added to the list of suffixes yet to be
     added in tree*/
    remainingSuffixCount++;

    /*set lastNewNode to NULL while starting a new phase,
     indicating there is no internal node waiting for
     it's suffix link reset in current phase*/
    lastNewNode = NULL;

    //Add all suffixes (yet to be added) one by one in tree
    while(remainingSuffixCount > 0) {

        if (activeLength == 0)
            activeEdge = pos; //APCFALZ

        // There is no outgoing edge starting with
        // activeEdge from activeNode
        if (activeNode->children] == NULL)
        {
            //Extension Rule 2 (A new leaf edge gets created)
            activeNode->children] =
                newNode(pos, &leafEnd);

            /*A new leaf edge is created in above line starting

```

```

from an existng node (the current activeNode), and
if there is any internal node waiting for it's suffix
link get reset, point the suffix link from that last
internal node to current activeNode. Then set lastNewNode
to NULL indicating no more node waiting for suffix link
reset.*/
if (lastNewNode != NULL)
{
    lastNewNode->suffixLink = activeNode;
    lastNewNode = NULL;
}
}
// There is an outgoing edge starting with activeEdge
// from activeNode
else
{
    // Get the next node at the end of edge starting
    // with activeEdge
    Node *next = activeNode->children];
    if (walkDown(next))//Do walkdown
    {
        //Start from next node (the new activeNode)
        continue;
    }
    /*Extension Rule 3 (current character being processed
     is already on the edge)*/
    if (text[next->start + activeLength] == text[pos])
    {
        //If a newly created node waiting for it's
        //suffix link to be set, then set suffix link
        //of that waiting node to curent active node
        if(lastNewNode != NULL && activeNode != root)
        {
            lastNewNode->suffixLink = activeNode;
            lastNewNode = NULL;
        }

        //APCFER3
        activeLength++;
        /*STOP all further processing in this phase
         and move on to next phase*/
        break;
    }

    /*We will be here when activePoint is in middle of
     the edge being traversed and current character
     being processed is not on the edge (we fall off
     the tree). In this case, we add a new internal node

```

```

        and a new leaf edge going out of that new node. This
        is Extension Rule 2, where a new leaf edge and a new
        internal node get created*/
splitEnd = (int*) malloc(sizeof(int));
*splitEnd = next->start + activeLength - 1;

//New internal node
Node *split = newNode(next->start, splitEnd);
activeNode->children] = split;

//New leaf coming out of new internal node
split->children] = newNode(pos, &leafEnd);
next->start += activeLength;
split->children] = next;

/*We got a new internal node here. If there is any
internal node created in last extensions of same
phase which is still waiting for it's suffix link
reset, do it now.*/
if (lastNewNode != NULL)
{
/*suffixLink of lastNewNode points to current newly
created internal node*/
    lastNewNode->suffixLink = split;
}

/*Make the current newly created internal node waiting
for it's suffix link reset (which is pointing to root
at present). If we come across any other internal node
(existing or newly created) in next extension of same
phase, when a new leaf edge gets added (i.e. when
Extension Rule 2 applies is any of the next extension
of same phase) at that point, suffixLink of this node
will point to that internal node.*/
lastNewNode = split;
}

/* One suffix got added in tree, decrement the count of
suffixes yet to be added.*/
remainingSuffixCount--;
if (activeNode == root && activeLength > 0) //APCFER2C1
{
    activeLength--;
    activeEdge = pos - remainingSuffixCount + 1;
}
else if (activeNode != root) //APCFER2C2
{
    activeNode = activeNode->suffixLink;
}

```

```

        }
    }
}

void print(int i, int j)
{
    int k;
    for (k=i; k<=j; k++)
        printf("%c", text[k]);
}

//Print the suffix tree as well along with setting suffix index
//So tree will be printed in DFS manner
//Each edge along with it's suffix index will be printed
void setSuffixIndexByDFS(Node *n, int labelHeight)
{
    if (n == NULL)    return;

    if (n->start != -1) //A non-root node
    {
        //Print the label on edge from parent to current node
        //Uncomment below line to print suffix tree
        // printf(n->start, *(n->end));
    }
    int leaf = 1;
    int i;
    for (i = 0; i < MAX_CHAR; i++)
    {
        if (n->children[i] != NULL)
        {
            //Uncomment below two lines to print suffix index
            // if (leaf == 1 && n->start != -1)
            //     printf(" [%d]\n", n->suffixIndex);

            //Current node is not a leaf as it has outgoing
            //edges from it.
            leaf = 0;
            setSuffixIndexByDFS(n->children[i], labelHeight +
                                edgeLength(n->children[i]));
        }
    }
    if (leaf == 1)
    {
        n->suffixIndex = size - labelHeight;
        //Uncomment below line to print suffix index
        //printf(" [%d]\n", n->suffixIndex);
    }
}

```

```

void freeSuffixTreeByPostOrder(Node *n)
{
    if (n == NULL)
        return;
    int i;
    for (i = 0; i < MAX_CHAR; i++)
    {
        if (n->children[i] != NULL)
        {
            freeSuffixTreeByPostOrder(n->children[i]);
        }
    }
    if (n->suffixIndex == -1)
        free(n->end);
    free(n);
}

/*Build the suffix tree and print the edge labels along with
suffixIndex. suffixIndex for leaf edges will be >= 0 and
for non-leaf edges will be -1*/
void buildSuffixTree()
{
    size = strlen(text);
    int i;
    rootEnd = (int*) malloc(sizeof(int));
    *rootEnd = - 1;

    /*Root is a special node with start and end indices as -1,
    as it has no parent from where an edge comes to root*/
    root = newNode(-1, rootEnd);

    activeNode = root; //First activeNode will be root
    for (i=0; i<size; i++)
        extendSuffixTree(i);
    int labelHeight = 0;
    setSuffixIndexByDFS(root, labelHeight);
}

void doTraversal(Node *n, int labelHeight, int* maxHeight,
int* substringStartIndex)
{
    if(n == NULL)
    {
        return;
    }
    int i=0;
    if(n->suffixIndex == -1) //If it is internal node

```

```

{
    for (i = 0; i < MAX_CHAR; i++)
    {
        if(n->children[i] != NULL)
        {
            doTraversal(n->children[i], labelHeight +
                        edgeLength(n->children[i]), maxHeight,
                        substringstartIndex);
        }
    }
    else if(n->suffixIndex > -1 &&
             (*maxHeight < labelHeight - edgeLength(n)))
    {
        *maxHeight = labelHeight - edgeLength(n);
        *substringstartIndex = n->suffixIndex;
    }
}

void getLongestRepeatedSubstring()
{
    int maxHeight = 0;
    int substringstartIndex = 0;
    doTraversal(root, 0, &maxHeight, &substringstartIndex);
//    printf("maxHeight %d, substringstartIndex %d\n", maxHeight,
//           substringstartIndex);
    printf("Longest Repeated Substring in %s is: ", text);
    int k;
    for (k=0; k<maxHeight; k++)
        printf("%c", text[k + substringstartIndex]);
    if(k == 0)
        printf("No repeated substring");
    printf("\n");
}

// driver program to test above functions
int main(int argc, char *argv[])
{
    strcpy(text, "GEEKSFORGEEKS$");
    buildSuffixTree();
    getLongestRepeatedSubstring();
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    strcpy(text, "AAAAAAA$");
    buildSuffixTree();
    getLongestRepeatedSubstring();
    //Free the dynamically allocated memory
}

```

```
freeSuffixTreeByPostOrder(root);

strcpy(text, "ABCDEFG$");
buildSuffixTree();
getLongestRepeatedSubstring();
//Free the dynamically allocated memory
freeSuffixTreeByPostOrder(root);

strcpy(text, "ABABABA$");
buildSuffixTree();
getLongestRepeatedSubstring();
//Free the dynamically allocated memory
freeSuffixTreeByPostOrder(root);

strcpy(text, "ATCGATCGA$");
buildSuffixTree();
getLongestRepeatedSubstring();
//Free the dynamically allocated memory
freeSuffixTreeByPostOrder(root);

strcpy(text, "banana$");
buildSuffixTree();
getLongestRepeatedSubstring();
//Free the dynamically allocated memory
freeSuffixTreeByPostOrder(root);

strcpy(text, "abcpqrabpqpq$");
buildSuffixTree();
getLongestRepeatedSubstring();
//Free the dynamically allocated memory
freeSuffixTreeByPostOrder(root);

strcpy(text, "pqrpqabab$");
buildSuffixTree();
getLongestRepeatedSubstring();
//Free the dynamically allocated memory
freeSuffixTreeByPostOrder(root);

return 0;
}
```

Output:

```
Longest Repeated Substring in GEEKSFORGEEKS$ is: GEEKS
Longest Repeated Substring in AAAAAAAA$ is: AAAAAAAA
Longest Repeated Substring in ABCDEFG$ is: No repeated substring
Longest Repeated Substring in ABABABA$ is: ABABA
```

Longest Repeated Substring in ATCGATCGA\$ is: ATCGA

Longest Repeated Substring in banana\$ is: ana

Longest Repeated Substring in abcpqrabpq\$ is: ab

Longest Repeated Substring in pqrpqbab\$ is: ab

In case of multiple LRS (As we see in last two test cases), this implementation prints the LRS which comes 1st lexicographically.

Ukkonen's Suffix Tree Construction takes O(N) time and space to build suffix tree for a string of length N and after that finding deepest node will take O(N). So it is linear in time and space.

Followup questions:

1. Find all repeated substrings in given text
2. Find all unique substrings in given text
3. Find all repeated substrings of a given length
4. Find all unique substrings of a given length
5. In case of multiple LRS in text, find the one which occurs most number of times

All these problems can be solved in linear time with few changes in above implementation.

We have published following more articles on suffix tree applications:

- [Suffix Tree Application 1 – Substring Check](#)
- [Suffix Tree Application 2 – Searching All Patterns](#)
- [Suffix Tree Application 4 – Build Linear Time Suffix Array](#)
- [Generalized Suffix Tree 1](#)
- [Suffix Tree Application 5 – Longest Common Substring](#)
- [Suffix Tree Application 6 – Longest Palindromic Substring](#)

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

Source

<https://www.geeksforgeeks.org/suffix-tree-application-3-longest-repeated-substring/>

Chapter 187

Suffix Tree Application 4 – Build Linear Time Suffix Array

Suffix Tree Application 4 - Build Linear Time Suffix Array - GeeksforGeeks

Given a string, build it's [Suffix Array](#)

We have already discussed following two ways of building suffix array:

- [Naive O\(\$n^2 \log n\$ \) algorithm](#)
- [Enhanced O\(\$n \log n\$ \) algorithm](#)

Please go through these to have the basic understanding.

Here we will see how to build suffix array in linear time using suffix tree.

As a prerequisite, we must know how to build a suffix tree in one or the other way.

Here we will build suffix tree using Ukkonen's Algorithm, discussed already as below:

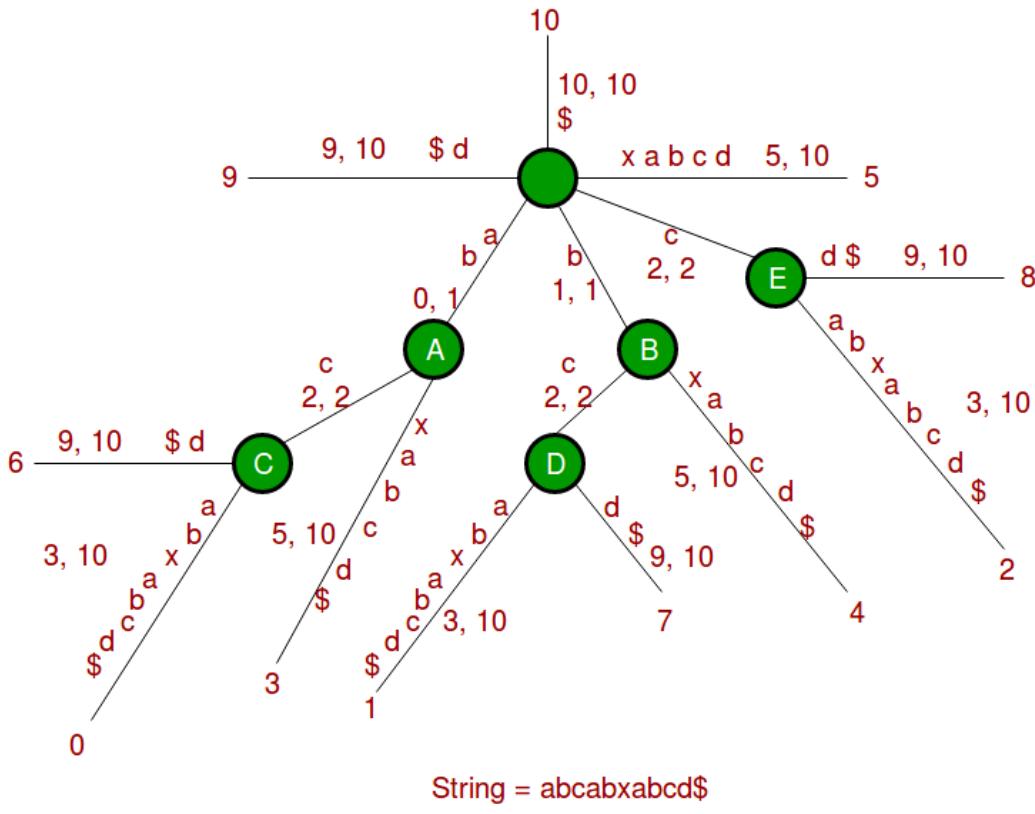
- [Ukkonen's Suffix Tree Construction – Part 1](#)
- [Ukkonen's Suffix Tree Construction – Part 2](#)
- [Ukkonen's Suffix Tree Construction – Part 3](#)
- [Ukkonen's Suffix Tree Construction – Part 4](#)
- [Ukkonen's Suffix Tree Construction – Part 5](#)
- [Ukkonen's Suffix Tree Construction – Part 6](#)

Lets consider string abcabxabcd.

It's suffix array would be:

0 6 3 1 7 4 2 8 9 5

Lets look at following figure:



If we do a DFS traversal, visiting edges in lexicographic order (we have been doing the same traversal in other Suffix Tree Application articles as well) and print suffix indices on leaves, we will get following:

10 0 6 3 1 7 4 2 8 9 5

"\$" is lexicographically lesser than [a-zA-Z].

The suffix index 10 corresponds to edge with "\$" label.

Except this 1st suffix index, the sequence of all other numbers gives the suffix array of the string.

So if we have a suffix tree of the string, then to get it's suffix array, we just need to do a lexicographic order DFS traversal and store all the suffix indices in resultant suffix array, except the very 1st suffix index.

```
// A C program to implement Ukkonen's Suffix Tree Construction
// And then create suffix array in linear time
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
```

```
#define MAX_CHAR 256

struct SuffixTreeNode {
    struct SuffixTreeNode *children[MAX_CHAR];

    //pointer to other node via suffix link
    struct SuffixTreeNode *suffixLink;

    /*(start, end) interval specifies the edge, by which the
     node is connected to its parent node. Each edge will
     connect two nodes, one parent and one child, and
     (start, end) interval of a given edge will be stored
     in the child node. Lets say there are two nodes A and B
     connected by an edge with indices (5, 8) then this
     indices (5, 8) will be stored in node B. */
    int start;
    int *end;

    /*for leaf nodes, it stores the index of suffix for
     the path from root to leaf*/
    int suffixIndex;
};

typedef struct SuffixTreeNode Node;

char text[100]; //Input string
Node *root = NULL; //Pointer to root node

/*lastNewNode will point to newly created internal node,
   waiting for it's suffix link to be set, which might get
   a new suffix link (other than root) in next extension of
   same phase. lastNewNode will be set to NULL when last
   newly created internal node (if there is any) got it's
   suffix link reset to new internal node created in next
   extension of same phase.*/
Node *lastNewNode = NULL;
Node *activeNode = NULL;

/*activeEdge is represented as input string character
   index (not the character itself)*/
int activeEdge = -1;
int activeLength = 0;

// remainingSuffixCount tells how many suffixes yet to
// be added in tree
int remainingSuffixCount = 0;
int leafEnd = -1;
int *rootEnd = NULL;
```

```

int *splitEnd = NULL;
int size = -1; //Length of input string

Node *newNode(int start, int *end)
{
    Node *node =(Node*) malloc(sizeof(Node));
    int i;
    for (i = 0; i < MAX_CHAR; i++)
        node->children[i] = NULL;

    /*For root node, suffixLink will be set to NULL
    For internal nodes, suffixLink will be set to root
    by default in current extension and may change in
    next extension*/
    node->suffixLink = root;
    node->start = start;
    node->end = end;

    /*suffixIndex will be set to -1 by default and
     actual suffix index will be set later for leaves
     at the end of all phases*/
    node->suffixIndex = -1;
    return node;
}

int edgeLength(Node *n) {
    if(n == root)
        return 0;
    return *(n->end) - (n->start) + 1;
}

int walkDown(Node *currNode)
{
    /*activePoint change for walk down (APCFWD) using
     Skip/Count Trick (Trick 1). If activeLength is greater
     than current edge length, set next internal node as
     activeNode and adjust activeEdge and activeLength
     accordingly to represent same activePoint*/
    if (activeLength >= edgeLength(currNode))
    {
        activeEdge += edgeLength(currNode);
        activeLength -= edgeLength(currNode);
        activeNode = currNode;
        return 1;
    }
    return 0;
}

```

```
void extendSuffixTree(int pos)
{
    /*Extension Rule 1, this takes care of extending all
     leaves created so far in tree*/
    leafEnd = pos;

    /*Increment remainingSuffixCount indicating that a
     new suffix added to the list of suffixes yet to be
     added in tree*/
    remainingSuffixCount++;

    /*set lastNewNode to NULL while starting a new phase,
     indicating there is no internal node waiting for
     it's suffix link reset in current phase*/
    lastNewNode = NULL;

    //Add all suffixes (yet to be added) one by one in tree
    while(remainingSuffixCount > 0) {

        if (activeLength == 0)
            activeEdge = pos; //APCFALZ

        // There is no outgoing edge starting with
        // activeEdge from activeNode
        if (activeNode->children] == NULL)
        {
            //Extension Rule 2 (A new leaf edge gets created)
            activeNode->children] =
                newNode(pos, &leafEnd);

            /*A new leaf edge is created in above line starting
             from an existng node (the current activeNode), and
             if there is any internal node waiting for it's suffix
             link get reset, point the suffix link from that last
             internal node to current activeNode. Then set lastNewNode
             to NULL indicating no more node waiting for suffix link
             reset.*/
            if (lastNewNode != NULL)
            {
                lastNewNode->suffixLink = activeNode;
                lastNewNode = NULL;
            }
        }
        // There is an outgoing edge starting with activeEdge
        // from activeNode
        else
        {
            // Get the next node at the end of edge starting
```

```

// with activeEdge
Node *next = activeNode->children];
if (walkDown(next))//Do walkdown
{
    //Start from next node (the new activeNode)
    continue;
}
/*Extension Rule 3 (current character being processed
   is already on the edge)*/
if (text[next->start + activeLength] == text[pos])
{
    //If a newly created node waiting for it's
    //suffix link to be set, then set suffix link
    //of that waiting node to current active node
    if(lastNewNode != NULL && activeNode != root)
    {
        lastNewNode->suffixLink = activeNode;
        lastNewNode = NULL;
    }

    //APCFER3
    activeLength++;
    /*STOP all further processing in this phase
       and move on to next phase*/
    break;
}

/*We will be here when activePoint is in middle of
   the edge being traversed and current character
   being processed is not on the edge (we fall off
   the tree). In this case, we add a new internal node
   and a new leaf edge going out of that new node. This
   is Extension Rule 2, where a new leaf edge and a new
   internal node get created*/
splitEnd = (int*) malloc(sizeof(int));
*splitEnd = next->start + activeLength - 1;

//New internal node
Node *split = newNode(next->start, splitEnd);
activeNode->children] = split;

//New leaf coming out of new internal node
split->children] = newNode(pos, &leafEnd);
next->start += activeLength;
split->children] = next;

/*We got a new internal node here. If there is any
   internal node created in last extensions of same

```

```

        phase which is still waiting for it's suffix link
        reset, do it now.*/
    if (lastNewNode != NULL)
    {
        /*suffixLink of lastNewNode points to current newly
         created internal node*/
        lastNewNode->suffixLink = split;
    }

    /*Make the current newly created internal node waiting
     for it's suffix link reset (which is pointing to root
     at present). If we come across any other internal node
     (existing or newly created) in next extension of same
     phase, when a new leaf edge gets added (i.e. when
     Extension Rule 2 applies is any of the next extension
     of same phase) at that point, suffixLink of this node
     will point to that internal node.*/
    lastNewNode = split;
}

/* One suffix got added in tree, decrement the count of
   suffixes yet to be added.*/
remainingSuffixCount--;
if (activeNode == root && activeLength > 0) //APCFER2C1
{
    activeLength--;
    activeEdge = pos - remainingSuffixCount + 1;
}
else if (activeNode != root) //APCFER2C2
{
    activeNode = activeNode->suffixLink;
}
}

void print(int i, int j)
{
    int k;
    for (k=i; k<=j; k++)
        printf("%c", text[k]);
}

//Print the suffix tree as well along with setting suffix index
//So tree will be printed in DFS manner
//Each edge along with it's suffix index will be printed
void setSuffixIndexByDFS(Node *n, int labelHeight)
{
    if (n == NULL)  return;
}

```

```
if (n->start != -1) //A non-root node
{
    //Print the label on edge from parent to current node
    //Uncomment below line to print suffix tree
    // print(n->start, *(n->end));
}
int leaf = 1;
int i;
for (i = 0; i < MAX_CHAR; i++)
{
    if (n->children[i] != NULL)
    {
        //Uncomment below two lines to print suffix index
        // if (leaf == 1 && n->start != -1)
        //     printf(" [%d]\n", n->suffixIndex);

        //Current node is not a leaf as it has outgoing
        //edges from it.
        leaf = 0;
        setSuffixIndexByDFS(n->children[i], labelHeight +
                           edgeLength(n->children[i]));
    }
}
if (leaf == 1)
{
    n->suffixIndex = size - labelHeight;
    //Uncomment below line to print suffix index
    //printf(" [%d]\n", n->suffixIndex);
}
}

void freeSuffixTreeByPostOrder(Node *n)
{
    if (n == NULL)
        return;
    int i;
    for (i = 0; i < MAX_CHAR; i++)
    {
        if (n->children[i] != NULL)
        {
            freeSuffixTreeByPostOrder(n->children[i]);
        }
    }
    if (n->suffixIndex == -1)
        free(n->end);
    free(n);
}
```

```
/*Build the suffix tree and print the edge labels along with
suffixIndex. suffixIndex for leaf edges will be >= 0 and
for non-leaf edges will be -1*/
void buildSuffixTree()
{
    size = strlen(text);
    int i;
    rootEnd = (int*) malloc(sizeof(int));
    *rootEnd = - 1;

    /*Root is a special node with start and end indices as -1,
    as it has no parent from where an edge comes to root*/
    root = newNode(-1, rootEnd);

    activeNode = root; //First activeNode will be root
    for (i=0; i<size; i++)
        extendSuffixTree(i);
    int labelHeight = 0;
    setSuffixIndexByDFS(root, labelHeight);
}

void doTraversal(Node *n, int suffixArray[], int *idx)
{
    if(n == NULL)
    {
        return;
    }
    int i=0;
    if(n->suffixIndex == -1) //If it is internal node
    {
        for (i = 0; i < MAX_CHAR; i++)
        {
            if(n->children[i] != NULL)
            {
                doTraversal(n->children[i], suffixArray, idx);
            }
        }
    }
    //If it is Leaf node other than "$" label
    else if(n->suffixIndex > -1 && n->suffixIndex < size)
    {
        suffixArray[(*idx)++] = n->suffixIndex;
    }
}

void buildSuffixArray(int suffixArray[])
{
```

```
int i = 0;
for(i=0; i< size; i++)
    suffixArray[i] = -1;
int idx = 0;
doTraversal(root, suffixArray, &idx);
printf("Suffix Array for String ");
for(i=0; i<size; i++)
    printf("%c", text[i]);
printf(" is: ");
for(i=0; i<size; i++)
    printf("%d ", suffixArray[i]);
printf("\n");
}

// driver program to test above functions
int main(int argc, char *argv[])
{
    strcpy(text, "banana$");
    buildSuffixTree();
    size--;
    int *suffixArray =(int*) malloc(sizeof(int) * size);
    buildSuffixArray(suffixArray);
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);
    free(suffixArray);

    strcpy(text, "GEEKSFORGEEKS$");
    buildSuffixTree();
    size--;
    suffixArray =(int*) malloc(sizeof(int) * size);
    buildSuffixArray(suffixArray);
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);
    free(suffixArray);

    strcpy(text, "AAAAAAAAAA$");
    buildSuffixTree();
    size--;
    suffixArray =(int*) malloc(sizeof(int) * size);
    buildSuffixArray(suffixArray);
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);
    free(suffixArray);

    strcpy(text, "ABCDEFG$");
    buildSuffixTree();
    size--;
    suffixArray =(int*) malloc(sizeof(int) * size);
```

```
buildSuffixArray(suffixArray);
//Free the dynamically allocated memory
freeSuffixTreeByPostOrder(root);
free(suffixArray);

strcpy(text, "ABABABA$");
buildSuffixTree();
size--;
suffixArray =(int*) malloc(sizeof(int) * size);
buildSuffixArray(suffixArray);
//Free the dynamically allocated memory
freeSuffixTreeByPostOrder(root);
free(suffixArray);

strcpy(text, "abcabxabcd$");
buildSuffixTree();
size--;
suffixArray =(int*) malloc(sizeof(int) * size);
buildSuffixArray(suffixArray);
//Free the dynamically allocated memory
freeSuffixTreeByPostOrder(root);
free(suffixArray);

strcpy(text, "CCAAACCCGATTA$");
buildSuffixTree();
size--;
suffixArray =(int*) malloc(sizeof(int) * size);
buildSuffixArray(suffixArray);
//Free the dynamically allocated memory
freeSuffixTreeByPostOrder(root);
free(suffixArray);

return 0;
}
```

Output:

```
Suffix Array for String banana is: 5 3 1 0 4 2
Suffix Array for String GEEKSFORGEEKS is: 9 1 10 2 5 8 0 11 3 6 7 12 4
Suffix Array for String AAAAAAAA is: 9 8 7 6 5 4 3 2 1 0
Suffix Array for String ABCDEFG is: 0 1 2 3 4 5 6
Suffix Array for String ABABABA is: 6 4 2 0 5 3 1
Suffix Array for String abcabxabcd is: 0 6 3 1 7 4 2 8 9 5
Suffix Array for String CCAAACCCGATTA is: 12 2 3 4 9 1 0 5 6 7 8 11 10
```

Ukkonen's Suffix Tree Construction takes $O(N)$ time and space to build suffix tree for a string of length N and after that, traversal of tree take $O(N)$ to build suffix array.

So overall, it's linear in time and space.

Can you see why traversal is $O(N)$?? Because a suffix tree of string of length N will have at most $N-1$ internal nodes and N leaves. Traversal of these nodes can be done in $O(N)$.

We have published following more articles on suffix tree applications:

- [Suffix Tree Application 1 – Substring Check](#)
- [Suffix Tree Application 2 – Searching All Patterns](#)
- [Suffix Tree Application 3 – Longest Repeated Substring](#)
- [Generalized Suffix Tree 1](#)
- [Suffix Tree Application 5 – Longest Common Substring](#)
- [Suffix Tree Application 6 – Longest Palindromic Substring](#)

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

Source

<https://www.geeksforgeeks.org/suffix-tree-application-4-build-linear-time-suffix-array/>

Chapter 188

Suffix Tree Application 5 – Longest Common Substring

Suffix Tree Application 5 - Longest Common Substring - GeeksforGeeks

Given two strings X and Y, find the [Longest Common Substring](#) of X and Y.

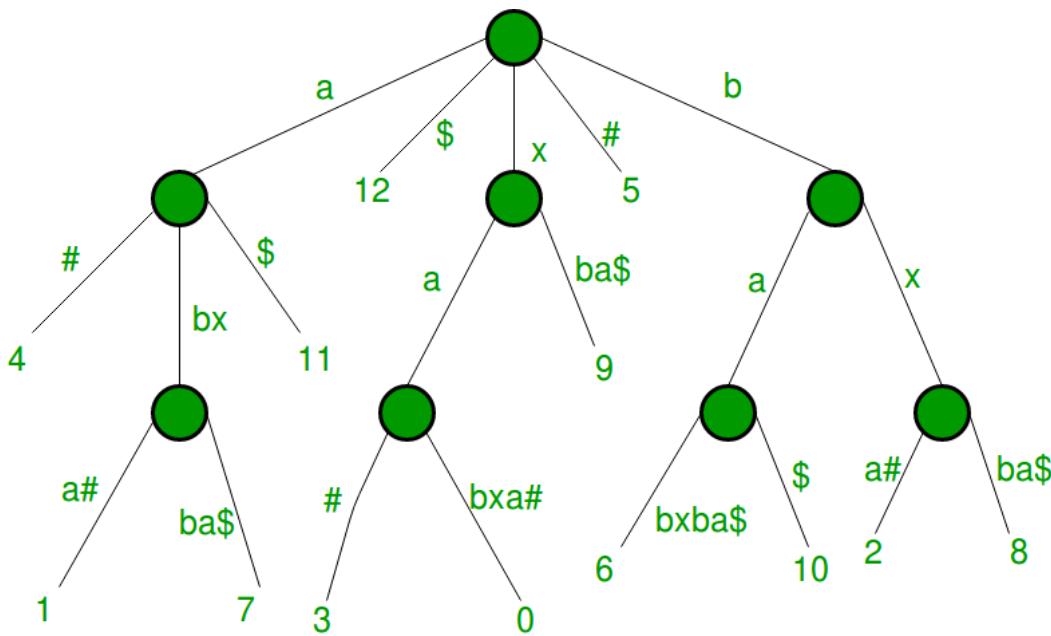
Naive [$O(N*M^2)$] and Dynamic Programming [$O(N*M)$] approaches are already discussed [here](#).

In this article, we will discuss a linear time approach to find LCS using suffix tree (The 5th Suffix Tree Application).

Here we will build generalized suffix tree for two strings X and Y as discussed already at: [Generalized Suffix Tree 1](#)

Lets take same example (X = xabxa, and Y = babxba) we saw in [Generalized Suffix Tree 1](#).

We built following suffix tree for X and Y there:



This is generalized suffix tree for $xabxa\#babxba\$$

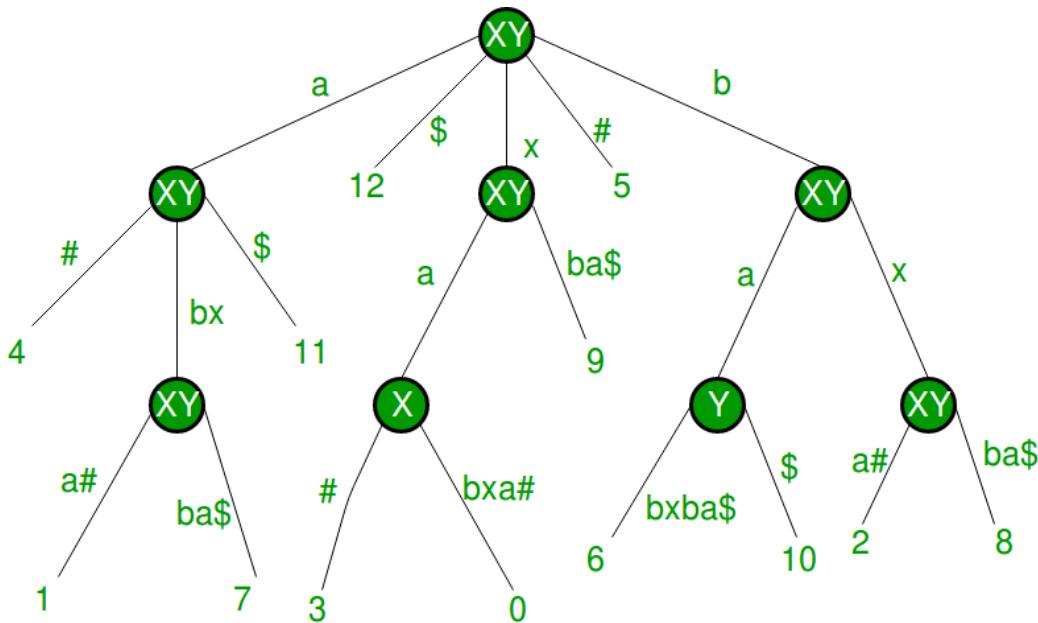
In above, leaves with suffix indices in $[0,4]$ are suffixes of string $xabxa$ and leaves with suffix indices in $[6,11]$ are suffixes of string $babxa$. Why ??

Because in concatenated string $xabxa\#babxba\$$, index of string $xabxa$ is 0 and it's length is 5, so indices of its suffixes would be 0, 1, 2, 3 and 4. Similarly index of string $babxa$ is 6 and it's length is 6, so indices of its suffixes would be 6, 7, 8, 9, 10 and 11.

With this, we can see that in the generalized suffix tree figure above, there are some internal nodes having leaves below it from

- both strings X and Y (i.e. there is at least one leaf with suffix index in $[0,4]$ and one leaf with suffix index in $[6, 11]$)
- string X only (i.e. all leaf nodes have suffix indices in $[0,4]$)
- string Y only (i.e. all leaf nodes have suffix indices in $[6,11]$)

Following figure shows the internal nodes marked as “XY”, “X” or “Y” depending on which string the leaves belong to, that they have below themselves.



What these “XY”, “X” or “Y” marking mean ?

Path label from root to an internal node gives a substring of X or Y or both.

For node marked as XY, substring from root to that node belongs to both strings X and Y.

For node marked as X, substring from root to that node belongs to string X only.

For node marked as Y, substring from root to that node belongs to string Y only.

By looking at above figure, can you see how to get LCS of X and Y ?

By now, it should be clear that how to get common substring of X and Y at least.

If we traverse the path from root to nodes marked as XY, we will get common substring of X and Y.

Now we need to find the longest one among all those common substrings.

Can you think how to get LCS now ? Recall how did we get [Longest Repeated Substring](#) in a given string using suffix tree already.

The path label from root to the deepest node marked as XY will give the LCS of X and Y. The deepest node is highlighted in above figure and path label “abx” from root to that node is the LCS of X and Y.

```
// A C program to implement Ukkonen's Suffix Tree Construction
// Here we build generalized suffix tree for two strings
// And then we find longest common substring of the two input strings
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define MAX_CHAR 256

struct SuffixTreeNode {
    struct SuffixTreeNode *children[MAX_CHAR];
```

```

//pointer to other node via suffix link
struct SuffixTreeNode *suffixLink;

/*(start, end) interval specifies the edge, by which the
node is connected to its parent node. Each edge will
connect two nodes, one parent and one child, and
(start, end) interval of a given edge will be stored
in the child node. Lets say there are two nodes A and B
connected by an edge with indices (5, 8) then this
indices (5, 8) will be stored in node B. */
int start;
int *end;

/*for leaf nodes, it stores the index of suffix for
the path from root to leaf*/
int suffixIndex;
};

typedef struct SuffixTreeNode Node;

char text[100]; //Input string
Node *root = NULL; //Pointer to root node

/*lastNewNode will point to newly created internal node,
waiting for it's suffix link to be set, which might get
a new suffix link (other than root) in next extension of
same phase. lastNewNode will be set to NULL when last
newly created internal node (if there is any) got it's
suffix link reset to new internal node created in next
extension of same phase. */
Node *lastNewNode = NULL;
Node *activeNode = NULL;

/*activeEdge is represented as input string character
index (not the character itself)*/
int activeEdge = -1;
int activeLength = 0;

// remainingSuffixCount tells how many suffixes yet to
// be added in tree
int remainingSuffixCount = 0;
int leafEnd = -1;
int *rootEnd = NULL;
int *splitEnd = NULL;
int size = -1; //Length of input string
int size1 = 0; //Size of 1st string

Node *newNode(int start, int *end)

```

```

{
    Node *node =(Node*) malloc(sizeof(Node));
    int i;
    for (i = 0; i < MAX_CHAR; i++)
        node->children[i] = NULL;

    /*For root node, suffixLink will be set to NULL
    For internal nodes, suffixLink will be set to root
    by default in current extension and may change in
    next extension*/
    node->suffixLink = root;
    node->start = start;
    node->end = end;

    /*suffixIndex will be set to -1 by default and
     actual suffix index will be set later for leaves
     at the end of all phases*/
    node->suffixIndex = -1;
    return node;
}

int edgeLength(Node *n) {
    if(n == root)
        return 0;
    return *(n->end) - (n->start) + 1;
}

int walkDown(Node *currNode)
{
    /*activePoint change for walk down (APCFWD) using
     Skip/Count Trick (Trick 1). If activeLength is greater
     than current edge length, set next internal node as
     activeNode and adjust activeEdge and activeLength
     accordingly to represent same activePoint*/
    if (activeLength >= edgeLength(currNode))
    {
        activeEdge += edgeLength(currNode);
        activeLength -= edgeLength(currNode);
        activeNode = currNode;
        return 1;
    }
    return 0;
}

void extendSuffixTree(int pos)
{
    /*Extension Rule 1, this takes care of extending all
     leaves created so far in tree*/
}

```

```

leafEnd = pos;

/*Increment remainingSuffixCount indicating that a
new suffix added to the list of suffixes yet to be
added in tree*/
remainingSuffixCount++;

/*set lastNewNode to NULL while starting a new phase,
indicating there is no internal node waiting for
it's suffix link reset in current phase*/
lastNewNode = NULL;

//Add all suffixes (yet to be added) one by one in tree
while(remainingSuffixCount > 0) {

    if (activeLength == 0)
        activeEdge = pos; //APCFALZ

    // There is no outgoing edge starting with
    // activeEdge from activeNode
    if (activeNode->children] == NULL)
    {
        //Extension Rule 2 (A new leaf edge gets created)
        activeNode->children] =
            newNode(pos, &leafEnd);

        /*A new leaf edge is created in above line starting
        from an existing node (the current activeNode), and
        if there is any internal node waiting for it's suffix
        link get reset, point the suffix link from that last
        internal node to current activeNode. Then set lastNewNode
        to NULL indicating no more node waiting for suffix link
        reset.*/
        if (lastNewNode != NULL)
        {
            lastNewNode->suffixLink = activeNode;
            lastNewNode = NULL;
        }
    }
    // There is an outgoing edge starting with activeEdge
    // from activeNode
    else
    {
        // Get the next node at the end of edge starting
        // with activeEdge
        Node *next = activeNode->children];
        if (walkDown(next))//Do walkdown
        {

```

```

        //Start from next node (the new activeNode)
        continue;
    }
/*Extension Rule 3 (current character being processed
   is already on the edge)*/
if (text[next->start + activeLength] == text[pos])
{
    //If a newly created node waiting for it's
    //suffix link to be set, then set suffix link
    //of that waiting node to current active node
    if(lastNewNode != NULL && activeNode != root)
    {
        lastNewNode->suffixLink = activeNode;
        lastNewNode = NULL;
    }

    //APCFER3
    activeLength++;
    /*STOP all further processing in this phase
    and move on to next phase*/
    break;
}

/*We will be here when activePoint is in middle of
   the edge being traversed and current character
   being processed is not on the edge (we fall off
   the tree). In this case, we add a new internal node
   and a new leaf edge going out of that new node. This
   is Extension Rule 2, where a new leaf edge and a new
   internal node get created*/
splitEnd = (int*) malloc(sizeof(int));
*splitEnd = next->start + activeLength - 1;

//New internal node
Node *split = newNode(next->start, splitEnd);
activeNode->children] = split;

//New leaf coming out of new internal node
split->children] = newNode(pos, &leafEnd);
next->start += activeLength;
split->children] = next;

/*We got a new internal node here. If there is any
   internal node created in last extensions of same
   phase which is still waiting for it's suffix link
   reset, do it now.*/
if (lastNewNode != NULL)
{

```

```

/*suffixLink of lastNewNode points to current newly
created internal node*/
lastNewNode->suffixLink = split;
}

/*Make the current newly created internal node waiting
for it's suffix link reset (which is pointing to root
at present). If we come across any other internal node
(existing or newly created) in next extension of same
phase, when a new leaf edge gets added (i.e. when
Extension Rule 2 applies is any of the next extension
of same phase) at that point, suffixLink of this node
will point to that internal node.*/
lastNewNode = split;
}

/* One suffix got added in tree, decrement the count of
suffixes yet to be added.*/
remainingSuffixCount--;
if (activeNode == root && activeLength > 0) //APCFER2C1
{
    activeLength--;
    activeEdge = pos - remainingSuffixCount + 1;
}
else if (activeNode != root) //APCFER2C2
{
    activeNode = activeNode->suffixLink;
}
}

void print(int i, int j)
{
    int k;
    for (k=i; k<=j && text[k] != '#'; k++)
        printf("%c", text[k]);
    if(k<=j)
        printf("#");
}

//Print the suffix tree as well along with setting suffix index
//So tree will be printed in DFS manner
//Each edge along with it's suffix index will be printed
void setSuffixIndexByDFS(Node *n, int labelHeight)
{
    if (n == NULL)  return;

    if (n->start != -1) //A non-root node

```

```

{
    //Print the label on edge from parent to current node
    //Uncomment below line to print suffix tree
    //printf(n->start, *(n->end));
}
int leaf = 1;
int i;
for (i = 0; i < MAX_CHAR; i++)
{
    if (n->children[i] != NULL)
    {
        //Uncomment below two lines to print suffix index
        //if (leaf == 1 && n->start != -1)
        //    printf(" [%d]\n", n->suffixIndex);

        //Current node is not a leaf as it has outgoing
        //edges from it.
        leaf = 0;
        setSuffixIndexByDFS(n->children[i], labelHeight +
                           edgeLength(n->children[i]));
    }
}
if (leaf == 1)
{
    for(i= n->start; i<= *(n->end); i++)
    {
        if(text[i] == '#')
        {
            n->end = (int*) malloc(sizeof(int));
            *(n->end) = i;
        }
    }
    n->suffixIndex = size - labelHeight;
    //Uncomment below line to print suffix index
    //printf(" [%d]\n", n->suffixIndex);
}
}

void freeSuffixTreeByPostOrder(Node *n)
{
    if (n == NULL)
        return;
    int i;
    for (i = 0; i < MAX_CHAR; i++)
    {
        if (n->children[i] != NULL)
        {
            freeSuffixTreeByPostOrder(n->children[i]);
        }
    }
}

```

```

        }
    }
    if (n->suffixIndex == -1)
        free(n->end);
    free(n);
}

/*Build the suffix tree and print the edge labels along with
suffixIndex. suffixIndex for leaf edges will be >= 0 and
for non-leaf edges will be -1*/
void buildSuffixTree()
{
    size = strlen(text);
    int i;
    rootEnd = (int*) malloc(sizeof(int));
    *rootEnd = - 1;

    /*Root is a special node with start and end indices as -1,
    as it has no parent from where an edge comes to root*/
    root = newNode(-1, rootEnd);

    activeNode = root; //First activeNode will be root
    for (i=0; i<size; i++)
        extendSuffixTree(i);
    int labelHeight = 0;
    setSuffixIndexByDFS(root, labelHeight);
}

int doTraversal(Node *n, int labelHeight, int* maxHeight,
int* substringStartIndex)
{
    if(n == NULL)
    {
        return;
    }
    int i=0;
    int ret = -1;
    if(n->suffixIndex < 0) //If it is internal node
    {
        for (i = 0; i < MAX_CHAR; i++)
        {
            if(n->children[i] != NULL)
            {
                ret = doTraversal(n->children[i], labelHeight +
                    edgeLength(n->children[i]),
                    maxHeight, substringStartIndex);

                if(n->suffixIndex == -1)

```

```

        n->suffixIndex = ret;
    else if((n->suffixIndex == -2 && ret == -3) ||
            (n->suffixIndex == -3 && ret == -2) ||
            n->suffixIndex == -4)
    {
        n->suffixIndex = -4;//Mark node as XY
        //Keep track of deepest node
        if(*maxHeight < labelHeight)
        {
            *maxHeight = labelHeight;
            *substringstartIndex = *(n->end) -
                labelHeight + 1;
        }
    }
}
else if(n->suffixIndex > -1 && n->suffixIndex < size1)//suffix of X
    return -2;//Mark node as X
else if(n->suffixIndex >= size1)//suffix of Y
    return -3;//Mark node as Y
return n->suffixIndex;
}

void getLongestCommonSubstring()
{
    int maxHeight = 0;
    int substringstartIndex = 0;
    doTraversal(root, 0, &maxHeight, &substringstartIndex);

    int k;
    for (k=0; k<maxHeight; k++)
        printf("%c", text[k + substringstartIndex]);
    if(k == 0)
        printf("No common substring");
    else
        printf(", of length: %d",maxHeight);
    printf("\n");
}

// driver program to test above functions
int main(int argc, char *argv[])
{
    size1 = 7;
    printf("Longest Common Substring in xabxac and abcabxabcd is: ");
    strcpy(text, "xabxac#abcabxabcd$"); buildSuffixTree();
    getLongestCommonSubstring();
    //Free the dynamically allocated memory
}

```

```
freeSuffixTreeByPostOrder(root);

size1 = 10;
printf("Longest Common Substring in xabxaabxa and babxba is: ");
strcpy(text, "xabxaabxa#babxba$"); buildSuffixTree();
getLongestCommonSubstring();
//Free the dynamically allocated memory
freeSuffixTreeByPostOrder(root);

size1 = 14;
printf("Longest Common Substring in GeeksforGeeks and GeeksQuiz is: ");
strcpy(text, "GeeksforGeeks#GeeksQuiz$"); buildSuffixTree();
getLongestCommonSubstring();
//Free the dynamically allocated memory
freeSuffixTreeByPostOrder(root);

size1 = 26;
printf("Longest Common Substring in OldSite:GeeksforGeeks.org");
printf(" and NewSite:GeeksQuiz.com is: ");
strcpy(text, "OldSite:GeeksforGeeks.org#NewSite:GeeksQuiz.com$");
buildSuffixTree();
getLongestCommonSubstring();
//Free the dynamically allocated memory
freeSuffixTreeByPostOrder(root);

size1 = 6;
printf("Longest Common Substring in abcde and fghie is: ");
strcpy(text, "abcde#fghie$"); buildSuffixTree();
getLongestCommonSubstring();
//Free the dynamically allocated memory
freeSuffixTreeByPostOrder(root);

size1 = 6;
printf("Longest Common Substring in pqrst and uvwxyz is: ");
strcpy(text, "pqrst#uvwxyz$"); buildSuffixTree();
getLongestCommonSubstring();
//Free the dynamically allocated memory
freeSuffixTreeByPostOrder(root);

return 0;
}
```

Output:

```
Longest Common Substring in xabxac and abcabxabcd is: abxa, of length: 4
Longest Common Substring in xabxaabxa and babxba is: abx, of length: 3
Longest Common Substring in GeeksforGeeks and GeeksQuiz is: Geeks, of length: 5
```

Longest Common Substring in OldSite:GeeksforGeeks.org and

NewSite:GeeksQuiz.com is: Site:Geeks, of length: 10

Longest Common Substring in abcde and fghie is: e, of length: 1

Longest Common Substring in pqrst and uvwxyz is: No common substring

If two strings are of size M and N, then Generalized Suffix Tree construction takes $O(M+N)$ and LCS finding is a DFS on tree which is again $O(M+N)$.

So overall complexity is linear in time and space.

Followup:

1. Given a pattern, check if it is substring of X or Y or both. If it is a substring, find all its occurrences along with which string (X or Y or both) it belongs to.
2. Extend the implementation to find LCS of more than two strings
3. Solve problem 1 for more than two strings
4. Given a string, find its [Longest Palindromic Substring](#)

We have published following more articles on suffix tree applications:

- [Suffix Tree Application 1 – Substring Check](#)
- [Suffix Tree Application 2 – Searching All Patterns](#)
- [Suffix Tree Application 3 – Longest Repeated Substring](#)
- [Suffix Tree Application 4 – Build Linear Time Suffix Array](#)
- [Suffix Tree Application 6 – Longest Palindromic Substring](#)
- [Generalized Suffix Tree 1](#)

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

Source

<https://www.geeksforgeeks.org/suffix-tree-application-5-longest-common-substring-2/>

Chapter 189

Suffix Tree Application 6 – Longest Palindromic Substring

Suffix Tree Application 6 - Longest Palindromic Substring - GeeksforGeeks

Given a string, find the longest substring which is palindrome.

We have already discussed Naïve [$O(n^3)$], quadratic [$O(n^2)$] and linear [$O(n)$] approaches in [Set 1](#), [Set 2](#) and [Manacher's Algorithm](#).

In this article, we will discuss another linear time approach based on suffix tree.

If given string is S, then approach is following:

- Reverse the string S (say reversed string is R)
- Get [Longest Common Substring](#) of S and R given that **LCS in S and R must be from same position in S**

Can you see why we say that **LCS in R and S must be from same position in S ?**

Let's look at following examples:

- For S = *xababayz* and R = *zyababax*, LCS and LPS both are *ababa* (SAME)
- For S = *abacdfgdcaba* and R = *abacdgcaba*, LCS is *abacd* and LPS is *aba* (DIFFERENT)
- For S = *pqrqpabcdgdcba* and R = *abcdgfdcbapqrqp*, LCS and LPS both are *pqrqp* (SAME)
- For S = *pqqpabcdghfdcba* and R = *abcdfhgfdcbapqqp*, LCS is *abcdf* and LPS is *pqqp* (DIFFERENT)

We can see that LCS and LPS are not same always. When they are different ?

When S has a reversed copy of a non-palindromic substring in it which is of same or longer length than the palindromic substring, then LCS and LPS will be different.

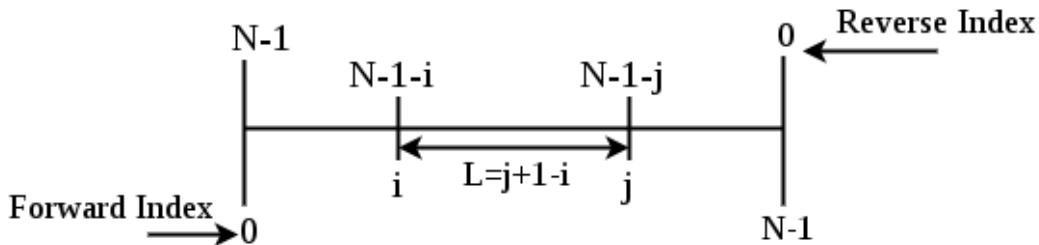
length than LPS in S, then LCS and LPS will be different.

In 2nd example above ($S = abacdgdcab$), for substring $abacd$, there exists a reverse copy $dcaba$ in S, which is of longer length than LPS aba and so LPS and LCS are different here. Same is the scenario in 4th example.

To handle this scenario we say that LPS in S is same as LCS in S and R given that **LCS in R and S must be from same position in S**.

If we look at 2nd example again, substring aba in R comes from exactly same position in S as substring aba in S which is ZERO (0th index) and so this is LPS.

The Position Constraint:



We will refer string S index as forward index (S_i) and string R index as reverse index (R_i). Based on above figure, a character with index i (forward index) in a string S of length N , will be at index $N-1-i$ (reverse index) in it's reversed string R.

If we take a substring of length L in string S with starting index i and ending index j ($j = i+L-1$), then in it's reversed string R, the reversed substring of the same will start at index $N-1-j$ and will end at index $N-1-i$.

If there is a common substring of length L at indices S_i (forward index) and R_i (reverse index) in S and R, then these will come from same position in S if $R_i = (N - 1) - (S_i + L - 1)$ where N is string length.

So to find LPS of string S, we find longest common string of S and R where both substrings satisfy above constraint, i.e. if substring in S is at index S_i , then same substring should be in R at index $(N - 1) - (S_i + L - 1)$. If this is not the case, then this substring is not LPS candidate.

Naive [$O(N*M^2)$] and Dynamic Programming [$O(N*M)$] approaches to find LCS of two strings are already discussed [here](#) which can be extended to add position constraint to give LPS of a given string.

Now we will discuss suffix tree approach which is nothing but an extension to [Suffix Tree LCS approach](#) where we will add the position constraint.

While finding LCS of two strings X and Y, we just take deepest node marked as XY (i.e. the node which has suffixes from both strings as it's children).

While finding LPS of string S, we will again find LCS of S and R with a condition that the common substring should satisfy the position constraint (the common substring should come from same position in S). To verify position constraint, we need to know all forward and reverse indices on each internal node (i.e. the suffix indices of all leaf children below the internal nodes).

In [Generalized Suffix Tree](#) of $S\#R\$$, a substring on the path from root to an internal node is a common substring if the internal node has suffixes from both strings S and R. The index

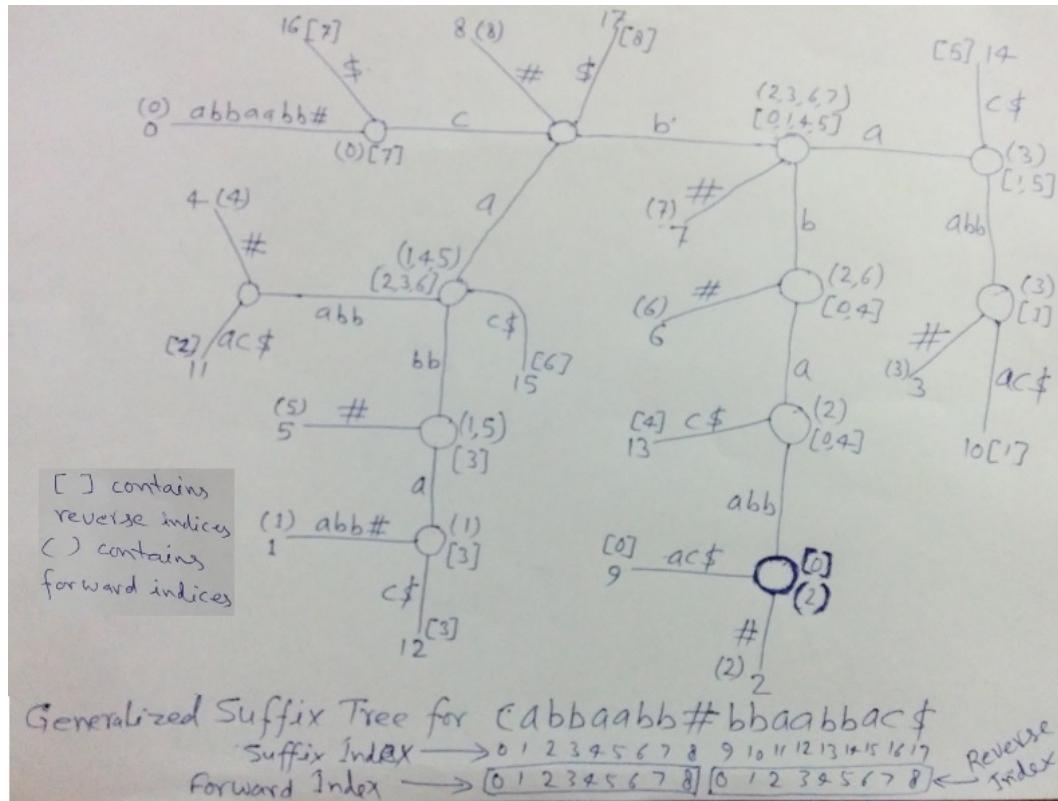
of the common substring in S and R can be found by looking at suffix index at respective leaf node.

If string $S\#$ is of length N then:

- If suffix index of a leaf is less than N, then that suffix belongs to S and same suffix index will become forward index of all ancestor nodes
- If suffix index of a leaf is greater than N, then that suffix belongs to R and reverse index for all ancestor nodes will be $N - \text{suffix index}$

Let's take string $S = cabbaabb$. The figure below is Generalized Suffix Tree for $cabbaabb\#bbaabbac\$$ where we have shown forward and reverse indices of all children suffixes on all internal nodes (except root).

Forward indices are in Parentheses () and reverse indices are in square bracket [].



In above figure, all leaf nodes will have one forward or reverse index depending on which string (S or R) they belong to. Then children's forward or reverse indices propagate to the parent.

Look at the figure to understand what would be the forward or reverse index on a leaf with a given suffix index. At the bottom of figure, it is shown that leaves with suffix indices from 0 to 8 will get same values (0 to 8) as their forward index in S and leaves with suffix indices 9 to 17 will get reverse index in R from 0 to 8.

For example, the highlighted internal node has two children with suffix indices 2 and 9. Leaf with suffix index 2 is from position 2 in S and so its forward index is 2 and shown in $()$. Leaf with suffix index 9 is from position 0 in R and so its reverse index is 0 and shown in $[]$. These indices propagate to parent and the parent has one leaf with suffix index 14 for which reverse index is 4. So on this parent node forward index is (2) and reverse index is $[0,4]$. And in same way, we should be able to understand the how forward and reverse indices are calculated on all nodes.

In above figure, all internal nodes have suffixes from both strings S and R, i.e. all of them represent a common substring on the path from root to themselves. Now we need to find deepest node satisfying position constraint. For this, we need to check if there is a forward index S_i on a node, then there must be a reverse index R_i with value $(N - 2) - (S_i + L - 1)$ where N is length of string $S\#$ and L is node depth (or substring length). If yes, then consider this node as a LPS candidate, else ignore it. In above figure, deepest node is highlighted which represents LPS as bbaabb.

We have not shown forward and reverse indices on root node in figure. Because root node itself doesn't represent any common substring (In code implementation also, forward and reverse indices will not be calculated on root node)

How to implement this approach to find LPS? Here are the things that we need:

- We need to know forward and reverse indices on each node.
- For a given forward index S_i on an internal node, we need to know if reverse index $R_i = (N - 2) - (S_i + L - 1)$ also present on same node.
- Keep track of deepest internal node satisfying above condition.

One way to do above is:

While DFS on suffix tree, we can store forward and reverse indices on each node in some way (storage will help to avoid repeated traversals on tree when we need to know forward and reverse indices on a node). Later on, we can do another DFS to look for nodes satisfying position constraint. For position constraint check, we need to search in list of indices.

What data structure is suitable here to do all these in quickest way ?

- If we store indices in array, it will require linear search which will make overall approach non-linear in time.
- If we store indices in tree (set in C++, TreeSet in Java), we may use binary search but still overall approach will be non-linear in time.
- If we store indices in hash function based set (unordered_set in C++, HashSet in Java), it will provide a constant search on average and this will make overall approach linear in time. *A hash function based set may take more space depending on values being stored.*

We will use two unordered_set (one for forward and other from reverse indices) in our implementation, added as a member variable in SuffixTreeNode structure.

```

// A C++ program to implement Ukkonen's Suffix Tree Construction
// Here we build generalized suffix tree for given string S
// and it's reverse R, then we find
// longest palindromic substring of given string S
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <iostream>
#include <unordered_set>
#define MAX_CHAR 256
using namespace std;

struct SuffixTreeNode {
    struct SuffixTreeNode *children[MAX_CHAR];

    //pointer to other node via suffix link
    struct SuffixTreeNode *suffixLink;

    /*(start, end) interval specifies the edge, by which the
     node is connected to its parent node. Each edge will
     connect two nodes, one parent and one child, and
     (start, end) interval of a given edge will be stored
     in the child node. Lets say there are two nodes A and B
     connected by an edge with indices (5, 8) then this
     indices (5, 8) will be stored in node B. */
    int start;
    int *end;

    /*for leaf nodes, it stores the index of suffix for
     the path from root to leaf*/
    int suffixIndex;

    //To store indices of children suffixes in given string
    unordered_set<int> *forwardIndices;

    //To store indices of children suffixes in reversed string
    unordered_set<int> *reverseIndices;
};

typedef struct SuffixTreeNode Node;

char text[100]; //Input string
Node *root = NULL; //Pointer to root node

/*lastNewNode will point to newly created internal node,
 waiting for it's suffix link to be set, which might get
 a new suffix link (other than root) in next extension of
 same phase. lastNewNode will be set to NULL when last

```

```

newly created internal node (if there is any) got it's
suffix link reset to new internal node created in next
extension of same phase. */
Node *lastNewNode = NULL;
Node *activeNode = NULL;

/*activeEdge is represeted as input string character
 index (not the character itself)*/
int activeEdge = -1;
int activeLength = 0;

// remainingSuffixCount tells how many suffixes yet to
// be added in tree
int remainingSuffixCount = 0;
int leafEnd = -1;
int *rootEnd = NULL;
int *splitEnd = NULL;
int size = -1; //Length of input string
int size1 = 0; //Size of 1st string
int reverseIndex; //Index of a suffix in reversed string
unordered_set<int>::iterator forwardIndex;

Node *newNode(int start, int *end)
{
    Node *node =(Node*) malloc(sizeof(Node));
    int i;
    for (i = 0; i < MAX_CHAR; i++)
        node->children[i] = NULL;

    /*For root node, suffixLink will be set to NULL
    For internal nodes, suffixLink will be set to root
    by default in current extension and may change in
    next extension*/
    node->suffixLink = root;
    node->start = start;
    node->end = end;

    /*suffixIndex will be set to -1 by default and
       actual suffix index will be set later for leaves
       at the end of all phases*/
    node->suffixIndex = -1;
    node->forwardIndices = new unordered_set<int>;
    node->reverseIndices = new unordered_set<int>;
    return node;
}

int edgeLength(Node *n) {
    if(n == root)

```

```

        return 0;
    return *(n->end) - (n->start) + 1;
}

int walkDown(Node *currNode)
{
    /*activePoint change for walk down (APCFWD) using
     Skip/Count Trick (Trick 1). If activeLength is greater
     than current edge length, set next internal node as
     activeNode and adjust activeEdge and activeLength
     accordingly to represent same activePoint*/
    if (activeLength >= edgeLength(currNode))
    {
        activeEdge += edgeLength(currNode);
        activeLength -= edgeLength(currNode);
        activeNode = currNode;
        return 1;
    }
    return 0;
}

void extendSuffixTree(int pos)
{
    /*Extension Rule 1, this takes care of extending all
     leaves created so far in tree*/
    leafEnd = pos;

    /*Increment remainingSuffixCount indicating that a
     new suffix added to the list of suffixes yet to be
     added in tree*/
    remainingSuffixCount++;

    /*set lastNewNode to NULL while starting a new phase,
     indicating there is no internal node waiting for
     it's suffix link reset in current phase*/
    lastNewNode = NULL;

    //Add all suffixes (yet to be added) one by one in tree
    while(remainingSuffixCount > 0) {

        if (activeLength == 0)
            activeEdge = pos; //APCFALZ

        // There is no outgoing edge starting with
        // activeEdge from activeNode
        if (activeNode->children] == NULL)
        {
            //Extension Rule 2 (A new leaf edge gets created)

```

```

activeNode->children] =
    newNode(pos, &leafEnd);

/*A new leaf edge is created in above line starting
from an existing node (the current activeNode), and
if there is any internal node waiting for it's suffix
link get reset, point the suffix link from that last
internal node to current activeNode. Then set lastNewNode
to NULL indicating no more node waiting for suffix link
reset.*/
if (lastNewNode != NULL)
{
    lastNewNode->suffixLink = activeNode;
    lastNewNode = NULL;
}
// There is an outgoing edge starting with activeEdge
// from activeNode
else
{
    // Get the next node at the end of edge starting
    // with activeEdge
    Node *next = activeNode->children] ;
    if (walkDown(next))//Do walkdown
    {
        //Start from next node (the new activeNode)
        continue;
    }
    /*Extension Rule 3 (current character being processed
     is already on the edge)*/
    if (text[next->start + activeLength] == text[pos])
    {
        //APCFER3
        activeLength++;
        /*STOP all further processing in this phase
         and move on to next phase*/
        break;
    }

    /*We will be here when activePoint is in middle of
     the edge being traversed and current character
     being processed is not on the edge (we fall off
     the tree). In this case, we add a new internal node
     and a new leaf edge going out of that new node. This
     is Extension Rule 2, where a new leaf edge and a new
     internal node get created*/
    splitEnd = (int*) malloc(sizeof(int));
    *splitEnd = next->start + activeLength - 1;
}

```

```

//New internal node
Node *split = newNode(next->start, splitEnd);
activeNode->children] = split;

//New leaf coming out of new internal node
split->children] = newNode(pos, &leafEnd);
next->start += activeLength;
split->children] = next;

/*We got a new internal node here. If there is any
internal node created in last extensions of same
phase which is still waiting for it's suffix link
reset, do it now.*/
if (lastNewNode != NULL)
{
/*suffixLink of lastNewNode points to current newly
created internal node*/
lastNewNode->suffixLink = split;
}

/*Make the current newly created internal node waiting
for it's suffix link reset (which is pointing to root
at present). If we come across any other internal node
(existing or newly created) in next extension of same
phase, when a new leaf edge gets added (i.e. when
Extension Rule 2 applies is any of the next extension
of same phase) at that point, suffixLink of this node
will point to that internal node.*/
lastNewNode = split;
}

/* One suffix got added in tree, decrement the count of
suffixes yet to be added.*/
remainingSuffixCount--;
if (activeNode == root && activeLength > 0) //APCFER2C1
{
    activeLength--;
    activeEdge = pos - remainingSuffixCount + 1;
}
else if (activeNode != root) //APCFER2C2
{
    activeNode = activeNode->suffixLink;
}
}

void print(int i, int j)

```

```

{
    int k;
    for (k=i; k<=j && text[k] != '#'; k++)
        printf("%c", text[k]);
    if(k<=j)
        printf("#");
}

//Print the suffix tree as well along with setting suffix index
//So tree will be printed in DFS manner
//Each edge along with it's suffix index will be printed
void setSuffixIndexByDFS(Node *n, int labelHeight)
{
    if (n == NULL)  return;

    if (n->start != -1) //A non-root node
    {
        //Print the label on edge from parent to current node
        //Uncomment below line to print suffix tree
        //printf(n->start, *(n->end));
    }
    int leaf = 1;
    int i;
    for (i = 0; i < MAX_CHAR; i++)
    {
        if (n->children[i] != NULL)
        {
            //Uncomment below two lines to print suffix index
            //if (leaf == 1 && n->start != -1)
            //    printf(" [%d]\n", n->suffixIndex);

            //Current node is not a leaf as it has outgoing
            //edges from it.
            leaf = 0;
            setSuffixIndexByDFS(n->children[i], labelHeight +
                                edgeLength(n->children[i]));
        }
        if(n != root)
        {
            //Add children's suffix indices in parent
            n->forwardIndices->insert(
                n->children[i]->forwardIndices->begin(),
                n->children[i]->forwardIndices->end());
            n->reverseIndices->insert(
                n->children[i]->reverseIndices->begin(),
                n->children[i]->reverseIndices->end());
        }
    }
}
}

```

```

if (leaf == 1)
{
    for(i= n->start; i<= *(n->end); i++)
    {
        if(text[i] == '#')
        {
            n->end = (int*) malloc(sizeof(int));
            *(n->end) = i;
        }
    }
    n->suffixIndex = size - labelHeight;

    if(n->suffixIndex < size1) //Suffix of Given String
        n->forwardIndices->insert(n->suffixIndex);
    else //Suffix of Reversed String
        n->reverseIndices->insert(n->suffixIndex - size1);

    //Uncomment below line to print suffix index
    // printf(" [%d]\n", n->suffixIndex);
}
}

void freeSuffixTreeByPostOrder(Node *n)
{
    if (n == NULL)
        return;
    int i;
    for (i = 0; i < MAX_CHAR; i++)
    {
        if (n->children[i] != NULL)
        {
            freeSuffixTreeByPostOrder(n->children[i]);
        }
    }
    if (n->suffixIndex == -1)
        free(n->end);
    free(n);
}

/*Build the suffix tree and print the edge labels along with
suffixIndex. suffixIndex for leaf edges will be >= 0 and
for non-leaf edges will be -1*/
void buildSuffixTree()
{
    size = strlen(text);
    int i;
    rootEnd = (int*) malloc(sizeof(int));
    *rootEnd = - 1;
}

```

```

/*Root is a special node with start and end indices as -1,
as it has no parent from where an edge comes to root*/
root = newNode(-1, rootEnd);

activeNode = root; //First activeNode will be root
for (i=0; i<size; i++)
    extendSuffixTree(i);
int labelHeight = 0;
setSuffixIndexByDFS(root, labelHeight);
}

void doTraversal(Node *n, int labelHeight, int* maxHeight,
int* substringStartIndex)
{
    if(n == NULL)
    {
        return;
    }
    int i=0;
    int ret = -1;
    if(n->suffixIndex < 0) //If it is internal node
    {
        for (i = 0; i < MAX_CHAR; i++)
        {
            if(n->children[i] != NULL)
            {
                doTraversal(n->children[i], labelHeight +
                    edgeLength(n->children[i]),
                    maxHeight, substringStartIndex);

                if(*maxHeight < labelHeight
                    && n->forwardIndices->size() > 0 &&
                    n->reverseIndices->size() > 0)
                {
                    for (forwardIndex=n->forwardIndices->begin();
                        forwardIndex!=n->forwardIndices->end();
                        ++forwardIndex)
                    {
                        reverseIndex = (size1 - 2) -
                            (*forwardIndex + labelHeight - 1);
                        //If reverse suffix comes from
                        //SAME position in given string
                        //Keep track of deepest node
                        if(n->reverseIndices->find(reverseIndex) !=
                            n->reverseIndices->end())
                        {
                            *maxHeight = labelHeight;
                        }
                    }
                }
            }
        }
    }
}

```

```

        *substringstartIndex = *(n->end) -
            labelHeight + 1;
        break;
    }
}
}
}
}
}
}

void getLongestPalindromicSubstring()
{
    int maxHeight = 0;
    int substringstartIndex = 0;
    doTraversal(root, 0, &maxHeight, &substringstartIndex);

    int k;
    for (k=0; k<maxHeight; k++)
        printf("%c", text[k + substringstartIndex]);
    if(k == 0)
        printf("No palindromic substring");
    else
        printf(", of length: %d",maxHeight);
    printf("\n");
}

// driver program to test above functions
int main(int argc, char *argv[])
{
    size1 = 9;
    printf("Longest Palindromic Substring in cabbaabb is: ");
    strcpy(text, "cabbaabb#bbaabbac$"); buildSuffixTree();
    getLongestPalindromicSubstring();
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    size1 = 17;
    printf("Longest Palindromic Substring in forgeeksskeegfor is: ");
    strcpy(text, "forgeeksskeegfor#rofgeeksskeegrof$"); buildSuffixTree();
    getLongestPalindromicSubstring();
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    size1 = 6;
    printf("Longest Palindromic Substring in abcde is: ");
    strcpy(text, "abcde#edcba$"); buildSuffixTree();
    getLongestPalindromicSubstring();
}

```

```
//Free the dynamically allocated memory
freeSuffixTreeByPostOrder(root);

size1 = 7;
printf("Longest Palindromic Substring in abcdae is: ");
strcpy(text, "abcdae#eadcba$"); buildSuffixTree();
getLongestPalindromicSubstring();
//Free the dynamically allocated memory
freeSuffixTreeByPostOrder(root);

size1 = 6;
printf("Longest Palindromic Substring in abacd is: ");
strcpy(text, "abacd#dcaba$"); buildSuffixTree();
getLongestPalindromicSubstring();
//Free the dynamically allocated memory
freeSuffixTreeByPostOrder(root);

size1 = 6;
printf("Longest Palindromic Substring in abcdc is: ");
strcpy(text, "abcdc#cdcba$"); buildSuffixTree();
getLongestPalindromicSubstring();
//Free the dynamically allocated memory
freeSuffixTreeByPostOrder(root);

size1 = 13;
printf("Longest Palindromic Substring in abacdfgdcba is: ");
strcpy(text, "abacdfgdcba#abacdfgdcba$"); buildSuffixTree();
getLongestPalindromicSubstring();
//Free the dynamically allocated memory
freeSuffixTreeByPostOrder(root);

size1 = 15;
printf("Longest Palindromic Substring in xyabacdfgdcba is: ");
strcpy(text, "xyabacdfgdcba#abacdfgdcabayx$"); buildSuffixTree();
getLongestPalindromicSubstring();
//Free the dynamically allocated memory
freeSuffixTreeByPostOrder(root);

size1 = 9;
printf("Longest Palindromic Substring in xababayz is: ");
strcpy(text, "xababayz#zyababax$"); buildSuffixTree();
getLongestPalindromicSubstring();
//Free the dynamically allocated memory
freeSuffixTreeByPostOrder(root);

size1 = 6;
printf("Longest Palindromic Substring in xabax is: ");
strcpy(text, "xabax#xabax$"); buildSuffixTree();
```

```
getLongestPalindromicSubstring();
//Free the dynamically allocated memory
freeSuffixTreeByPostOrder(root);

    return 0;
}
```

Output:

```
Longest Palindromic Substring in cabbaabb is: bbaabb, of length: 6
Longest Palindromic Substring in forgeeksskeegfor is: geeksskeeg, of length: 10
Longest Palindromic Substring in abcde is: a, of length: 1
Longest Palindromic Substring in abcdae is: a, of length: 1
Longest Palindromic Substring in abacd is: aba, of length: 3
Longest Palindromic Substring in abcdc is: cdc, of length: 3
Longest Palindromic Substring in abacdfgdcaba is: aba, of length: 3
Longest Palindromic Substring in xyabacdfgdcaba is: aba, of length: 3
Longest Palindromic Substring in xababayz is: ababa, of length: 5
Longest Palindromic Substring in xabax is: xabax, of length: 5
```

Followup:

Detect ALL palindromes in a given string.

e.g. For string abcddcbefgf, all possible palindromes are a, b, c, d, e, f, g, dd, fgf, cddc, bcddcb.

We have published following more articles on suffix tree applications:

- [Suffix Tree Application 1 – Substring Check](#)
- [Suffix Tree Application 2 – Searching All Patterns](#)
- [Suffix Tree Application 3 – Longest Repeated Substring](#)
- [Suffix Tree Application 4 – Build Linear Time Suffix Array](#)
- [Suffix Tree Application 5 – Longest Common Substring](#)
- [Generalized Suffix Tree 1](#)

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

Source

<https://www.geeksforgeeks.org/suffix-tree-application-6-longest-palindromic-substring/>

Chapter 190

Sum of K largest elements in BST using O(1) Extra space

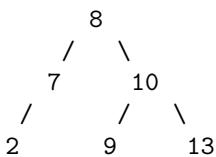
Sum of K largest elements in BST using O(1) Extra space - GeeksforGeeks

Given a BST, the task is to find the sum of all elements greater than or equal to K-th largest element in O(1) space.

Examples:

Examples:

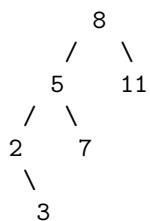
Input : K = 3



Output : 32

Explanation: 3rd largest element is 9 so sum of all elements greater than or equal to 9 are
9 + 10 + 13 = 32.

Input : K = 2



Output : 19

Explanation: 2nd largest element is 8 so sum of all

```
elements greater than or equal to 8 are  
8 + 11 = 19.
```

Approach: The approach here is to do reverse inorder traversal, and while doing it simply keep a count of the number of nodes visited. Until the count of visited nodes is less than equal to K, keep on adding the current node's data. Use the fact that the reverse inorder traversal of a BST gives us a list which is sorted in decreasing order. But recursion or stack/queue based approach to do reverse inorder traversal because both these techniques consume **O(n)** extra memory, instead make use of [Reverse Morris Traversal](#) to do inorder tree traversal, which is a memory efficient and faster method to do reverse inorder tree traversal based on [threaded binary trees](#).

Given below is the **algorithm**:

- 1) Initialize Current as root.
- 2) Initialize a "count" and "sum" variable to 0.
- 3) While current is not NULL :
 - 3.1) If the current has no right child
 - a) Increment count and check if count is less than or equal to K.
 - 1) Simply add the current node's data in "sum" variable.
 - b) Otherwise, Move to the left child of current.
 - 3.2) Else, here we have 2 cases:
 - a) Find the inorder successor of current Node.
Inorder successor is the left most Node
in the right subtree or right child itself.
 - b) If the left child of the inorder successor is NULL:
 - 1) Set current as the left child of its inorder successor.
 - 2) Move current Node to its right child.
 - c) Else, if the threaded link between the current Node
and it's inorder successor already exists :
 - 1) Set left pointer of the inorder successor as NULL.
 - 2) Increment count and check if the count is less than or equal to K.
 - 2.a) Simply add the current node's data in "sum" variable.
 - 3) Otherwise, Move current to it's left child.
 - 4) After the traversal is complete simply return the sum.

Below is the implementation of the above approach:

```
// C++ program to find sum of  
// K largest elements in BST using  
// Reverse Morris Traversal  
#include <bits/stdc++.h>  
using namespace std;
```

```
struct node {
    int data;
    struct node *left, *right;
};

// Add a new node
node* newNode(int item)
{
    node* temp = new node;
    temp->data = item;
    temp->left = temp->right = NULL;

    return temp;
}

// Function to find the sum of the K largest elements
// space efficient method used
int SumKLargestUsingReverseMorrisTraversal(node* root, int k)
{
    node* curr = root;
    int sum = 0;
    int count = 0;

    // while doing reverse inorder traversal
    // keep track of visited nodes
    while (curr) {
        if (curr->right == NULL) {

            // till count is less than k
            if (++count <= k) {
                sum += curr->data;
            }

            curr = curr->left;
        }
        else {
            // finding the inorder successor node
            // inorder successor is the left most in right subtree
            node* succ = curr->right;
            while (succ->left && succ->left != curr)
                succ = succ->left;

            if (succ->left == NULL) {
                succ->left = curr;

                curr = curr->right;
            }
        }
    }
}
```

```
}

// if the threaded link already exists then simply
// revert back the tree to original form.
else {
    succ->left = NULL;

    if (++count <= k)
        sum += curr->data;

    curr = curr->left;
}
}

return sum;
}

// Driver Code
int main()
{
    /* Constructed binary tree is
       8
      /   \
     7     10
    /   /   \
   2   9   13
*/
    struct node* root = newNode(8);

    root->right = newNode(10);
    root->left = newNode(7);
    root->left->left = newNode(2);

    root->right->left = newNode(9);
    root->right->right = newNode(13);

    cout << SumKLargestUsingReverseMorrisTraversal(root, 3);

    return 0;
}
```

Output:

Time Complexity: $O(N)$

Space complexity: $O(1)$

Source

<https://www.geeksforgeeks.org/sum-of-k-largest-elements-in-bst-using-o1-extra-space/>

Chapter 191

Sum over Subsets Dynamic Programming

Sum over Subsets Dynamic Programming - GeeksforGeeks

Prerequisite: [Basic Dynamic Programming, Bitmasks](#)

Consider the following problem where we will use Sum over subset Dynamic Programming to solve it.

Given an array of 2^n integers, we need to calculate function $F(x) = \sum_{i \in x} A_i$ such that $x \& i == i$ for all x . i.e, i is a bitwise subset of x . i will be a bitwise subset of mask x , if $x \& i == i$.

Examples:

Input: $A[] = \{7, 12, 14, 16\}$, $n = 2$
Output: 7, 19, 21, 49
Explanation: There will be 4 values of x : 0,1,2,3
So, we need to calculate $F(0), F(1), F(2), F(3)$.
Now, $F(0) = A_0 = 7$
 $F(1) = A_0 + A_1 = 19$
 $F(2) = A_0 + A_2 = 21$
 $F(3) = A_0 + A_1 + A_2 + A_3 = 49$

Input: $A[] = \{7, 11, 13, 16\}$, $n = 2$
Output: 7, 18, 20, 47
Explanation: There will be 4 values of x : 0,1,2,3
So, we need to calculate $F(0), F(1), F(2), F(3)$.
Now, $F(0) = A_0 = 7$
 $F(1) = A_0 + A_1 = 18$
 $F(2) = A_0 + A_2 = 20$
 $F(3) = A_0 + A_1 + A_2 + A_3 = 47$

Brute-Force Approach:

Iterate for all the x from 0 to $(2^n - 1)$. Calculate the **bitwise subsets** of all the x and sum it up for every x.

Time-Complexity: $O(4^n)$

Below is the implementation of above idea:

CPP

```
// CPP program for brute force
// approach of SumOverSubsets DP
#include <bits/stdc++.h>
using namespace std;

// function to print the sum over subsets value
void SumOverSubsets(int a[], int n) {

    // array to store the SumOverSubsets
    int sos[1 << n] = {0};

    // iterate for all possible x
    for (int x = 0; x < (1 << n); x++) {

        // iterate for all possible bitwise subsets
        for (int i = 0; i < (1 << n); i++) {

            // if i is a bitwise subset of x
            if ((x & i) == i)
                sos[x] += a[i];
        }
    }

    // prints all the subsets
    for (int i = 0; i < (1 << n); i++)
        cout << sos[i] << " ";
}

// Driver Code
int main() {
    int a[] = {7, 12, 14, 16};
    int n = 2;
    SumOverSubsets(a, n);
    return 0;
}
```

Java

```
// Java program for brute force
```

```
// approach of SumOverSubsets DP

class GFG{

    // function to print the
    // sum over subsets value
    static void SumOverSubsets(int a[], int n) {

        // array to store the SumOverSubsets
        int sos[] = new int [1 << n];

        // iterate for all possible x
        for (int x = 0; x < (1 << n); x++) {

            // iterate for all possible
            // bitwise subsets
            for (int i = 0; i < (1 << n); i++) {

                // if i is a bitwise subset of x
                if ((x & i) == i)
                    sos[x] += a[i];
            }
        }

        // prints all the subsets
        for (int i = 0; i < (1 << n); i++)
            System.out.printf("%d ", sos[i]);
    }

    // Driver Code
    public static void main(String[] args) {
        int a[] = {7, 12, 14, 16};
        int n = 2;
        SumOverSubsets(a, n);
    }
}

// This code is contributed by
// Smitha Dinesh Semwal
```

Python3

```
# Python 3 program
# for brute force
# approach of SumOverSubsets DP

# function to print the
```

```
# sum over subsets value
def SumOverSubsets(a, n):

    # array to store
    # the SumOverSubsets
    sos = [0] * (1 << n)

    # iterate for all possible x
    for x in range(0,(1 << n)):

        # iterate for all
        # possible bitwise subsets
        for i in range(0,(1 << n)):

            # if i is a bitwise subset of x
            if ((x & i) == i):
                sos[x] += a[i]

    # prints all the subsets
    for i in range(0,(1 << n)):
        print(sos[i],end = " ")

# Driver Code
a = [7, 12, 14, 16]
n = 2
SumOverSubsets(a, n)

# This code is contributed by
# Smitha Dinesh Semwal
```

C#

```
// C# program for brute force
// approach of SumOverSubsets DP
using System;

class GFG {

    // function to print the
    // sum over subsets value
    static void SumOverSubsets(int []a, int n)
    {

        // array to store the SumOverSubsets
        int []sos = new int [1 << n];
```

```
// iterate for all possible x
for (int x = 0; x < (1 << n); x++)
{
    // iterate for all possible
    // bitwise subsets
    for (int i = 0; i < (1 << n); i++)
    {
        // if i is a bitwise subset of x
        if ((x & i) == i)
            sos[x] += a[i];
    }
}

// prints all the subsets
for (int i = 0; i < (1 << n); i++)
    Console.Write(sos[i] + " ");
}

// Driver function
public static void Main()
{
    int []a = {7, 12, 14, 16};
    int n = 2;
    SumOverSubsets(a, n);
}
}

// This code is contributed by Sam007
```

PHP

```
<?php
// PHP program for brute force
// approach of SumOverSubsets DP

// function to print the sum
// over subsets value
function SumOverSubsets($a, $n)
{

    // array to store the SumOverSubsets
    $sos = array(1 << $n);

    for($i = 0 ;$i < (1 << $n); $i++)
}
```

```

$sos[$i] = 0;

// iterate for all possible x
for ($x = 0; $x < (1 << $n); $x++)
{
    // iterate for all possible
    // bitwise subsets
    for($i = 0; $i < (1 << $n); $i++)
    {
        // if i is a bitwise
        // subset of x
        if (($x & $i) == $i)
            $sos[$x] += $a[$i];
    }
}

// prints all the subsets
for ($i = 0; $i < (1 << $n); $i++)
    echo $sos[$i] . " ";
}

// Driver Code
$a = array(7, 12, 14, 16);
$n = 2;
SumOverSubsets($a, $n);

// This code is contributed by Sam007
?>

```

Output:

7 19 21 49

Sub-Optimal Approach:

The brute-force algorithm can be easily improved by just iterating over bitwise subsets. Instead of iterating for every i , we can simply iterate for the bitwise subsets only. Iterating backward for $i=(i-1)\&x$ gives us every bitwise subset, where i starts from x and ends at 1. If the mask x has k set bits, we do 2^k iterations. A number of k set bits will have 2^k

bitwise subsets. Therefore total number of mask x with k set bits is $\binom{3^n}{k}$. Therefore the total number of iterations is $\binom{3^n}{k} \cdot 2^k = 3^n$

Time Complexity: $O(3^n)$

Below is the implementation of above idea:

C++

```
// CPP program for sub-optimal
// approach of SumOverSubsets DP
#include <bits/stdc++.h>
using namespace std;

// function to print the sum over subsets value
void SumOverSubsets(int a[], int n) {

    // array to store the SumOverSubsets
    int sos[1 << n] = {0};

    // iterate for all possible x
    for (int x = 0; x < (1 << n); x++) {
        sos[x] = a[0];

        // iterate for the bitwise subsets only
        for (int i = x; i > 0; i = (i - 1) & x)
            sos[x] += a[i];
    }

    // print all the subsets
    for (int i = 0; i < (1 << n); i++)
        cout << sos[i] << " ";
}

// Driver Code
int main() {
    int a[] = {7, 12, 14, 16};
    int n = 2;
    SumOverSubsets(a, n);
    return 0;
}
```

Java

```
// java program for sub-optimal
// approach of SumOverSubsets DP
public class GFG {

    // function to print the sum over
    // subsets value
    static void SumOverSubsets(int a[], int n)
    {
```

```

// array to store the SumOverSubsets
int sos[] = new int[(1 << n)];

// iterate for all possible x
for (int x = 0; x < (1 << n); x++) {
    sos[x] = a[0];

    // iterate for the bitwise subsets only
    for (int i = x; i > 0; i = (i - 1) & x)
        sos[x] += a[i];
}

// print all the subsets
for (int i = 0; i < (1 << n); i++)
    System.out.print(sos[i] + " ");
}

// Driver code
public static void main(String args[])
{
    int a[] = {7, 12, 14, 16};
    int n = 2;

    SumOverSubsets(a, n);
}
}

// This code is contributed by Sam007

```

C#

```

// C# program for sub-optimal
// approach of SumOverSubsets DP
using System;

class GFG {

    // function to print the sum over
    // subsets value
    static void SumOverSubsets(int []a, int n)
    {

        // array to store the SumOverSubsets
        int []sos = new int[(1 << n)];

        // iterate for all possible x
        for (int x = 0; x < (1 << n); x++) {
            sos[x] = a[0];

```

```
// iterate for the bitwise subsets only
for (int i = x; i > 0; i = (i - 1) & x)
    sos[x] += a[i];
}

// print all the subsets
for (int i = 0; i < (1 << n); i++)
    Console.WriteLine(sos[i] + " ");
}

// Driver code
static void Main()
{
    int []a = {7, 12, 14, 16};
    int n = 2;

    SumOverSubsets(a, n);
}
}

// This code is contributed by Sam007.
```

PHP

```
<?php
// PHP program for sub-optimal
// approach of SumOverSubsets DP

// function to print the
// sum over subsets value
function SumOverSubsets($a,$n)
{

    // array to store the SumOverSubsets
    $sos=array(1 << $n);

    // iterate for all possible x
    for ($x = 0; $x < (1 << $n); $x++)
    {
        $sos[$x] = $a[0];

        // iterate for the bitwise
        // subsets only
        for ($i = $x; $i > 0; $i = ($i - 1) & $x)
            $sos[$x] += $a[$i];
    }
}
```

```

// print all the subsets
for ($i = 0; $i < (1 << $n); $i++)
    echo $sos[$i] . " ";
}

// Driver Code
$a = array(7, 12, 14, 16);
$n = 2;
SumOverSubsets($a, $n);

// This code is contributed by Sam007.
?>

```

Output:

7 19 21 49

Dynamic Programming Approach:

In the brute-force approach, we iterated for every possible i for each mask x . We check if it was a bitwise subset and then summed it. In the sub-optimal approach, we iterated over the bitwise subsets only which reduced the complexity from $O(4^n)$ to $O(3^n)$. Having a closer look at the mask and the bitwise subset of every mask, we observe that we are performing repetitive calculations which can be reduced by memoization using Dynamic Programming. An index which has an off bit or an on bit is being visited by 2^n masks more than once.

Let $\text{DP}(\text{mask}, i)$ be the set of only those subsets of mask which differ in first i bits (zero-based from right).

For example Let mask be 10110101 in binary and i be 3, than those subsets which differ in first i bits (zero-based from right).

Example:

(1011010, 1010010, 1011000, 1010000).

We will find repetitive masks whose first i bits will be same then the same bitwise subsets will be formed. We can memoize to obtain the previous results and reduce the number of steps by a significant amount.

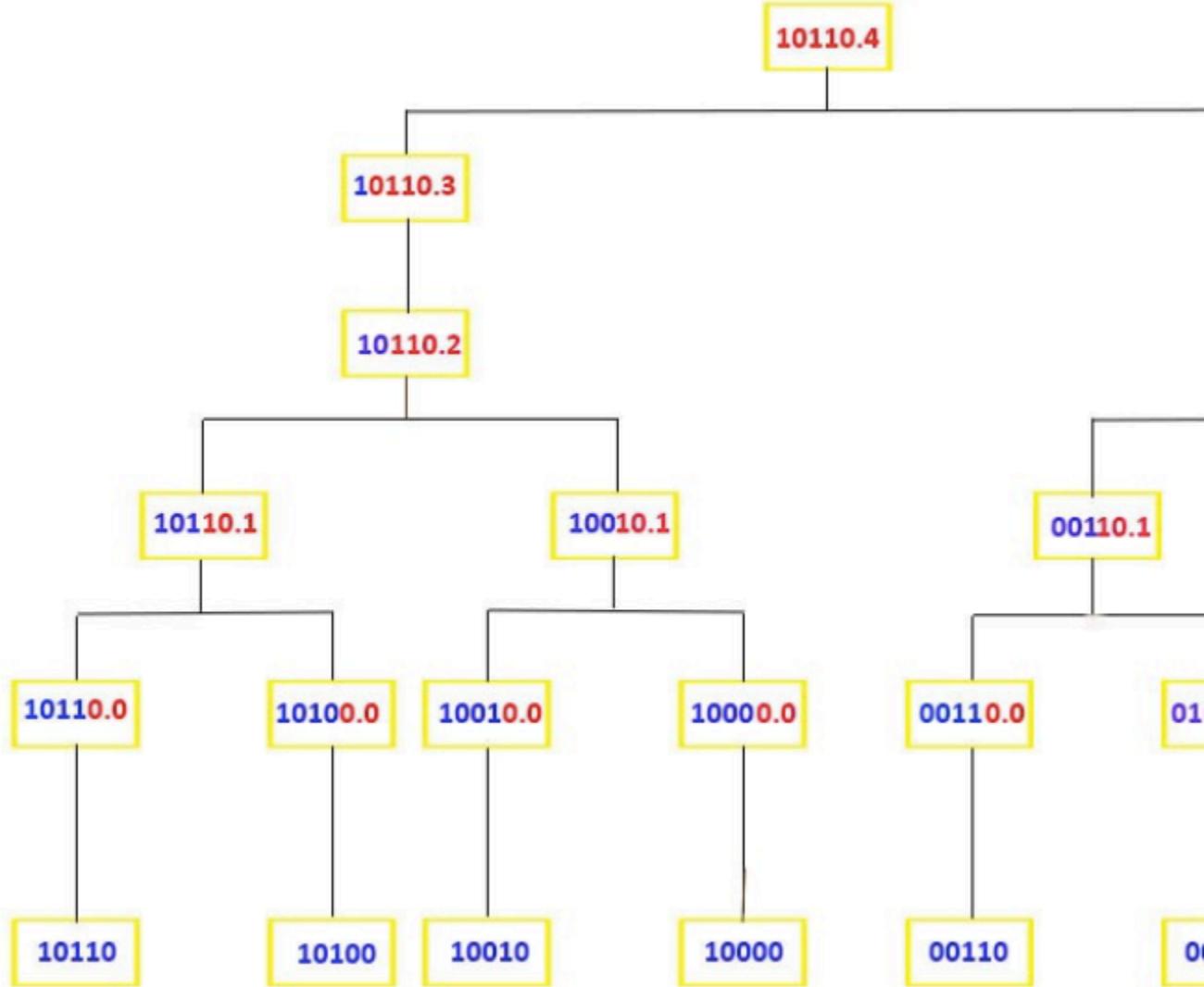
Let's consider the **i-th bit to be 0**, then no subset can differ from the mask in the i -th bit as it would mean that the numbers will have a 1 at i -th bit where the mask has a 0 which would mean that it is not a subset of the mask. Thus we conclude that the numbers now differ in the first $(i-1)$ bits only. Hence,

$\text{DP}(\text{mask}, i) = \text{DP}(\text{mask}, i-1)$

Now the **second case, if the i-th bit is 1**, it can be divided into two non-intersecting sets. One containing numbers with i -th bit as 1 and differing from mask in the next $(i-1)$

bits. Second containing numbers with ith bit as 0 and differing from mask $\oplus_{j=1}^{i-1}(2^j)$ in next $(i-1)$ bits. Hence,

$$DP(\text{mask}, i) = DP(\text{mask}, i-1) \cup DP(\text{mask} \oplus_{j=1}^{i-1}(2^j), i-1).$$



The above diagram explains how we can relate the $DP(\text{mask}, i)$ sets on each other. The mask is represented in binary and is separated by a “.” with i . Elements of any set $DP(\text{mask}, i)$ are the leaves in its subtree. The red-blue prefixes depict that this part of the mask will be common to all its members/children while the red part of the mask is allowed to differ.

Looking at the rooted tree, we can figure out that for the same value of i , it can have a different value of mask.

Hence the two recurrence relations are:

When i -th bit is off:

$$1. \text{DP}(\text{mask}, i) = \text{DP}(\text{mask}, i-1)$$

When i -th bit is on:

$$2. \text{DP}(\text{mask}, i) = \text{DP}(\text{mask}, i-1) \cup \text{DP}(\text{mask} \oplus (1 \ll i), i-1).$$

Below is the implementation of above idea:

```
// CPP program for Dynamic Programming
// approach of SumOverSubsets DP

#include <bits/stdc++.h>
using namespace std;

const int N = 1000;

// function to print the sum over subsets value
void SumOverSubsets(int a[], int n) {

    // array to store the SumOverSubsets
    int sos[1 << n] = {0};

    int dp[N][N];

    // iterate for all possible x
    for (int x = 0; x < (1 << n); x++) {
        // iterate till n
        for (int i = 0; i < n; i++) {
            // if i-th bit is set
            if (x & (1 << i)) {
                if (i == 0)
                    dp[x][i] = a[x] + a[x ^ (1 << i)];
                else // dp recurrence
                    dp[x][i] = dp[x][i - 1] +
                               dp[x ^ (1 << i)][i - 1];
            }
            else // i-th bit is not set
            {
                if (i == 0)
                    dp[x][i] = a[x];
                else
                    dp[x][i] = dp[x][i - 1];
            }
        }
    }

    cout << "Sum over Subsets value is: " << sos[1 << n] << endl;
}
```

```
        dp[x][i] = a[x]; // base condition
    else
        dp[x][i] = dp[x][i - 1]; // dp recurrence
    }
}

// stores the final sum of subset of mask x
sos[x] = dp[x][n - 1];
}

// print all the sum of subsets
for (int i = 0; i < (1 << n); i++)
    cout << sos[i] << " ";
}

// Driver Code
int main()
{
    int a[] = {7, 12, 14, 16};
    int n = 2;
    SumOverSubsets(a, n);
    return 0;
}
```

Output:

7 19 21 49

Time Complexity: $O(n \cdot 2^n)$
Auxiliary Space: $O(n^2)$

Reference:

<http://home.iitk.ac.in/~gsahil/cs498a/report.pdf>

Improved By : [Sam007](#)

Source

<https://www.geeksforgeeks.org/sum-subsets-dynamic-programming/>

Chapter 192

Summed Area Table – Submatrix Summation

Summed Area Table - Submatrix Summation - GeeksforGeeks

Given a matrix of size M x N, there are large number of queries to find submatrix sums. Inputs to queries are left top and right bottom indexes of submatrix whose sum is to find out.

How to preprocess the matrix so that submatrix sum queries can be performed in O(1) time.

Example:

```
tli : Row number of top left of query submatrix  
tlj : Column number of top left of query submatrix  
rbi : Row number of bottom right of query submatrix  
rbj : Column number of bottom right of query submatrix
```

```
Input: mat[M][N] = {{1, 2, 3, 4, 6},  
                    {5, 3, 8, 1, 2},  
                    {4, 6, 7, 5, 5},  
                    {2, 4, 8, 9, 4}};  
Query1: tli = 0, tlj = 0, rbi = 1, rbj = 1  
Query2: tli = 2, tlj = 2, rbi = 3, rbj = 4  
Query3: tli = 1, tlj = 2, rbi = 3, rbj = 3;
```

```
Output:  
Query1: 11 // Sum between (0, 0) and (1, 1)  
Query2: 38 // Sum between (2, 2) and (3, 4)  
Query3: 38 // Sum between (1, 2) and (3, 3)
```

Naive Algorithm:

We can loop all the queries and calculate each query in $O(q^*(N*M))$ worst case which is too large for a large range of numbers.

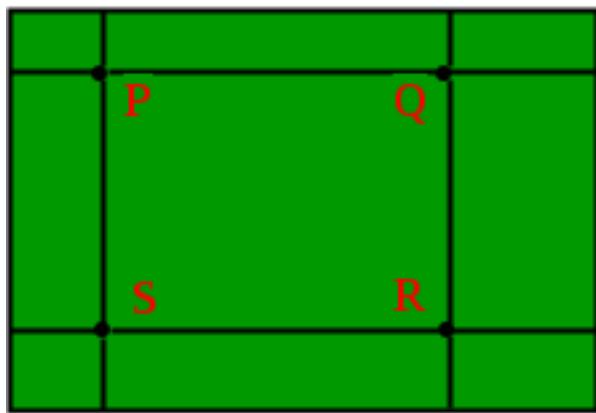
```
// Pseudo code of Naive algorithm.
Arr [] [] = input_matrix
For each query:
    Input tli, tlj, rbi, rbj
    sum = 0
    for i from tli to tbi (inclusive):
        for j from tlj to rbj(inclusive):
            sum += Arr[i] [j]
    print(sum)
```

Optimized Solution :

[Summed Area Table](#) can reduce this type of query into preprocessing time of $O(M*N)$ and each query will execute in $O(1)$.

Summed Area Table is a data structure and algorithm for quickly and efficiently generating the sum of values in a rectangular subset of a grid.

The value at any point (x, y) in the summed area table is just the sum of all the values above and to the left of (x, y) , inclusive :



$$\text{Sum} = R - Q - S + P$$

The optimized solution is implemented in below post.

[Implementation of optimized approach](#)

Source

<https://www.geeksforgeeks.org/summed-area-table-submatrix-summation/>

Chapter 193

Tango Tree Data Structure

Tango Tree Data Structure - GeeksforGeeks

Tango Tree is an [online algorithm](#). It is a type of binary search tree. It is better than the [offline weight balanced binary search tree](#) as it achieves a [competitive ratio](#) of $O(\log \log n)$ as compared to competitive ratio of $O(\log n)$ of offline weight balanced binary search tree. It takes only $O(\log \log n)$ of additional bits of memory at every node of the tree to increase the performance over weight balanced binary search tree.

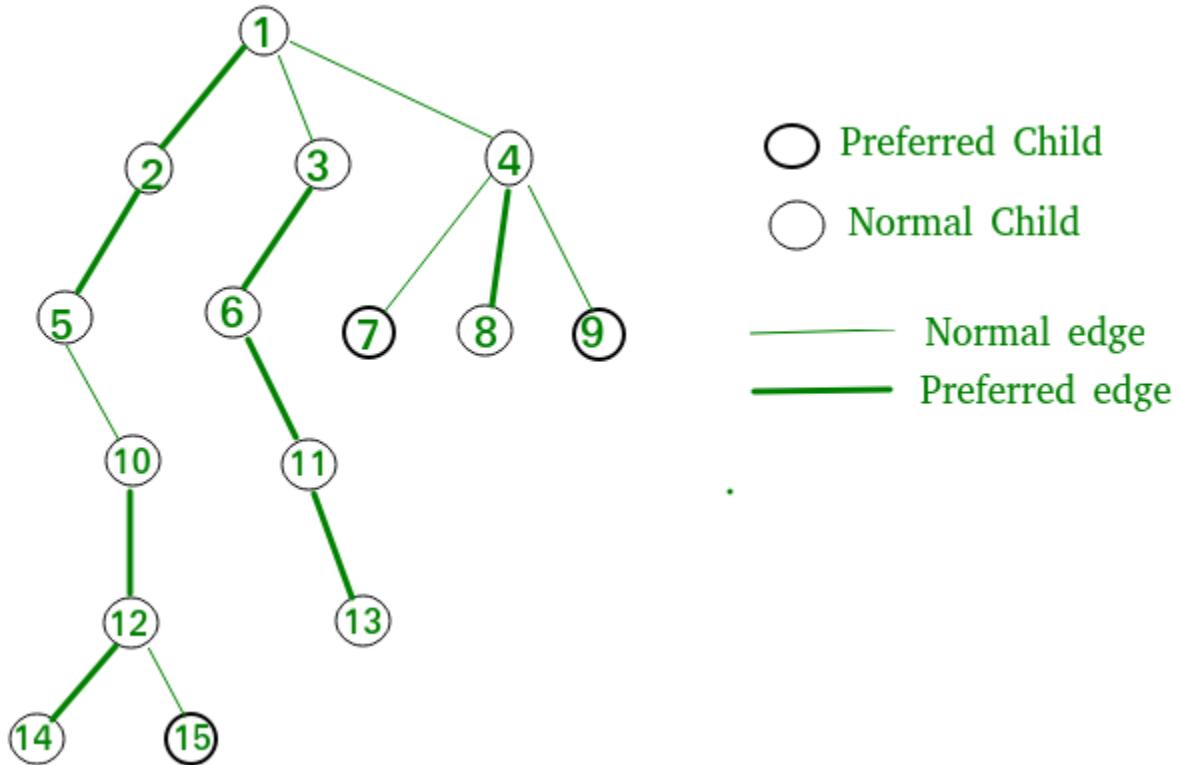
Tango tree is conceptually a tree of trees as it partitions a binary search tree into *preferred paths* and *non preferred paths* and these *preferred paths* are themselves stored in the form of auxiliary trees.

Preferred child and Preferred Paths

Preferred child of any node of Tango tree is defined as the most recently touched child by a normal binary search tree lookup technique. Elaborating more on preferred child let us assume a subtree T rooted at node n with right child as p and left child as q. Call p as the preferred child of n if most recently accessed node of subtree rooted at n is present in the subtree rooted at p.

A **Preferred path** is defined as the path starting from root node and following all the

preferred children along a path to reach a leaf node.



Tango Tree

Auxiliary Trees for Preferred Paths

A preferred path is represented by storing the nodes of the preferred path in a balanced binary search tree specifically red black tree. Then, for every non-leaf node of path P there is a non-preferred child q . This non preferred child acts as the root of the new auxiliary tree and then attach this new auxiliary tree rooted at q to its parent and in this way, it is easy to link the auxiliary trees together. There is also option to store some extra information in each node of the auxiliary tree as per our own needs like minimum depth of the nodes in the subtree below it, etc.

Operations on Tango Tree

The two most common operations on Tango Trees are *Searching* and *Updating*

- Updating in a Tango Tree
 1. Cut process
 2. Join process

Searching in a tango tree

To search in a Tango tree we simply carry out the search in the conceptual binary search tree. First search the preferred path by searching its auxiliary tree. If the node cannot be found in the given auxiliary tree then, search the auxiliary tree of the next path and so on until search of the entire tree or could not find the desired node.

Source

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

Chapter 194

Tarjan's off-line lowest common ancestors algorithm

Tarjan's off-line lowest common ancestors algorithm - GeeksforGeeks

Prerequisite : [LCA basics](#), [Disjoint Set Union by Rank and Path Compression](#)

We are given a tree(can be extended to a DAG) and we have many queries of form LCA(u, v), i.e., find LCA of nodes 'u' and 'v'.

We can perform those queries in **O(N + QlogN)** time [using RMQ](#), where O(N) time for pre-processing and O(log N) for answering the queries, where

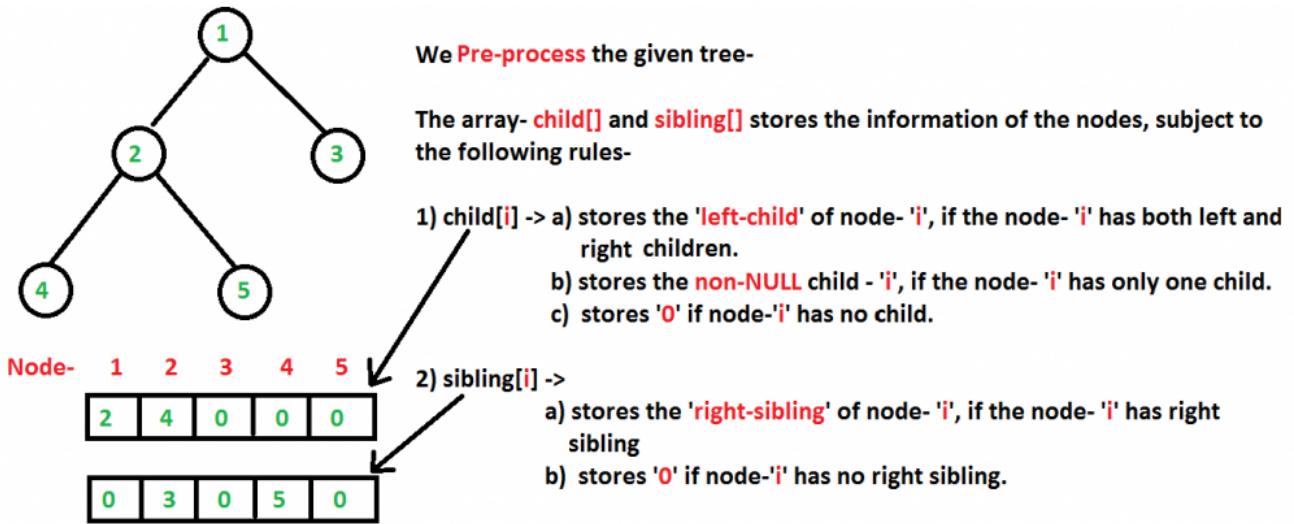
N = number of nodes and

Q = number of queries to be answered.

Can we do better than this? Can we do in linear(almost) time? Yes.

The article presents an offline algorithm which performs those queries in approximately **O(N + Q)** time. Although, this is not exactly linear, as there is an Inverse Ackermann function involved in the time complexity analysis. For more details on Inverse Ackermann function see [this](#). Just as a summary, we can say that the Inverse Ackermann Function remains less than 4, for any value of input size that can be written in physical inverse. Thus, we consider this as almost linear.

We consider the input tree as shown below. We will **Pre-Process** the tree and fill two arrays- **child[]** and **sibling[]** according to the below explanation-



Let we want to process these queries- **LCA(5,4)**, **LCA(1,3)**, **LCA(2,3)**

Now, after pre-processing, we perform a **LCA walk** starting from the root of the tree(here-node '1'). But prior to the LCA walk, we colour all the nodes with **WHITE**. During the whole LCA walk, we use three disjoint set union functions- `makeSet()`, `findSet()`, `unionSet()`. These functions use the technique of union by rank and path compression to improve the running time. During the LCA walk, our queries gets processed and outputted (in a random order). After the LCA walk of the whole tree, all the nodes gets coloured **BLACK**.

Tarjan Offline LCA Algorithm steps from CLRS, Section-21-3, Pg 584, 2nd /3rd edition.

21-3 Tarjan's off-line least-common-ancestors algorithm

The **least common ancestor** of two nodes u and v in a rooted tree T is the node w that is an ancestor of both u and v and that has the greatest depth in T . In the **off-line least-common-ancestors problem**, we are given a rooted tree T and an arbitrary set $P = \{\{u, v\}\}$ of unordered pairs of nodes in T , and we wish to determine the least common ancestor of each pair in P .

To solve the off-line least-common-ancestors problem, the following procedure performs a tree walk of T with the initial call $\text{LCA}(T.\text{root})$. We assume that each node is colored **WHITE** prior to the walk.

$\text{LCA}(u)$

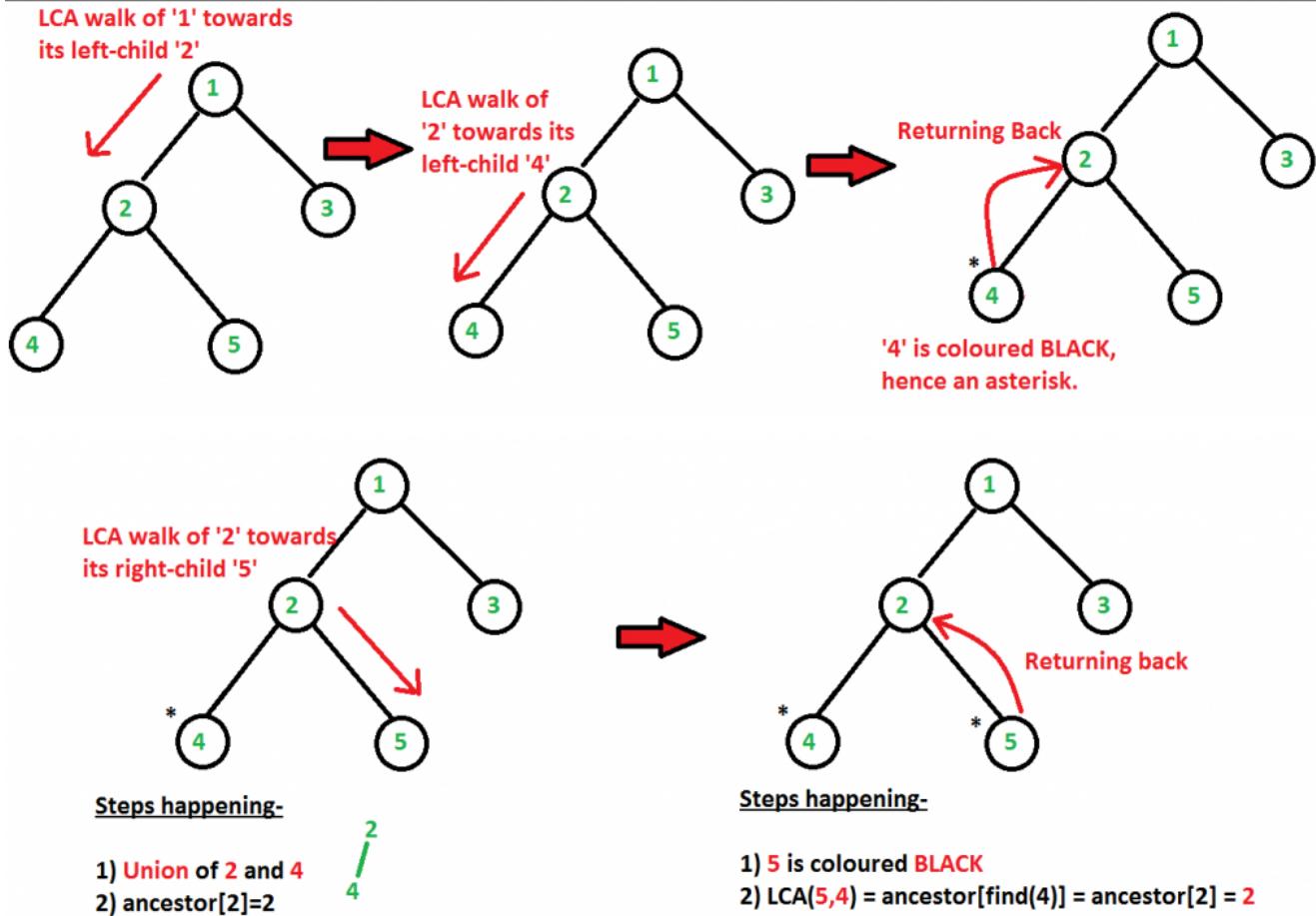
```

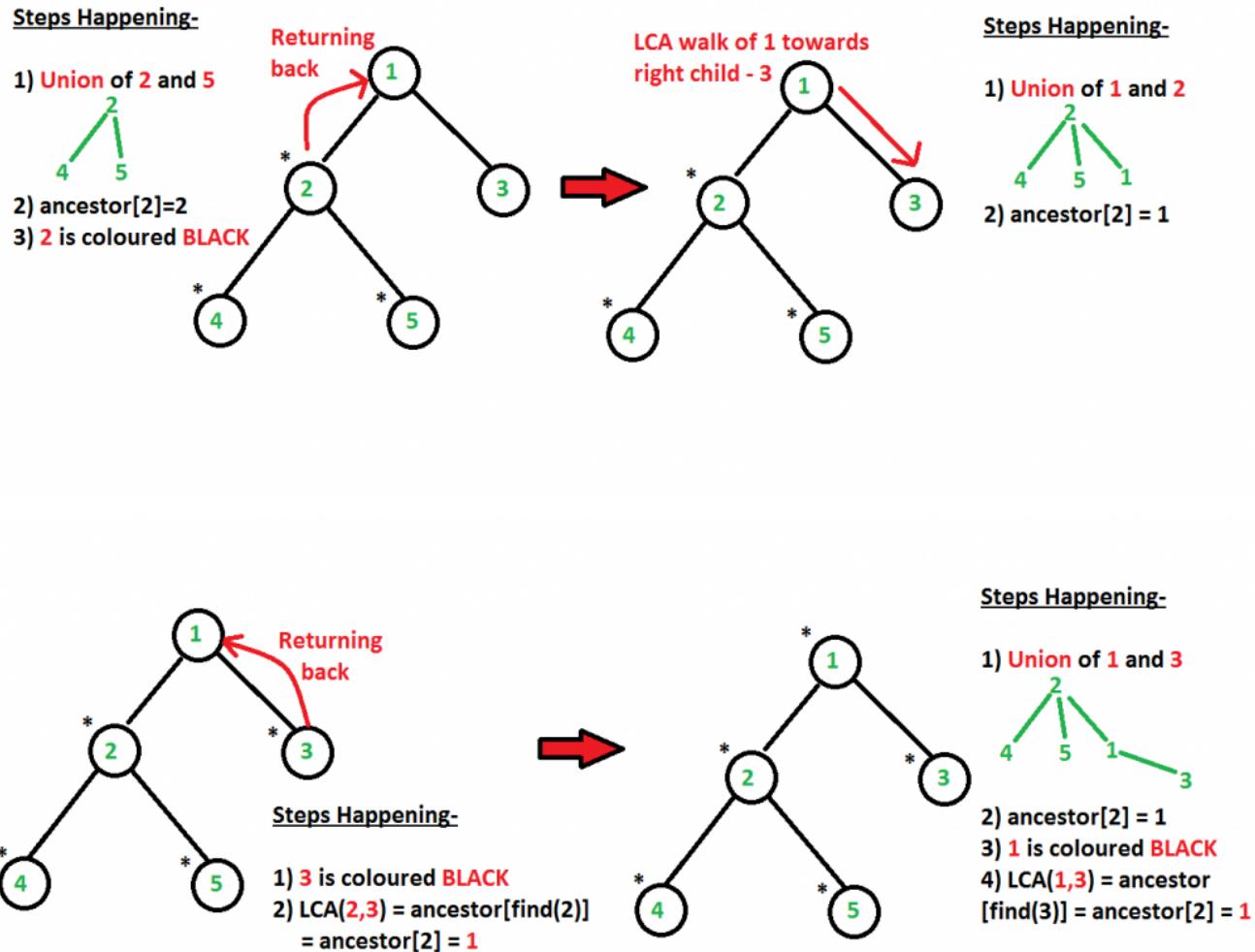
1  MAKE-SET( $u$ )
2  FIND-SET( $u$ ).ancestor =  $u$ 
3  for each child  $v$  of  $u$  in  $T$ 
4    LCA( $v$ )
5    UNION( $u, v$ )
6    FIND-SET( $u$ ).ancestor =  $u$ 
7   $u.\text{color}$  = BLACK
8  for each node  $v$  such that  $\{u, v\} \in P$ 
9    if  $v.\text{color} == \text{BLACK}$ 
10       print "The least common ancestor of"
            $u$  "and"  $v$  "is" FIND-SET( $v$ ).ancestor
```

Note- The queries may not be processed in the original order. We can easily modify the process and sort them according to the input order.

The below pictures clearly depict all the steps happening. The red arrow shows the direction of travel of our **recursive function LCA()**.

Initially, prior to the LCA walk, all nodes are coloured **WHITE**, and during the LCA walk the nodes gets coloured **BLACK**. We denote a node to be coloured **BLACK**, by putting an asterisk '*' beside that node, and a **WHITE** node doesn't has an asterisk '*' beside it. Note that LCA walk of a node consists of two part- **towards its left-child** and **towards its right-child**.





As, we can clearly see from the above pictures, the queries are processed in the following order, LCA(5,4), LCA(2,3), LCA(1,3) which is not in the same order as the input(LCA(5,4), LCA(1,3), LCA(2,3)).

Below is C++ implementation.

```
// A C++ Program to implement Tarjan Offline LCA Algorithm
#include <bits/stdc++.h>

#define V 5      // number of nodes in input tree
#define WHITE 1  // COLOUR 'WHITE' is assigned value 1
#define BLACK 2  // COLOUR 'BLACK' is assigned value 2
```

```

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

/*
subset[i].parent-->Holds the parent of node-'i'
subset[i].rank-->Holds the rank of node-'i'
subset[i].ancestor-->Holds the LCA queries answers
subset[i].child-->Holds one of the child of node-'i'
    if present, else -'0'
subset[i].sibling-->Holds the right-sibling of node-'i'
    if present, else -'0'
subset[i].color-->Holds the colour of node-'i'
*/
struct subset
{
    int parent, rank, ancestor, child, sibling, color;
};

// Structure to represent a query
// A query consists of (L,R) and we will process the
// queries offline a/c to Tarjan's offline LCA algorithm
struct Query
{
    int L, R;
};

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

//A utility function to make set
void makeSet(struct subset subsets[], int i)
{
    if (i < 1 || i > V)
        return;

    subsets[i].color = WHITE;
}

```

```

subsets[i].parent = i;
subsets[i].rank = 0;

return;
}

// A utility function to find set of an element i
// (uses path compression technique)
int findSet(struct subset subsets[], int i)
{
    // find root and make root as parent of i (path compression)
    if (subsets[i].parent != i)
        subsets[i].parent = findSet (subsets, subsets[i].parent);

    return subsets[i].parent;
}

// A function that does union of two sets of x and y
// (uses union by rank)
void unionSet(struct subset subsets[], int x, int y)
{
    int xroot = findSet (subsets, x);
    int yroot = findSet (subsets, y);

    // Attach smaller rank tree under root of high rank tree
    // (Union by Rank)
    if (subsets[xroot].rank < subsets[yroot].rank)
        subsets[xroot].parent = yroot;
    else if (subsets[xroot].rank > subsets[yroot].rank)
        subsets[yroot].parent = xroot;

    // If ranks are same, then make one as root and increment
    // its rank by one
    else
    {
        subsets[yroot].parent = xroot;
        (subsets[xroot].rank)++;
    }
}

// The main function that prints LCAs. u is root's data.
// m is size of q[]
void lcaWalk(int u, struct Query q[], int m,
             struct subset subsets[])
{
    // Make Sets
    makeSet(subsets, u);
}

```

```

// Initially, each node's ancestor is the node
// itself.
subsets[findSet(subsets, u)].ancestor = u;

int child = subsets[u].child;

// This while loop doesn't run for more than 2 times
// as there can be at max. two children of a node
while (child != 0)
{
    lcaWalk(child, q, m, subsets);
    unionSet (subsets, u, child);
    subsets[findSet(subsets, u)].ancestor = u;
    child = subsets[child].sibling;
}

subsets[u].color = BLACK;

for (int i = 0; i < m; i++)
{
    if (q[i].L == u)
    {
        if (subsets[q[i].R].color == BLACK)
        {
            printf("LCA(%d %d) -> %d\n",
                   q[i].L,
                   q[i].R,
                   subsets[findSet(subsets,q[i].R)].ancestor);
        }
    }
    else if (q[i].R == u)
    {
        if (subsets[q[i].L].color == BLACK)
        {
            printf("LCA(%d %d) -> %d\n",
                   q[i].L,
                   q[i].R,
                   subsets[findSet(subsets,q[i].L)].ancestor);
        }
    }
}

return;
}

// This is basically an inorder traversal and
// we preprocess the arrays-> child[]
// and sibling[] in "struct subset" with

```

```

// the tree structure using this function.
void preprocess(Node * node, struct subset subsets[])
{
    if (node == NULL)
        return;

    // Recur on left child
    preprocess(node->left, subsets);

    if (node->left != NULL&&node->right != NULL)
    {
        /* Note that the below two lines can also be this-
        subsets[node->data].child = node->right->data;
        subsets[node->right->data].sibling =
            node->left->data;

        This is because if both left and right children of
        node-'i' are present then we can store any of them
        in subsets[i].child and correspondingly its sibling*/
        subsets[node->data].child = node->left->data;
        subsets[node->left->data].sibling =
            node->right->data;

    }
    else if ((node->left != NULL && node->right == NULL)
             || (node->left == NULL && node->right != NULL))
    {
        if(node->left != NULL && node->right == NULL)
            subsets[node->data].child = node->left->data;
        else
            subsets[node->data].child = node->right->data;
    }

    //Recur on right child
    preprocess (node->right, subsets);
}

// A function to initialise prior to pre-processing and
// LCA walk
void initialise(struct subset subsets[])
{
    // Initialising the structure with 0's
    memset(subsets, 0, (V+1) * sizeof(struct subset));

    // We colour all nodes WHITE before LCA Walk.
    for (int i=1; i<=V; i++)
        subsets[i].color=WHITE;
}

```

```

        return;
    }

// Prints LCAs for given queries q[0..m-1] in a tree
// with given root
void printLCAs(Node *root, Query q[], int m)
{
    // Allocate memory for V subsets and nodes
    struct subset * subsets = new subset[V+1];

    // Creates subsets and colors them WHITE
    initialise(subsets);

    // Preprocess the tree
    preprocess(root, subsets);

    // Perform a tree walk to process the LCA queries
    // offline
    lcaWalk(root->data , q, m, subsets);
}

// Driver program to test above functions
int main()
{
    /*
     * We construct a binary tree :-
     *
     *      1
     *     / \
     *    2   3
     *   / \
     *  4   5
     */
    Node *root = newNode(1);
    root->left      = newNode(2);
    root->right     = newNode(3);
    root->left->left  = newNode(4);
    root->left->right = newNode(5);

    // LCA Queries to answer
    Query q[] = {{5, 4}, {1, 3}, {2, 3}};
    int m = sizeof(q)/sizeof(q[0]);

    printLCAs(root, q, m);

    return 0;
}

```

Output :

```
LCA(5 4) -> 2  
LCA(2 3) -> 1  
LCA(1 3) -> 1
```

Time Complexity : Super-linear, i.e- barely slower than linear. $O(N + Q)$ time, where $O(N)$ time for pre-processing and almost $O(1)$ time for answering the queries.

Auxiliary Space : We use many arrays- parent[], rank[], ancestor[] which are used in Disjoint Set Union Operations each with the size equal to the number of nodes. We also use the arrays- child[], sibling[], color[] which are useful in this offline algorithm. Hence, we use $O(N)$.

For convenience, all these arrays are put up in a structure- struct subset to hold these arrays.

References

https://en.wikipedia.org/wiki/Tarjan%27s_off-line_lowest_common_ancestors_algorithm

CLRS, Section-21-3, Pg 584, 2nd /3rd edition

http://wcipeg.com/wiki/Lowest_common_ancestor#Offline

Source

<https://www.geeksforgeeks.org/tarjans-off-line-lowest-common-ancestors-algorithm/>

Chapter 195

Ternary Search Tree

Ternary Search Tree - GeeksforGeeks

A ternary search tree is a special trie data structure where the child nodes of a standard trie are ordered as a binary search tree.

Representation of ternary search trees:

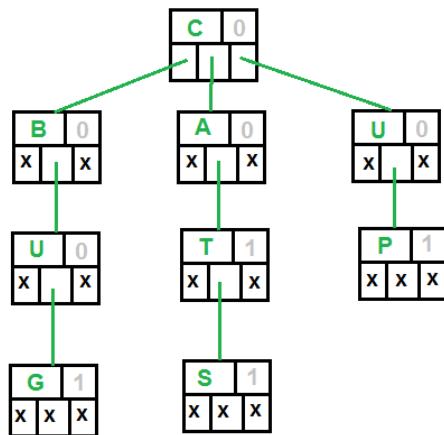
Unlike trie(standard) data structure where each node contains 26 pointers for its children, each node in a ternary search tree contains only 3 pointers:

1. The left pointer points to the node whose value is less than the value in the current node.
2. The equal pointer points to the node whose value is equal to the value in the current node.
3. The right pointer points to the node whose value is greater than the value in the current node.

Apart from above three pointers, each node has a field to indicate data(character in case of dictionary) and another field to mark end of a string.

So, more or less it is similar to BST which stores data based on some order. However, data in a ternary search tree is distributed over the nodes. e.g. It needs 4 nodes to store the word “Geek”.

Below figure shows how exactly the words in a ternary search tree are stored?



Ternary Search Tree for CAT, BUG, CATS, UP

Following are the 5 fields in a node

- 1) The data (a character)
- 2) isEndOfString bit (0 or 1). It may be 1 for nonleaf nodes (the node with character T)
- 3) Left Pointer
- 4) Equal Pointer
- 5) Right Pointer

One of the advantage of using ternary search trees over tries is that ternary search trees are a more space efficient (involve only three pointers per node as compared to 26 in standard tries). Further, ternary search trees can be used any time a hashtable would be used to store strings.

Tries are suitable when there is a proper distribution of words over the alphabets so that spaces are utilized most efficiently. Otherwise ternary search trees are better. Ternary search trees are efficient to use(in terms of space) when the strings to be stored share a common prefix.

Applications of ternary search trees:

1. Ternary search trees are efficient for queries like “Given a word, find the next word in dictionary(near-neighbor lookups)” or “Find all telephone numbers starting with 9342 or “typing few starting characters in a web browser displays all website names with this prefix”(Auto complete feature)”.
2. Used in spell checks: Ternary search trees can be used as a dictionary to store all the words. Once the word is typed in an editor, the word can be parallelly searched in the ternary search tree to check for correct spelling.

Implementation:

Following is C implementation of ternary search tree. The operations implemented are, search, insert and traversal.

```

// C program to demonstrate Ternary Search Tree (TST) insert, traverse
// and search operations
#include <stdio.h>
#include <stdlib.h>
#define MAX 50

// A node of ternary search tree
struct Node
{

```

```

char data;

// True if this character is last character of one of the words
unsigned isEndOfString: 1;

struct Node *left, *eq, *right;
};

// A utility function to create a new ternary search tree node
struct Node* newNode(char data)
{
    struct Node* temp = (struct Node*) malloc(sizeof( struct Node ));
    temp->data = data;
    temp->isEndOfString = 0;
    temp->left = temp->eq = temp->right = NULL;
    return temp;
}

// Function to insert a new word in a Ternary Search Tree
void insert(struct Node** root, char *word)
{
    // Base Case: Tree is empty
    if (!(*root))
        *root = newNode(*word);

    // If current character of word is smaller than root's character,
    // then insert this word in left subtree of root
    if ((*word) < (*root)->data)
        insert(&(*root)->left), word);

    // If current character of word is greater than root's character,
    // then insert this word in right subtree of root
    else if ((*word) > (*root)->data)
        insert(&(*root)->right), word);

    // If current character of word is same as root's character,
    else
    {
        if (*(word+1))
            insert(&(*root)->eq), word+1);

        // the last character of the word
        else
            (*root)->isEndOfString = 1;
    }
}

// A recursive function to traverse Ternary Search Tree

```

```
void traverseTSTUtil(struct Node* root, char* buffer, int depth)
{
    if (root)
    {
        // First traverse the left subtree
        traverseTSTUtil(root->left, buffer, depth);

        // Store the character of this node
        buffer[depth] = root->data;
        if (root->isEndOfString)
        {
            buffer[depth+1] = '\0';
            printf( "%s\n", buffer);
        }

        // Traverse the subtree using equal pointer (middle subtree)
        traverseTSTUtil(root->eq, buffer, depth + 1);

        // Finally Traverse the right subtree
        traverseTSTUtil(root->right, buffer, depth);
    }
}

// The main function to traverse a Ternary Search Tree.
// It mainly uses traverseTSTUtil()
void traverseTST(struct Node* root)
{
    char buffer[MAX];
    traverseTSTUtil(root, buffer, 0);
}

// Function to search a given word in TST
int searchTST(struct Node *root, char *word)
{
    if (!root)
        return 0;

    if (*word < (root)->data)
        return searchTST(root->left, word);

    else if (*word > (root)->data)
        return searchTST(root->right, word);

    else
    {
        if (*(word+1) == '\0')
            return root->isEndOfString;
```

```
        return searchTST(root->eq, word+1);
    }
}

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

    insert(&root, "cat");
    insert(&root, "cats");
    insert(&root, "up");
    insert(&root, "bug");

    printf("Following is traversal of ternary search tree\n");
    traverseTST(root);

    printf("\nFollowing are search results for cats, bu and cat respectively\n");
    searchTST(root, "cats")? printf("Found\n"): printf("Not Found\n");
    searchTST(root, "bu")? printf("Found\n"): printf("Not Found\n");
    searchTST(root, "cat")? printf("Found\n"): printf("Not Found\n");

    return 0;
}
```

Output:

```
Following is traversal of ternary search tree
bug
cat
cats
up

Following are search results for cats, bu and cat respectively
Found
Not Found
Found
```

Time Complexity: The time complexity of the ternary search tree operations is similar to that of binary search tree. i.e. the insertion, deletion and search operations take time proportional to the height of the ternary search tree. The space is proportional to the length of the string to be stored.

Reference:

http://en.wikipedia.org/wiki/Ternary_search_tree

This article is compiled by **Aashish Barnwal** and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

<https://www.geeksforgeeks.org/ternary-search-tree/>

Chapter 196

Ternary Search Tree (Deletion)

Ternary Search Tree (Deletion) - GeeksforGeeks

In the [SET 1](#) post on TST we have described how to insert and search a node in TST. In this article we will discuss algorithm on how to delete a node from TST.

During delete operation we delete the key in bottom up manner using recursion. The following are possible cases when deleting a key from trie.

1. Key may not be there in TST.

Solution : Delete operation should not modify TST.

2. Key present as unique key (no part of key contains another key (prefix), nor the key itself is prefix of another key in TST).

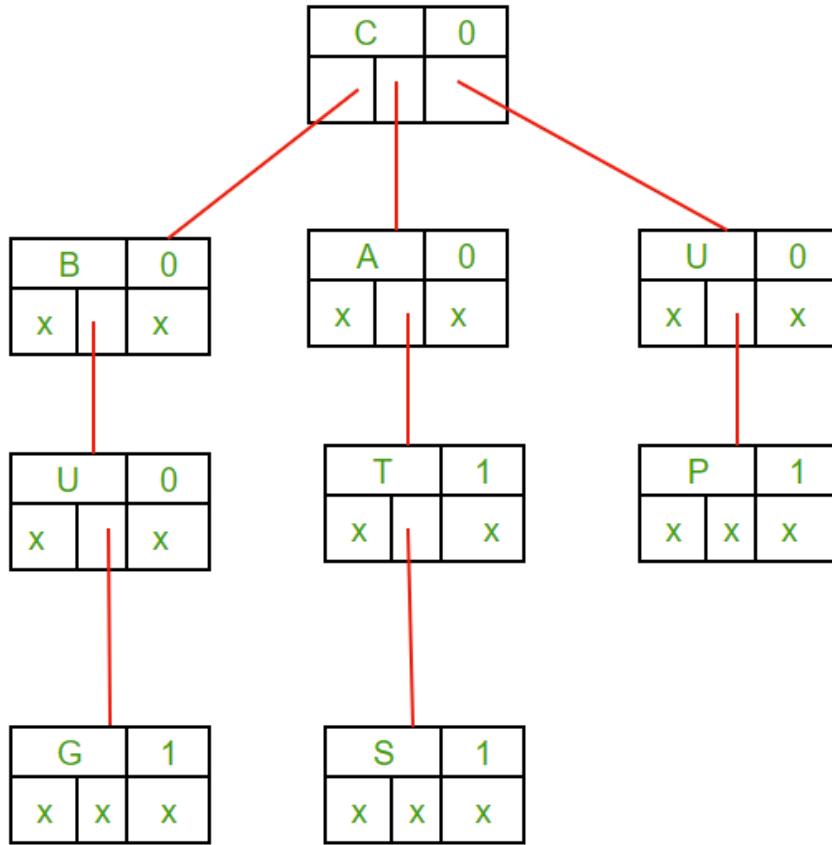
Solution : Delete all the nodes.

3. Key is prefix key of another long key in TST.

Solution : Unmark the leaf node.

4. Key present in TST, having atleast one other key as prefix key.

Solution : Delete nodes from end of key until first leaf node of longest prefix key.



Explanation for delete_node function

1. Let suppose we want to delete string “BIG”, since it is not present in TST so after matching with first character ‘B’, delete_node function will return zero. Hence nothing is deleted.
2. Now we want to delete string “BUG”, it is Uniquely present in TST i.e neither it has part which is the prefix of other string nor it is prefix to any other string, so it will be deleted completely.
3. Now we want to delete string “CAT”, since it is prefix of string “CATS”, we cannot delete anything from the string “CAT” and we can only unmark the leaf node which will ensure that “CAT” is no longer a member string of TST.
4. Now we want to delete string “CATS”, since it has a prefix string “CAT” which also is a member string of TST so we can only delete last character of string “CATS” which will ensure that string “CAT” still remains the part of TST.

C

```
// C program to demonstrate deletion in
// Ternary Search Tree (TST). For insert
// and other functions, refer
// https://www.geeksforgeeks.org/ternary-search-tree/
#include<stdio.h>
#include<stdlib.h>

// structure of a node in TST
struct Node
{
    char key;
    int isleaf;
    struct Node *left;
    struct Node *eq;
    struct Node *right;
};

// function to create a Node in TST
struct Node *createNode(char key)
{
    struct Node *temp =
        (struct Node*)malloc(sizeof(struct Node));
    temp->key = key;
    temp->isleaf = 0;
    temp->left = NULL;
    temp->eq = NULL;
    temp->right = NULL;
    return temp;
};

// function to insert a Node in TST
void insert_node(struct Node **root ,char *s)
{
    if (!(*root))
        (*root) = createNode(*s);

    if ((*s)<(*root)->key)
        insert_node( &(*root)->left ,s);

    else if ((*s)>(*root)->key)
        insert_node( &(*root)->right ,s);

    else if ((*s) == (*root)->key)
    {
        if (*(s+1) == '\0')
        {
            (*root)->isleaf = 1;
            return;
        }
    }
}
```

```
        }
        insert_node( &(*root)->eq ,s+1);
    }
}

// function to display the TST
void display(struct Node *root, char str[], int level)
{
    if (!root)
        return;

    display(root->left ,str ,level);
    str[level] = root->key;

    if (root->isleaf == 1)
    {
        str[level+1] = '\0';
        printf("%s\n",str);
    }

    display(root->eq ,str ,level+1);
    display(root->right ,str ,level);
}

// to check if current Node is leaf node or not
int isLeaf(struct Node *root)
{
    return root->isleaf == 1;
}

// to check if current node has any child node or not
int isFreeNode(struct Node *root)
{
    if (root->left ||root->eq ||root->right)
        return 0;
    return 1;
}

// function to delete a string in TST
int delete_node(struct Node *root, char str[],
                int level ,int n)
{
    if (root == NULL)
        return 0;

    // CASE 4 Key present in TST, having
    // atleast one other key as prefix key.
```

```

if (str[level+1] == '\0')
{
    // Unmark leaf node if present
    if (isLeaf(root))
    {
        root->isleaf=0;
        return isFreeNode(root);
    }

    // else string is not present in TST and
    // return 0
    else
        return 0;
}
else
{
    // CASE 3 Key is prefix key of another
    // long key in TST.
    if (str[level] < root->key)
        delete_node(root->left ,str ,level ,n);
    else if (str[level] > root->key)
        delete_node(root->right ,str ,level ,n);

    // CASE 1 Key may not be there in TST.
    else if (str[level] == root->key)
    {
        // CASE 2 Key present as unique key
        if( delete_node(root->eq ,str ,level+1 ,n) )
        {
            // delete the last node, neither it
            // has any child
            // nor it is part of any other string
            free(root->eq);
            return !isLeaf(root) && isFreeNode(root);
        }
    }
}

return 0;
}

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

    insert_node(&temp , "CAT");
    insert_node(&temp , "BUGS");
}

```

```
insert_node(&temp , "CATS");
insert_node(&temp , "UP");

int level = 0;
char str[20];
int p = 0;

printf( "1.Content of the TST before "
        "deletion:\n" );
display(temp ,str ,level);

level = 0;
delete_node(temp , "CAT" ,level ,5);

level = 0;
printf("\n2.Content of the TST after "
        "deletion:\n");
display(temp, str, level);
return 0;
}
```

C++

```
// C++ program to demonstrate deletion in
// Ternary Search Tree (TST)
// For insert and other functions, refer
// https://www.geeksforgeeks.org/ternary-search-tree

#include<iostream>
using namespace std;

// structure of a node in TST
struct Node
{
    char key;
    int isleaf;
    struct Node *left;
    struct Node *eq;
    struct Node *right;
};

// function to create a node in TST
struct Node *createNode(char key)
{
    struct Node *temp = new Node;
    temp->key = key;
    temp->isleaf = 0;
    temp->left = NULL;
```

```
temp->eq = NULL;
temp->right = NULL;
return temp;
};

// function to insert a Node in TST
void insert_node(struct Node **root, char *s)
{
    if (!(*root))
    {
        (*root) = createNode(*s);
    }

    if ((*s)<(*root)->key)
        insert_node( &(*root)->left, s);

    else if ((*s)>(*root)->key)
        insert_node( &(*root)->right, s);

    else if ((*s) == (*root)->key)
    {
        if (*(s+1) == '\0')
        {
            (*root)->isleaf = 1;
            return;
        }
        insert_node( &(*root)->eq, s+1);
    }
}

// function to display the TST
void display(struct Node *root, char str[], int level)
{
    if (!root)
        return;

    display(root->left, str, level);
    str[level] = root->key;

    if (root->isleaf == 1)
    {
        str[level+1] = '\0';
        cout<< str << endl;
    }

    display(root->eq, str, level+1);
    display(root->right, str, level);
}
```

```
//to check if current node is leaf node or not
int isLeaf(struct Node *root)
{
    return root->isleaf == 1;
}

// to check if current node has any child
// node or not
int isFreeNode(struct Node *root)
{
    if (root->left ||root->eq ||root->right)
        return 0;
    return 1;
}

// function to delete a string in TST
int delete_node(struct Node *root, char str[],
                int level, int n)
{
    if (root == NULL)
        return 0;

    // CASE 4 Key present in TST, having atleast
    // one other key as prefix key.
    if (str[level+1] == '\0')
    {
        // Unmark leaf node if present
        if (isLeaf(root))
        {
            root->isleaf = 0;
            return isFreeNode(root);
        }
    }

    // else string is not present in TST and
    // return 0
    else
        return 0;
}

// CASE 3 Key is prefix key of another long
// key in TST.
if (str[level] < root->key)
    return delete_node(root->left, str, level, n);
if (str[level] > root->key)
    return delete_node(root->right, str, level, n);
```

```
// CASE 1 Key may not be there in TST.  
if (str[level] == root->key)  
{  
    // CASE 2 Key present as unique key  
    if (delete_node(root->eq, str, level+1, n))  
    {  
        // delete the last node, neither it has  
        // any child nor it is part of any other  
        // string  
        delete(root->eq);  
        return !isLeaf(root) && isFreeNode(root);  
    }  
}  
  
return 0;  
}  
  
// Driver function  
int main()  
{  
    struct Node *temp = NULL;  
  
    insert_node(&temp, "CAT");  
    insert_node(&temp, "BUGS");  
    insert_node(&temp, "CATS");  
    insert_node(&temp, "UP");  
  
    int level = 0;  
    char str[20];  
    int p = 0;  
  
    cout << "1.Content of the TST before deletion:\n";  
    display(temp, str, level);  
  
    level = 0;  
    delete_node(temp,"CAT", level, 5);  
  
    level = 0;  
    cout << "\n2.Content of the TST after deletion:\n";  
    display(temp, str, level);  
    return 0;  
}
```

Output:

```
1.Content of the TST before deletion:  
BUGS
```

CAT
CATS
UP

2.Content of the TST after deletion:

BUGS
CATS
UP

Source

<https://www.geeksforgeeks.org/ternary-search-tree-deletion/>

Chapter 197

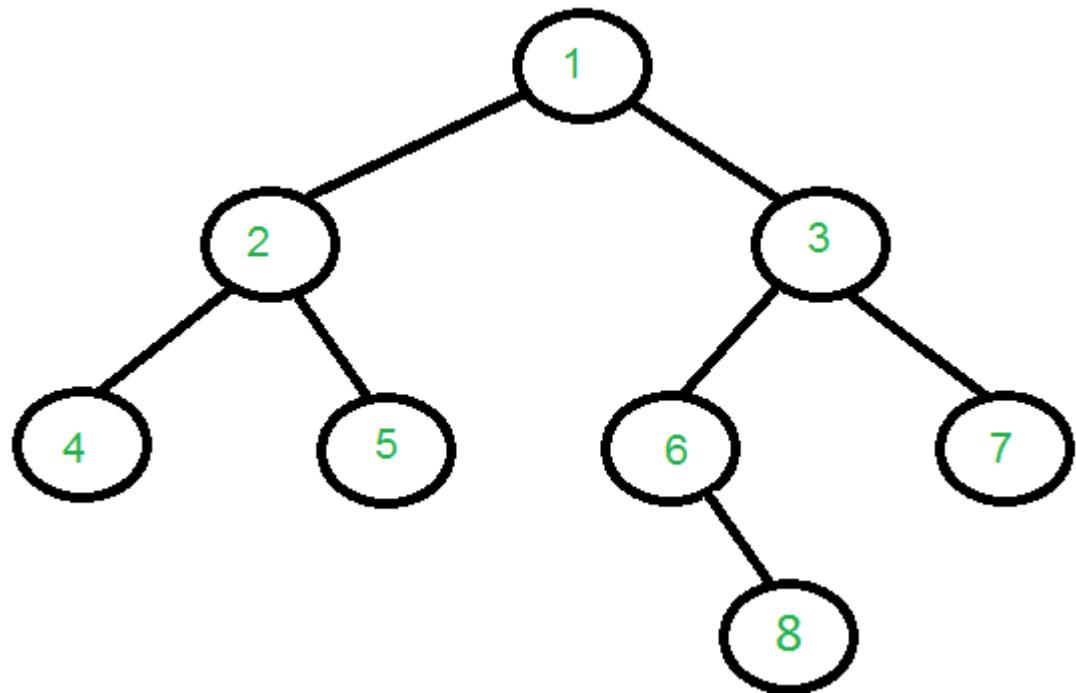
Total nodes traversed in Euler Tour Tree

Total nodes traversed in Euler Tour Tree - GeeksforGeeks

[*Euler tour of tree*](#) has been already discussed which flattens the hierarchical structure of tree into array which contains exactly $2*N-1$ values. In this post, the task is to prove that the degree of Euler Tour Tree is 2 times the number of nodes minus one. Here degree means the total number of nodes get traversed in Euler Tour.

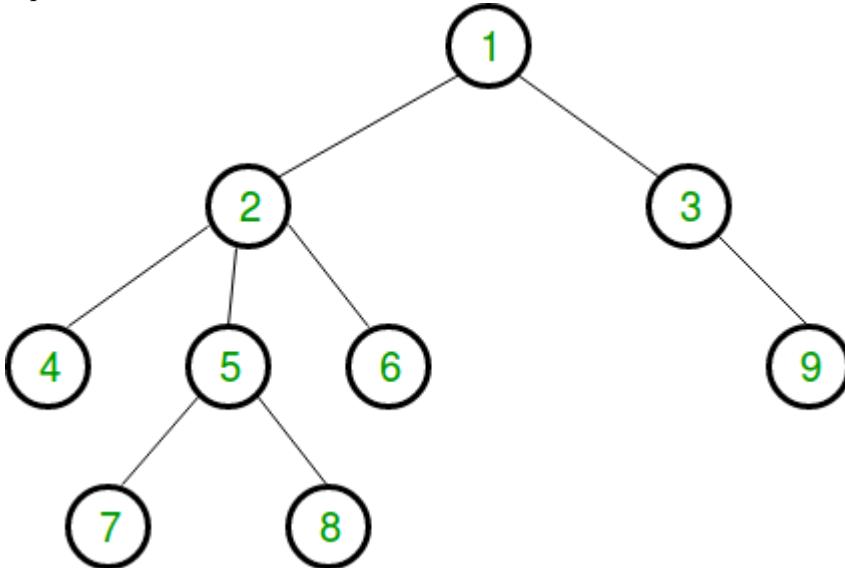
Examples:

Input:



Output: 15

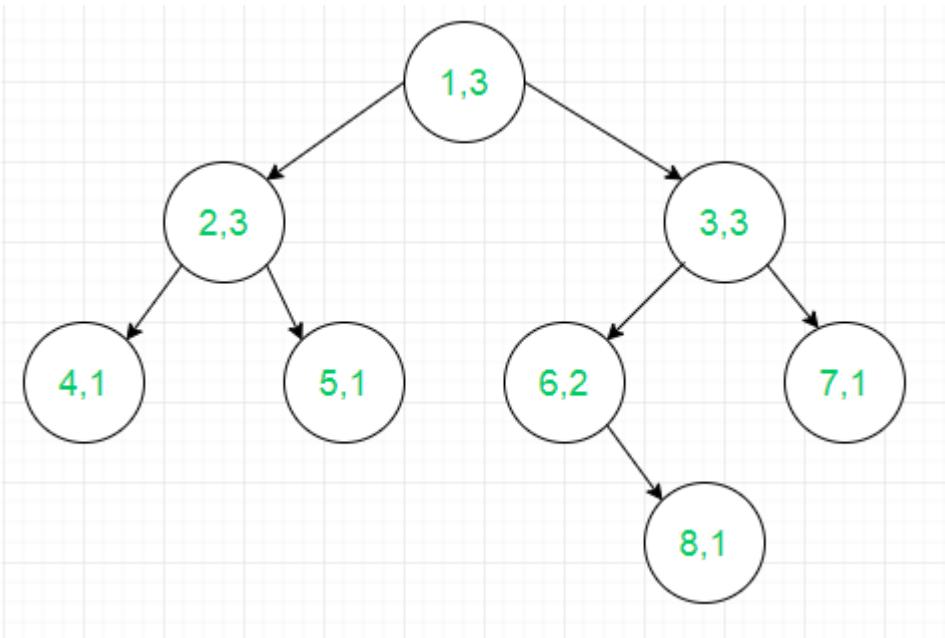
Input:



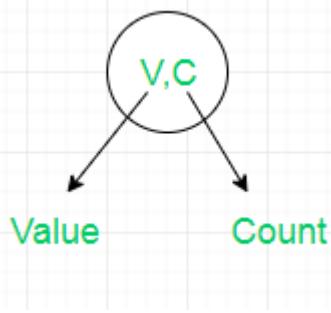
Output: 17

Explanation:

Using Example 1:



where



It can be seen that each node's count in Euler Tour is exactly equal to the out-degree of node plus 1.

Mathematically, it can be represented as:

$$Count = \sum_{node_i=1}^{n-1} OutDeg[node_i] + 1$$

$$Count = n + \sum_{node_i=1}^{n-1} OutDeg[node_i]$$

Where

Total represents total number of nodes in Euler Tour Tree

node_i represents ith node in given Tree

N represents the total number of node in given Tree

adj[i].size() represents number of child of **node_i**

```
// C++ program to check the number of nodes
// in Euler Tour tree.
#include <bits/stdc++.h>
using namespace std;

#define MAX 1001

// Adjacency list representation of tree
vector<int> adj[MAX];

// Function to add edges to tree
void add_edge(int u, int v)
{
    adj[u].push_back(v);
}

// Program to check if calculated Value is
// equal to 2*size-1
void checkTotalNumberofNodes(int actualAnswer,
                             int size)
{
    int calculatedAnswer = size;

    // Add out-degree of each node
    for (int i = 1; i <= size; i++)
        calculatedAnswer += adj[i].size();

    if (actualAnswer == calculatedAnswer)
        cout << "Calculated Answer is " << calculatedAnswer
            << " and is Equal to Actual Answer\n";
    else
        cout << "Calculated Answer is Incorrect\n";
}

int main()
{ // Constructing 1st tree from example
    int N = 8;
    add_edge(1, 2);
    add_edge(1, 3);
    add_edge(2, 4);
```

```
    add_edge(2, 5);
    add_edge(3, 6);
    add_edge(3, 7);
    add_edge(6, 8);

    // Out_deg[node[i]] is equal to adj[i].size()
    checkTotalNumberofNodes(2 * N - 1, N);

    // clear previous stored tree
    for (int i = 1; i <= N; i++)
        adj[i].clear();

    // Constructing 2nd tree from example
    N = 9;
    add_edge(1, 2);
    add_edge(1, 3);
    add_edge(2, 4);
    add_edge(2, 5);
    add_edge(2, 6);
    add_edge(3, 9);
    add_edge(5, 7);
    add_edge(5, 8);

    // Out_deg[node[i]] is equal to adj[i].size()
    checkTotalNumberofNodes(2 * N - 1, N);

    return 0;
}
```

Output:

```
Calculated Answer is 15 and is Equal to Actual Answer
Calculated Answer is 17 and is Equal to Actual Answer
```

Source

<https://www.geeksforgeeks.org/total-nodes-traversed-in-euler-tour-tree/>

Chapter 198

Tournament Tree (Winner Tree) and Binary Heap

Tournament Tree (Winner Tree) and Binary Heap - GeeksforGeeks

Given a team of N players. How many minimum games are required to find second best player?

We can use adversary arguments based on tournament tree (Binary Heap).

[Tournament tree](#) is a form of min (max) heap which is a complete binary tree. Every external node represents a player and internal node represents winner. In a tournament tree every internal node contains winner and every leaf node contains one player.

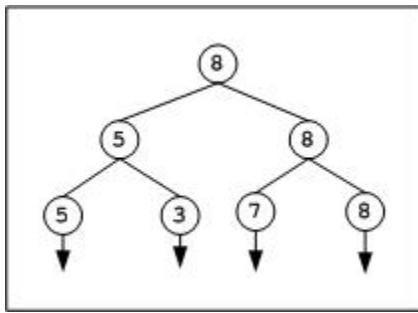
There will be $N - 1$ internal nodes in a binary tree with N leaf (external) nodes. For details see [this post](#) (put $n = 2$ in equation given in the post).

It is obvious that to select the best player among N players, $(N - 1)$ players to be eliminated, i.e. we need minimum of $(N - 1)$ games (comparisons). Mathematically we can prove it. In a binary tree $I = E - 1$, where I is number of internal nodes and E is number of external nodes. It means to find maximum or minimum element of an array, we need $N - 1$ (internal nodes) comparisons.

Second Best Player

The information explored during best player selection can be used to minimize the number of comparisons in tracing the next best players. For example, we can pick second best player in $(N + \log_2 N - 2)$ comparisons.

The following diagram displays a tournament tree (*winner tree*) as a max heap. Note that the concept of *loser tree* is different.



The above tree contains 4 leaf nodes that represent players and have 3 levels 0, 1 and 2. Initially 2 games are conducted at level 2, one between 5 and 3 and another one between 7 and 8. In the next move, one more game is conducted between 5 and 8 to conclude the final winner. Overall we need 3 comparisons. For second best player we need to trace the candidates participated with final winner, that leads to 7 as second best.

Median of Sorted Arrays

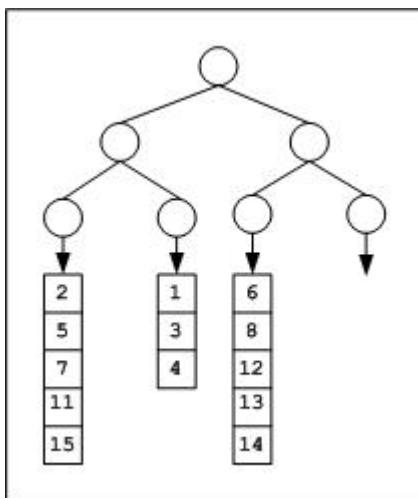
Tournament tree can effectively be used to find median of sorted arrays. Assume, given M sorted arrays of equal size L (for simplicity). We can attach all these sorted arrays to the tournament tree, one array per leaf. We need a tree of height **CEIL ($\log_2 M$)** to have atleast M external nodes.

Consider an example. Given 3 ($M = 3$) sorted integer arrays of maximum size 5 elements.

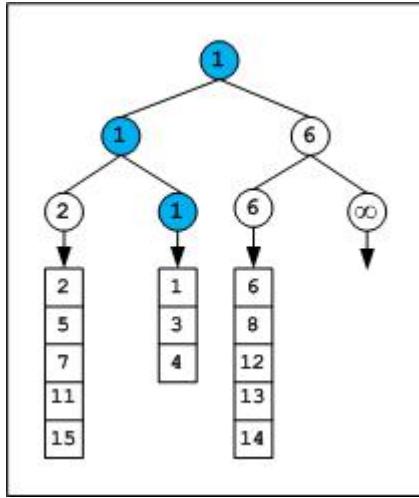
```

{ 2, 5, 7, 11, 15 } ---- Array1
{1, 3, 4} ----- Array2
{6, 8, 12, 13, 14} ----- Array3
  
```

What should be the height of tournament tree? We need to construct a tournament tree of height $\log_2 3 = 1.585 = 2$ rounded to next integer. A binary tree of height 2 will have 4 leaves to which we can attach the arrays as shown in the below figure.



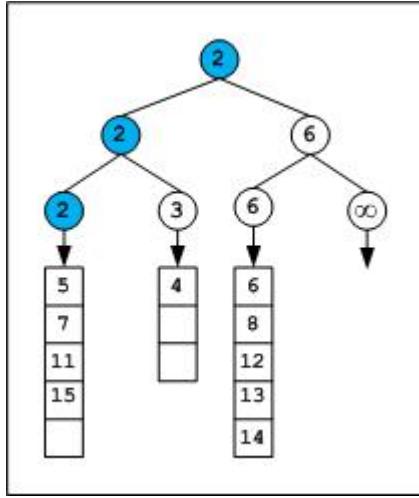
After the first tournament, the tree appears as below,



We can observe that the winner is from Array2. Hence the next element from Array2 will dive-in and games will be played along the winner path of previous tournament.

Note that infinity is used as sentinel element. Based on data being hold in nodes we can select the sentinel character. For example we usually store the pointers in nodes rather than keys, so NULL can serve as sentinel. If any of the array exhausts we will fill the corresponding leaf and upcoming internal nodes with sentinel.

After the second tournament, the tree appears as below,



The next winner is from Array1, so next element of Array1 array which is 5 will dive-in to the next round, and next tournament played along the path of 2.

The tournaments can be continued till we get median element which is $(5+3+5)/2 = 7$ th element. Note that there are even better algorithms for finding median of union of sorted arrays, for details see the related links given below.

In general with M sorted lists of size $L_1, L_2 \dots L_m$ requires time complexity of $O((L_1 + L_2 + \dots + L_m) * \log M)$ to merge all the arrays, and $O(m * \log M)$ time to find median, where m is median position.

Select smallest one million elements from one billion unsorted elements:

As a simple solution, we can sort the billion numbers and select first one million.

On a limited memory system sorting billion elements and picking the first one million seems to be impractical. We can use tournament tree approach. At any time only elements of tree to be in memory.

Split the large array (perhaps stored on disk) into smaller size arrays of size one million each (or even smaller that can be sorted by the machine). Sort these 1000 small size arrays and store them on disk as individual files. Construct a tournament tree which can have atleast 1000 leaf nodes (tree to be of height 10 since $2^9 < 1000 < 2^{10}$, if the individual file size is even smaller we will need more leaf nodes). Every leaf node will have an engine that picks next element from the sorted file stored on disk. We can play the tournament tree game to extract first one million elements.

Total cost = sorting 1000 lists of one million each + tree construction + tournaments

Implementation

We need to build the tree in bottom-up manner. All the leaf nodes filled first. Start at the left extreme of tree and fill along the breadth (i.e. from 2^{k-1} to $2^k - 1$ where k is depth of tree) and play the game. After practicing with few examples it will be easy to write code. Implementation is discussed in below code

[Second minimum element using minimum comparisons](#)

Related Posts

[Find the smallest and second smallest element in an array.](#)

[Second minimum element using minimum comparisons](#)

— by [Venki](#). Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

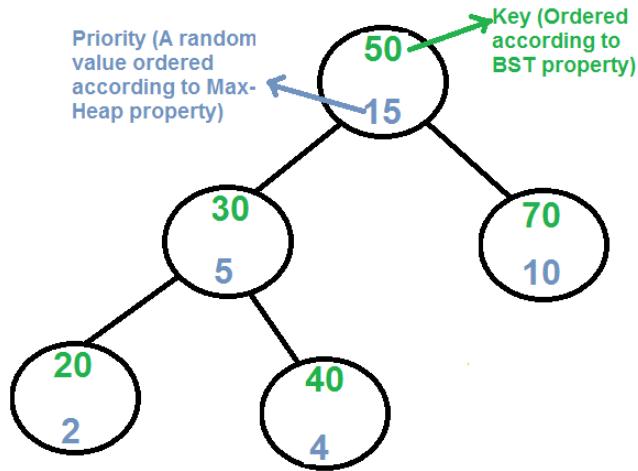
<https://www.geeksforgeeks.org/tournament-tree-and-binary-heap/>

Chapter 199

Treap (A Randomized Binary Search Tree)

Treap (A Randomized Binary Search Tree) - GeeksforGeeks

Like Red-Black and AVL Trees, Treap is a Balanced Binary Search Tree, but not guaranteed to have height as $O(\log n)$. The idea is to use Randomization and Binary Heap property to maintain balance with high probability. The expected time complexity of search, insert and delete is $O(\log n)$.



Every node of Treap maintains two values.

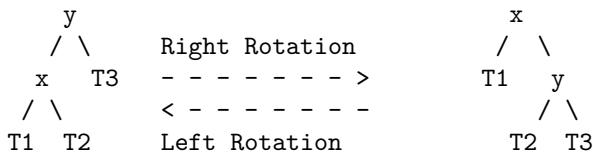
- 1) **Key** Follows standard BST ordering (left is smaller and right is greater)
- 2) **Priority** Randomly assigned value that follows Max-Heap property.

Basic Operation on Treap:

Like other self-balancing Binary Search Trees, Treap uses rotations to maintain Max-Heap

property during insertion and deletion.

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



Keys in both of the above trees follow the following order

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

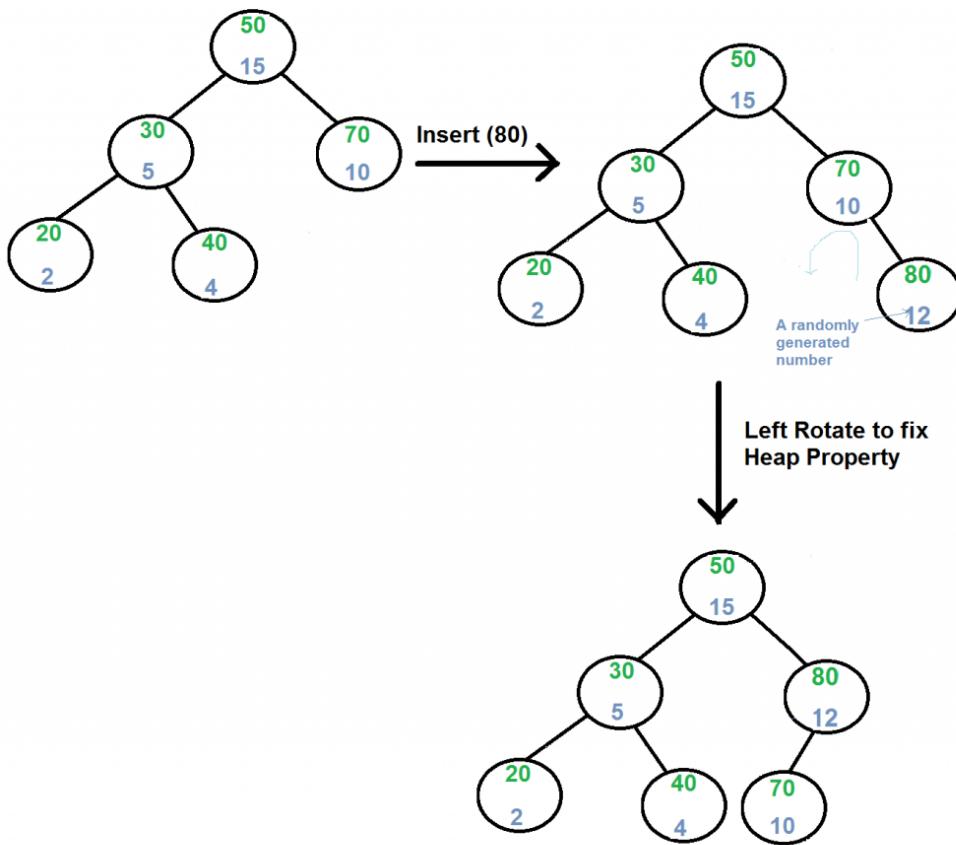
So BST property is not violated anywhere.

search(x)

Perform standard [BST Search](#) to find x.

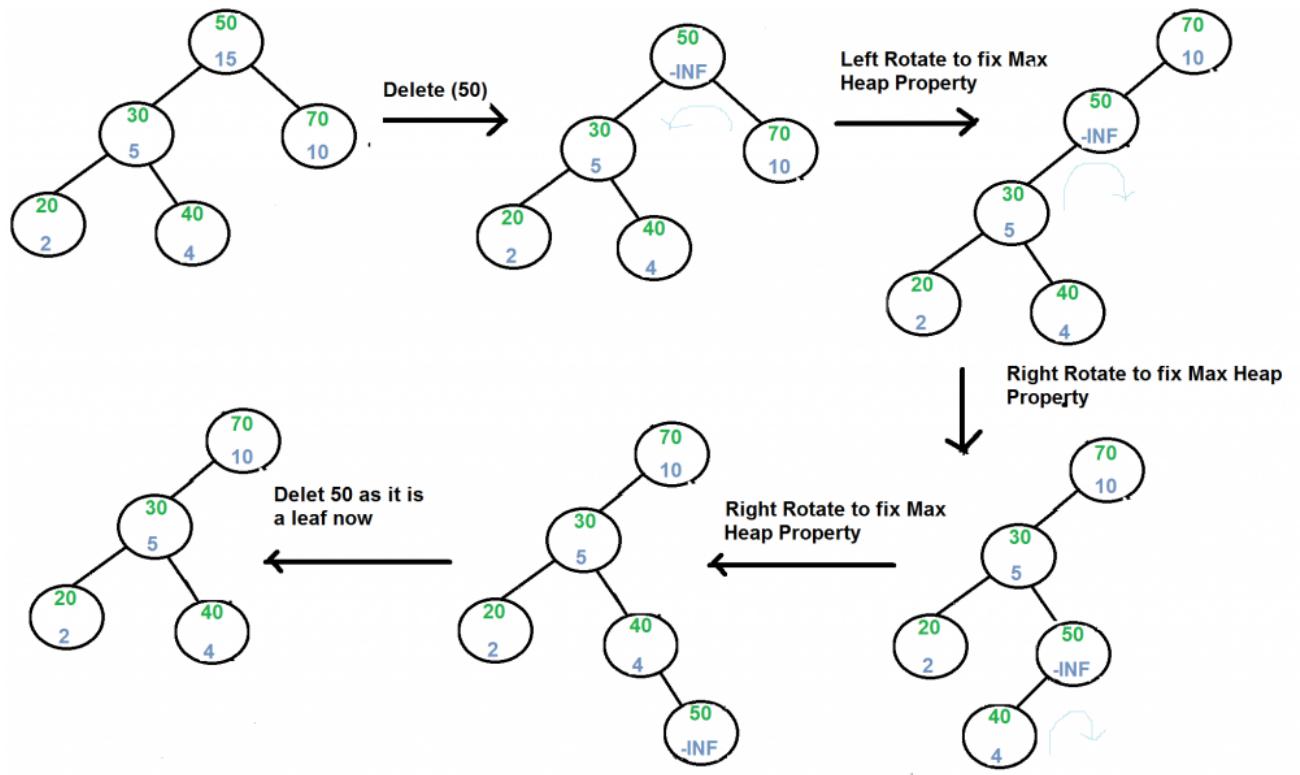
Insert(x):

- 1) Create new node with key equals to x and value equals to a random value.
- 2) Perform standard [BST insert](#).
- 3) Use rotations to make sure that inserted node's priority follows max heap property.



Delete(x):

- 1) If node to be deleted is a leaf, delete it.
- 2) Else replace node's priority with minus infinite (-INF), and do appropriate rotations to bring the node down to a leaf.



Refer [Implementation of Treap Search, Insert and Delete](#) for more details.

References:

- <https://en.wikipedia.org/wiki/Treap>
- <https://courses.cs.washington.edu/courses/cse326/00wi/handouts/lecture19/sld017.htm>

Source

<https://www.geeksforgeeks.org/treap-a-randomized-binary-search-tree/>

Chapter 200

Treap Set 2 (Implementation of Search, Insert and Delete)

Treap Set 2 (Implementation of Search, Insert and Delete) - GeeksforGeeks

We strongly recommend to refer set 1 as a prerequisite of this post.

[Treap \(A Randomized Binary Search Tree\)](#)

In this post, implementations of search, insert and delete are discussed.

Search:

Same as [standard BST search](#). Priority is not considered for search.

```
// C function to search a given key in a given BST
TreapNode* search(TreapNode* root, int key)
{
    // Base Cases: root is null or key is present at root
    if (root == NULL || root->key == key)
        return root;

    // Key is greater than root's key
    if (root->key < key)
        return search(root->right, key);

    // Key is smaller than root's key
    return search(root->left, key);
}
```

Insert

- 1) Create new node with key equals to x and value equals to a random value.
- 2) Perform standard [BST insert](#).

- 3) A newly inserted node gets a random priority, so Max-Heap property may be violated.. Use rotations to make sure that inserted node's priority follows max heap property.

During insertion, we recursively traverse all ancestors of the inserted node.

a) If new node is inserted in left subtree and root of left subtree has higher priority, perform right rotation.

b) If new node is inserted in right subtree and root of right subtree has higher priority, perform left rotation.

```
/* Recursive implementation of insertion in Treap */
TreapNode* insert(TreapNode* root, int key)
{
    // If root is NULL, create a new node and return it
    if (!root)
        return newNode(key);

    // If key is smaller than root
    if (key <= root->key)
    {
        // Insert in left subtree
        root->left = insert(root->left, key);

        // Fix Heap property if it is violated
        if (root->left->priority > root->priority)
            root = rightRotate(root);
    }
    else // If key is greater
    {
        // Insert in right subtree
        root->right = insert(root->right, key);

        // Fix Heap property if it is violated
        if (root->right->priority > root->priority)
            root = leftRotate(root);
    }
    return root;
}
```

Delete:

The delete implementation here is slightly different from the steps discussed in [previous post](#).

- 1) If node is a leaf, delete it.
- 2) If node has one child NULL and other as non-NULL, replace node with the non-empty child.
- 3) If node has both children as non-NULL, find max of left and right children.
 -a) If priority of right child is greater, perform left rotation at node
 -b) If priority of left child is greater, perform right rotation at node.

The idea of step 3 is to move the node to down so that we end up with either case 1 or case 2.

```
/* Recursive implementation of Delete() */
TreapNode* deleteNode(TreapNode* root, int key)
{
    // Base case
    if (root == NULL) return root;

    // IF KEYS IS NOT AT ROOT
    if (key < root->key)
        root->left = deleteNode(root->left, key);
    else if (key > root->key)
        root->right = deleteNode(root->right, key);

    // IF KEY IS AT ROOT

    // If left is NULL
    else if (root->left == NULL)
    {
        TreapNode *temp = root->right;
        delete(root);
        root = temp; // Make right child as root
    }

    // If Right is NULL
    else if (root->right == NULL)
    {
        TreapNode *temp = root->left;
        delete(root);
        root = temp; // Make left child as root
    }

    // If ksy is at root and both left and right are not NULL
    else if (root->left->priority < root->right->priority)
    {
        root = leftRotate(root);
        root->left = deleteNode(root->left, key);
    }
    else
    {
        root = rightRotate(root);
        root->right = deleteNode(root->right, key);
    }

    return root;
}
```

A Complete Program to Demonstrate All Operations

```

// C++ program to demonstrate search, insert and delete in Treap
#include <bits/stdc++.h>
using namespace std;

// A Treap Node
struct TreapNode
{
    int key, priority;
    TreapNode *left, *right;
};

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

      y                               x
      / \     Right Rotation         / \
      x   T3   - - - - - - - >   T1   y
      / \     < - - - - - - - -   / \
      T1   T2   Left Rotation     T2   T3 */

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

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

    // Return new root
    return x;
}

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

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

    // Return new root
    return y;
}

```

```
/* Utility function to add a new key */
TreapNode* newNode(int key)
{
    TreapNode* temp = new TreapNode;
    temp->key = key;
    temp->priority = rand()%100;
    temp->left = temp->right = NULL;
    return temp;
}

// C function to search a given key in a given BST
TreapNode* search(TreapNode* root, int key)
{
    // Base Cases: root is null or key is present at root
    if (root == NULL || root->key == key)
        return root;

    // Key is greater than root's key
    if (root->key < key)
        return search(root->right, key);

    // Key is smaller than root's key
    return search(root->left, key);
}

/* Recursive implementation of insertion in Treap */
TreapNode* insert(TreapNode* root, int key)
{
    // If root is NULL, create a new node and return it
    if (!root)
        return newNode(key);

    // If key is smaller than root
    if (key <= root->key)
    {
        // Insert in left subtree
        root->left = insert(root->left, key);

        // Fix Heap property if it is violated
        if (root->left->priority > root->priority)
            root = rightRotate(root);
    }
    else // If key is greater
    {
        // Insert in right subtree
        root->right = insert(root->right, key);
    }
}
```

```
// Fix Heap property if it is violated
if (root->right->priority > root->priority)
    root = leftRotate(root);
}
return root;
}

/* Recursive implementation of Delete() */
TreapNode* deleteNode(TreapNode* root, int key)
{
    if (root == NULL)
        return root;

    if (key < root->key)
        root->left = deleteNode(root->left, key);
    else if (key > root->key)
        root->right = deleteNode(root->right, key);

    // IF KEY IS AT ROOT

    // If left is NULL
    else if (root->left == NULL)
    {
        TreapNode *temp = root->right;
        delete(root);
        root = temp; // Make right child as root
    }

    // If Right is NULL
    else if (root->right == NULL)
    {
        TreapNode *temp = root->left;
        delete(root);
        root = temp; // Make left child as root
    }

    // If ksy is at root and both left and right are not NULL
    else if (root->left->priority < root->right->priority)
    {
        root = leftRotate(root);
        root->left = deleteNode(root->left, key);
    }
    else
    {
        root = rightRotate(root);
        root->right = deleteNode(root->right, key);
    }
}
```

```
        return root;
    }

// A utility function to print tree
void inorder(TreapNode* root)
{
    if (root)
    {
        inorder(root->left);
        cout << "key: " << root->key << " | priority: %d "
            << root->priority;
        if (root->left)
            cout << " | left child: " << root->left->key;
        if (root->right)
            cout << " | right child: " << root->right->key;
        cout << endl;
        inorder(root->right);
    }
}

// Driver Program to test above functions
int main()
{
    srand(time(NULL));

    struct TreapNode *root = NULL;
    root = insert(root, 50);
    root = insert(root, 30);
    root = insert(root, 20);
    root = insert(root, 40);
    root = insert(root, 70);
    root = insert(root, 60);
    root = insert(root, 80);

    cout << "Inorder traversal of the given tree \n";
    inorder(root);

    cout << "\nDelete 20\n";
    root = deleteNode(root, 20);
    cout << "Inorder traversal of the modified tree \n";
    inorder(root);

    cout << "\nDelete 30\n";
    root = deleteNode(root, 30);
    cout << "Inorder traversal of the modified tree \n";
    inorder(root);
```

```
cout << "\nDelete 50\n";
root = deleteNode(root, 50);
cout << "Inorder traversal of the modified tree \n";
inorder(root);

TreapNode *res = search(root, 50);
(res == NULL)? cout << "\n50 Not Found ":
               cout << "\n50 found";

return 0;
}
```

Output:

```
Inorder traversal of the given tree
key: 20 | priority: %d 92 | right child: 50
key: 30 | priority: %d 48 | right child: 40
key: 40 | priority: %d 21
key: 50 | priority: %d 73 | left child: 30 | right child: 60
key: 60 | priority: %d 55 | right child: 70
key: 70 | priority: %d 50 | right child: 80
key: 80 | priority: %d 44

Delete 20
Inorder traversal of the modified tree
key: 30 | priority: %d 48 | right child: 40
key: 40 | priority: %d 21
key: 50 | priority: %d 73 | left child: 30 | right child: 60
key: 60 | priority: %d 55 | right child: 70
key: 70 | priority: %d 50 | right child: 80
key: 80 | priority: %d 44

Delete 30
Inorder traversal of the modified tree
key: 40 | priority: %d 21
key: 50 | priority: %d 73 | left child: 40 | right child: 60
key: 60 | priority: %d 55 | right child: 70
key: 70 | priority: %d 50 | right child: 80
key: 80 | priority: %d 44

Delete 50
Inorder traversal of the modified tree
key: 40 | priority: %d 21
key: 60 | priority: %d 55 | left child: 40 | right child: 70
key: 70 | priority: %d 50 | right child: 80
key: 80 | priority: %d 44
```

50 Not Found

Explanation of above output:

Every node is written as key(priority)

The above code constructs below tree

```
20(92)
  \
  50(73)
 /   \
30(48) 60(55)
 \   \
40(21) 70(50)
   \
   80(44)
```

After deleteNode(20)

```
50(73)
 /   \
30(48) 60(55)
 \   \
40(21) 70(50)
   \
   80(44)
```

After deleteNode(30)

```
50(73)
 /   \
40(21) 60(55)
   \
   70(50)
   \
   80(44)
```

After deleteNode(50)

```
60(55)
 /   \
40(21) 70(50)
   \
   80(44)
```

Thanks to **Jai Goyal** for providing initial implementation. Please write comments if you

find anything incorrect, or you want to share more information about the topic discussed above.

Source

<https://www.geeksforgeeks.org/treap-set-2-implementation-of-search-insert-and-delete/>

Chapter 201

Trie memory optimization using hash map

Trie memory optimization using hash map - GeeksforGeeks

We introduced and discussed an implementation in below post.

[Trie \(Insert and Search\) – GeeksforGeeks](#)

The implementation used in above post uses an array of alphabet size with every node. It can be made memory efficient. One way to implementing Trie is linked set of nodes, where each node contains an array of child pointers, one for each symbol in the alphabet. This is not efficient in terms of time as we can't quickly find a particular child.

The efficient way is an implementation where we use hash map to store children of a node. Now we allocate memory only for alphabets in use, and don't waste space storing null pointers.

```
// A memory optimized CPP implementation of trie
// using unordered_map
#include <iostream>
#include <unordered_map>
using namespace std;

struct Trie {

    // isEndOfWord is true if the node
    // represents end of a word
    bool isEndOfWord;

    /* nodes store a map to child node */
    unordered_map<char, Trie*> map;
};

/*function to make a new trie*/
```

```
Trie* getNewTrieNode()
{
    Trie* node = new Trie;
    node->isEndOfWord = false;
    return node;
}

/*function to insert in trie*/
void insert(Trie*& root, const string& str)
{
    if (root == nullptr)
        root = getNewTrieNode();

    Trie* temp = root;
    for (int i = 0; i < str.length(); i++) {
        char x = str[i];

        /* make a new node if there is no path */
        if (temp->map.find(x) == temp->map.end())
            temp->map[x] = getNewTrieNode();

        temp = temp->map[x];
    }

    temp->isEndOfWord = true;
}

/*function to search in trie*/
bool search(Trie* root, const string& str)
{
    /*return false if Trie is empty*/
    if (root == nullptr)
        return false;

    Trie* temp = root;
    for (int i = 0; i < str.length(); i++) {

        /* go to next node*/
        temp = temp->map[str[i]];

        if (temp == nullptr)
            return false;
    }

    return temp->isEndOfWord;
}

/*Driver function*/
```

```
int main()
{
    Trie* root = nullptr;

    insert(root, "geeks");
    cout << search(root, "geeks") << " ";

    insert(root, "for");
    cout << search(root, "for") << " ";

    cout << search(root, "geekk") << " ";

    insert(root, "gee");
    cout << search(root, "gee") << " ";

    insert(root, "science");
    cout << search(root, "science") << endl;

    return 0;
}
```

Output:

1 1 0 1 1

Space used here with every node here is proportional to number of children which is much better than proportional to alphabet size, especially if alphabet is large.

Source

<https://www.geeksforgeeks.org/trie-memory-optimization-using-hash-map/>

Chapter 202

Trie (Delete)

Trie (Delete) - GeeksforGeeks

In the [previous post](#) on [trie](#) we have described how to insert and search a node in trie. Here is an algorithm how to delete a node from trie.

During delete operation we delete the key in bottom up manner using recursion. The following are possible conditions when deleting key from trie,

1. Key may not be there in trie. Delete operation should not modify trie.
2. Key present as unique key (no part of key contains another key (prefix), nor the key itself is prefix of another key in trie). Delete all the nodes.
3. Key is prefix key of another long key in trie. Unmark the leaf node.
4. Key present in trie, having atleast one other key as prefix key. Delete nodes from end of key until first leaf node of longest prefix key.

The highlighted code presents algorithm to implement above conditions. (One may be in dilemma how a pointer passed to delete helper is reflecting changes from deleteHelper to deleteKey. Note that we are holding trie as an ADT in `trie_t` node, which is passed by reference or pointer).

C

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

#define ARRAY_SIZE(a) sizeof(a)/sizeof(a[0])

// Alphabet size (# of symbols)
```

```
#define ALPHABET_SIZE (26)
#define INDEX(c) ((int)c - (int)'a')

#define FREE(p) \
    free(p);      \
    p = NULL;

// forward declaration
typedef struct trie_node trie_node_t;

// trie node
struct trie_node
{
    int value; // non zero if leaf
    trie_node_t *children[ALPHABET_SIZE];
};

// trie ADT
typedef struct trie trie_t;

struct trie
{
    trie_node_t *root;
    int count;
};

trie_node_t *getNode(void)
{
    trie_node_t *pNode = NULL;

    pNode = (trie_node_t *)malloc(sizeof(trie_node_t));

    if( pNode )
    {
        int i;

        pNode->value    = 0;

        for(i = 0; i < ALPHABET_SIZE; i++)
        {
            pNode->children[i] = NULL;
        }
    }

    return pNode;
}

void initialize(trie_t *pTrie)
```

```

{
    pTrie->root = getNode();
    pTrie->count = 0;
}

void insert(trie_t *pTrie, char key[])
{
    int level;
    int length = strlen(key);
    int index;
    trie_node_t *pCrawl;

    pTrie->count++;
    pCrawl = pTrie->root;

    for( level = 0; level < length; level++ )
    {
        index = INDEX(key[level]);

        if( pCrawl->children[index] )
        {
            // Skip current node
            pCrawl = pCrawl->children[index];
        }
        else
        {
            // Add new node
            pCrawl->children[index] = getNode();
            pCrawl = pCrawl->children[index];
        }
    }

    // mark last node as leaf (non zero)
    pCrawl->value = pTrie->count;
}

int search(trie_t *pTrie, char key[])
{
    int level;
    int length = strlen(key);
    int index;
    trie_node_t *pCrawl;

    pCrawl = pTrie->root;

    for( level = 0; level < length; level++ )
    {
        index = INDEX(key[level]);
}

```

```

    if( !pCrawl->children[index] )
    {
        return 0;
    }

    pCrawl = pCrawl->children[index];
}

return (0 != pCrawl && pCrawl->value);
}

int leafNode(trie_node_t *pNode)
{
    return (pNode->value != 0);
}

int isItFreeNode(trie_node_t *pNode)
{
    int i;
    for(i = 0; i < ALPHABET_SIZE; i++)
    {
        if( pNode->children[i] )
            return 0;
    }

    return 1;
}

bool deleteHelper(trie_node_t *pNode, char key[], int level, int len)
{
    if( pNode )
    {
        // Base case
        if( level == len )
        {
            if( pNode->value )
            {
                // Unmark leaf node
                pNode->value = 0;

                // If empty, node to be deleted
                if( isItFreeNode(pNode) )
                {
                    return true;
                }
            }
        }
    }

    return false;
}

```

```

        }
    }
    else // Recursive case
    {
        int index = INDEX(key[level]);
        if( deleteHelper(pNode->children[index], key, level+1, len) )
        {
            // last node marked, delete it
            FREE(pNode->children[index]);
            // recursively climb up, and delete eligible nodes
            return ( !leafNode(pNode) && isItFreeNode(pNode) );
        }
    }
}

return false;
}

void deleteKey(trie_t *pTrie, char key[])
{
    int len = strlen(key);

    if( len > 0 )
    {
        deleteHelper(pTrie->root, key, 0, len);
    }
}

int main()
{
    char keys[] [8] = {"she", "sells", "sea", "shore", "the", "by", "sheer"};
    trie_t trie;

    initialize(&trie);

    for(int i = 0; i < ARRAY_SIZE(keys); i++)
    {
        insert(&trie, keys[i]);
    }

    deleteKey(&trie, keys[0]);

    printf("%s %s\n", "she", search(&trie, "she") ? "Present in trie" : "Not present in trie");

    return 0;
}

```

Python

```
# Python program for delete operation
# in a Trie

class TrieNode(object):
    """
    Trie node class
    """

    def __init__(self):
        self.children = [None]*26

        # non zero if leaf
        self.value = 0

    def leafNode(self):
        """
        Check if node is leaf node or not
        """
        return self.value != 0

    def isItFreeNode(self):
        """
        If node have no children then it is free
        If node have children return False else True
        """
        for c in self.children:
            if c: return False
        return True


class Trie(object):
    """
    Trie data structure class
    """

    def __init__(self):
        self.root = self.getNode()

        # keep count on number of keys
        # inserted in trie
        self.count = 0;

    def _Index(self, ch):
        """
        private helper function
        Converts key current character into index
        use only 'a' through 'z' and lower case
        """

```

```

        return ord(ch)-ord('a')

def getNode(self):
    """
    Returns new trie node (initialized to NULLs)
    """
    return TrieNode()

def insert(self, key):
    """
    If not present, inserts key into trie
    If the key is prefix of trie node, mark
    it as leaf(non zero)
    """
    length = len(key)
    pCrawl = self.root
    self.count += 1

    for level in range(length):
        index = self._Index(key[level])

        if pCrawl.children[index]:
            # skip current node
            pCrawl = pCrawl.children[index]
        else:
            # add new node
            pCrawl.children[index] = self.getNode()
            pCrawl = pCrawl.children[index]

    # mark last node as leaf (non zero)
    pCrawl.value = self.count

def search(self, key):
    """
    Search key in the trie
    Returns true if key presents in trie, else false
    """
    length = len(key)
    pCrawl = self.root
    for level in range(length):
        index = self._Index(key[level])
        if not pCrawl.children[index]:
            return False
        pCrawl = pCrawl.children[index]

    return pCrawl != None and pCrawl.value != 0

```

```

def _deleteHelper(self,pNode,key,length):
    """
    Helper function for deleting key from trie
    """
    if pNode:
        # Base case
        if level == length:
            if pNode.value:
                # unmark leaf node
                pNode.value = 0

            # if empty, node to be deleted
            return pNode.isItFreeNode()

        # recursive case
    else:
        index = self._Index(key[level])
        if self._deleteHelper(pNode.children[index],\
                               key,level+1,length):

            # last node marked, delete it
            del pNode.children[index]

            # recursively climb up and delete
            # eligible nodes
            return (not pNode.leafNode() and \
                    pNode.isItFreeNode())

    return False

def deleteKey(self,key):
    """
    Delete key from trie
    """
    length = len(key)
    if length > 0:
        self._deleteHelper(self.root,key,0,length)

def main():
    keys = ["she","sells","sea","shore","the","by","sheer"]
    trie = Trie()
    for key in keys:
        trie.insert(key)

    trie.deleteKey(keys[0])

    print("{} {}".format(keys[0],\

```

```
"Present in trie" if trie.search(keys[0]) \
else "Not present in trie"))

print("{} {}".format(keys[6],\
"Present in trie" if trie.search(keys[6]) \
else "Not present in trie"))

if __name__ == '__main__':
    main()

# This code is contributed by Atul Kumar
# (www.facebook.com/atul.kumar.007)
```

— **Venki**. 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/trie-delete/>

Chapter 203

Trie (Display Content)

Trie (Display Content) - GeeksforGeeks

Trie is an efficient information retrieval data structure. In our [previous](#) post on trie we have discussed about basics of trie and how to insert and search a key in trie. In this post we will discuss about displaying all of the content of a trie. That is, to display all of the keys present in the Trie.

Examples:

```
Input: If Trie is      root
        /   \   \
        t   a   b
        |   |   |
        h   n   y
        |   |   \
        e   s   y   e
        /   |   |
        i   r   w
        |   |   |
        r   e   e
                    |
                    r
```

Output: Contents of Trie:

```
answer
any
bye
their
there
```

The idea to do this is to start traversing from the root node of trie, whenever we find a NON-NULL child node, we add parent key of child node in the “string str” at the current

index(level) and then recursively call the same process for the child node and same goes on till we find the node which is a leafnode, which actually marks the end of the string.
Below is the C++ implementation of above idea:

```
// CPP program to display content of Trie
#include <iostream>
#include <string.h>
#define alpha_size 26
#define ARRAY_SIZE(a) sizeof(a) / sizeof(a[0])

using namespace std;

// Trie node
struct TrieNode
{
    struct TrieNode* children[alpha_size];

    bool isLeaf;
};

// Returns new trie node (initialized to NULLs)
struct TrieNode* createNode()
{
    struct TrieNode* pNode = new TrieNode;

    for (int i = 0; i < alpha_size; i++)
        pNode->children[i] = NULL;

    pNode->isLeaf = false;

    return pNode;
}

// function to insert a node in Trie
void insert_node(struct TrieNode* root, char* key)
{
    int level;
    int length = strlen(key);
    struct TrieNode* pCrawl = root;

    for (level = 0; level < length; level++)
    {
        int index = key[level] - 'a';

        if (pCrawl->children[index] == NULL)
            pCrawl->children[index] = createNode();

        pCrawl = pCrawl->children[index];
    }
}
```

```
}

    pCrawl->isLeaf = true;
}

// function to check if current node is leaf node or not
bool isLeafNode(struct TrieNode* root)
{
    return root->isLeaf != false;
}

// function to display the content of Trie
void display(struct TrieNode* root, char str[], int level)
{
    // If node is leaf node, it indicates end
    // of string, so a null character is added
    // and string is displayed
    if (isLeafNode(root))
    {
        str[level] = '\0';
        cout << str << endl;
    }

    int i;
    for (i = 0; i < alpha_size; i++)
    {
        // if NON NULL child is found
        // add parent key to str and
        // call the display function recursively
        // for child node
        if (root->children[i])
        {
            str[level] = i + 'a';
            display(root->children[i], str, level + 1);
        }
    }
}

// Driver program to test above functions
int main()
{
    // Keys to be inserted in Trie
    char keys[] [8] = { "the", "a", "there", "answer",
                       "any", "by", "bye", "their" };

    struct TrieNode* root = createNode();

    // Inserting keys in Trie
```

```
for (int j = 0; j < ARRAY_SIZE(keys); j++)
    insert_node(root, keys[j]);

int level = 0;
char str[20];

// Displaying content of Trie
cout << "Content of Trie: " << endl;
display(root, str, level);
}
```

Output:

Content of Trie:

```
a
answer
any
by
bye
the
their
there
```

NOTE: The above algorithm displays the content of Trie in **Lexographically** Sorted order.

Some useful applications of Trie are:

- [Implementing Auto-Correct and Auto complete Features](#)
- [Implementing a Dictionary](#)

Source

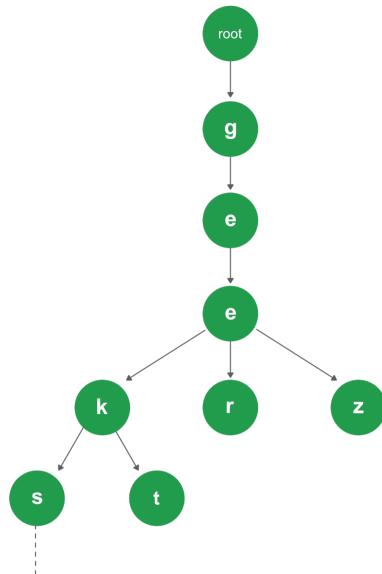
<https://www.geeksforgeeks.org/trie-display-content/>

Chapter 204

Trie (Insert and Search)

Trie (Insert and Search) - GeeksforGeeks

Trie is an efficient information retrieval data structure. Using Trie, search complexities can be brought to optimal limit (key length). If we store keys in binary search tree, a well balanced BST will need time proportional to $M * \log N$, where M is maximum string length and N is number of keys in tree. Using Trie, we can search the key in $O(M)$ time. However the penalty is on Trie storage requirements (Please refer [Applications of Trie](#) for more details)



Every node of Trie consists of multiple branches. Each branch represents a possible character of keys. We need to mark the last node of every key as end of word node. A Trie node field *isEndOfWord* is used to distinguish the node as end of word node. A simple

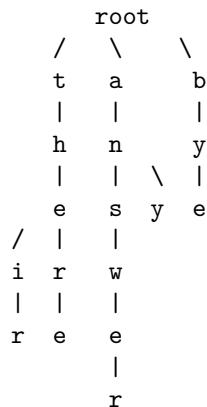
structure to represent nodes of English alphabet can be as following,

```
// Trie node struct TrieNode {
    struct TrieNode children[ALPHABET_SIZE];
    // isEndOfWord is true if the node
    // represents end of a word
    bool isEndOfWord;
};
```

Inserting a key into Trie is simple approach. Every character of input key is inserted as an individual Trie node. Note that the *children* is an array of pointers (or references) to next level trie nodes. The key character acts as an index into the array *children*. If the input key is new or an extension of existing key, we need to construct non-existing nodes of the key, and mark end of word for last node. If the input key is prefix of existing key in Trie, we simply mark the last node of key as end of word. The key length determines Trie depth.

Searching for a key is similar to insert operation, however we only compare the characters and move down. The search can terminate due to end of string or lack of key in trie. In the former case, if the *isEndofWord* field of last node is true, then the key exists in trie. In the second case, the search terminates without examining all the characters of key, since the key is not present in trie.

The following picture explains construction of trie using keys given in the example below,



In the picture, every character is of type *trie_node_t*. For example, the *root* is of type *trie_node_t*, and it's children *a*, *b* and *t* are filled, all other nodes of root will be NULL. Similarly, “a” at the next level is having only one child (“n”), all other children are NULL. The leaf nodes are in blue.

Insert and search costs $O(\text{key_length})$, however the memory requirements of Trie is $O(\text{ALPHABET_SIZE} * \text{key_length} * N)$ where N is number of keys in Trie. There are efficient representation of trie nodes (e.g. compressed trie, [ternary search tree](#), etc.) to minimize memory requirements of trie.

C++

```
// C++ implementation of search and insert
// operations on Trie
```

```
#include <bits/stdc++.h>
using namespace std;

const int ALPHABET_SIZE = 26;

// trie node
struct TrieNode
{
    struct TrieNode *children[ALPHABET_SIZE];

    // isEndOfWord is true if the node represents
    // end of a word
    bool isEndOfWord;
};

// Returns new trie node (initialized to NULLs)
struct TrieNode *getNode(void)
{
    struct TrieNode *pNode = new TrieNode;

    pNode->isEndOfWord = false;

    for (int i = 0; i < ALPHABET_SIZE; i++)
        pNode->children[i] = NULL;

    return pNode;
}

// If not present, inserts key into trie
// If the key is prefix of trie node, just
// marks leaf node
void insert(struct TrieNode *root, string key)
{
    struct TrieNode *pCrawl = root;

    for (int i = 0; i < key.length(); i++)
    {
        int index = key[i] - 'a';
        if (!pCrawl->children[index])
            pCrawl->children[index] = getNode();

        pCrawl = pCrawl->children[index];
    }

    // mark last node as leaf
    pCrawl->isEndOfWord = true;
}
```

```
// Returns true if key presents in trie, else
// false
bool search(struct TrieNode *root, string key)
{
    struct TrieNode *pCrawl = root;

    for (int i = 0; i < key.length(); i++)
    {
        int index = key[i] - 'a';
        if (!pCrawl->children[index])
            return false;

        pCrawl = pCrawl->children[index];
    }

    return (pCrawl != NULL && pCrawl->isEndOfWord);
}

// Driver
int main()
{
    // Input keys (use only 'a' through 'z'
    // and lower case)
    string keys[] = {"the", "a", "there",
                     "answer", "any", "by",
                     "bye", "their"};
    int n = sizeof(keys)/sizeof(keys[0]);

    struct TrieNode *root = getNode();

    // Construct trie
    for (int i = 0; i < n; i++)
        insert(root, keys[i]);

    // Search for different keys
    search(root, "the")? cout << "Yes\n" :
                           cout << "No\n";
    search(root, "these")? cout << "Yes\n" :
                           cout << "No\n";
    return 0;
}
```

C

```
// C implementation of search and insert operations
// on Trie
#include <stdio.h>
#include <stdlib.h>
```

```
#include <string.h>
#include <stdbool.h>

#define ARRAY_SIZE(a) sizeof(a)/sizeof(a[0])

// Alphabet size (# of symbols)
#define ALPHABET_SIZE (26)

// Converts key current character into index
// use only 'a' through 'z' and lower case
#define CHAR_TO_INDEX(c) ((int)c - (int)'a')

// trie node
struct TrieNode
{
    struct TrieNode *children[ALPHABET_SIZE];

    // isEndOfWord is true if the node represents
    // end of a word
    bool isEndOfWord;
};

// Returns new trie node (initialized to NULLs)
struct TrieNode *getNode(void)
{
    struct TrieNode *pNode = NULL;

    pNode = (struct TrieNode *)malloc(sizeof(struct TrieNode));

    if (pNode)
    {
        int i;

        pNode->isEndOfWord = false;

        for (i = 0; i < ALPHABET_SIZE; i++)
            pNode->children[i] = NULL;
    }

    return pNode;
}

// If not present, inserts key into trie
// If the key is prefix of trie node, just marks leaf node
void insert(struct TrieNode *root, const char *key)
{
    int level;
    int length = strlen(key);
```

```
int index;

struct TrieNode *pCrawl = root;

for (level = 0; level < length; level++)
{
    index = CHAR_TO_INDEX(key[level]);
    if (!pCrawl->children[index])
        pCrawl->children[index] = getNode();

    pCrawl = pCrawl->children[index];
}

// mark last node as leaf
pCrawl->isEndOfWord = true;
}

// Returns true if key presents in trie, else false
bool search(struct TrieNode *root, const char *key)
{
    int level;
    int length = strlen(key);
    int index;
    struct TrieNode *pCrawl = root;

    for (level = 0; level < length; level++)
    {
        index = CHAR_TO_INDEX(key[level]);

        if (!pCrawl->children[index])
            return false;

        pCrawl = pCrawl->children[index];
    }

    return (pCrawl != NULL && pCrawl->isEndOfWord);
}

// Driver
int main()
{
    // Input keys (use only 'a' through 'z' and lower case)
    char keys[] [8] = {"the", "a", "there", "answer", "any",
                      "by", "bye", "their"};

    char output[] [32] = {"Not present in trie", "Present in trie"};
}
```

```
struct TrieNode *root = getNode();

// Construct trie
int i;
for (i = 0; i < ARRAY_SIZE(keys); i++)
    insert(root, keys[i]);

// Search for different keys
printf("%s --- %s\n", "the", output[search(root, "the")]);
printf("%s --- %s\n", "these", output[search(root, "these")]);
printf("%s --- %s\n", "their", output[search(root, "their")]);
printf("%s --- %s\n", "thaw", output[search(root, "thaw")]);

return 0;
}
```

Java

```
// Java implementation of search and insert operations
// on Trie
public class Trie {

    // Alphabet size (# of symbols)
    static final int ALPHABET_SIZE = 26;

    // trie node
    static class TrieNode
    {
        TrieNode[] children = new TrieNode[ALPHABET_SIZE];

        // isEndOfWord is true if the node represents
        // end of a word
        boolean isEndOfWord;

        TrieNode(){
            isEndOfWord = false;
            for (int i = 0; i < ALPHABET_SIZE; i++)
                children[i] = null;
        }
    };

    static TrieNode root;

    // If not present, inserts key into trie
    // If the key is prefix of trie node,
    // just marks leaf node
    static void insert(String key)
    {
```

```
int level;
int length = key.length();
int index;

TrieNode pCrawl = root;

for (level = 0; level < length; level++)
{
    index = key.charAt(level) - 'a';
    if (pCrawl.children[index] == null)
        pCrawl.children[index] = new TrieNode();

    pCrawl = pCrawl.children[index];
}

// mark last node as leaf
pCrawl.isEndOfWord = true;
}

// Returns true if key presents in trie, else false
static boolean search(String key)
{
    int level;
    int length = key.length();
    int index;
    TrieNode pCrawl = root;

    for (level = 0; level < length; level++)
    {
        index = key.charAt(level) - 'a';

        if (pCrawl.children[index] == null)
            return false;

        pCrawl = pCrawl.children[index];
    }

    return (pCrawl != null && pCrawl.isEndOfWord);
}

// Driver
public static void main(String args[])
{
    // Input keys (use only 'a' through 'z' and lower case)
    String keys[] = {"the", "a", "there", "answer", "any",
                     "by", "bye", "their"};

    String output[] = {"Not present in trie", "Present in trie"};
}
```

```
root = new TrieNode();

// Construct trie
int i;
for (i = 0; i < keys.length ; i++)
    insert(keys[i]);

// Search for different keys
if(search("the") == true)
    System.out.println("the --- " + output[1]);
else System.out.println("the --- " + output[0]);

if(search("these") == true)
    System.out.println("these --- " + output[1]);
else System.out.println("these --- " + output[0]);

if(search("their") == true)
    System.out.println("their --- " + output[1]);
else System.out.println("their --- " + output[0]);

if(search("thaw") == true)
    System.out.println("thaw --- " + output[1]);
else System.out.println("thaw --- " + output[0]);

}

}

// This code is contributed by Sumit Ghosh
```

Python

```
# Python program for insert and search
# operation in a Trie

class TrieNode:

    # Trie node class
    def __init__(self):
        self.children = [None]*26

    # isEndOfWord is True if node represent the end of the word
    self.isEndOfWord = False

class Trie:

    # Trie data structure class
    def __init__(self):
```

```
    self.root = self.getNode()

def getNode(self):

    # Returns new trie node (initialized to NULLs)
    return TrieNode()

def _charToIndex(self, ch):

    # private helper function
    # Converts key current character into index
    # use only 'a' through 'z' and lower case

    return ord(ch)-ord('a')

def insert(self, key):

    # If not present, inserts key into trie
    # If the key is prefix of trie node,
    # just marks leaf node
    pCrawl = self.root
    length = len(key)
    for level in range(length):
        index = self._charToIndex(key[level])

        # if current character is not present
        if not pCrawl.children[index]:
            pCrawl.children[index] = self.getNode()
            pCrawl = pCrawl.children[index]

    # mark last node as leaf
    pCrawl.isEndOfWord = True

def search(self, key):

    # Search key in the trie
    # Returns true if key presents
    # in trie, else false
    pCrawl = self.root
    length = len(key)
    for level in range(length):
        index = self._charToIndex(key[level])
        if not pCrawl.children[index]:
            return False
        pCrawl = pCrawl.children[index]

    return pCrawl != None and pCrawl.isEndOfWord
```

```
# driver function
def main():

    # Input keys (use only 'a' through 'z' and lower case)
    keys = ["the", "a", "there", "anaswe", "any",
            "by", "their"]
    output = ["Not present in trie",
              "Present in tire"]

    # Trie object
    t = Trie()

    # Construct trie
    for key in keys:
        t.insert(key)

    # Search for different keys
    print("{} ---- {}".format("the",output[t.search("the")]))
    print("{} ---- {}".format("these",output[t.search("these")]))
    print("{} ---- {}".format("their",output[t.search("their")]))
    print("{} ---- {}".format("thaw",output[t.search("thaw")]))

if __name__ == '__main__':
    main()

# This code is contributed by Atul Kumar (www.facebook.com/atul.kr.007)
```

Output :

```
the --- Present in trie
these --- Not present in trie
their --- Present in trie
thaw --- Not present in trie
```

NOTE : In video, **isEndOfWord** is referred as **isLeaf**.

Related Articles:

- [Trie Delete](#)
- [Displaying content of Trie](#)
- [Applications of Trie](#)
- [Auto-complete feature using Trie](#)
- [Minimum Word Break](#)
- [Sorting array of strings \(or words\) using Trie](#)

- Pattern Searching using a Trie of all Suffixes

Practice Problems :

1. Trie Search and Insert
2. Trie Delete
3. Unique rows in a binary matrix
4. Count of distinct substrings
5. Word Boggle

Recent Articles on Trie

This article is contributed by **Venki**. 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/trie-insert-and-search/>

Chapter 205

Two Dimensional Binary Indexed Tree or Fenwick Tree

Two Dimensional Binary Indexed Tree or Fenwick Tree - GeeksforGeeks

Prerequisite – [Fenwick Tree](#)

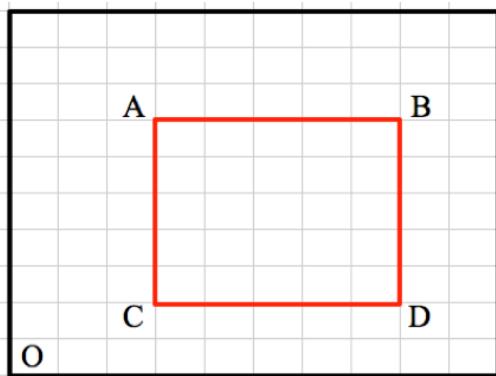
We know that to answer range sum queries on a 1-D array efficiently, binary indexed tree (or Fenwick Tree) is the best choice (even better than segment tree due to less memory requirements and a little faster than segment tree).

Can we answer sub-matrix sum queries efficiently using Binary Indexed Tree ?

The answer is yes. This is possible using a **2D BIT** which is nothing but an array of 1D BIT.

Algorithm:

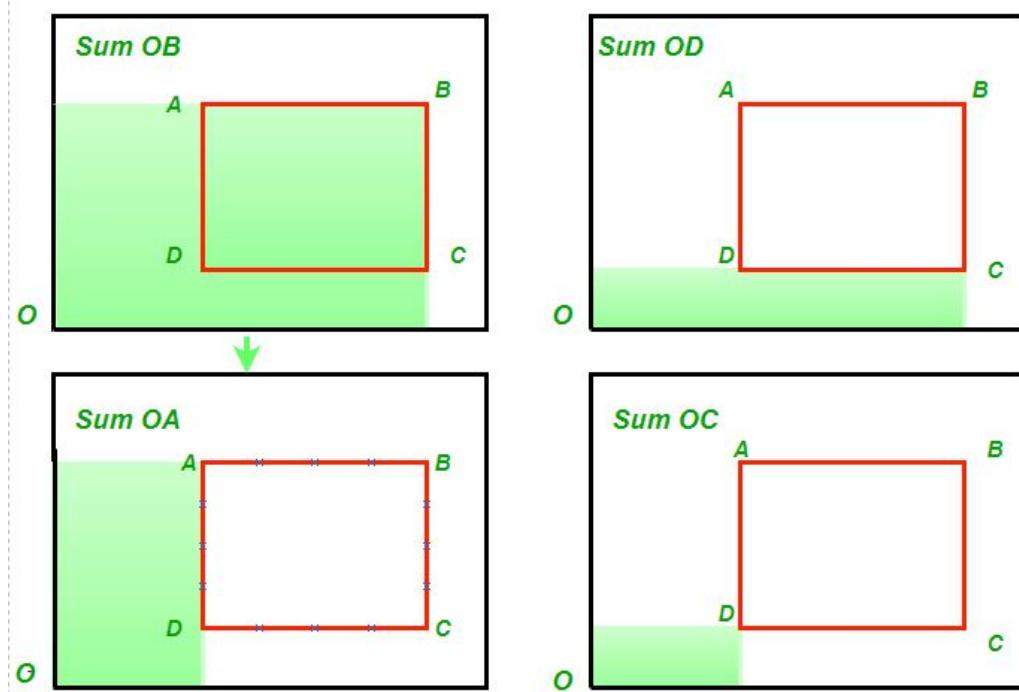
We consider the below example. Suppose we have to find the sum of all numbers inside the highlighted area-



70	37	23	57	27	22	90	99	22	59
47	63	33	1	42	46	6	70	98	93
36	62	50	21	92	27	60	29	15	34
53	3	88	45	57	39	83	81	79	56
28	63	89	20	47	15	84	18	82	33
26	87	11	76	79	5	94	55	73	51
17	82	86	10	96	5	42	43	51	6
44	76	51	4	15	99	52	11	70	89
66	36	92	85	50	21	72	27	52	65
60	0	67	37	59	14	33	13	36	36

We assume the origin of the matrix at the bottom – O. Then a 2D BIT exploits the fact that-

Sum under the marked area = Sum(OB) - Sum(OD) -
Sum(OA) + Sum(OC)



In our program, we use the `getSum(x, y)` function which finds the sum of the matrix from $(0, 0)$ to (x, y) .

Hence the below formula :

Sum under the marked area = Sum(OB) - Sum(OD) -
Sum(OA) + Sum(OC)

The above formula gets reduced to,

```
Query(x1,y1,x2,y2) = getSum(x2, y2) -
                      getSum(x2, y1-1) -
                      getSum(x1-1, y2) +
                      getSum(x1-1, y1-1)
```

where,

x1, y1 = x and y coordinates of C

x2, y2 = x and y coordinates of B

The `updateBIT(x, y, val)` function updates all the elements under the region $-(x, y)$ to (N, M) where,

N = maximum X co-ordinate of the whole matrix.

M = maximum Y co-ordinate of the whole matrix.

The rest procedure is quite similar to that of 1D Binary Indexed Tree. Below is the C++ implementation of 2D indexed tree

```
/* C++ program to implement 2D Binary Indexed Tree

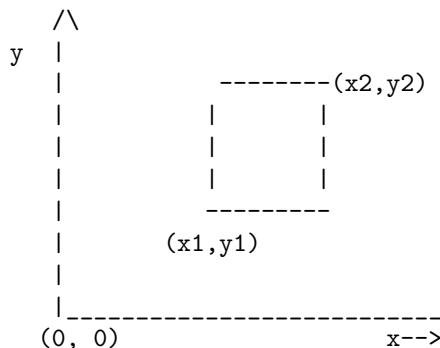
2D BIT is basically a BIT where each element is another BIT.
Updating by adding v on (x, y) means it's effect will be found
throughout the rectangle [(x, y), (max_x, max_y)],
and query for (x, y) gives you the result of the rectangle
[(0, 0), (x, y)], assuming the total rectangle is
[(0, 0), (max_x, max_y)]. So when you query and update on
this BIT, you have to be careful about how many times you are
subtracting a rectangle and adding it. Simple set union formula
works here.
```

So if you want to get the result of a specific rectangle
[(x1, y1), (x2, y2)], the following steps are necessary:

```
Query(x1,y1,x2,y2) = getSum(x2, y2)-getSum(x2, y1-1) -
                      getSum(x1-1, y2)+getSum(x1-1, y1-1)
```

Here 'Query(x1,y1,x2,y2)' means the sum of elements enclosed
in the rectangle with bottom-left corner's co-ordinates
(x1, y1) and top-right corner's co-ordinates - (x2, y2)

Constraints -> x1<=x2 and y1<=y2



In this program we have assumed a square matrix. The
program can be easily extended to a rectangular one. */

```
#include<bits/stdc++.h>
using namespace std;

#define N 4 // N-->max_x and max_y
```

```

// A structure to hold the queries
struct Query
{
    int x1, y1; // x and y co-ordinates of bottom left
    int x2, y2; // x and y co-ordinates of top right
};

// A function to update the 2D BIT
void updateBIT(int BIT[][] [N+1], int x, int y, int val)
{
    for (; x <= N; x += (x & -x))
    {
        // This loop update all the 1D BIT inside the
        // array of 1D BIT = BIT[x]
        for (; y <= N; y += (y & -y))
            BIT[x] [y] += val;
    }
    return;
}

// A function to get sum from (0, 0) to (x, y)
int getSum(int BIT[][] [N+1], int x, int y)
{
    int sum = 0;

    for(; x > 0; x -= x&-x)
    {
        // This loop sum through all the 1D BIT
        // inside the array of 1D BIT = BIT[x]
        for(; y > 0; y -= y&-y)
        {
            sum += BIT[x] [y];
        }
    }
    return sum;
}

// A function to create an auxiliary matrix
// from the given input matrix
void constructAux(int mat[][] [N], int aux[][] [N+1])
{
    // Initialise Auxiliary array to 0
    for (int i=0; i<=N; i++)
        for (int j=0; j<=N; j++)
            aux[i] [j] = 0;

    // Construct the Auxiliary Matrix
}

```

```

        for (int j=1; j<=N; j++)
            for (int i=1; i<=N; i++)
                aux[i][j] = mat[N-j][i-1];

        return;
    }

// A function to construct a 2D BIT
void construct2DBIT(int mat[][][N], int BIT[][][N+1])
{
    // Create an auxiliary matrix
    int aux[N+1][N+1];
    constructAux(mat, aux);

    // Initialise the BIT to 0
    for (int i=1; i<=N; i++)
        for (int j=1; j<=N; j++)
            BIT[i][j] = 0;

    for (int j=1; j<=N; j++)
    {
        for (int i=1; i<=N; i++)
        {
            // Creating a 2D-BIT using update function
            // everytime we/ encounter a value in the
            // input 2D-array
            int v1 = getSum(BIT, i, j);
            int v2 = getSum(BIT, i, j-1);
            int v3 = getSum(BIT, i-1, j-1);
            int v4 = getSum(BIT, i-1, j);

            // Assigning a value to a particular element
            // of 2D BIT
            updateBIT(BIT, i, j, aux[i][j]-(v1-v2-v4+v3));
        }
    }

    return;
}

// A function to answer the queries
void answerQueries(Query q[], int m, int BIT[][][N+1])
{
    for (int i=0; i<m; i++)
    {
        int x1 = q[i].x1 + 1;
        int y1 = q[i].y1 + 1;
        int x2 = q[i].x2 + 1;
    }
}

```

```

int y2 = q[i].y2 + 1;

int ans = getSum(BIT, x2, y2)-getSum(BIT, x2, y1-1)-
          getSum(BIT, x1-1, y2)+getSum(BIT, x1-1, y1-1);

printf ("Query(%d, %d, %d, %d) = %d\n",
       q[i].x1, q[i].y1, q[i].x2, q[i].y2, ans);
}

return;
}

// Driver program
int main()
{
    int mat[N][N] = {{1, 2, 3, 4},
                      {5, 3, 8, 1},
                      {4, 6, 7, 5},
                      {2, 4, 8, 9};

    // Create a 2D Binary Indexed Tree
    int BIT[N+1][N+1];
    construct2DBIT(mat, BIT);

    /* Queries of the form - x1, y1, x2, y2
       For example the query- {1, 1, 3, 2} means the sub-matrix-
       y
       \ \
       3 |      1 2 3 4      Sub-matrix
       2 |      5 3 8 1      {1,1,3,2}      --->      3 8 1
       1 |      4 6 7 5
       0 |      2 4 8 9
       |
       --|----- 0 1 2 3 -----> x
       |

    Hence sum of the sub-matrix = 3+8+1+6+7+5 = 30

    */
}

Query q[] = {{1, 1, 3, 2}, {2, 3, 3, 3}, {1, 1, 1, 1}};
int m = sizeof(q)/sizeof(q[0]);

answerQueries(q, m, BIT);

return(0);
}

```

Output:

```
Query(1, 1, 3, 2) = 30
Query(2, 3, 3, 3) = 7
Query(1, 1, 1, 1) = 6
```

Time Complexity:

- Both updateBIT(x, y, val) function and getSum(x, y) function takes $O(\log(NM))$ time.
- Building the 2D BIT takes $O(NM \log(NM))$.
- Since in each of the queries we are calling getSum(x, y) function so answering all the Q queries takes $O(Q \cdot \log(NM))$ time.

Hence the overall time complexity of the program is $O((NM+Q) \cdot \log(NM))$ where,

N = maximum X co-ordinate of the whole matrix.

M = maximum Y co-ordinate of the whole matrix.

Q = Number of queries.

Auxiliary Space: $O(NM)$ to store the BIT and the auxiliary array

References: <https://www.topcoder.com/community/data-science/data-science-tutorials/binary-indexed-trees/>

Source

<https://www.geeksforgeeks.org/two-dimensional-binary-indexed-tree-or-fenwick-tree/>

Chapter 206

Two Dimensional Segment Tree Sub-Matrix Sum

Two Dimensional Segment Tree Sub-Matrix Sum - GeeksforGeeks

Given a rectangular matrix $M[0...n-1][0...m-1]$, and queries are asked to find the sum / minimum / maximum on some sub-rectangles $M[a...b][e...f]$, as well as queries for modification of individual matrix elements (i.e $M[x][y] = p$).

We can also answer sub-matrix queries using [Two Dimensional Binary Indexed Tree](#).

In this article, We will focus on solving sub-matrix queries using two dimensional segment tree.Two dimensional segment tree is nothing but segment tree of segment trees.

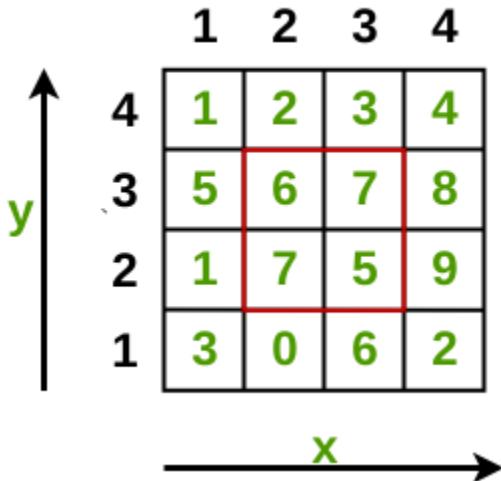
Prerequisite : [Segment Tree – Sum of given range](#)

Algorithm :

We will build a two-dimensional tree of segments by the following principle:

- 1 . In First step, We will construct an ordinary one-dimensional segment tree, working only with the first coordinate say ‘x’ and ‘y’ as constant. Here, we will not write number in inside the node as in the one-dimensional segment tree, but an entire tree of segments.
2. The second step is to combine the values of segmented trees. Assume that in second step instead of combining the elements we are combining the segment trees obtained from the step first.

Consider the below example. Suppose we have to find the sum of all numbers inside the highlighted red area



Step 1 : We will first create the segment tree of each strip of y- axis. We represent the segment tree here as an array where child node is $2n$ and $2n+1$ where $n > 0$.

Segment Tree for strip $y=1$

10	3	7	1	2	3	4
----	---	---	---	---	---	---

Segment Tree for Strip $y = 2$

26	11	15	5	6	7	8
----	----	----	---	---	---	---

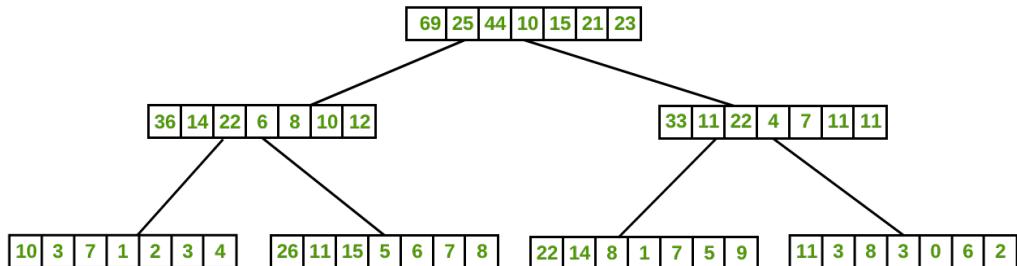
Segment Tree for Strip $y = 3$

22	14	8	1	7	5	9
----	----	---	---	---	---	---

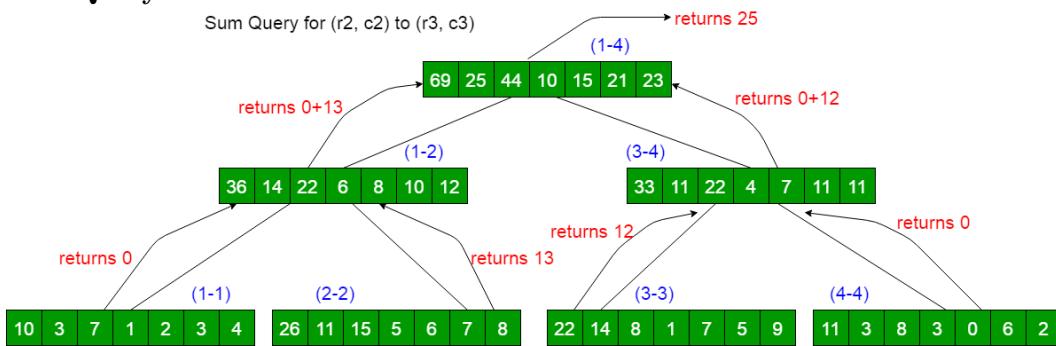
Segment Tree for Strip $y = 4$

11	3	8	3	0	6	2
----	---	---	---	---	---	---

Step 2: In this step, we create the segment tree for the rectangular matrix where the base node are the strips of y-axis given above. The task is to merge above segment trees.



Sum Query :



Thanks to **Sahil Bansal** for contributing this image.

Processing Query :

We will respond to the two-dimensional query by the following principle: first to break the query on the first coordinate, and then, when we reached some vertex of the tree of segments with the first coordinate and then we call the corresponding tree of segments on the second coordinate.

This function works in time **O(log n * log m)**, because it first descends the tree in the first coordinate, and for each traversed vertex of that tree, it makes a query from the usual tree of segments along the second coordinate.

Modification Query :

We want to learn how to modify the tree of segments in accordance with the change in the value of an element $M[x][y] = p$. It is clear that the changes will occur only in those vertices of the first tree of segments that cover the coordinate x , and for the trees of the segments corresponding to them, the changes will only occur in those vertices that cover the coordinate y . Therefore, the implementation of the modification request will not be very different from the one-dimensional case, only now we first descend the first coordinate, and then the second.

Output for the highlighted area will be 25.

Below is the implementation of above approach :

```
// C++ program for implementation
// of 2D segment tree.
#include <bits/stdc++.h>
using namespace std;

// Base node of segment tree.
int ini_seg[1000][1000] = { 0 };

// final 2d-segment tree.
int fin_seg[1000][1000] = { 0 };

// Rectangular matrix.
int rect[4][4] = {
    { 1, 2, 3, 4 },
    { 5, 6, 7, 8 },
    { 1, 7, 5, 9 },
    { 3, 0, 6, 2 },
};

// size of x coordinate.
int size = 4;

/*
 * A recursive function that constructs
 * Initial Segment Tree for array rect[][] = { }.
 * 'pos' is index of current node in segment
 * tree seg[]. 'strip' is the enumeration
 * for the y-axis.
 */
int segment(int low, int high,
            int pos, int strip)
{
    if (high == low) {
        ini_seg[strip][pos] = rect[strip][low];
    }
    else {
        int mid = (low + high) / 2;
        segment(low, mid, 2 * pos, strip);
        segment(mid + 1, high, 2 * pos + 1, strip);
        ini_seg[strip][pos] = ini_seg[strip][2 * pos] +
                            ini_seg[strip][2 * pos + 1];
    }
}

/*
 * A recursive function that constructs
 * Final Segment Tree for array ini_seg[][] = { }.
```

```

*/
int finalSegment(int low, int high, int pos)
{
    if (high == low) {

        for (int i = 1; i < 2 * size; i++)
            fin_seg[pos][i] = ini_seg[low][i];
    }
    else {
        int mid = (low + high) / 2;
        finalSegment(low, mid, 2 * pos);
        finalSegment(mid + 1, high, 2 * pos + 1);

        for (int i = 1; i < 2 * size; i++)
            fin_seg[pos][i] = fin_seg[2 * pos][i] +
                fin_seg[2 * pos + 1][i];
    }
}

/*
* Return sum of elements in range from index
* x1 to x2 . It uses the final_seg[][] array
* created using finalsegment() function.
* 'pos' is index of current node in
* segment tree fin_seg[][] .
*/
int finalQuery(int pos, int start, int end,
               int x1, int x2, int node)
{
    if (x2 < start || end < x1) {
        return 0;
    }

    if (x1 <= start && end <= x2) {
        return fin_seg[node][pos];
    }

    int mid = (start + end) / 2;
    int p1 = finalQuery(2 * pos, start, mid,
                        x1, x2, node);

    int p2 = finalQuery(2 * pos + 1, mid + 1,
                        end, x1, x2, node);

    return (p1 + p2);
}
/*

```

```

* This fuction calls the finalQuery fuction
* for elements in range from index x1 to x2 .
* This fuction queries the yth coordinate.
*/
int query(int pos, int start, int end,
          int y1, int y2, int x1, int x2)
{
    if (y2 < start || end < y1) {
        return 0;
    }

    if (y1 <= start && end <= y2) {
        return (finalQuery(1, 1, 4, x1, x2, pos));
    }

    int mid = (start + end) / 2;
    int p1 = query(2 * pos, start,
                   mid, y1, y2, x1, x2);
    int p2 = query(2 * pos + 1, mid + 1,
                   end, y1, y2, x1, x2);

    return (p1 + p2);
}

/* A recursive function to update the nodes
   which for the given index. The following
   are parameters : pos --> index of current
   node in segment tree fin_seg[][] . x ->
   index of the element to be updated. val -->
   Value to be change at node idx
*/
int finalUpdate(int pos, int low, int high,
                int x, int val, int node)
{
    if (low == high) {
        fin_seg[node][pos] = val;
    }
    else {
        int mid = (low + high) / 2;

        if (low <= x && x <= mid) {
            finalUpdate(2 * pos, low, mid, x, val, node);
        }
        else {
            finalUpdate(2 * pos + 1, mid + 1, high,
                        x, val, node);
        }
    }
}

```

```

        fin_seg[node][pos] = fin_seg[node][2 * pos] +
                            fin_seg[node][2 * pos + 1];
    }
}

/*
This function call the final update function after
visiting the yth coordinate in the segment tree fin_seg[][].
*/
int update(int pos, int low, int high, int x, int y, int val)
{
    if (low == high) {
        finalUpdate(1, 1, 4, x, val, pos);
    }
    else {
        int mid = (low + high) / 2;

        if (low <= y && y <= mid) {
            update(2 * pos, low, mid, x, y, val);
        }
        else {
            update(2 * pos + 1, mid + 1, high, x, y, val);
        }

        for (int i = 1; i < size; i++)
            fin_seg[pos][i] = fin_seg[2 * pos][i] +
                               fin_seg[2 * pos + 1][i];
    }
}

// Driver program to test above functions
int main()
{
    int pos = 1;
    int low = 0;
    int high = 3;

    // Call the ini_segment() to create the
    // initial segment tree on x- coordinate
    for (int strip = 0; strip < 4; strip++)
        segment(low, high, 1, strip);

    // Call the final function to built the 2d segment tree.
    finalSegment(low, high, 1);

    /*
Query:
* To request the query for sub-rectangle y1, y2=(2, 3) x1, x2=(2, 3)

```

```
* update the value of index (3, 3)=100;
* To request the query for sub-rectangle y1, y2=(2, 3) x1, x2=(2, 3)
*/
    cout << "The sum of the submatrix (y1, y2)->(2, 3), "
        << "(x1, x2)->(2, 3) is "
        << query(1, 1, 4, 2, 3, 2, 3) << endl;

// Function to update the value
update(1, 1, 4, 2, 3, 100);

cout << "The sum of the submatrix (y1, y2)->(2, 3), "
        << "(x1, x2)->(2, 3) is "
        << query(1, 1, 4, 2, 3, 2, 3) << endl;

return 0;
}
```

Output:

```
The sum of the submatrix (y1, y2)->(2, 3), (x1, x2)->(2, 3) is 25
The sum of the submatrix (y1, y2)->(2, 3), (x1, x2)->(2, 3) is 118
```

Time complexity :

Processing Query : $O(\log n * \log m)$
Modification Query: $O(2 * n * \log n * \log m)$
Space Complexity : $O(4 * m * n)$

Source

<https://www.geeksforgeeks.org/two-dimensional-segment-tree-sub-matrix-sum/>

Chapter 207

Ukkonen's Suffix Tree Construction – Part 1

Ukkonen's Suffix Tree Construction - Part 1 - GeeksforGeeks

Suffix Tree is very useful in numerous string processing and computational biology problems. Many books and e-resources talk about it theoretically and in few places, code implementation is discussed. But still, I felt something is missing and it's not easy to implement code to construct suffix tree and it's usage in many applications. This is an attempt to bridge the gap between theory and complete working code implementation. Here we will discuss Ukkonen's Suffix Tree Construction Algorithm. We will discuss it in step by step detailed way and in multiple parts from theory to implementation. We will start with brute force way and try to understand different concepts, tricks involved in Ukkonen's algorithm and in the last part, code implementation will be discussed.

Note: You may find some portion of the algorithm difficult to understand while 1st or 2nd reading and it's perfectly fine. With few more attempts and thought, you should be able to understand such portions.

Book [Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology](#) by **Dan Gusfield** explains the concepts very well.

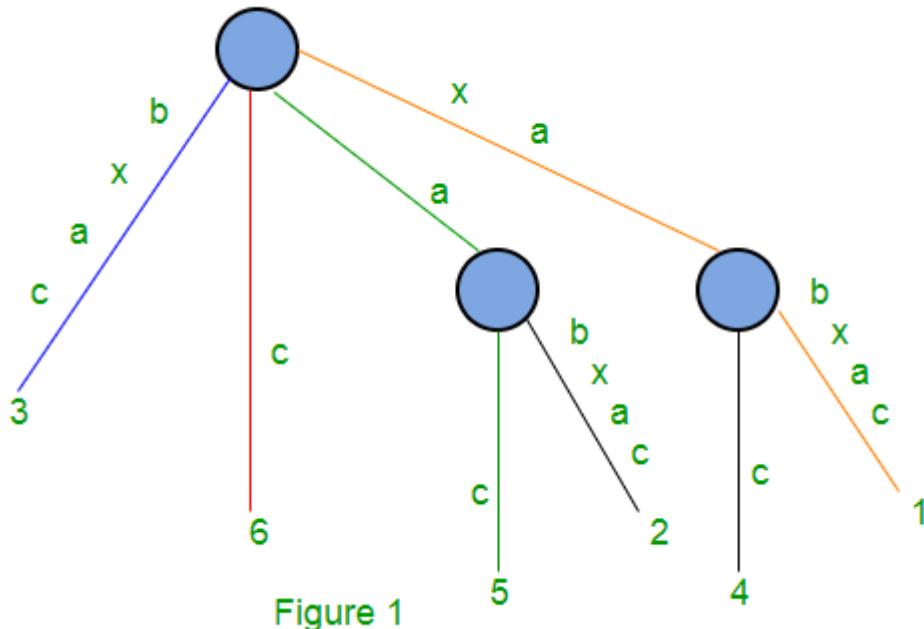
A suffix tree **T** for a m-character string **S** is a rooted directed tree with exactly m leaves numbered 1 to **m**. (Given that last string character is unique in string)

- Root can have zero, one or more children.
- Each internal node, other than the root, has at least two children.
- Each edge is labelled with a nonempty substring of **S**.
- No two edges coming out of same node can have edge-labels beginning with the same character.

Concatenation of the edge-labels on the path from the root to leaf i gives the suffix of **S** that starts at position i, i.e. **S[i...m]**.

Note: Position starts with 1 (it's not zero indexed, but later, while code implementation, we will used zero indexed position)

For string $S = \text{xabxac}$ with $m = 6$, suffix tree will look like following:



It has one root node and two internal nodes and 6 leaf nodes.

String Depth of red path is 1 and it represents suffix c starting at position 6

String Depth of blue path is 4 and it represents suffix bxaca starting at position 3

String Depth of green path is 2 and it represents suffix ac starting at position 5

String Depth of orange path is 6 and it represents suffix xabxac starting at position 1

Edges with labels a (green) and xa (orange) are non-leaf edge (which ends at an internal node). All other edges are leaf edge (ends at a leaf)

If one suffix of S matches a prefix of another suffix of S (when last character in not unique in string), then path for the first suffix would not end at a leaf.

For String $S = \text{xabxa}$, with $m = 5$, following is the suffix tree:

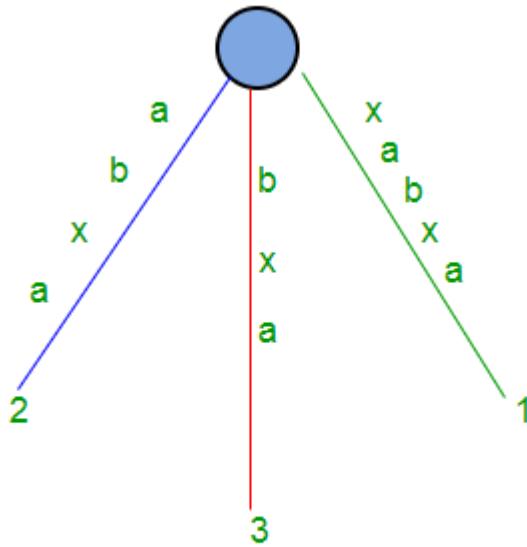


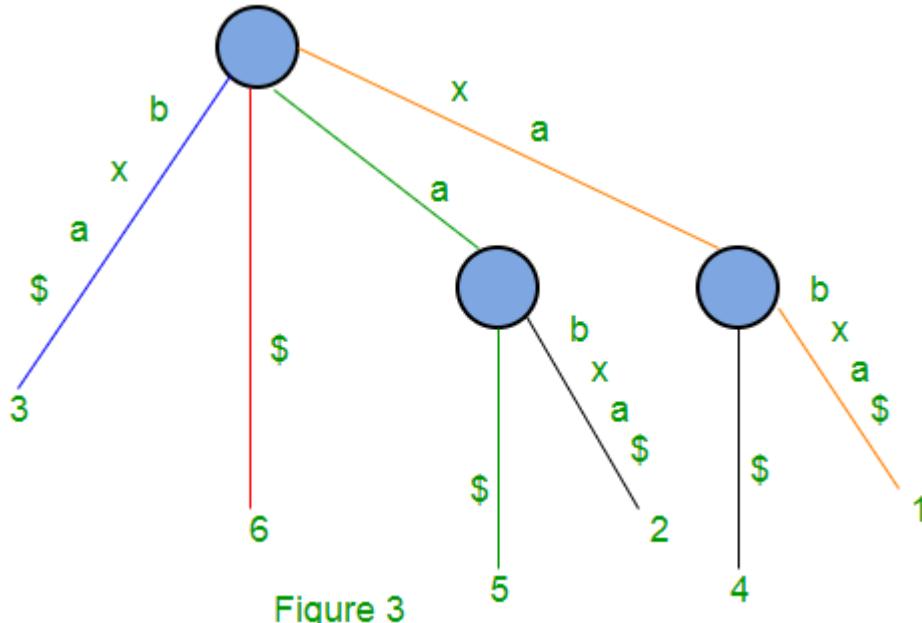
Figure 2

Here we will have 5 suffixes: xabxa, abxa, bxa, xa and a.

Path for suffixes ‘xa’ and ‘a’ do not end at a leaf. A tree like above (Figure 2) is called implicit suffix tree as some suffixes (‘xa’ and ‘a’) are not seen explicitly in tree.

To avoid this problem, we add a character which is not present in string already. We normally use \$, # etc as termination characters.

Following is the suffix tree for string $S = \text{xabxa\$}$ with $m = 6$ and now all 6 suffixes end at leaf.



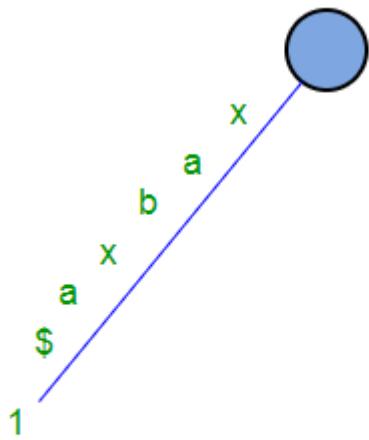
A naive algorithm to build a suffix tree

Given a string S of length m , enter a single edge for suffix $S[1..m]\$$ (the entire string) into the tree, then successively enter suffix $S[i..m]\$$ into the growing tree, for i increasing from 2 to m . Let N_i denote the intermediate tree that encodes all the suffixes from 1 to i . So N_{i+1} is constructed from N_i as follows:

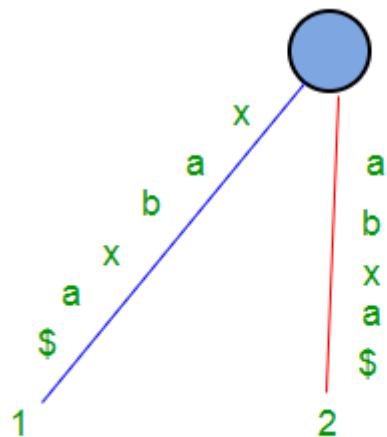
- Start at the root of N_i
- Find the longest path from the root which matches a prefix of $S[i+1..m]\$$
- Match ends either at the node (say w) or in the middle of an edge [say (u, v)].
- If it is in the middle of an edge (u, v) , break the edge (u, v) into two edges by inserting a new node w just after the last character on the edge that matched a character in $S[i+1..m]$ and just before the first character on the edge that mismatched. The new edge (u, w) is labelled with the part of the (u, v) label that matched with $S[i+1..m]$, and the new edge (w, v) is labelled with the remaining part of the (u, v) label.
- Create a new edge $(w, i+1)$ from w to a new leaf labelled $i+1$ and it labels the new edge with the unmatched part of suffix $S[i+1..m]$

This takes $O(m^2)$ to build the suffix tree for the string S of length m .

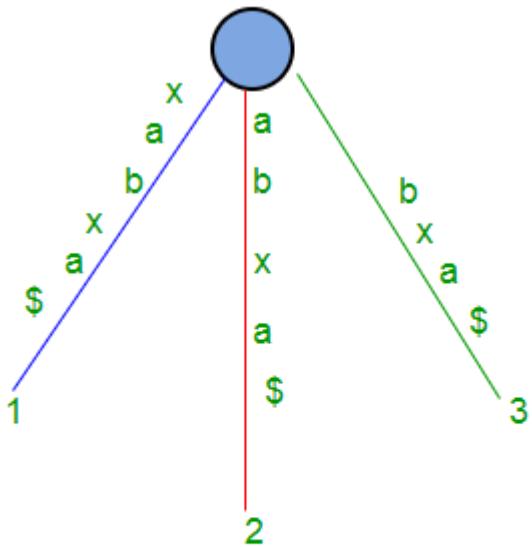
Following are few steps to build suffix tree based for string “ $xabxa\$$ ” based on above algorithm:



Tree with suffix N1, S[1....6]
Figure 4

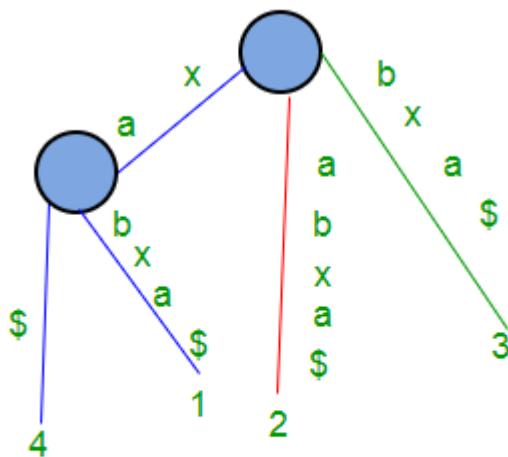


Tree with suffix N1, S[1....6] and N2, S[2...6]
Figure 5



Tree with suffixes N1,N2 and N3

Figure 6



Tree with suffix N1, N2, N3 and N4

Figure 7

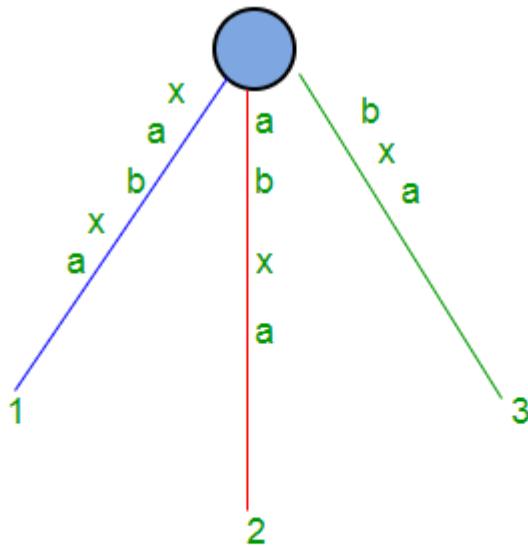
Implicit suffix tree

While generating suffix tree using Ukkonen's algorithm, we will see implicit suffix tree in intermediate steps few times depending on characters in string S. In implicit suffix trees, there will be no edge with \$ (or # or any other termination character) label and no internal

node with only one edge going out of it.

To get implicit suffix tree from a suffix tree $S\$$,

- Remove all terminal symbol $\$$ from the edge labels of the tree,
- Remove any edge that has no label
- Remove any node that has only one edge going out of it and merge the edges.



**Figure 8 : Implicit suffix tree for storing xabxa
Suffix tree shown in Figure 3**

High Level Description of Ukkonen's algorithm

Ukkonen's algorithm constructs an implicit suffix tree T_i for each prefix $S[1..i]$ of S (of length m).

It first builds T_1 using 1st character, then T_2 using 2nd character, then T_3 using 3rd character, ..., T_m using m^{th} character.

Implicit suffix tree T_{i+1} is built on top of implicit suffix tree T_i .

The true suffix tree for S is built from T_m by adding $\$$.

At any time, Ukkonen's algorithm builds the suffix tree for the characters seen so far and so it has **on-line** property that may be useful in some situations.

Time taken is $O(m)$.

Ukkonen's algorithm is divided into m phases (one phase for each character in the string with length m)

In phase $i+1$, tree T_{i+1} is built from tree T_i .

Each phase $i+1$ is further divided into $i+1$ extensions, one for each of the $i+1$ suffixes of $S[1..i+1]$

In extension j of phase $i+1$, the algorithm first finds the end of the path from the root labelled with substring $S[j..i]$.

It then extends the substring by adding the character $S(i+1)$ to its end (if it is not there already).

In extension 1 of phase $i+1$, we put string $S[1..i+1]$ in the tree. Here $S[1..i]$ will already be present in tree due to previous phase i . We just need to add $S[i+1]$ th character in tree (if not there already).

In extension 2 of phase $i+1$, we put string $S[2..i+1]$ in the tree. Here $S[2..i]$ will already be present in tree due to previous phase i . We just need to add $S[i+1]$ th character in tree (if not there already)

In extension 3 of phase $i+1$, we put string $S[3..i+1]$ in the tree. Here $S[3..i]$ will already be present in tree due to previous phase i . We just need to add $S[i+1]$ th character in tree (if not there already)

.

.

In extension $i+1$ of phase $i+1$, we put string $S[i+1..i+1]$ in the tree. This is just one character which may not be in tree (if character is seen first time so far). If so, we just add a new leaf edge with label $S[i+1]$.

High Level Ukkonen's algorithm

Construct tree T_1

For i from 1 to $m-1$ do

begin {phase $i+1$ }

For j from 1 to $i+1$

begin {extension j }

Find the end of the path from the root labelled $S[j..i]$ in the current tree.

Extend that path by adding character $S[i+1]$ if it is not there already

end;

end;

Suffix extension is all about adding the next character into the suffix tree built so far.

In extension j of phase $i+1$, algorithm finds the end of $S[j..i]$ (which is already in the tree due to previous phase i) and then it extends $S[j..i]$ to be sure the suffix $S[j..i+1]$ is in the tree.

There are 3 extension rules:

Rule 1: If the path from the root labelled $S[j..i]$ ends at leaf edge (i.e. $S[i]$ is last character on leaf edge) then character $S[i+1]$ is just added to the end of the label on that leaf edge.

Rule 2: If the path from the root labelled $S[j..i]$ ends at non-leaf edge (i.e. there are more characters after $S[i]$ on path) and next character is not $s[i+1]$, then a new leaf edge with label $s[i+1]$ and number j is created starting from character $S[i+1]$.

A new internal node will also be created if $s[1..i]$ ends inside (in-between) a non-leaf edge.

Rule 3: If the path from the root labelled $S[j..i]$ ends at non-leaf edge (i.e. there are more characters after $S[i]$ on path) and next character is $s[i+1]$ (already in tree), do nothing.

One important point to note here is that from a given node (root or internal), there will be one and only one edge starting from one character. There will not be more than one edges going out of any node, starting with same character.

Following is a step by step suffix tree construction of string `xabxac` using Ukkonen's

algorithm:

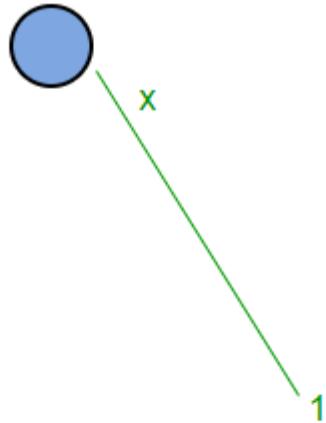


Figure 9 : T1 for S[1...1]
Adding suffixes of x(x)
Rule2-A new leaf node

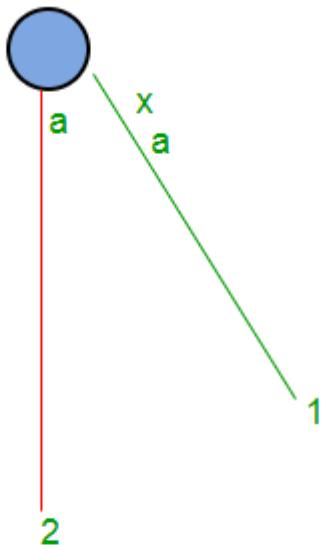


Figure 10 : T2 for S[1...2]
Adding suffixes of xa(xa and a)
Rule1-Extending path label in existing leaf edge
Rule2-A new leaf node

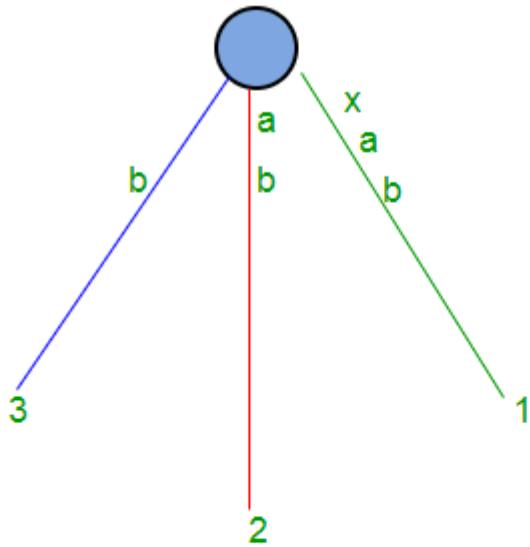


Figure 11 : T3 for S[1...3]
Adding suffixes of xab(xab, ab and b)

Rule1-Extending path label in existing leaf edge
Rule2-A new leaf node

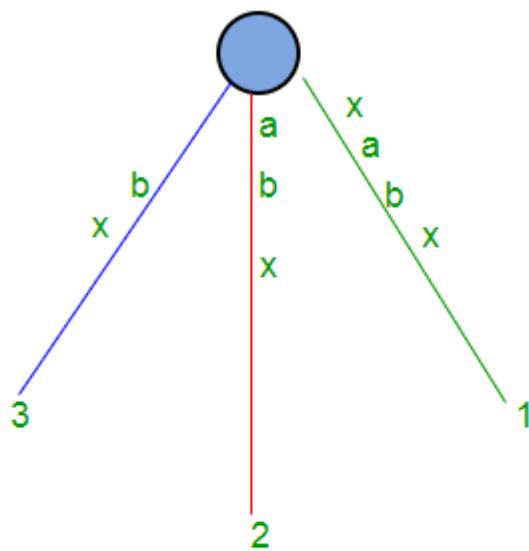


Figure 12 : T4 for S[1...4]
Adding suffixes of xabx(xabx, abx, bx and x)

Rule1-Extending path label in existing leaf edge
Rule3-Do nothing(path with label x already present)

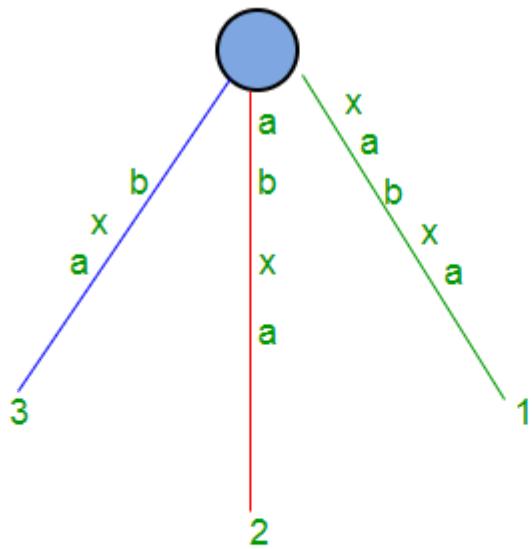


Figure 13 : T5 for S[1..5]

Adding suffixes of xabxa(xabxa,abxa,bxa,xa and x)

Rule1-Extending path label in existing leaf edge

Rule3-Do nothing(path with label xa and a already present)

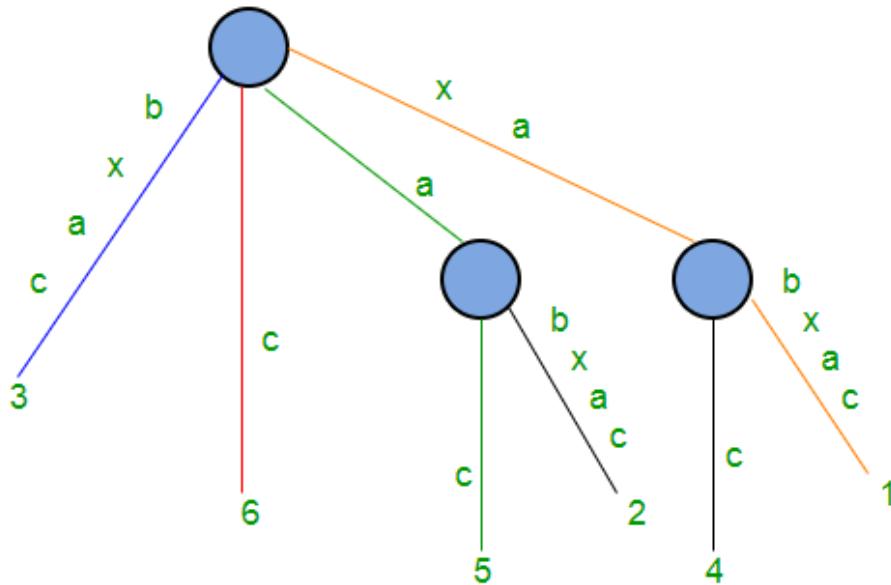


Figure 14 : T6 for $S[1..6]$

Adding suffixes of $xabxac$ ($xabxac$, $abxac$, $bxac$, xac , ac , c)

Rule1-Extending path label in Existing leaf edge.

Rule2-Three new leaf edges and two new internal nodes

In next parts ([Part 2](#), [Part 3](#), [Part 4](#) and [Part 5](#)), we will discuss suffix links, active points, few tricks and finally code implementations ([Part 6](#)).

References:

<http://web.stanford.edu/~mjkay/gusfield.pdf>

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

Source

<https://www.geeksforgeeks.org/ukkonens-suffix-tree-construction-part-1/>

Chapter 208

Ukkonen's Suffix Tree Construction – Part 2

Ukkonen's Suffix Tree Construction - Part 2 - GeeksforGeeks

In [Ukkonen's Suffix Tree Construction – Part 1](#), we have seen high level Ukkonen's Algorithm. This 2nd part is continuation of [Part 1](#).

Please go through [Part 1](#), before looking at current article.

In Suffix Tree Construction of string S of length m , there are m phases and for a phase j ($1 \leq j \leq m$), we add j^{th} character in tree built so far and this is done through j extensions. All extensions follow one of the three extension rules (discussed in [Part 1](#)).

To do j^{th} extension of phase $i+1$ (adding character $S[i+1]$), we first need to find end of the path from the root labelled $S[j..i]$ in the current tree. One way is start from root and traverse the edges matching $S[j..i]$ string. This will take $O(m^3)$ time to build the suffix tree. Using few observations and implementation tricks, it can be done in $O(m)$ which we will see now.

Suffix links

For an internal node v with path-label xA , where x denotes a single character and A denotes a (possibly empty) substring, if there is another node $s(v)$ with path-label A , then a pointer from v to $s(v)$ is called a suffix link.

If A is empty string, suffix link from internal node will go to root node.

There will not be any suffix link from root node (As it's not considered as internal node).

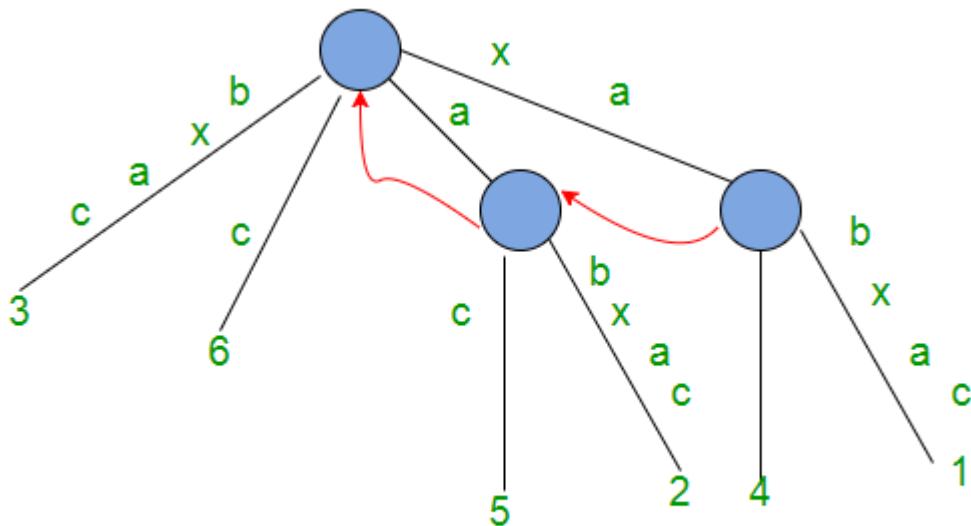


Figure 15 : Suffix links in red arrows

In extension j of some phase i , if a new internal node v with path-label xA is added, then in extension $j+1$ in the same phase i :

- Either the path labelled A already ends at an internal node (or root node if A is empty)
- OR a new internal node at the end of string A will be created

In extension $j+1$ of same phase i , we will create a suffix link from the internal node created in j^{th} extension to the node with path labelled A .

So in a given phase, any newly created internal node (with path-label xA) will have a suffix link from it (pointing to another node with path-label A) by the end of the next extension.

In any implicit suffix tree T_i after phase i , if internal node v has path-label xA , then there is a node $s(v)$ in T_i with path-label A and node v will point to node $s(v)$ using suffix link.

At any time, all internal nodes in the changing tree will have suffix links from them to another internal node (or root) except for the most recently added internal node, which will receive its suffix link by the end of the next extension.

How suffix links are used to speed up the implementation?

In extension j of phase $i+1$, we need to find the end of the path from the root labelled $S[j..i]$ in the current tree. One way is start from root and traverse the edges matching $S[j..i]$ string. Suffix links provide a short cut to find end of the path.

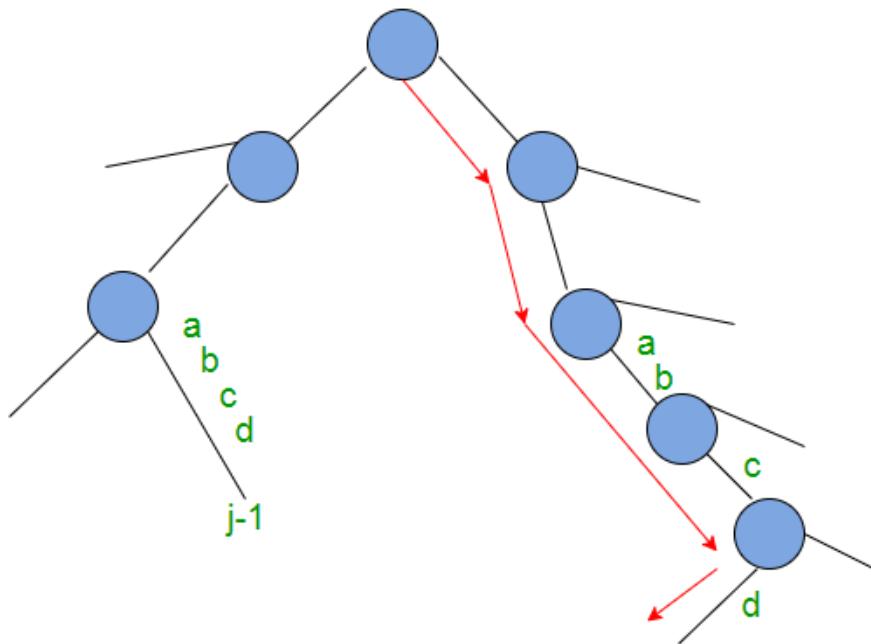


Figure 16 : Traversal from root to leaf in extension j of phase $i+1$, to find end of $S[j\dots i]$, when suffix link is not used.

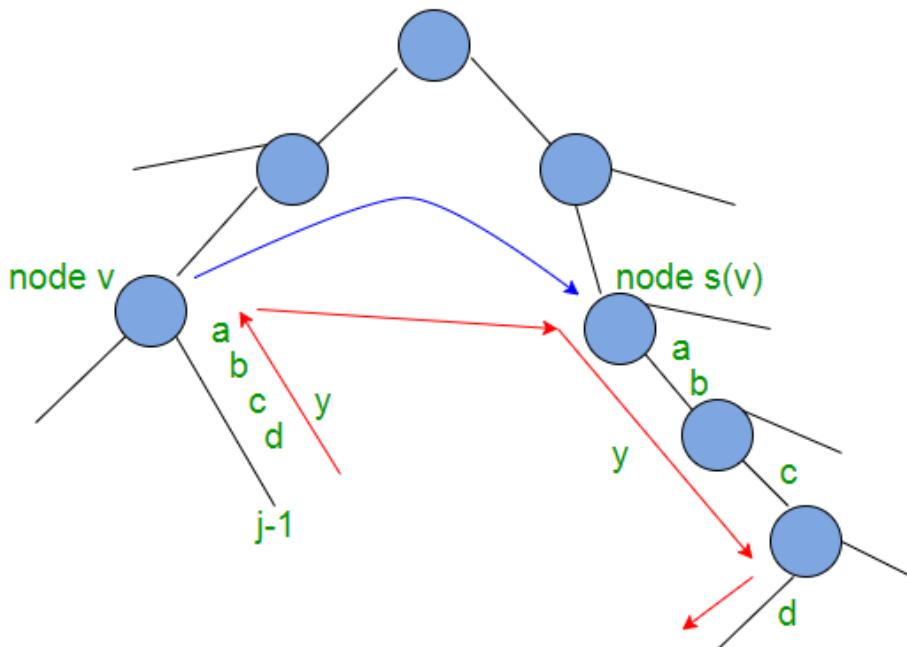


Figure 17 : Traversal from root to leaf in extension j of phase $i+1$, to find end of $S[j\dots i]$, when suffix link(blue arrow) is used.

So we can see that, to find end of path $S[j..i]$, we need not traverse from root. We can start from the end of path $S[j-1..i]$, walk up one edge to node v (i.e. go to parent node), follow the suffix link to $s(v)$, then walk down the path y (which is abcd here in Figure 17). This shows the use of suffix link is an improvement over the process.

Note: In the next part 3, we will introduce activePoint which will help to avoid “walk up”. We can directly go to node $s(v)$ from node v .

When there is a suffix link from node v to node $s(v)$, then if there is a path labelled with string y from node v to a leaf, then there must be a path labelled with string y from node $s(v)$ to a leaf. In Figure 17, there is a path label “abcd” from node v to a leaf, then there is a path with same label “abcd” from node $s(v)$ to a leaf.

This fact can be used to improve the walk from $s(v)$ to leaf along the path y . This is called “skip/count” trick.

Skip/Count Trick

When walking down from node $s(v)$ to leaf, instead of matching path character by character as we travel, we can directly skip to the next node if number of characters on the edge is less than the number of characters we need to travel. If number of characters on the edge is more than the number of characters we need to travel, we directly skip to the last character on that edge.

If implementation is such a way that number of characters on any edge, character at a given position in string S should be obtained in constant time, then skip/count trick will do the walk down in proportional to the number of nodes on it rather than the number of characters on it.

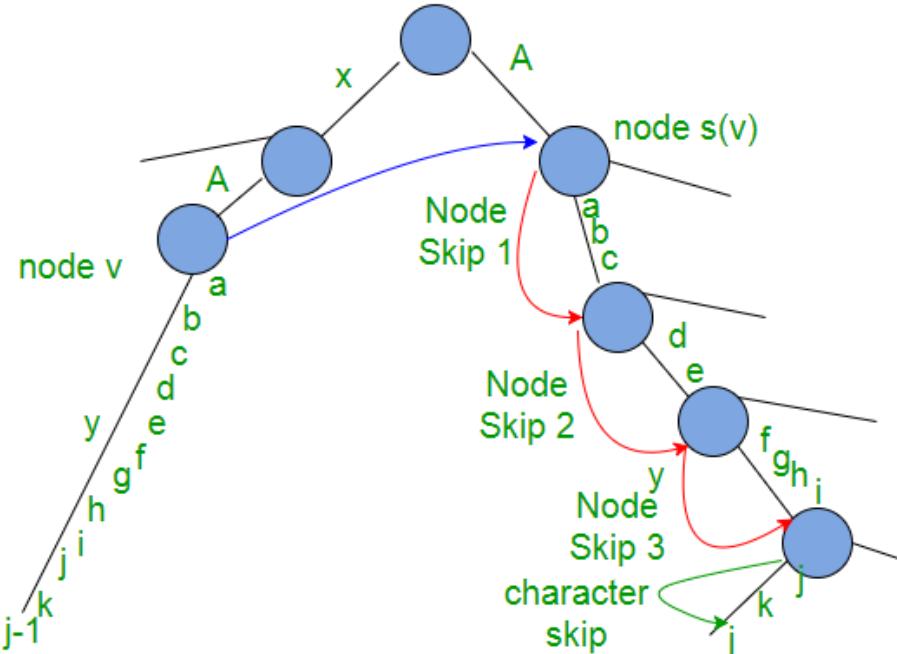


Figure 18 : skip/count trick : substring y from node v has length 11. Substring y from node $s(v)$ is two characters down the last node, after 3 node skips

Using suffix link along with skip/count trick, suffix tree can be built in $O(m^2)$ as there are

m phases and each phase takes $O(m)$.

Edge-label compression

So far, path labels are represented as characters in string. Such a suffix tree will take $O(m^2)$ space to store the path labels. To avoid this, we can use two pair of indices (start, end) on each edge for path labels, instead of substring itself. The indices start and end tells the path label start and end position in string S. With this, suffix tree needs $O(m)$ space.

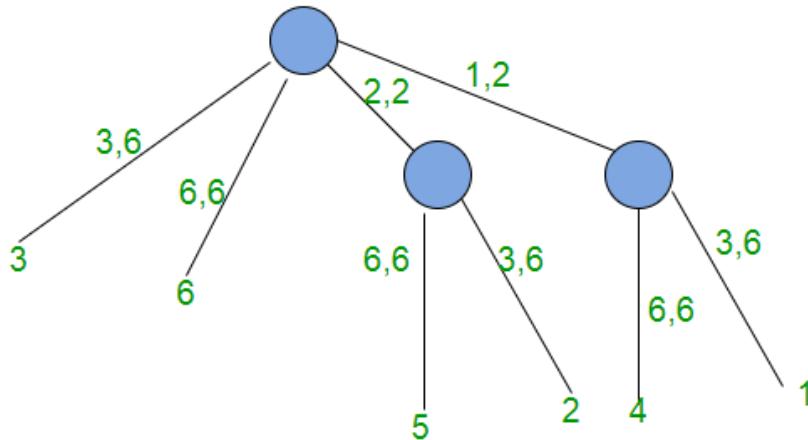


Figure 19 : Suffix tree for string xabxac with edge-label compression

Figure 14 shows same suffix tree without edge-label compression

There are two observations about the way extension rules interact in successive extensions and phases. These two observations lead to two more implementation tricks (first trick “skip/count” is seen already while walk down).

Observation 1: Rule 3 is show stopper

In a phase i, there are i extensions (1 to i) to be done.

When rule 3 applies in any extension j of phase i+1 (i.e. path labelled $S[j..i]$ continues with character $S[i+1]$), then it will also apply in all further extensions of same phase (i.e. extensions $j+1$ to $i+1$ in phase i+1). That’s because if path labelled $S[j..i]$ continues with character $S[i+1]$, then path labelled $S[j+1..i]$, $S[j+2..i]$, $S[j+3..i]$, ..., $S[i..i]$ will also continue with character $S[i+1]$.

Consider Figure 11, Figure12 and Figure 13 in [Part 1](#) where Rule 3 is applied.

In Figure 11, “xab” is added in tree and in Figure 12 (Phase 4), we add next character “x”. In this, 3 extensions are done (which adds 3 suffixes). Last suffix “x” is already present in tree.

In Figure 13, we add character “a” in tree (Phase 5). First 3 suffixes are added in tree and last two suffixes “xa” and “a” are already present in tree. This shows that if suffix $S[j..i]$ present in tree, then ALL the remaining suffixes $S[j+1..i]$, $S[j+2..i]$, $S[j+3..i]$, ..., $S[i..i]$ will also be there in tree and no work needed to add those remaining suffixes.

So no more work needed to be done in any phase as soon as rule 3 applies in any extension in that phase. If a new internal node v gets created in extension j and rule 3 applies in next extension $j+1$, then we need to add suffix link from node v to current node (if we are on internal node) or root node. ActiveNode, which will be discussed in part 3, will help while setting suffix links.

Trick 2

Stop the processing of any phase as soon as rule 3 applies. All further extensions are already present in tree implicitly.

Observation 2: Once a leaf, always a leaf

Once a leaf is created and labelled j (for suffix starting at position j in string S), then this leaf will always be a leaf in successive phases and extensions. Once a leaf is labelled as j , extension rule 1 will always apply to extension j in all successive phases.

Consider Figure 9 to Figure 14 in Part 1.

In Figure 10 (Phase 2), Rule 1 is applied on leaf labelled 1. After this, in all successive phases, rule 1 is always applied on this leaf.

In Figure 11 (Phase 3), Rule 1 is applied on leaf labelled 2. After this, in all successive phases, rule 1 is always applied on this leaf.

In Figure 12 (Phase 4), Rule 1 is applied on leaf labelled 3. After this, in all successive phases, rule 1 is always applied on this leaf.

In any phase i , there is an initial sequence of consecutive extensions where rule 1 or rule 2 are applied and then as soon as rule 3 is applied, phase i ends.

Also rule 2 creates a new leaf always (and internal node sometimes).

If J_i represents the last extension in phase i when rule 1 or 2 was applied (i.e after i^{th} phase, there will be J_i leaves labelled 1, 2, 3, ..., J_i), then $J_i \leq J_{i+1}$

J_i will be equal to J_{i+1} when there are no new leaf created in phase $i+1$ (i.e rule 3 is applied in J_{i+1} extension)

In Figure 11 (Phase 3), Rule 1 is applied in 1st two extensions and Rule 2 is applied in 3rd extension, so here $J_3 = 3$

In Figure 12 (Phase 4), no new leaf created (Rule 1 is applied in 1st 3 extensions and then rule 3 is applied in 4th extension which ends the phase). Here $J_4 = 3 = J_3$

In Figure 13 (Phase 5), no new leaf created (Rule 1 is applied in 1st 3 extensions and then rule 3 is applied in 4th extension which ends the phase). Here $J_5 = 3 = J_4$

J_i will be less than J_{i+1} when few new leaves are created in phase $i+1$.

In Figure 14 (Phase 6), new leaf created (Rule 1 is applied in 1st 3 extensions and then rule 2 is applied in last 3 extension which ends the phase). Here $J_6 = 6 > J_5$

So we can see that in phase $i+1$, only rule 1 will apply in extensions 1 to J_i (which really doesn't need much work, can be done in constant time and that's the trick 3), extension J_{i+1} onwards, rule 2 may apply to zero or more extensions and then finally rule 3, which ends the phase.

Now edge labels are represented using two indices (start, end), for any leaf edge, end will always be equal to phase number i.e. for phase i , end = i for leaf edges, for phase $i+1$, end = $i+1$ for leaf edges.

Trick 3

In any phase i , leaf edges may look like (p, i) , (q, i) , (r, i) , ... where p, q, r are starting position of different edges and i is end position of all. Then in phase $i+1$, these leaf edges will look like $(p, i+1)$, $(q, i+1)$, $(r, i+1)$, ... This way, in each phase, end position has to be incremented in all leaf edges. For this, we need to traverse through all leaf edges and increment end position for them. To do same thing in constant time, maintain a global index e and e will be equal to phase number. So now leaf edges will look like (p, e) , (q, e) , (r, e) , ... In any phase, just increment e and extension on all leaf edges will be done. Figure 19 shows this.

So using suffix links and tricks 1, 2 and 3, a suffix tree can be built in linear time.

Tree T_m could be implicit tree if a suffix is prefix of another. So we can add a \$ terminal symbol first and then run algorithm to get a true suffix tree (A true suffix tree contains all suffixes explicitly). To label each leaf with corresponding suffix starting position (all leaves are labelled as global index e), a linear time traversal can be done on tree.

At this point, we have gone through most of the things we needed to know to create suffix tree using Ukkonen's algorithm. In next [Part 3](#), we will take string $S = "abcabxabcd"$ as an example and go through all the things step by step and create the tree. While building the tree, we will discuss few more implementation issues which will be addressed by ActivePoints. We will continue to discuss the algorithm in [Part 4](#) and [Part 5](#). Code implementation will be discussed in [Part 6](#).

References:

<http://web.stanford.edu/~mjkay/gusfield.pdf>

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

Source

<https://www.geeksforgeeks.org/ukkonens-suffix-tree-construction-part-2/>

Chapter 209

Ukkonen's Suffix Tree Construction – Part 3

Ukkonen's Suffix Tree Construction - Part 3 - GeeksforGeeks

This article is continuation of following two articles:

[Ukkonen's Suffix Tree Construction – Part 1](#)

[Ukkonen's Suffix Tree Construction – Part 2](#)

Please go through [Part 1](#) and [Part 2](#), before looking at current article, where we have seen few basics on suffix tree, high level ukkonen's algorithm, suffix link and three implementation tricks.

Here we will take string $S = \text{"abcabxabcd"}$ as an example and go through all the things step by step and create the tree.

We will add \$ (discussed in [Part 1](#) why we do this) so string S would be "abcabxabcd\$".

While building suffix tree for string S of length m :

- There will be m phases 1 to m (one phase for each character)
In our current example, m is 11, so there will be 11 phases.
- First phase will add first character 'a' in the tree, second phase will add second character 'b' in tree, third phase will add third character 'c' in tree, , m^{th} phase will add m^{th} character in tree (This makes Ukkonen's algorithm an online algorithm)
- Each phase i will go through at-most i extensions (from 1 to i). If current character being added in tree is not seen so far, all i extensions will be completed (Extension Rule 3 will not apply in this phase). If current character being added in tree is seen before, then phase i will complete early (as soon as Extension Rule 3 applies) without going through all i extensions
- There are three extension rules (1, 2 and 3) and each extension j (from 1 to i) of any phase i will adhere to one of these three rules.
- Rule 1 adds a new character on existing leaf edge

- Rule 2 creates a new leaf edge (And may also create new internal node, if the path label ends in between an edge)
- Rule 3 ends the current phase (when current character is found in current edge being traversed)
- Phase 1 will read first character from the string, will go through 1 extension.
(In figures, we are showing characters on edge labels just for explanation, while writing code, we will only use start and end indices – The Edge-label compression discussed in Part 2)
Extension 1 will add suffix “a” in tree. We start from root and traverse path with label ‘a’. There is no path from root, going out with label ‘a’, so create a leaf edge (Rule 2).

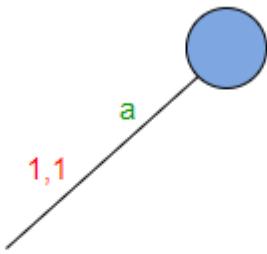


Figure 20 : Phase 1, extension 1-Rule2 applied. Created a leaf edge(1,1) Phase 1 completes here

Phase 1 completes with the completion of extension 1 (As a phase i has at most i extensions)

For any string, Phase 1 will have only one extension and it will always follow Rule 2.

- Phase 2 will read second character, will go through at least 1 and at most 2 extensions. In our example, phase 2 will read second character ‘b’. Suffixes to be added are “ab” and “b”.
Extension 1 adds suffix “ab” in tree.
Path for label ‘a’ ends at leaf edge, so add ‘b’ at the end of this edge.
Extension 1 just increments the end index by 1 (from 1 to 2) on first edge (Rule 1).

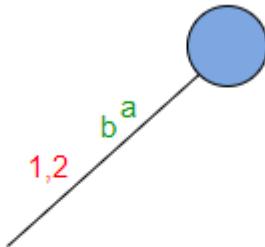
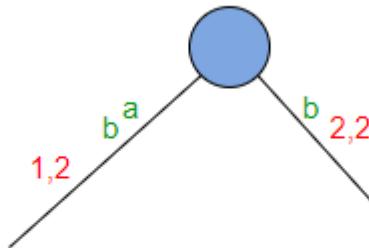


Figure 21 : Phase 2, Extension1-Rule1 applied extended the leaf edge from (1,1) to (1,2)

Extension 2 adds suffix “b” in tree. There is no path from root, going out with label ‘b’, so creates a leaf edge (Rule 2).



**Figure 22 : Phase 2, Extension2-Rule2 applied
Created a leaf edge(2,2)
Phase 2 completes here**

Phase 2 completes with the completion of extension 2.

Phase 2 went through two extensions here. Rule 1 applied in 1st Extension and Rule 2 applied in 2nd Extension.

- Phase 3 will read third character, will go through at least 1 and at most 3 extensions. In our example, phase 3 will read third character ‘c’. Suffixes to be added are “abc”, “bc” and “c”.

Extension 1 adds suffix “abc” in tree.

Path for label ‘ab’ ends at leaf edge, so add ‘c’ at the end of this edge.

Extension 1 just increments the end index by 1 (from 2 to 3) on this edge (Rule 1).

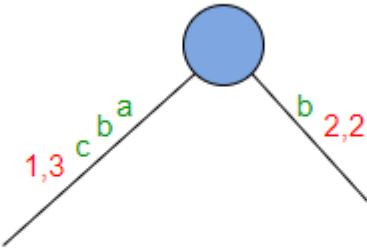


Figure 23 : Phase 3, Extension1-Rule1 applied
Extended the leaf edge from (1,2) to (1,3)

Extension 2 adds suffix “bc” in tree.

Path for label ‘b’ ends at leaf edge, so add ‘c’ at the end of this edge.

Extension 2 just increments the end index by 1 (from 2 to 3) on this edge (Rule 1).

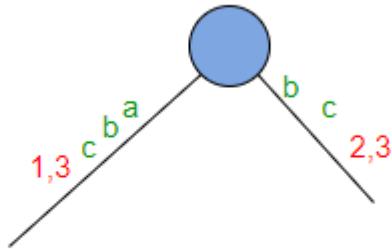
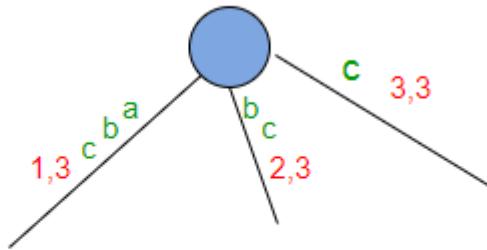


Figure 24 : Phase 3, Extension2-Rule1 applied
Extended the leaf edge from (2,2) to (2,3)

Extension 3 adds suffix “c” in tree. There is no path from root, going out with label ‘c’, so creates a leaf edge (Rule 2).

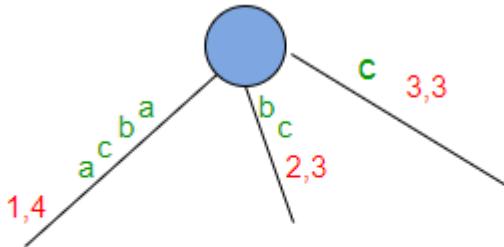


**Figure 25 : Phase 3, Extension3-Rule2 applied
Created a leaf edge(3,3)
Phase 3 completes here**

Phase 3 completes with the completion of extension 3.

Phase 3 went through three extensions here. Rule 1 applied in first two Extensions and Rule 2 applied in 3rd Extension.

- Phase 4 will read fourth character, will go to at least 1 and at most 4 extensions. In our example, phase 4 will read fourth character ‘a’. Suffixes to be added are “abca”, “bca”, “ca” and “a”. Extension 1 adds suffix “abca” in tree. Path for label ‘abc’ ends at leaf edge, so add ‘a’ at the end of this edge. Extension 1 just increments the end index by 1 (from 3 to 4) on this edge (Rule 1).

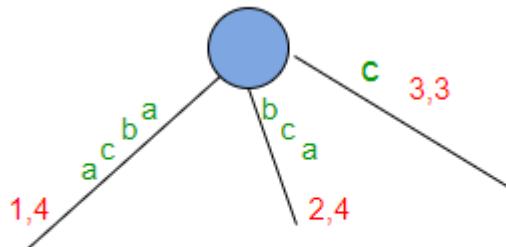


**Figure 26 : Phase 4, Extension 1
Rule 1 applied**

Extension 2 adds suffix “bca” in tree.

Path for label ‘bc’ ends at leaf edge, so add ‘a’ at the end of this edge.

Extension 2 just increments the end index by 1 (from 3 to 4) on this edge (Rule 1).

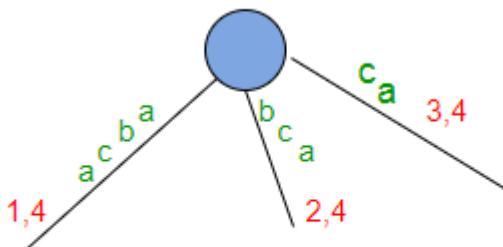


**Figure 27 : Phase 4, Extension 2
Rule 1 applied**

Extension 3 adds suffix “ca” in tree.

Path for label ‘c’ ends at leaf edge, so add ‘a’ at the end of this edge.

Extension 3 just increments the end index by 1 (from 3 to 4) on this edge (Rule 1).



**Figure 28 : Phase 4, Extension 3
Rule 1 applied**

Extension 4 adds suffix “a” in tree.

Path for label ‘a’ exists in the tree. No more work needed and Phase 4 ends here (Rule 3 and Trick 2). This is an example of implicit suffix tree. Here suffix “a” is not seen explicitly (because it doesn’t end at a leaf edge) but it is in the tree implicitly. So there is no change in tree structure after extension 4. It will remain as above in Figure 28.

Phase 4 completes as soon as Rule 3 is applied while Extension 4.

Phase 4 went through four extensions here. Rule 1 applied in first three Extensions and Rule 3 applied in 4th Extension.

Now we will see few observations and how to implement those.

- At the end of any phase i , there are at most i leaf edges (if i^{th} character is not seen so far, there will be i leaf edges, else there will be less than i leaf edges).
e.g. After phases 1, 2 and 3 in our example, there are 1, 2 and 3 leaf edges respectively, but after phase 4, there are 3 leaf edges only (not 4).

2. After completing phase i , “end” indices of all leaf edges are i . How do we implement this in code? Do we need to iterate through all those extensions, find leaf edges by traversing from root to leaf and increment the “end” index? Answer is “NO”.

For this, we will maintain a global variable (say “END”) and we will just increment this global variable “END” and all leaf edge end indices will point to this global variable. So this way, if we have j leaf edges after phase i , then in phase $i+1$, first j extensions (1 to j) will be done by just incrementing variable “END” by 1 (END will be $i+1$ at the point).

Here we just implemented the trick 3 – **Once a leaf, always a leaf**. This trick processes all the j leaf edges (i.e. extension 1 to j) using rule 1 in a constant time in any phase. Rule 1 will not apply to subsequent extensions in the same phase. This can be verified in the four phases we discussed above. If at all Rule 1 applies in any phase, it only applies in initial few phases continuously (say 1 to j). Rule 1 never applies later in a given phase once Rule 2 or Rule 3 is applied in that phase.

3. In the example explained so far, in each extension (where trick 3 is not applied) of any phase to add a suffix in tree, we are traversing from root by matching path labels against the suffix being added. If there are j leaf edges after phase i , then in phase $i+1$, first j extensions will follow Rule 1 and will be done in constant time using trick 3. There are $i+1-j$ extensions yet to be performed. For these extensions, which node (root or some other internal node) to start from and which path to go? Answer to this depends on how previous phase i is completed.

If previous phase i went through all the i extensions (when i^{th} character is unique so far), then in next phase $i+1$, trick 3 will take care of first i suffixes (the i leaf edges) and then extension $i+1$ will start from root node and it will insert just one character $[(i+1)^{\text{th}}]$ suffix in tree by creating a leaf edge using Rule 2.

If previous phase i completes early (and this will happen if and only if rule 3 applies – when i^{th} character is already seen before), say at j^{th} extension (i.e. rule 3 is applied at j^{th} extension), then there are $j-1$ leaf edges so far.

We will state few more facts (which may be a repeat, but we want to make sure it’s clear to you at this point) here based on discussion so far:

- *Phase 1 starts with Rule 2, all other phases start with Rule 1*
- *Any phase ends with either Rule 2 or Rule 3*
- *Any phase i may go through a series of j extensions ($1 \leq j \leq i$). In these j extensions, first p ($0 \leq p < i$) extensions will follow Rule 1, next q ($0 \leq q \leq i-p$) extensions will follow Rule 2 and next r ($0 \leq r \leq 1$) extensions will follow Rule 3. The order in which Rule 1, Rule 2 and Rule 3 apply, is never intermixed in a phase. They apply in order of their number (if at all applied), i.e. in a phase, Rule 1 applies 1st, then Rule 2 and then Rule 3*
- *In a phase i , $p + q + r \leq i$*
- *At the end of any phase i , there will be $p+q$ leaf edges and next phase $i+1$ will go through Rule 1 for first $p+q$ extensions*

In the next phase $i+1$, trick 3 (Rule 1) will take care of first $j-1$ suffixes (the $j-1$ leaf edges), then extension j will start where we will add j^{th} suffix in tree. For this, we need to find the best possible matching edge and then add new character at the end of that edge. How to find the end of best matching edge? Do we need to traverse

from root node and match tree edges against the j^{th} suffix being added character by character? This will take time and overall algorithm will not be linear. activePoint comes to the rescue here.

In previous phase i , while j^{th} extension, path traversal ended at a point (which could be an internal node or some point in the middle of an edge) where i^{th} character being added was found in tree already and Rule 3 applied, j^{th} extension of phase $i+1$ will start exactly from the same point and we start matching path against $(i+1)^{\text{th}}$ character. activePoint helps to avoid unnecessary path traversal from root in any extension based on the knowledge gained in traversals done in previous extension. There is no traversal needed in 1st p extensions where Rule 1 is applied. Traversal is done where Rule 2 or Rule 3 gets applied and that's where activePoint tells the starting point for traversal where we match the path against the current character being added in tree. Implementation is done in such a way that, in any extension where we need a traversal, activePoint is set to right location already (with one exception case **APCFALZ** discussed below) and at the end of current extension, we reset activePoint as appropriate so that next extension (of same phase or next phase) where a traversal is required, activePoint points to the right place already.

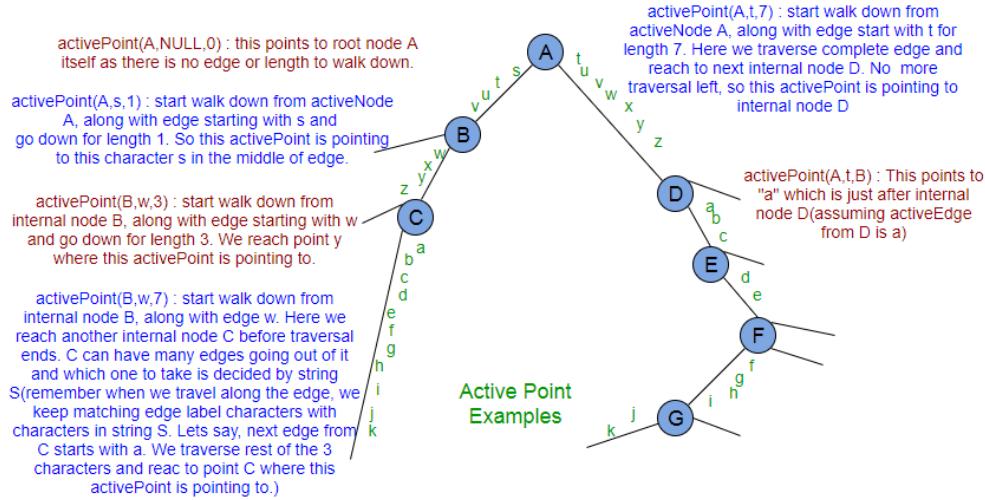
activePoint: This could be root node, any internal node or any point in the middle of an edge. This is the point where traversal starts in any extension. For the 1st extension of phase 1, activePoint is set to root. Other extension will get activePoint set correctly by previous extension (with one exception case **APCFALZ** discussed below) and it is the responsibility of current extension to reset activePoint appropriately at the end, to be used in next extension where Rule 2 or Rule 3 is applied (of same or next phase).

To accomplish this, we need a way to store activePoint. We will store this using three variables: **activeNode**, **activeEdge**, **activeLength**.

activeNode: This could be root node or an internal node.

activeEdge: When we are on root node or internal node and we need to walk down, we need to know which edge to choose. activeEdge will store that information. In case, activeNode itself is the point from where traversal starts, then activeEdge will be set to next character being processed in next phase.

activeLength: This tells how many characters we need to walk down (on the path represented by activeEdge) from activeNode to reach the activePoint where traversal starts. In case, activeNode itself is the point from where traversal starts, then activeLength will be ZERO.



After phase i, if there are j leaf edges then in phase $i+1$, first j extensions will be done by trick 3. activePoint will be needed for the extensions from $j+1$ to $i+1$ and activePoint may or may not change between two extensions depending on the point where previous extension ends.

activePoint change for extension rule 3 (APCFER3): When rule 3 applies in any phase i , then before we move on to next phase $i+1$, we increment activeLength by 1. There is no change in activeNode and activeEdge. Why? Because in case of rule 3, the current character from string S is matched on the same path represented by current activePoint, so for next activePoint, activeNode and activeEdge remain the same, only activeLength is increased by 1 (because of matched character in current phase). This new activePoint (same node, same edge and incremented length) will be used in phase $i+1$.

activePoint change for walk down (APCFWD): activePoint may change at the end of an extension based on extension rule applied. activePoint may also change during the extension when we do walk down. Let's consider an activePoint is $(A, s, 11)$ in the above activePoint example figure. If this is the activePoint at the start of some extension, then while walk down from activeNode A, other internal nodes will be seen. Anytime if we encounter an internal node while walk down, that node will become activeNode (it will change activeEdge and activeLength as appropriate so that new activePoint represents the same point as earlier). In this walk down, below is the sequence of changes in activePoint:

$(A, s, 11) \rightarrow (B, w, 7) \rightarrow (C, a, 3)$

All above three activePoints refer to same point 'c'

Let's take another example.

If activePoint is $(D, a, 11)$ at the start of an extension, then while walk down, below is the sequence of changes in activePoint:

$(D, a, 10) \rightarrow (E, d, 7) \rightarrow (F, f, 5) \rightarrow (G, j, 1)$

All above activePoints refer to same point 'k'.

If activePoints are $(A, s, 3)$, $(A, t, 5)$, $(B, w, 1)$, $(D, a, 2)$ etc when no internal node comes in the way while walk down, then there will be no change in activePoint for APCFWD.

The idea is that, at any time, the closest internal node from the point, where we want to reach, should be the activePoint. Why? This will minimize the length of traversal in the next extension.

activePoint change for Active Length ZERO (APCFALZ): Let's consider an activePoint (A, s, 0) in the above activePoint example figure. And let's say current character being processed from string S is 'x' (or any other character). At the start of extension, when activeLength is ZERO, activeEdge is set to the current character being processed, i.e. 'x', because there is no walk down needed here (as activeLength is ZERO) and so next character we look for is current character being processed.

4. While code implementation, we will loop through all the characters of string S one by one. Each loop for i^{th} character will do processing for phase i . Loop will run one or more time depending on how many extensions are left to be performed (Please note that in a phase $i+1$, we don't really have to perform all $i+1$ extensions explicitly, as trick 3 will take care of j extensions for all j leaf edges coming from previous phase i). We will use a variable **remainingSuffixCount**, to track how many extensions are yet to be performed explicitly in any phase (after trick 3 is performed). Also, at the end of any phase, if remainingSuffixCount is ZERO, this tells that all suffixes supposed to be added in tree, are added explicitly and present in tree. If remainingSuffixCount is non-zero at the end of any phase, that tells that suffixes of that many count are not added in tree explicitly (because of rule 3, we stopped early), but they are in tree implicitly though (Such trees are called implicit suffix tree). These implicit suffixes will be added explicitly in subsequent phases when a unique character comes in the way.

We will continue our discussion in [Part 4](#) and [Part 5](#). Code implementation will be discussed in [Part 6](#).

References:

<http://web.stanford.edu/~mjkay/gusfield.pdf>
[Ukkonen's suffix tree algorithm in plain English](#)

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

Source

<https://www.geeksforgeeks.org/ukkonens-suffix-tree-construction-part-3/>

Chapter 210

Ukkonen's Suffix Tree Construction – Part 4

Ukkonen's Suffix Tree Construction - Part 4 - GeeksforGeeks

This article is continuation of following three articles:

[Ukkonen's Suffix Tree Construction – Part 1](#)

[Ukkonen's Suffix Tree Construction – Part 2](#)

[Ukkonen's Suffix Tree Construction – Part 3](#)

Please go through [Part 1](#), [Part 2](#) and [Part 3](#), before looking at current article, where we have seen few basics on suffix tree, high level ukkonen's algorithm, suffix link and three implementation tricks and some details on activePoint along with an example string “abcabxabcd” where we went through four phases of building suffix tree.

Let's revisit those four phases we have seen already in [Part 3](#), in terms of trick 2, trick 3 and activePoint.

- activePoint is initialized to (root, NULL, 0), i.e. activeNode is root, activeEdge is NULL (for easy understanding, we are giving character value to activeEdge, but in code implementation, it will be index of the character) and activeLength is ZERO.
- The global variable END and remainingSuffixCount are initialized to ZERO

*****Phase 1*****

In Phase 1, we read 1st character (a) from string S

- Set END to 1
- Increment remainingSuffixCount by 1 (remainingSuffixCount will be 1 here, i.e. there is 1 extension left to be performed)
- Run a loop remainingSuffixCount times (i.e. one time) as below:

- If activeLength is ZERO, set activeEdge to the current character (here activeEdge will be ‘a’). This is **APCFALZ**.
- Check if there is an edge going out from activeNode (which is root in this phase 1) for the activeEdge. If not, create a leaf edge. If present, walk down. In our example, leaf edge gets created (Rule 2).
- Once extension is performed, decrement the remainingSuffixCount by 1
- At this point, activePoint is (root, a, 0)

At the end of phase 1, remainingSuffixCount is ZERO (All suffixes are added explicitly). Figure 20 in [Part 3](#) is the resulting tree after phase 1.

*******Phase 2*******

In Phase 2, we read 2nd character (b) from string S

Set END to 2 (This will do extension 1)

Increment remainingSuffixCount by 1 (remainingSuffixCount will be 1 here, i.e. there is 1 extension left to be performed)

Run a loop remainingSuffixCount times (i.e. one time) as below:

- If activeLength is ZERO, set activeEdge to the current character (here activeEdge will be ‘b’). This is **APCFALZ**.
- Check if there is an edge going out from activeNode (which is root in this phase 2) for the activeEdge. If not, create a leaf edge. If present, walk down. In our example, leaf edge gets created.
- Once extension is performed, decrement the remainingSuffixCount by 1
- At this point, activePoint is (root, b, 0)

At the end of phase 2, remainingSuffixCount is ZERO (All suffixes are added explicitly). Figure 22 in [Part 3](#) is the resulting tree after phase 2.

*******Phase 3*******

In Phase 3, we read 3rd character (c) from string S

Set END to 3 (This will do extensions 1 and 2)

Increment remainingSuffixCount by 1 (remainingSuffixCount will be 1 here, i.e. there is 1 extension left to be performed)

Run a loop remainingSuffixCount times (i.e. one time) as below:

- If activeLength is ZERO, set activeEdge to the current character (here activeEdge will be ‘c’). This is **APCFALZ**.
- Check if there is an edge going out from activeNode (which is root in this phase 3) for the activeEdge. If not, create a leaf edge. If present, walk down. In our example, leaf edge gets created.

- Once extension is performed, decrement the remainingSuffixCount by 1
- At this point, activePoint is (root, c, 0)

At the end of phase 3, remainingSuffixCount is ZERO (All suffixes are added explicitly). Figure 25 in [Part 3](#) is the resulting tree after phase 3.

*****Phase 4*****

In Phase 4, we read 4th character (a) from string S

Set END to 4 (This will do extensions 1, 2 and 3)

Increment remainingSuffixCount by 1 (remainingSuffixCount will be 1 here, i.e. there is 1 extension left to be performed)

Run a loop remainingSuffixCount times (i.e. one time) as below:

- If activeLength is ZERO, set activeEdge to the current character (here activeEdge will be 'a'). This is **APCFALZ**.
- Check if there is an edge going out from activeNode (which is root in this phase 3) for the activeEdge. If not, create a leaf edge. If present, walk down (The trick 1 – skip/count). In our example, edge 'a' is present going out of activeNode (i.e. root). No walk down needed as activeLength < edgeLength. We increment activeLength from zero to 1 (**APCFER3**) and stop any further processing (Rule 3).
- At this point, activePoint is (root, a, 1) and remainingSuffixCount remains set to 1 (no change there)

At the end of phase 4, remainingSuffixCount is 1 (One suffix 'a', the last one, is not added explicitly in tree, but it is there in tree implicitly).

Figure 28 in [Part 3](#) is the resulting tree after phase 4.

Revisiting completed for 1st four phases, we will continue building the tree and see how it goes.

*****Phase 5*****

In phase 5, we read 5th character (b) from string S

Set END to 5 (This will do extensions 1, 2 and 3). See Figure 29 shown below.

Increment remainingSuffixCount by 1 (remainingSuffixCount will be 2 here, i.e. there are 2 extension left to be performed, which are extensions 4 and 5. Extension 4 is supposed to add suffix "ab" and extension 5 is supposed to add suffix "b" in tree)

Run a loop remainingSuffixCount times (i.e. two times) as below:

- Check if there is an edge going out from activeNode (which is root in this phase 3) for the activeEdge. If not, create a leaf edge. If present, walk down. In our example, edge 'a' is present going out of activeNode (i.e. root).
- Do a walk down (The trick 1 – skip/count) if necessary. In current phase 5, no walk down needed as activeLength < edgeLength. Here activePoint is (root, a, 1) for extension 4 (remainingSuffixCount = 2)

- Check if current character of string S (which is ‘b’) is already present after the activePoint. If yes, no more processing (rule 3). Same is the case in our example, so we increment activeLength from 1 to 2 (**APCFER3**) and we stop here (Rule 3).
- At this point, activePoint is (root, a, 2) and remainingSuffixCount remains set to 2 (no change in remainingSuffixCount)

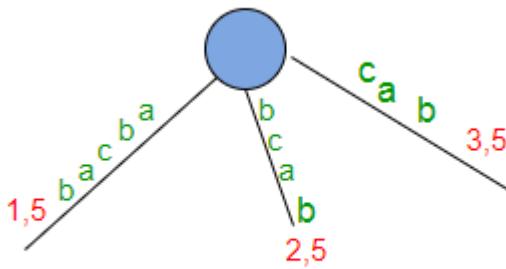


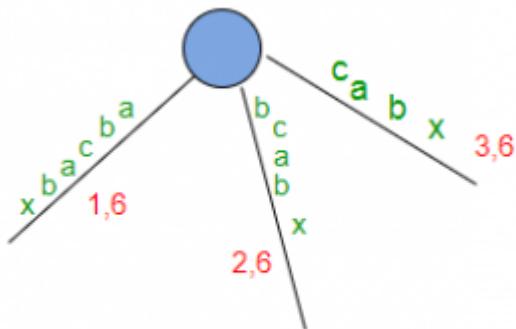
Figure 29 : Phase 5

At the end of phase 5, remainingSuffixCount is 2 (Two suffixes, ‘ab’ and ‘b’, the last two, are not added explicitly in tree, but they are in tree implicitly).

*****Phase 6*****

In phase 6, we read 6th character (x) from string S

Set END to 6 (This will do extensions 1, 2 and 3)



Phase 6, Extension 3

Increment remainingSuffixCount by 1 (remainingSuffixCount will be 3 here, i.e. there are 3 extension left to be performed, which are extensions 4, 5 and 6 for suffixes “abx”, “bx” and “x” respectively)

Run a loop remainingSuffixCount times (i.e. three times) as below:

- While extension 4, the activePoint is (root, a, 2) which points to ‘b’ on edge starting with ‘a’.
- In extension 4, current character ‘x’ from string S doesn’t match with the next character on the edge after activePoint, so this is the case of extension rule 2. So a leaf edge is created here with edge label x. Also here traversal ends in middle of an edge, so a new internal node also gets created at the end of activePoint.
- Decrement the remainingSuffixCount by 1 (from 3 to 2) as suffix “abx” added in tree.

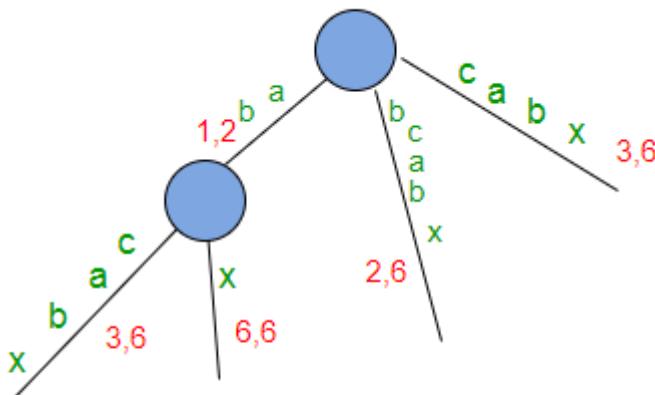


Figure 30 : Phase 6, Extension 4
Rule 2 applied- Created a leaf edge and
also a new internal node

Now activePoint will change after applying rule 2. Three other cases, (**APCFER3**, **APCFWD** and **APCFALZ**) where activePoint changes, are already discussed in [Part 3](#).

activePoint change for extension rule 2 (**APCFER2**):

Case 1 (APCFER2C1): If activeNode is root and activeLength is greater than ZERO, then decrement the activeLength by 1 and activeEdge will be set “ $S[i - \text{remainingSuffixCount} + 1]$ ” where i is current phase number. Can you see why this change in activePoint? Look at current extension we just discussed above for phase 6 (i=6) again where we added suffix “abx”. There activeLength is 2 and activeEdge is ‘a’. Now in next extension, we need to add suffix “bx” in the tree, i.e. path label in next extension should start with ‘b’. So ‘b’ (the 5th character in string S) should be active edge for next extension and index of b will be “ $i - \text{remainingSuffixCount} + 1$ ” ($6 - 2 + 1 = 5$). activeLength is decremented by 1 because activePoint gets closer to root by length 1 after every extension.

What will happen If activeNode is root and activeLength is ZERO? This case is already taken care by **APCFALZ**.

Case 2 (APCFER2C2): If activeNode is not root, then follow the suffix link from current activeNode. The new node (which can be root node or another internal node) pointed by suffix link will be the activeNode for next extension. No change in activeLength and

activeEdge. Can you see why this change in activePoint? This is because: If two nodes are connected by a suffix link, then labels on all paths going down from those two nodes, starting with same character, will be exactly same and so for two corresponding similar point on those paths, activeEdge and activeLength will be same and the two nodes will be the activeNode. Look at Figure 18 in [Part 2](#). Let's say in phase i and extension j, suffix 'xAabcdedg' was added in tree. At that point, let's say activePoint was (Node-V, a, 7), i.e. point 'g'. So for next extension $j+1$, we would add suffix 'Aabcdefg' and for that we need to traverse 2nd path shown in Figure 18. This can be done by following suffix link from current activeNode v. Suffix link takes us to the path to be traversed somewhere in between [Node s(v)] below which the path is exactly same as how it was below the previous activeNode v. As said earlier, "activePoint gets closer to root by length 1 after every extension", this reduction in length will happen above the node s(v) but below s(v), no change at all. So when activeNode is not root in current extension, then for next extension, only activeNode changes (No change in activeEdge and activeLength).

- At this point in extension 4, current activePoint is (root, a, 2) and based on **APCFER2C1**, new activePoint for next extension 5 will be (root, b, 1)
- Next suffix to be added is 'bx' (with remainingSuffixCount 2).
- Current character 'x' from string S doesn't match with the next character on the edge after activePoint, so this is the case of extension rule 2. So a leaf edge is created here with edge label x. Also here traversal ends in middle of an edge, so a new internal node also gets created at the end of activePoint.
Suffix link is also created from previous internal node (of extension 4) to the new internal node created in current extension 5.
- Decrement the remainingSuffixCount by 1 (from 2 to 1) as suffix "bx" added in tree.

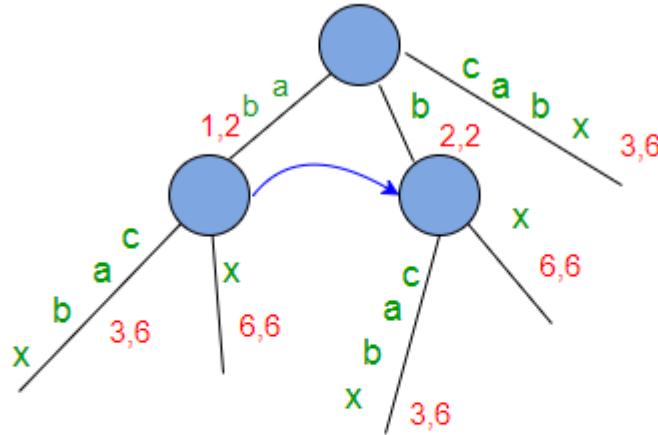


Figure 31 : Phase 6, Extension 5-Rule2 applied
Created a leaf edge, a new internal node and suffix link
from previous internal node of extension 4 to the current
newly internal node

- At this point in extension 5, current activePoint is (root, b, 1) and based on **APCFER2C1** new activePoint for next extension 6 will be (root, x, 0)
- Next suffix to be added is 'x' (with remainingSuffixCount 1).
- In the next extension 6, character x will not match to any existing edge from root, so a new edge with label x will be created from root node. Also suffix link from previous extension's internal node goes to root (as no new internal node created in current extension 6).
- Decrement the remainingSuffixCount by 1 (from 1 to 0) as suffix "x" added in tree

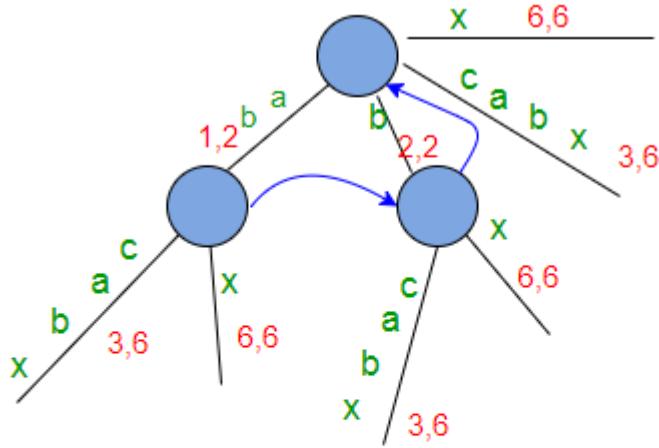


Figure 32 : Phase 6, Extension 6-Rule2 applied.
Created a leaf edge and suffix link from previous internal node of extension 5 to root node(as no new internal node created in extension 6, so suffix link goes to roots)

This completes the phase 6.

Note that phase 6 has completed all its 6 extensions (Why? Because the current character c was not seen in string so far, so rule 3, which stops further extensions never got chance to get applied in phase 6) and so the tree generated after phase 6 is a true suffix tree (i.e. not an implicit tree) for the characters ‘abcabx’ read so far and it has all suffixes explicitly in the tree.

While building the tree above, following facts were noticed:

- A newly created internal node in extension i, points to another internal node or root (if activeNode is root in extension i+1) by the end of extension i+1 via suffix link (Every internal node MUST have a suffix link pointing to another internal node or root)
- Suffix link provides short cut while searching path label end of next suffix
- With proper tracking of activePoints between extensions/phases, unnecessary walk-down from root can be avoided.

We will go through rest of the phases (7 to 11) in [Part 5](#) and build the tree completely and after that, we will see the code for the algorithm in [Part 6](#).

References:

<http://web.stanford.edu/~mjkay/gusfield.pdf>
[Ukkonen's suffix tree algorithm in plain English](#)

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

Source

<https://www.geeksforgeeks.org/ukkonens-suffix-tree-construction-part-4/>

Chapter 211

Ukkonen's Suffix Tree Construction – Part 5

Ukkonen's Suffix Tree Construction - Part 5 - GeeksforGeeks

This article is continuation of following four articles:

[Ukkonen's Suffix Tree Construction – Part 1](#)
[Ukkonen's Suffix Tree Construction – Part 2](#)
[Ukkonen's Suffix Tree Construction – Part 3](#)
[Ukkonen's Suffix Tree Construction – Part 4](#)

Please go through [Part 1](#), [Part 2](#), [Part 3](#) and [Part 4](#), before looking at current article, where we have seen few basics on suffix tree, high level ukkonen's algorithm, suffix link and three implementation tricks and some details on activePoint along with an example string “abcabxabcd” where we went through six phases of building suffix tree.

Here, we will go through rest of the phases (7 to 11) and build the tree completely.

*****Phase 7*****

In phase 7, we read 7th character (a) from string S

- Set END to 7 (This will do extensions 1, 2, 3, 4, 5 and 6) – because we have 6 leaf edges so far by the end of previous phase 6.

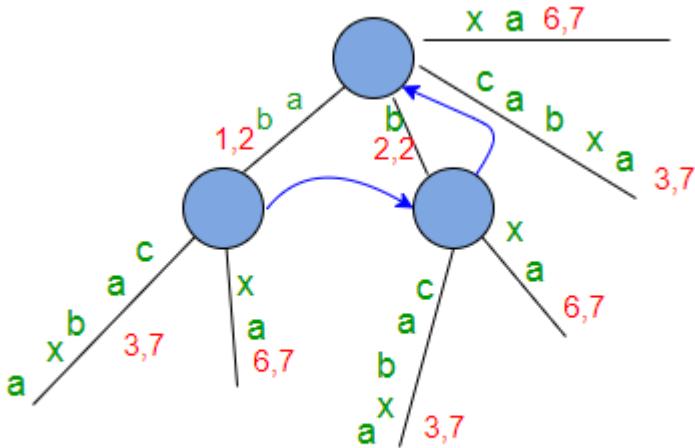


Figure 33 : Phase 7, Extension 6-Rule 1 applied

Increment remainingSuffixCount by 1 (remainingSuffixCount will be 1 here, i.e. there is only 1 extension left to be performed, which is extensions 7 for suffix ‘a’)

Run a loop remainingSuffixCount times (i.e. one time) as below:

- If activeLength is ZERO [activePoint in previous phase was (root, x, 0)], set activeEdge to the current character (here activeEdge will be ‘a’). This is **APCFALZ**. Now activePoint becomes (root, ‘a’, 0).
- Check if there is an edge going out from activeNode (which is root in this phase 7) for the activeEdge. If not, create a leaf edge. If present, walk down. In our example, edge ‘a’ is present going out of activeNode (i.e. root), here we increment activeLength from zero to 1 (**APCFER3**) and stop any further processing.
- At this point, activePoint is (root, a, 1) and remainingSuffixCount remains set to 1 (no change there)

At the end of phase 7, remainingSuffixCount is 1 (One suffix ‘a’, the last one, is not added explicitly in tree, but it is there in tree implicitly).

Above Figure 33 is the resulting tree after phase 7.

*****Phase 8*****

In phase 8, we read 8th character (b) from string S

- Set END to 8 (This will do extensions 1, 2, 3, 4, 5 and 6) – because we have 6 leaf edges so far by the end of previous phase 7 (Figure 34).

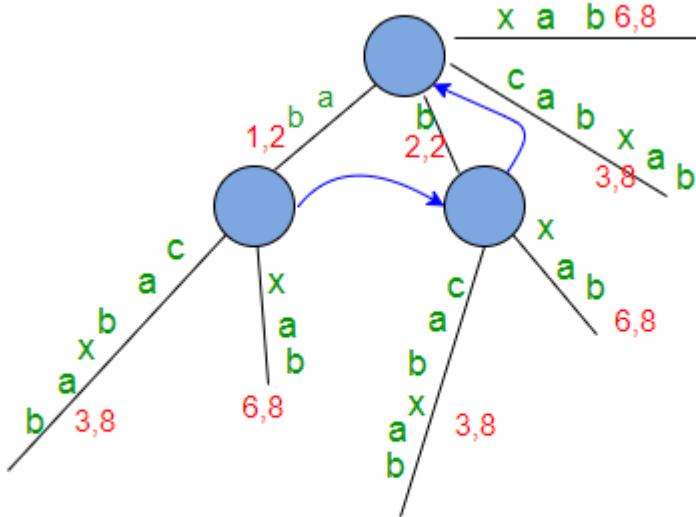


Figure 34 : Phase 8, Extension 6-Rule 1 applied

Increment remainingSuffixCount by 1 (remainingSuffixCount will be 2 here, i.e. there are two extensions left to be performed, which are extensions 7 and 8 for suffixes ‘ab’ and ‘b’ respectively)

Run a loop remainingSuffixCount times (i.e. two times) as below:

- Check if there is an edge going out from activeNode (which is root in this phase 8) for the activeEdge. If not, create a leaf edge. If present, walk down. In our example, edge ‘a’ is present going out of activeNode (i.e. root).
- Do a walk down (The trick 1 – skip/count) if necessary. In current phase 8, no walk down needed as activeLength < edgeLength. Here activePoint is (root, a, 1) for extension 7 (remainingSuffixCount = 2)
- Check if current character of string S (which is ‘b’) is already present after the activePoint. If yes, no more processing (rule 3). Same is the case in our example, so we increment activeLength from 1 to 2 (**APCFER3**) and we stop here (Rule 3).
- At this point, activePoint is (root, a, 2) and remainingSuffixCount remains set to 2 (no change in remainingSuffixCount)

At the end of phase 8, remainingSuffixCount is 2 (Two suffixes, ‘ab’ and ‘b’, the last two, are not added explicitly in tree explicitly, but they are in tree implicitly).

*****Phase 9*****

In phase 9, we read 9th character (c) from string S

- Set END to 9 (This will do extensions 1, 2, 3, 4, 5 and 6) – because we have 6 leaf edges so far by the end of previous phase 8.

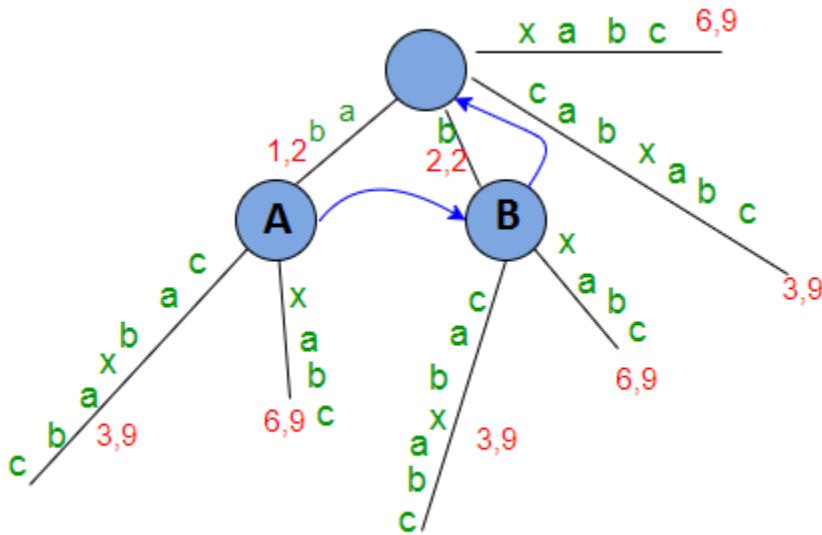


Figure 35 : Phase 9, Extension 6-Rule1 applied

Increment remainingSuffixCount by 1 (remainingSuffixCount will be 3 here, i.e. there are three extensions left to be performed, which are extensions 7, 8 and 9 for suffixes 'abc', 'bc' and 'c' respectively)

Run a loop remainingSuffixCount times (i.e. three times) as below:

- Check if there is an edge going out from activeNode (which is root in this phase 9) for the activeEdge. If not, create a leaf edge. If present, walk down. In our example, edge 'a' is present going out of activeNode (i.e. root).
- Do a walk down (The trick 1 – skip/count) if necessary. In current phase 9, walk down needed as activeLength(2) \geq edgeLength(2). While walk down, activePoint changes to (Node A, c, 0) based on **APCFWD** (This is first time **APCFWD** is being applied in our example).
- Check if current character of string S (which is 'c') is already present after the activePoint. If yes, no more processing (rule 3). Same is the case in our example, so we increment activeLength from 0 to 1 (**APCFER3**) and we stop here (Rule 3).
- At this point, activePoint is (Node A, c, 1) and remainingSuffixCount remains set to 3 (no change in remainingSuffixCount)

At the end of phase 9, remainingSuffixCount is 3 (Three suffixes, 'abc', 'bc' and 'c', the last three, are not added explicitly in tree explicitly, but they are in tree implicitly).

*****Phase 10*****

In phase 10, we read 10th character (d) from string S

- Set END to 10 (This will do extensions 1, 2, 3, 4, 5 and 6) – because we have 6 leaf edges so far by the end of previous phase 9.

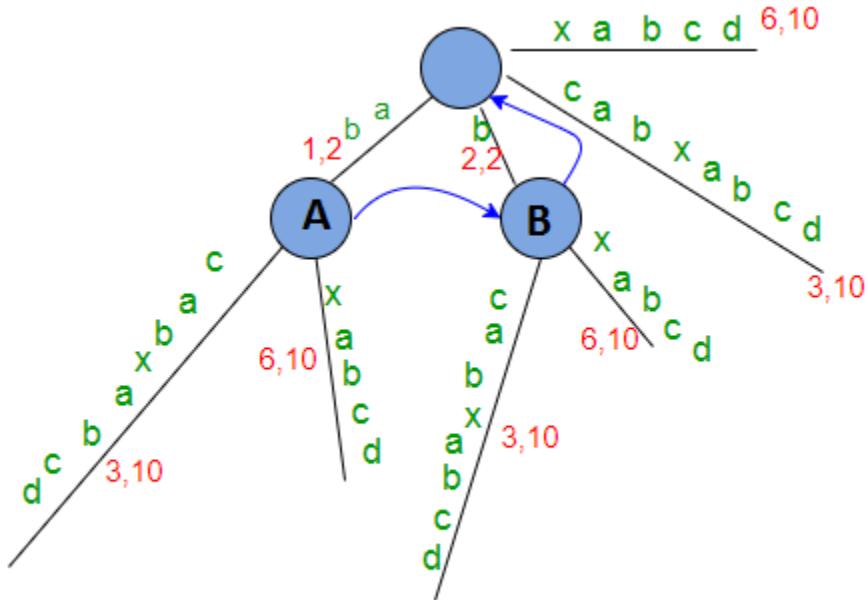
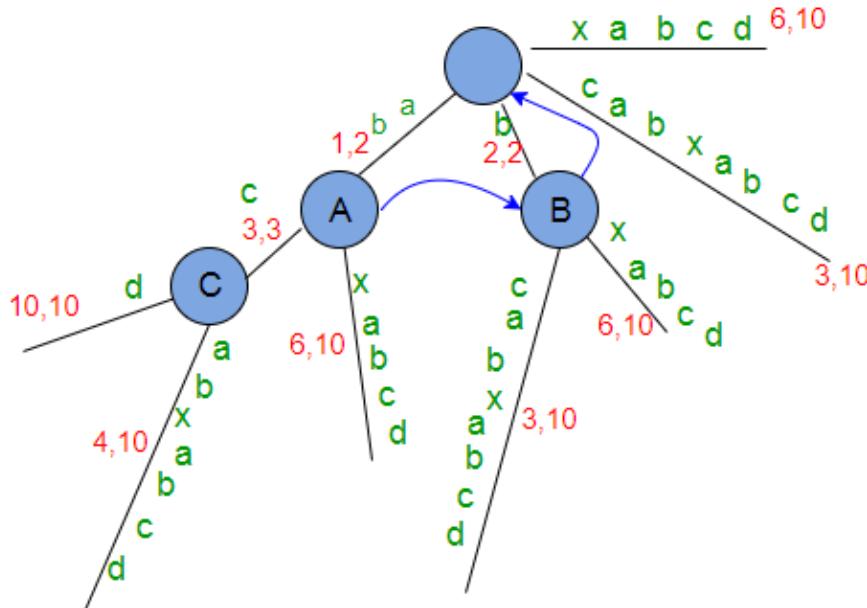


Figure 36 : Phase 10, Extension 6-Rule1 applied

Increment remainingSuffixCount by 1 (remainingSuffixCount will be 4 here, i.e. there are four extensions left to be performed, which are extensions 7, 8, 9 and 10 for suffixes ‘abcd’, ‘bcd’, ‘cd’ and ‘d’ respectively)

Run a loop remainingSuffixCount times (i.e. four times) as below:

- Check if there is an edge going out from activeNode (Node A) for the activeEdge(c). If not, create a leaf edge. If present, walk down. In our example, edge ‘c’ is present going out of activeNode (Node A).
- Do a walk down (The trick 1 – skip/count) if necessary. In current Extension 7, no walk down needed as activeLength < edgeLength.
- Check if current character of string S (which is ‘d’) is already present after the activePoint. If not, rule 2 will apply. In our example, there is no path starting with ‘d’ going out of activePoint, so we create a leaf edge with label ‘d’. Since activePoint ends in the middle of an edge, we will create a new internal node just after the activePoint (Rule 2)



**Figure 37 : Phase 10, Extension 7-Rule2 applied.
New leaf edge and new internal node created**

- Decrement the remainingSuffixCount by 1 (from 4 to 3) as suffix “abcd” added in tree.
- Now activePoint will change for next extension 8. Current activeNode is an internal node (Node A), so there must be a suffix link from there and we will follow that to get new activeNode and that’s going to be ‘Node B’. There is no change in activeEdge and activeLength (This is **APCFER2C2**). So new activePoint is (Node B, c, 1).
- Now in extension 8 (here we will add suffix ‘bcd’), while adding character ‘d’ after the current activePoint, exactly same logic will apply as previous extension 7. In previous extension 7, we added character ‘d’ at activePoint (Node A, c, 1) and in current extension 8, we are going to add same character ‘d’ at activePoint (Node B, c, 1). So logic will be same and here we a new leaf edge with label ‘d’ and a new internal node will be created. And the new internal node (C) of previous extension will point to the new node (D) of current extension via suffix link.

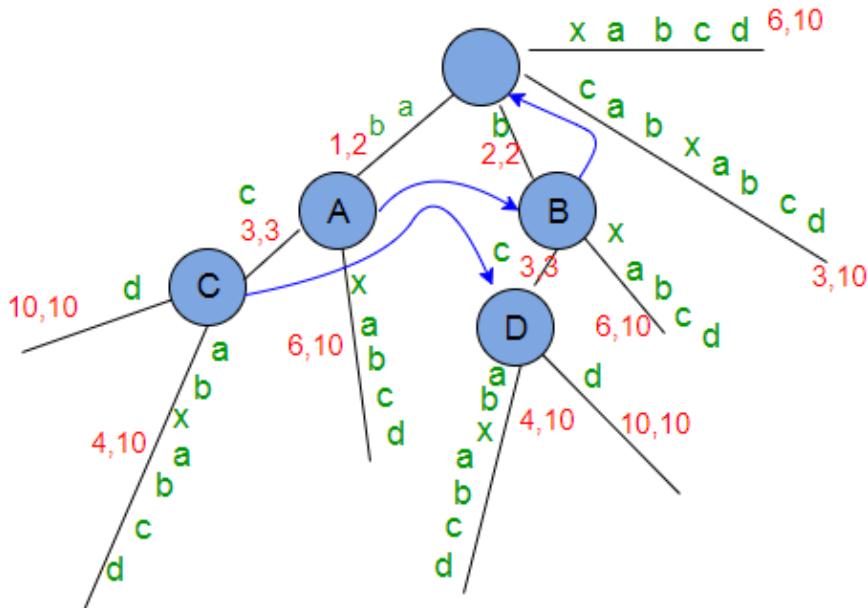


Figure 38 : Phase 10, Extension 8-Rule2 applied.
New leaf edge created and a new internal node created.
**Also the internal node C created in previous extension 7, pointing
 to the internal node D via suffix link**

- Decrement the remainingSuffixCount by 1 (from 3 to 2) as suffix “bcd” added in tree.
- Now activePoint will change for next extension 9. Current activeNode is an internal node (Node B), so there must be a suffix link from there and we will follow that to get new activeNode and that is ‘Root Node’. There is no change in activeEdge and activeLength (This is **APCFER2C2**). So new activePoint is (root, c, 1).
- Now in extension 9 (here we will add suffix ‘cd’), while adding character ‘d’ after the current activePoint, exactly same logic will apply as previous extensions 7 and 8. Note that internal node D created in previous extension 8, now points to internal node E (created in current extension) via suffix link.

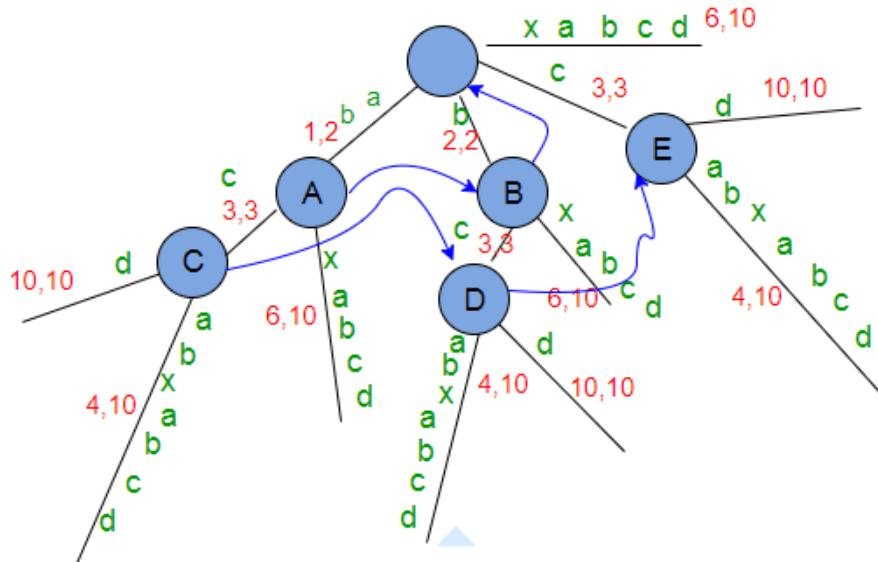


Figure 39 : Phase 10, Extension 9-Rule 2 applied.
 New leaf edge created and a new internal node created.
 Also the internal node D created in previous extension 8, pointing to the
 internal node E via suffix link

- Decrement the remainingSuffixCount by 1 (from 2 to 1) as suffix “cd” added in tree.
- Now activePoint will change for next extension 10. Current activeNode is root and activeLength is 1, based on **APCFER2C1**, activeNode will remain ‘root’, activeLength will be decremented by 1 (from 1 to ZERO) and activeEdge will be ‘d’. So new activePoint is (root, d, 0).
- Now in extension 10 (here we will add suffix ‘d’), while adding character ‘d’ after the current activePoint, there is no edge starting with d going out of activeNode root, so a new leaf edge with label d is created (Rule 2). Note that internal node E created in previous extension 9, now points to root node via suffix link (as no new internal node created in this extension).

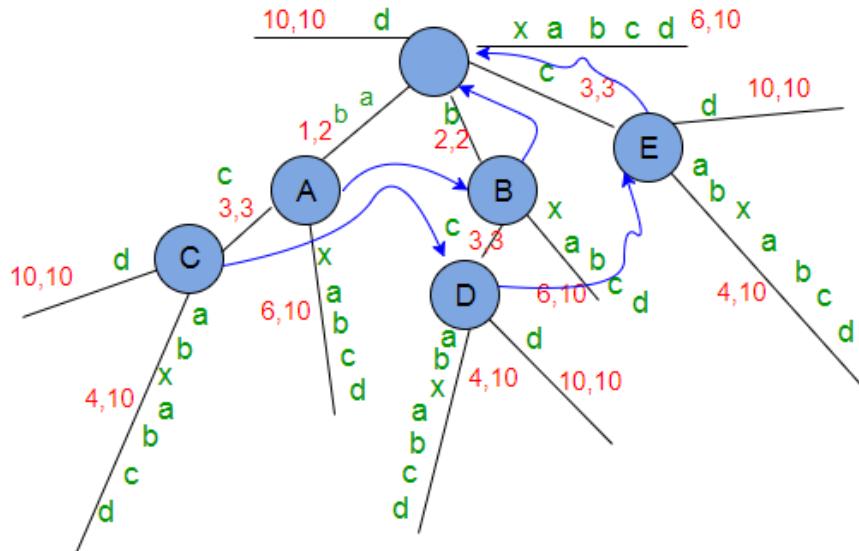


Figure 40 : Phase 10, Extension 10-Rule 2 applied. New leaf edge created.
Also the internal node E created in previous extension 9, pointing to root
node via suffix link

- Decrement the remainingSuffixCount by 1 (from 1 to 0) as suffix “d” added in tree. That means no more suffix is there to add and so the phase 10 ends here. Note that this tree is an explicit tree as all suffixes are added in tree explicitly (Why ?? because character d was not seen before in string S so far)
- activePoint for next phase 11 is (root, d, 0).

We see following facts in Phase 10:

- Internal Nodes connected via suffix links have exactly same tree below them, e.g. In above Figure 40, A and B have same tree below them, similarly C, D and E have same tree below them.
- Due to above fact, in any extension, when current activeNode is derived via suffix link from previous extension’s activeNode, then exactly same extension logic apply in current extension as previous extension. (In Phase 10, same extension logic is applied in extensions 7, 8 and 9)
- If a new internal node gets created in extension j of any phase i, then this newly created internal node will get it’s suffix link set by the end of next extension j+1 of same phase i. e.g. node C got created in extension 7 of phase 10 (Figure 37) and it got its suffix link set to node D in extension 8 of same phase 10 (Figure 38). Similarly node D got created in extension 8 of phase 10 (Figure 38) and it got its suffix link set to node E in extension 9 of same phase 10 (Figure 39). Similarly node E got created in extension 9 of phase 10 (Figure 39) and it got its suffix link set to root in extension 10 of same phase 10 (Figure 40).

- Based on above fact, every internal node will have a suffix link to some other internal node or root. Root is not an internal node and it will not have suffix link.

*****Phase 11*****

In phase 11, we read 11th character (\$) from string S

- Set END to 11 (This will do extensions 1 to 10) – because we have 10 leaf edges so far by the end of previous phase 10.

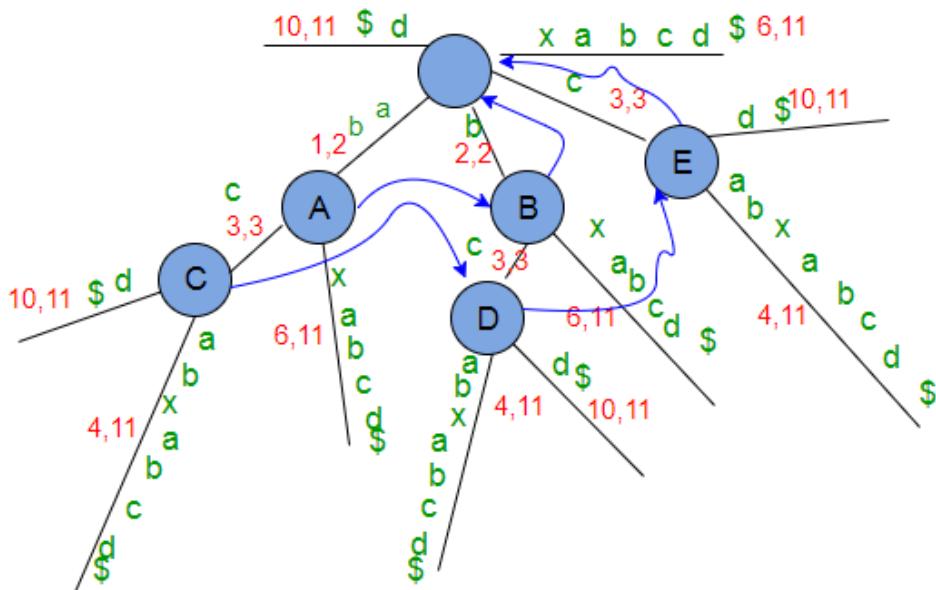


Figure 41 : Phase 11, Extension 10- Rule 1 applied

- Increment remainingSuffixCount by 1 (from 0 to 1), i.e. there is only one suffix ‘\$’ to be added in tree.
- Since activeLength is ZERO, activeEdge will change to current character ‘\$’ of string S being processed (**APCFALZ**).
- There is no edge going out from activeNode root, so a leaf edge with label ‘\$’ will be created (Rule 2).

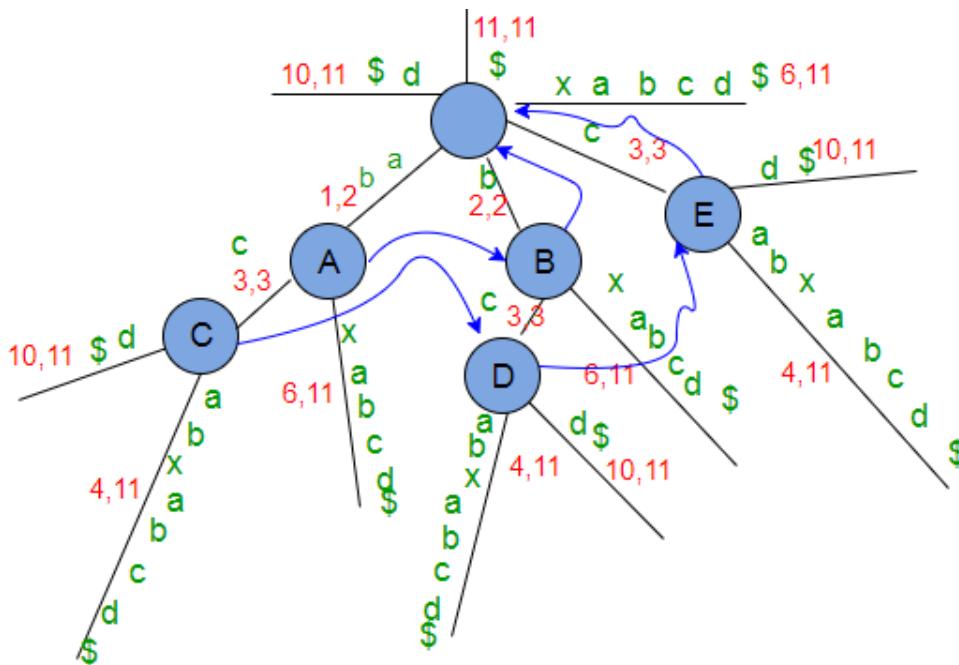


Figure 42 : Phase 11, Extension 11- Rule 2 applied

- Decrement the remainingSuffixCount by 1 (from 1 to 0) as suffix “\$” added in tree. That means no more suffix is there to add and so the phase 11 ends here. Note that this tree is an explicit tree as all suffixes are added in tree explicitly (Why ?? because character \$ was not seen before in string S so far)

Now we have added all suffixes of string ‘abcabxabcd\$’ in suffix tree. There are 11 leaf ends in this tree and labels on the path from root to leaf end represents one suffix. Now the only one thing left is to assign a number (suffix index) to each leaf end and that number would be the suffix starting position in the string S. This can be done by a DFS traversal on tree. While DFS traversal, keep track of label length and when a leaf end is found, set the suffix index as “stringSize – labelSize + 1”. Indexed suffix tree will look like below:

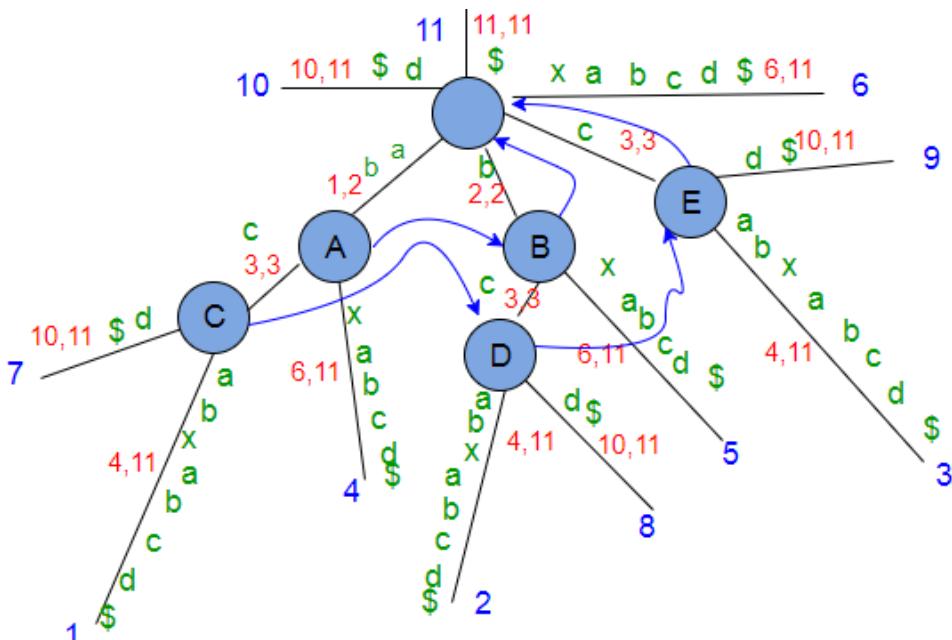


Figure 43 : Final suffix tree for String $S=abcabxabcd\$$

In above Figure, suffix indices are shown as character position starting with 1 (It's not zero indexed). In code implementation, suffix index will be set as zero indexed, i.e. where we see suffix index j (1 to m for string of length m) in above figure, in code implementation, it will be $j-1$ (0 to $m-1$)

And we are done !!!!

Data Structure to represent suffix tree

How to represent the suffix tree?? There are nodes, edges, labels and suffix links and indices.

Below are some of the operations/query we will be doing while building suffix tree and later on while using the suffix tree in different applications/usages:

- What length of path label on some edge?
- What is the path label on some edge?
- Check if there is an outgoing edge for a given character from a node.
- What is the character value on an edge at some given distance from a node?
- Where an internal node is pointing via suffix link?
- What is the suffix index on a path from root to leaf?
- Check if a given string present in suffix tree (as substring, suffix or prefix)?

We may think of different data structures which can fulfil these requirements.

In the next Part 6, we will discuss the data structure we will use in our code implementation and the code as well.

References:

<http://web.stanford.edu/~mjkay/gusfield.pdf>
Ukkonen's suffix tree algorithm in plain English

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

Source

<https://www.geeksforgeeks.org/ukkonens-suffix-tree-construction-part-5/>

Chapter 212

Ukkonen's Suffix Tree Construction – Part 6

Ukkonen's Suffix Tree Construction - Part 6 - GeeksforGeeks

This article is continuation of following five articles:

[Ukkonen's Suffix Tree Construction – Part 1](#)

[Ukkonen's Suffix Tree Construction – Part 2](#)

[Ukkonen's Suffix Tree Construction – Part 3](#)

[Ukkonen's Suffix Tree Construction – Part 4](#)

[Ukkonen's Suffix Tree Construction – Part 5](#)

Please go through [Part 1](#), [Part 2](#), [Part 3](#), [Part 4](#) and [Part 5](#), before looking at current article, where we have seen few basics on suffix tree, high level ukkonen's algorithm, suffix link and three implementation tricks and activePoints along with an example string “abcabxabcd” where we went through all phases of building suffix tree.

Here, we will see the data structure used to represent suffix tree and the code implementation.

At that end of [Part 5](#) article, we have discussed some of the operations we will be doing while building suffix tree and later when we use suffix tree in different applications.

There could be different possible data structures we may think of to fulfill the requirements where some data structure may be slow on some operations and some fast. Here we will use following in our implementation:

We will have SuffixTreeNode structure to represent each node in tree. SuffixTreeNode structure will have following members:

- **children** – This will be an array of alphabet size. This will store all the children nodes of current node on different edges starting with different characters.
- **suffixLink** – This will point to other node where current node should point via suffix link.

- **start, end** – These two will store the edge label details from parent node to current node. (start, end) interval specifies the edge, by which the node is connected to its parent node. Each edge will connect two nodes, one parent and one child, and (start, end) interval of a given edge will be stored in the child node. Lets say there are two nodes A (parent) and B (Child) connected by an edge with indices (5, 8) then this indices (5, 8) will be stored in node B.
- **suffixIndex** – This will be non-negative for leaves and will give index of suffix for the path from root to this leaf. For non-leaf node, it will be -1 .

This data structure will answer to the required queries quickly as below:

- How to check if a node is root ? — Root is a special node, with no parent and so it's start and end will be -1, for all other nodes, start and end indices will be non-negative.
- How to check if a node is internal or leaf node ? — suffixIndex will help here. It will be -1 for internal node and non-negative for leaf nodes.
- What is the length of path label on some edge? — Each edge will have start and end indices and length of path label will be end-start+1
- What is the path label on some edge ? — If string is S, then path label will be substring of S from start index to end index inclusive, [start, end].
- How to check if there is an outgoing edge for a given character c from a node A ? — If A->children[c] is not NULL, there is a path, if NULL, no path.
- What is the character value on an edge at some given distance d from a node A ? — Character at distance d from node A will be S[A->start + d], where S is the string.
- Where an internal node is pointing via suffix link ? — Node A will point to A->suffixLink
- What is the suffix index on a path from root to leaf ? — If leaf node is A on the path, then suffix index on that path will be A->suffixIndex

Following is C implementation of Ukkonen's Suffix Tree Construction. The code may look a bit lengthy, probably because of a good amount of comments.

```
// A C program to implement Ukkonen's Suffix Tree Construction
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define MAX_CHAR 256

struct SuffixTreeNode {
    struct SuffixTreeNode *children[MAX_CHAR];
    //pointer to other node via suffix link
    struct SuffixTreeNode *suffixLink;
}
```

```

/*
*(start, end) interval specifies the edge, by which the
node is connected to its parent node. Each edge will
connect two nodes, one parent and one child, and
(start, end) interval of a given edge will be stored
in the child node. Lets say there are two nodes A and B
connected by an edge with indices (5, 8) then this
indices (5, 8) will be stored in node B. */
int start;
int *end;

/*for leaf nodes, it stores the index of suffix for
the path from root to leaf*/
int suffixIndex;
};

typedef struct SuffixTreeNode Node;

char text[100]; //Input string
Node *root = NULL; //Pointer to root node

/*lastNewNode will point to newly created internal node,
waiting for it's suffix link to be set, which might get
a new suffix link (other than root) in next extension of
same phase. lastNewNode will be set to NULL when last
newly created internal node (if there is any) got it's
suffix link reset to new internal node created in next
extension of same phase.*/
Node *lastNewNode = NULL;
Node *activeNode = NULL;

/*activeEdge is represented as input string character
index (not the character itself)*/
int activeEdge = -1;
int activeLength = 0;

// remainingSuffixCount tells how many suffixes yet to
// be added in tree
int remainingSuffixCount = 0;
int leafEnd = -1;
int *rootEnd = NULL;
int *splitEnd = NULL;
int size = -1; //Length of input string

Node *newNode(int start, int *end)
{
    Node *node =(Node*) malloc(sizeof(Node));
    int i;

```

```

for (i = 0; i < MAX_CHAR; i++)
    node->children[i] = NULL;

/*For root node, suffixLink will be set to NULL
For internal nodes, suffixLink will be set to root
by default in current extension and may change in
next extension*/
node->suffixLink = root;
node->start = start;
node->end = end;

/*suffixIndex will be set to -1 by default and
actual suffix index will be set later for leaves
at the end of all phases*/
node->suffixIndex = -1;
return node;
}

int edgeLength(Node *n) {
    return *(n->end) - (n->start) + 1;
}

int walkDown(Node *currNode)
{
    /*activePoint change for walk down (APCFWD) using
    Skip/Count Trick (Trick 1). If activeLength is greater
    than current edge length, set next internal node as
    activeNode and adjust activeEdge and activeLength
    accordingly to represent same activePoint*/
    if (activeLength >= edgeLength(currNode))
    {
        activeEdge += edgeLength(currNode);
        activeLength -= edgeLength(currNode);
        activeNode = currNode;
        return 1;
    }
    return 0;
}

void extendSuffixTree(int pos)
{
    /*Extension Rule 1, this takes care of extending all
    leaves created so far in tree*/
    leafEnd = pos;

    /*Increment remainingSuffixCount indicating that a
    new suffix added to the list of suffixes yet to be
    added in tree*/
}

```

```

remainingSuffixCount++;

/*set lastNewNode to NULL while starting a new phase,
 indicating there is no internal node waiting for
 it's suffix link reset in current phase*/
lastNewNode = NULL;

//Add all suffixes (yet to be added) one by one in tree
while(remainingSuffixCount > 0) {

    if (activeLength == 0)
        activeEdge = pos; //APCFALZ

    // There is no outgoing edge starting with
    // activeEdge from activeNode
    if (activeNode->children] == NULL)
    {
        //Extension Rule 2 (A new leaf edge gets created)
        activeNode->children] =
            newNode(pos, &leafEnd);

        /*A new leaf edge is created in above line starting
         from an existing node (the current activeNode), and
         if there is any internal node waiting for it's suffix
         link get reset, point the suffix link from that last
         internal node to current activeNode. Then set lastNewNode
         to NULL indicating no more node waiting for suffix link
         reset.*/
        if (lastNewNode != NULL)
        {
            lastNewNode->suffixLink = activeNode;
            lastNewNode = NULL;
        }
    }
    // There is an outgoing edge starting with activeEdge
    // from activeNode
    else
    {
        // Get the next node at the end of edge starting
        // with activeEdge
        Node *next = activeNode->children];
        if (walkDown(next))//Do walkdown
        {
            //Start from next node (the new activeNode)
            continue;
        }
        /*Extension Rule 3 (current character being processed
         is already on the edge)*/
    }
}

```

```

if (text[next->start + activeLength] == text[pos])
{
    //If a newly created node waiting for it's
    //suffix link to be set, then set suffix link
    //of that waiting node to current active node
    if(lastNewNode != NULL && activeNode != root)
    {
        lastNewNode->suffixLink = activeNode;
        lastNewNode = NULL;
    }

    //APCFER3
    activeLength++;
    /*STOP all further processing in this phase
    and move on to next phase*/
    break;
}

/*We will be here when activePoint is in middle of
the edge being traversed and current character
being processed is not on the edge (we fall off
the tree). In this case, we add a new internal node
and a new leaf edge going out of that new node. This
is Extension Rule 2, where a new leaf edge and a new
internal node get created*/
splitEnd = (int*) malloc(sizeof(int));
*splitEnd = next->start + activeLength - 1;

//New internal node
Node *split = newNode(next->start, splitEnd);
activeNode->children] = split;

//New leaf coming out of new internal node
split->children] = newNode(pos, &leafEnd);
next->start += activeLength;
split->children] = next;

/*We got a new internal node here. If there is any
internal node created in last extensions of same
phase which is still waiting for it's suffix link
reset, do it now.*/
if (lastNewNode != NULL)
{
    /*suffixLink of lastNewNode points to current newly
    created internal node*/
    lastNewNode->suffixLink = split;
}

```

```

/*Make the current newly created internal node waiting
   for it's suffix link reset (which is pointing to root
   at present). If we come across any other internal node
   (existing or newly created) in next extension of same
   phase, when a new leaf edge gets added (i.e. when
   Extension Rule 2 applies is any of the next extension
   of same phase) at that point, suffixLink of this node
   will point to that internal node.*/
lastNewNode = split;
}

/* One suffix got added in tree, decrement the count of
   suffixes yet to be added.*/
remainingSuffixCount--;
if (activeNode == root && activeLength > 0) //APCFER2C1
{
    activeLength--;
    activeEdge = pos - remainingSuffixCount + 1;
}
else if (activeNode != root) //APCFER2C2
{
    activeNode = activeNode->suffixLink;
}
}

void print(int i, int j)
{
    int k;
    for (k=i; k<=j; k++)
        printf("%c", text[k]);
}

//Print the suffix tree as well along with setting suffix index
//So tree will be printed in DFS manner
//Each edge along with it's suffix index will be printed
void setSuffixIndexByDFS(Node *n, int labelHeight)
{
    if (n == NULL) return;

    if (n->start != -1) //A non-root node
    {
        //Print the label on edge from parent to current node
        print(n->start, *(n->end));
    }
    int leaf = 1;
    int i;
    for (i = 0; i < MAX_CHAR; i++)

```

```

{
    if (n->children[i] != NULL)
    {
        if (leaf == 1 && n->start != -1)
            printf(" [%d]\n", n->suffixIndex);

        //Current node is not a leaf as it has outgoing
        //edges from it.
        leaf = 0;
        setSuffixIndexByDFS(n->children[i], labelHeight +
                            edgeLength(n->children[i]));
    }
}
if (leaf == 1)
{
    n->suffixIndex = size - labelHeight;
    printf(" [%d]\n", n->suffixIndex);
}
}

void freeSuffixTreeByPostOrder(Node *n)
{
    if (n == NULL)
        return;
    int i;
    for (i = 0; i < MAX_CHAR; i++)
    {
        if (n->children[i] != NULL)
        {
            freeSuffixTreeByPostOrder(n->children[i]);
        }
    }
    if (n->suffixIndex == -1)
        free(n->end);
    free(n);
}

/*Build the suffix tree and print the edge labels along with
suffixIndex. suffixIndex for leaf edges will be >= 0 and
for non-leaf edges will be -1*/
void buildSuffixTree()
{
    size = strlen(text);
    int i;
    rootEnd = (int*) malloc(sizeof(int));
    *rootEnd = - 1;

    /*Root is a special node with start and end indices as -1,

```

```

        as it has no parent from where an edge comes to root*/
        root = newNode(-1, rootEnd);

        activeNode = root; //First activeNode will be root
        for (i=0; i<size; i++)
            extendSuffixTree(i);
        int labelHeight = 0;
        setSuffixIndexByDFS(root, labelHeight);

        //Free the dynamically allocated memory
        freeSuffixTreeByPostOrder(root);
    }

// driver program to test above functions
int main(int argc, char *argv[])
{
// strcpy(text, "abc"); buildSuffixTree();
// strcpy(text, "xabxac#"); buildSuffixTree();
// strcpy(text, "xabxa"); buildSuffixTree();
// strcpy(text, "xabxa$"); buildSuffixTree();
// strcpy(text, "abxabcd$"); buildSuffixTree();
// strcpy(text, "geeksforgeeks$"); buildSuffixTree();
// strcpy(text, "THIS IS A TEST TEXT$"); buildSuffixTree();
// strcpy(text, "AABAACAADAABAAABAA$"); buildSuffixTree();
    return 0;
}

```

Output (Each edge of Tree, along with suffix index of child node on edge, is printed in DFS order. To understand the output better, match it with the last figure no 43 in previous [Part 5](#) article):

```

$ [10]
ab [-1]
c [-1]
abxabcd$ [0]
d$ [6]
xabcd$ [3]
b [-1]
c [-1]
abxabcd$ [1]
d$ [7]
xabcd$ [4]
c [-1]
abxabcd$ [2]
d$ [8]
d$ [9]
xabcd$ [5]

```

Now we are able to build suffix tree in linear time, we can solve many string problem in efficient way:

- Check if a given pattern P is substring of text T (Useful when text is fixed and pattern changes, [KMP](#) otherwise)
- Find all occurrences of a given pattern P present in text T
- Find longest repeated substring
- [Linear Time Suffix Array Creation](#)

The above basic problems can be solved by DFS traversal on suffix tree.
We will soon post articles on above problems and others like below:

- Build [Generalized suffix tree](#)
- Linear Time [Longest common substring problem](#)
- Linear Time [Longest palindromic substring](#)

And [More](#).

Test your understanding?

1. Draw suffix tree (with proper suffix link, suffix indices) for string “AABAA-CAADAABAAABAA\$” on paper and see if that matches with code output.
2. Every extension must follow one of the three rules: Rule 1, Rule 2 and Rule 3.
Following are the rules applied on five consecutive extensions in some Phase i ($i > 5$), which ones are valid:
A) Rule 1, Rule 2, Rule 2, Rule 3, Rule 3
B) Rule 1, Rule 2, Rule 2, Rule 3, Rule 2
C) Rule 2, Rule 1, Rule 1, Rule 3, Rule 3
D) Rule 1, Rule 1, Rule 1, Rule 1, Rule 1
E) Rule 2, Rule 2, Rule 2, Rule 2, Rule 2
F) Rule 3, Rule 3, Rule 3, Rule 3
3. What are the valid sequences in above for Phase 5
4. Every internal node MUST have its suffix link set to another node (internal or root).
Can a newly created node point to already existing internal node or not ? Can it happen that a new node created in extension j, may not get its right suffix link in next extension $j+1$ and get the right one in later extensions like $j+2, j+3$ etc ?
5. Try solving the basic problems discussed above.

We have published following articles on suffix tree applications:

- [Suffix Tree Application 1 – Substring Check](#)

- Suffix Tree Application 2 – Searching All Patterns
- Suffix Tree Application 3 – Longest Repeated Substring
- Suffix Tree Application 4 – Build Linear Time Suffix Array
- Generalized Suffix Tree 1
- Suffix Tree Application 5 – Longest Common Substring
- Suffix Tree Application 6 – Longest Palindromic Substring

References:

<http://web.stanford.edu/~mjkay/gusfield.pdf>
Ukkonen's suffix tree algorithm in plain English

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

Source

<https://www.geeksforgeeks.org/ukkonens-suffix-tree-construction-part-6/>

Chapter 213

Union-Find Algorithm (Union By Rank and Find by Optimized Path Compression)

Union-Find Algorithm (Union By Rank and Find by Optimized Path Compression) - Geeks-forGeeks

Check whether a given graph contains a cycle or not.

Example 1:

Input:

Output: Graph contains Cycle.

Example 2:

Input:

Output: Graph does not contain Cycle.

Prerequisites: [Disjoint Set \(Or Union-Find\)](#), [Union By Rank and Path Compression](#)

We have already discussed [union-find to detect cycle](#). Here we discuss find by path compression, where it is slightly modified to work faster than the original method as we are skipping one level each time we are going up the graph. Implementation of find function is iterative, so there is no overhead involved. Time complexity of optimized find function is $O(\log^*(n))$, i.e iterated logarithm, which converges to $O(1)$ for repeated calls.

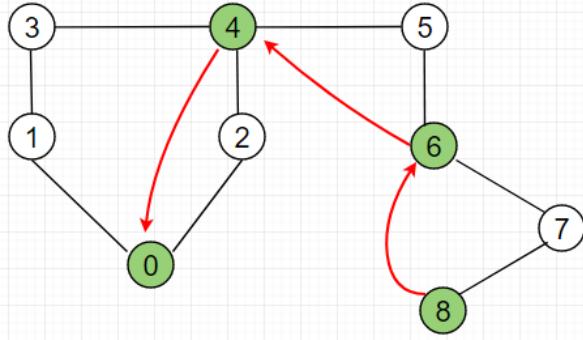
Refer this link for

[Proof of \$\log^*\(n\)\$ complexity of Union-Find](#)

Explanation of find function:

Take Example 1 to understand find function:

(1) call find(8) for **first time** and mappings will be done like this:



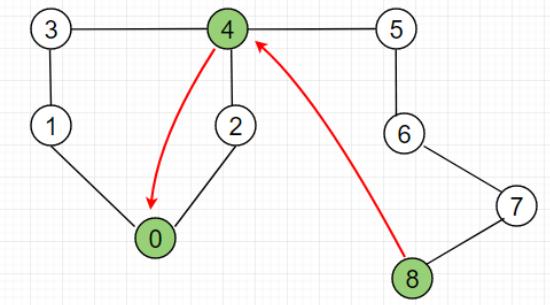
It took 3 mappings for find function to get the root of node 8. Mappings are illustrated below:

From node 8, skipped node 7, Reached node 6.

From node 6, skipped node 5, Reached node 4.

From node 4, skipped node 2, Reached node 0.

(2) call find(8) for **second time** and mappings will be done like this:

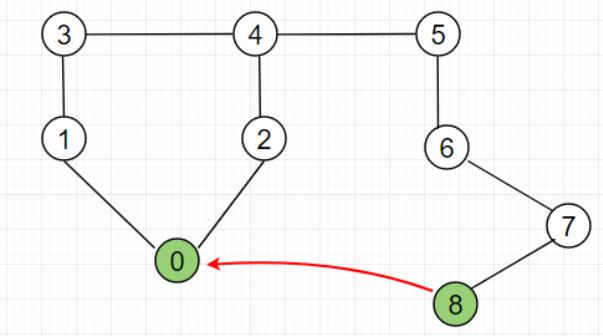


It took 2 mappings for find function to get the root of node 8. Mappings are illustrated below:

From node 8, skipped node 5, node 6 and node 7, Reached node 4.

From node 4, skipped node 2, Reached node 0.

(3) call find(8) for **third time** and mappings will be done like this:



Finally, we see it took only 1 mapping for find function to get the root of node 8. Mappings are illustrated below:

From node 8, skipped node 5, node 6, node 7, node 4, and node 2, Reached node 0.
That is how it converges path from certain mappings to single mapping.

Explanation of example 1:

Initially array size and Arr look like:

Arr[9] = {0, 1, 2, 3, 4, 5, 6, 7, 8}
size[9] = {1, 1, 1, 1, 1, 1, 1, 1, 1}

Consider the edges in the graph, and add them one by one to the disjoint-union set as follows:

Edge 1: 0-1

find(0)=>0, find(1)=>1, both have different root parent

Put these in single connected component as currently they doesn't belong to different connected components.

Arr[1]=0, size[0]=2;

Edge 2: 0-2

find(0)=>0, find(2)=>2, both have different root parent

Arr[2]=0, size[0]=3;

Edge 3: 1-3

find(1)=>0, find(3)=>3, both have different root parent

Arr[3]=0, size[0]=3;

Edge 4: 3-4

find(3)=>1, find(4)=>4, both have different root parent

Arr[4]=0, size[0]=4;

Edge 5: 2-4

find(2)=>0, find(4)=>0, both have same root parent

Hence, There is a cycle in graph.

We stop further checking for cycle in graph.

```
// CPP progxrm to implement Union-Find with union
// by rank and path compression.
#include <bits/stdc++.h>
using namespace std;

const int MAX_VERTEX = 101;

// Arr to represent parent of index i
int Arr[MAX_VERTEX];

// Size to represent the number of nodes
// in subgxrph rooted at index i
int size[MAX_VERTEX];

// set parent of every node to itself and
```

```
// size of node to one
void initialize(int n)
{
    for (int i = 0; i <= n; i++) {
        Arr[i] = i;
        size[i] = 1;
    }
}

// Each time we follow a path, find function
// compresses it further until the path length
// is greater than or equal to 1.
int find(int i)
{
    // while we reach a node whose parent is
    // equal to itself
    while (Arr[i] != i)
    {
        Arr[i] = Arr[Arr[i]]; // Skip one level
        i = Arr[i]; // Move to the new level
    }
    return i;
}

// A function that does union of two nodes x and y
// where xr is root node of x and yr is root node of y
void _union(int xr, int yr)
{
    if (size[xr] < size[yr]) // Make yr parent of xr
    {
        Arr[xr] = Arr[yr];
        size[yr] += size[xr];
    }
    else // Make xr parent of yr
    {
        Arr[yr] = Arr[xr];
        size[xr] += size[yr];
    }
}

// The main function to check whether a given
// gxrph contains cycle or not
int isCycle(vector<int> adj[], int V)
{
    // Iterte through all edges of gxrph, find
    // nodes connecting them.
    // If root nodes of both are same, then there is
    // cycle in gxrph.
```

```

for (int i = 0; i < V; i++) {
    for (int j = 0; j < adj[i].size(); j++) {
        int x = find(i); // find root of i
        int y = find(adj[i][j]); // find root of adj[i][j]

        if (x == y)
            return 1; // If same parent
        _union(x, y); // Make them connect
    }
}
return 0;
}

// Driver program to test above functions
int main()
{
    int V = 3;

    // Initialize the values for array Arr and Size
    initialize(V);

    /* Let us create following graph
       0
       |
       |
       |   \
       |   \
       1---2 */
}

vector<int> adj[V]; // Adjacency list for graph

adj[0].push_back(1);
adj[0].push_back(2);
adj[1].push_back(2);

// call is_cycle to check if it contains cycle
if (isCycle(adj, V))
    cout << "Graph contains Cycle.\n";
else
    cout << "Graph does not contain Cycle.\n";

return 0;
}

```

Output:

Graph contains Cycle.

Time Complexity(Find) : O(log*(n))

Time Complexity(union) : O(1)

Source

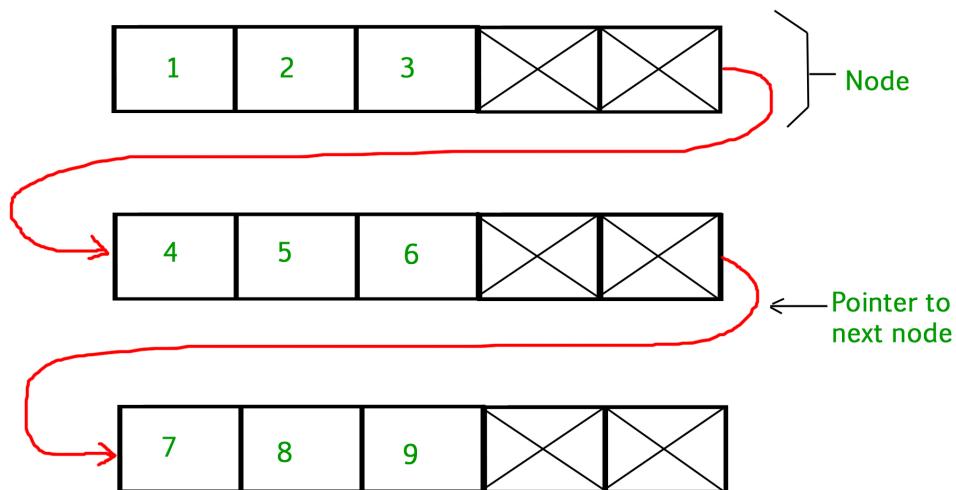
<https://www.geeksforgeeks.org/union-find-algorithm-union-by-rank-and-find-by-optimized-path-compression/>

Chapter 214

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 215

Wavelet Trees Introduction

Wavelet Trees Introduction - GeeksforGeeks

A wavelet tree is a data structure that recursively partitions a stream into two parts until we're left with homogeneous data. The name derives from an analogy with the wavelet transform for signals, which recursively decomposes a signal into low-frequency and high-frequency components. Wavelet trees can be used to answer range queries efficiently.

Consider the problem to find number of elements in a range $[L, R]$ of a given array A which are less than x. One way to solve this problem efficiently is using [Persistent Segment Tree](#) data structure. But we can also solve this easily using Wavelet Trees. Let us see how!

Constructing Wavelet Trees

Every node in a wavelet tree is represented by an array which is the subsequence of original array and a range $[L, R]$. Here $[L, R]$ is the range in which elements of array falls. That is, 'R' denotes maximum element in the array and 'L' denotes the smallest element. So, the root node will contain the original array in which elements are in range $[L, R]$. Now we will calculate the middle of the range $[L, R]$ and stable partition the array in two halves for the left and right childs. Therefore, the left child will contains elements that lies in range $[L, mid]$ and right child will contain elements that lies in the range $[mid+1, R]$.

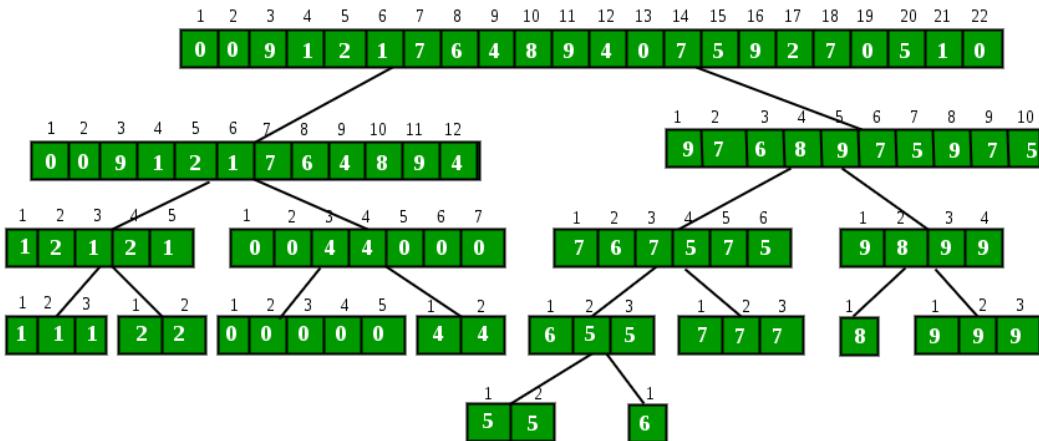
Suppose we are given an array of integers. Now we compute the mid ($\text{Max} + \text{Min} / 2$) and form two children.

Left Children: Integers less than/equal to Mid

Right Children: Integers greater than Mid

We recursively perform this operation until all node of similar elements are formed.

Given array : 0 0 9 1 2 1 7 6 4 8 9 4 3 7 5 9 2 7 0 5 1 0



To construct a Wavelet Tree, let us see what will we need to store at each node. So at each node of the tree, we will store two arrays say $S[]$ and $\text{freq}[]$. The array $S[]$ will be a subsequence of the original array $A[]$ and the array $\text{freq}[]$ will store the count of the elements that will go to left and right childs of the node. That is, $\text{freq}[i]$ will denote the count of elements from the first i elements of $S[]$ that will go to left child. Therefore, count of elements that will go to right child can be easily calculated as $(i - \text{freq}[i])$.

Below example shows how to maintain $\text{freq}[]$ array:

```

Array : 1 5 2 6 4 4
Mid = (1 + 6) / 2 = 3
Left Child : 1 2
Right Child : 5 6 4 4
    
```

To maintain frequency array, we will check if the element is less than Mid or not. If yes, then we will add 1 to last element of frequency array, else 0 and push back again.

For, above array :

Freq array :{1, 1, 2, 2, 2, 2}

It implies 1 element will go to left child of this node from index 1 and 2, and 2 elements will go to left child from indices 3 to 6. This can be easily depicted from the above given array. To compute the number of elements moving to right subtree, we subtract $\text{freq}[i]$ from i.

```

From index 1, 0 elements go to right subtree.
From index 2, 1 element go to right subtree.
From index 3, 1 element go to right subtree.
From index 4, 2 elements go to right subtree.
From index 5, 3 elements go to right subtree.
From index 6, 4 elements go to right subtree.
    
```

We can use the `stable_partition` function and `lambda expression` in C++ STL to easily stable partition the array around a pivot without distorting the order of elements in original

sequence. It is highly recommended to go through the stable_partition and lambda expression articles before moving onto implementation.

Below is the implementation of construction of Wavelet Trees:

```
// CPP code to implement wavelet trees
#include <bits/stdc++.h>
using namespace std;
#define N 100000

// Given array
int arr[N];

// wavelet tree class
class wavelet_tree {
public:

    // Range to elements
    int low, high;

    // Left and Right children
    wavelet_tree* l, *r;

    vector<int> freq;

    // Default constructor
    // Array is in range [x, y]
    // Indices are in range [from, to]
    wavelet_tree(int* from, int* to, int x, int y)
    {
        // Initialising low and high
        low = x, high = y;

        // Array is of 0 length
        if (from >= to)
            return;

        // Array is homogenous
        // Example : 1 1 1 1 1
        if (high == low) {

            // Assigning storage to freq array
            freq.reserve(to - from + 1);

            // Initialising the Freq array
            freq.push_back(0);

            // Assigning values
            for (auto it = from; it != to; it++)
                freq[*(it)]++;
        }
    }

    // Range Sum Query
    int query(int l, int r) {
        if (l > high || r < low)
            return 0;
        if (l >= low && r <= high)
            return freq[r - low];
        else
            return l <= low ? l <= r ? query(l, r, l, r) : query(l, r, l, high) + query(l, r, high, r) : query(l, r, low, high);
    }

    // Range Update
    void update(int l, int r, int val) {
        if (l > high || r < low)
            return;
        if (l >= low && r <= high) {
            freq[l - low] += val;
            freq[r - low] -= val;
        } else
            update(l, r, val, l, r, low, high);
    }

    // Print the tree
    void print() {
        cout << "Low: " << low << " High: " << high << endl;
        if (l)
            l->print();
        if (r)
            r->print();
    }
};
```

```

        // freq will be increasing as there'll
        // be no further sub-tree
        freq.push_back(freq.back() + 1);

        return;
    }

    // Computing mid
    int mid = (low + high) / 2;

    // Lambda function to check if a number is
    // less than or equal to mid
    auto lessThanMid = [mid](int x) {
        return x <= mid;
    };

    // Assigning storage to freq array
    freq.reserve(to - from + 1);

    // Initialising the freq array
    freq.push_back(0);

    // Assigning value to freq array
    for (auto it = from; it != to; it++)

        // If lessThanMid returns 1(true), we add
        // 1 to previous entry. Otherwise, we add
        // 0 (element goes to right sub-tree)
        freq.push_back(freq.back() + lessThanMid(*it));

    // std::stable_partition partitions the array w.r.t Mid
    auto pivot = stable_partition(from, to, lessThanMid);

    // Left sub-tree's object
    l = new wavelet_tree(from, pivot, low, mid);

    // Right sub-tree's object
    r = new wavelet_tree(pivot, to, mid + 1, high);
}

};

// Driver code
int main()
{
    int size = 5, high = INT_MIN;
    int arr[] = {1, 2, 3, 4, 5};
    for (int i = 0; i < size; i++)

```

```

        high = max(high, arr[i]);

    // Object of class wavelet tree
    wavelet_tree obj(arr, arr + size, 1, high);

    return 0;
}

```

Height of the tree: $O(\log(\max(A)))$, where $\max(A)$ is the maximum element in the array $A[]$.

Querying in Wavelet Trees

We have already constructed our wavelet tree for the given array. Now we will move on to our problem to calculate number of elements less than or equal to x in range $[L, R]$ in the given array.

So, for each node we have a subsequence of original array, lowest and highest values present in the array and count of elements in left and right child.

Now,

```

If high <= x,
    we return R - L + 1.
i.e. all the elements in the current range is less than x.

```

Otherwise, We will use variable $LtCount = freq[L-1]$ (i.e. elements going to left sub-tree from $L-1$) , $RtCount = freq[R]$ (i.e. elements going to right sub-tree from R)

Now, we recursively call and add the return values of :

```

left sub-tree with range[ LtCount + 1, RtCount ] and,
right sub-tree with range[ L - Ltcount,R - RtCount ]

```

Below is the implementation in C++:

```

// CPP program for querying in
// wavelet tree Data Structure
#include <bits/stdc++.h>
using namespace std;
#define N 100000

// Given Array
int arr[N];

// wavelet tree class
class wavelet_tree {

```

```

public:
    // Range to elements
    int low, high;

    // Left and Right child
    wavelet_tree* l, *r;

    vector<int> freq;

    // Default constructor
    // Array is in range [x, y]
    // Indices are in range [from, to]
    wavelet_tree(int* from, int* to, int x, int y)
    {
        // Initialising low and high
        low = x, high = y;

        // Array is of 0 length
        if (from >= to)
            return;

        // Array is homogenous
        // Example : 1 1 1 1 1
        if (high == low) {
            // Assigning storage to freq array
            freq.reserve(to - from + 1);

            // Initialising the Freq array
            freq.push_back(0);

            // Assigning values
            for (auto it = from; it != to; it++)
                freq.push_back(freq.back() + 1);
        }

        return;
    }

    // Computing mid
    int mid = (low + high) / 2;

    // Lambda function to check if a number
    // is less than or equal to mid
    auto lessThanMid = [mid](int x) {
        return x <= mid;
    };

```

```

// Assigning storage to freq array
freq.reserve(to - from + 1);

// Initialising the freq array
freq.push_back(0);

// Assigning value to freq array
for (auto it = from; it != to; it++)

    // If lessThanMid returns 1(true), we add
    // 1 to previous entry. Otherwise, we add 0
    // (element goes to right sub-tree)
    freq.push_back(freq.back() + lessThanMid(*it));

// std::stable_partition partitions the array w.r.t Mid
auto pivot = stable_partition(from, to, lessThanMid);

// Left sub-tree's object
l = new wavelet_tree(from, pivot, low, mid);

// Right sub-tree's object
r = new wavelet_tree(pivot, to, mid + 1, high);
}

// Count of numbers in range[L..R] less than
// or equal to k
int kOrLess(int l, int r, int k)
{
    // No elements int range is less than k
    if (l > r or k < low)
        return 0;

    // All elements in the range are less than k
    if (high <= k)
        return r - l + 1;

    // Computing LtCount and RtCount
    int LtCount = freq[l - 1];
    int RtCount = freq[r];

    // Answer is (no. of element <= k) in
    // left + (those <= k) in right
    return (this->l->kOrLess(LtCount + 1, RtCount, k) +
            this->r->kOrLess(l - LtCount, r - RtCount, k));
}

};


```

```
// Driver code
int main()
{
    int size = 5, high = INT_MIN;
    int arr[] = {1, 2, 3, 4, 5};

    // Array : 1 2 3 4 5
    for (int i = 0; i < size; i++)
        high = max(high, arr[i]);

    // Object of class wavelet tree
    wavelet_tree obj(arr, arr + size, 1, high);

    // count of elements less than 2 in range [1,3]
    cout << obj.kOrLess(0, 3, 2) << '\n';

    return 0;
}
```

Output :

2

Time Complexity: $O(\log(\max(A)))$, where $\max(A)$ is the maximum element in the array $A[]$.

In this post we have discussed about a single problem on range queries without update. In further we will be discussing on range updates also.

References :

- <https://users.dcc.uchile.cl/~jperez/papers/ioiconf16.pdf>
- https://en.wikipedia.org/wiki/Wavelet_Tree
- <https://www.youtube.com/watch?v=K7tju9j7UWU>

Source

<https://www.geeksforgeeks.org/wavelet-trees-introduction/>

Chapter 216

Weighted Prefix Search

Weighted Prefix Search - GeeksforGeeks

Given **n** strings and a weight associated with each string. The task is to find the maximum weight of string having the given prefix. Print “-1” if no string is present with given prefix.

Examples:

```
Input :  
s1 = "geeks", w1 = 15  
s2 = "geeksfor", w2 = 30  
s3 = "geeksforgeeks", w3 = 45  
prefix = geek  
Output : 45
```

All the string contain the given prefix, but maximum weight of string is 45 among all.

Method 1: (Brute Force)

Check all the string for given prefix, if string contains the prefix, compare its weight with maximum value so far.

Below is the implementation of above idea :

C++

```
// C++ program to find the maximum weight with given prefix.  
// Brute Force based C++ program to find the  
// string with maximum weight and given prefix.  
#include<bits/stdc++.h>  
#define MAX 1000  
using namespace std;
```

```
// Return the maximum weight of string having
// given prefix.
int maxWeight(char str[MAX][MAX], int weight[],
              int n, char prefix[])
{
    int ans = -1;
    bool check;

    // Traversing all strings
    for (int i = 0; i < n; i++)
    {
        check = true;

        // Checking if string contain given prefix.
        for (int j=0, k=0; j < strlen(str[i]) &&
                         k < strlen(prefix); j++, k++)
        {
            if (str[i][j] != prefix[k])
            {
                check = false;
                break;
            }
        }

        // If contain prefix then finding
        // the maximum value.
        if (check)
            ans = max(ans, weight[i]);
    }

    return ans;
}

// Driven program
int main()
{
    int n = 3;
    char str[3][MAX] = { "geeks", "geeksfor", "geeksforgeeks" };
    int weight[] = {15, 30, 45};
    char prefix[] = "geek";

    cout << maxWeight(str, weight, n, prefix) << endl;

    return 0;
}
```

Java

```
// Java program to find the maximum
// weight with given prefix.

class GFG {
    static final int MAX = 1000;

    // Return the maximum weight of string having
    // given prefix.
    static int maxWeight(String str[], int weight[],
                          int n, String prefix)
    {
        int ans = -1;
        boolean check;

        // Traversing all strings
        for (int i = 0; i < n; i++)
        {
            check = true;

            // Checking if string contain given prefix.
            for (int j=0, k=0; j < str[i].length() &&
                  k < prefix.length(); j++, k++)
            {
                if (str[i].charAt(j) != prefix.charAt(k))
                {
                    check = false;
                    break;
                }
            }

            // If contain prefix then finding
            // the maximum value.
            if (check)
                ans = Math.max(ans, weight[i]);
        }

        return ans;
    }

    // Driven program
    public static void main(String args[])
    {
        int n = 3;
        String str[] = { "geeks", "geeksfor", "geeksforgeeks" };
        int weight[] = {15, 30, 45};
        String prefix = "geek";

        System.out.println(maxWeight(str, weight, n, prefix));
    }
}
```

```
        }
    }
//This code is contributed by Sumit Ghosh

C#
// C# program to find the maximum weight
// with given prefix.
using System;

class GFG
{

    // Return the maximum weight of
    // string having given prefix.
    static int maxWeight(string []str, int []weight,
                          int n, string prefix)
    {
        int ans = -1;
        bool check;

        // Traversing all strings
        for (int i = 0; i < n; i++)
        {
            check = true;

            // Checking if string contain given prefix.
            for (int j=0, k=0; j < str[i].Length &&
                 k < prefix.Length; j++, k++)
            {
                if (str[i][j] != prefix[k])
                {
                    check = false;
                    break;
                }
            }

            // If contain prefix then finding
            // the maximum value.
            if (check)
                ans = Math.Max(ans, weight[i]);
        }

        return ans;
    }

    // Driver Code
    public static void Main()

```

```
{  
    int n = 3;  
    String []str = {"geeks", "geeksfor",  
                    "geeksforgeeks"};  
    int []weight = {15, 30, 45};  
    String prefix = "geek";  
  
    Console.WriteLine(maxWeight(str, weight,  
                                n, prefix));  
}  
}  
  
// This code is contributed by vt_m.
```

Output:

45

Method 2 (efficient):

The idea is to create and maintain a Trie. Instead of the normal Trie where we store the character, store a number with it, which is maximum value of its prefix. When we encounter the prefix again update the value with maximum of existing and new one.

Now, search prefix for maximum value, run through the characters starting from the root, if one of character is missing return -1, else return the number stored in the root.

Below is the implementation of the above idea :

C++

```
// C++ program to find the maximum weight  
// with given prefix.  
#include<bits/stdc++.h>  
#define MAX 1000  
using namespace std;  
  
// Structure of a trie node  
struct trienode  
{  
    // Pointer its children.  
    struct trienode *children[26];  
  
    // To store weight of string.  
    int weight;  
};
```

```

// Create and return a Trie node
struct trieNode* getNode()
{
    struct trieNode *node = new trieNode;
    node -> weight = INT_MIN;

    for (int i = 0; i < 26; i++)
        node -> children[i] = NULL;
}

// Inserting the node in the Trie.
struct trieNode* insert(char str[], int wt, int idx,
                       struct trieNode* root)
{
    int cur = str[idx] - 'a';

    if (!root -> children[cur])
        root -> children[cur] = getNode();

    // Assigning the maximum weight
    root->children[cur]->weight =
        max(root->children[cur]->weight, wt);

    if (idx + 1 != strlen(str))
        root -> children[cur] =
            insert(str, wt, idx + 1, root -> children[cur]);

    return root;
}

// Search and return the maximum weight.
int searchMaximum(char prefix[], struct trieNode *root)
{
    int idx = 0, n = strlen(prefix), ans = -1;

    // Searching the prefix in TRie.
    while (idx < n)
    {
        int cur = prefix[idx] - 'a';

        // If prefix not found return -1.
        if (!root->children[cur])
            return -1;

        ans = root->children[cur]->weight;
        root = root->children[cur];
        ++idx;
    }
}

```

```
        return ans;
    }

// Return the maximum weight of string having given prefix.
int maxWeight(char str[MAX][MAX], int weight[], int n,
               char prefix[])
{
    struct trieNode* root = getNode();

    // Inserting all string in the Trie.
    for (int i = 0; i < n; i++)
        root = insert(str[i], weight[i], 0, root);

    return searchMaximum(prefix, root);
}

// Driven Program
int main()
{
    int n = 3;
    char str[3][MAX] = {"geeks", "geeksfor", "geeksforgeeks"};
    int weight[] = {15, 30, 45};
    char prefix[] = "geek";

    cout << maxWeight(str, weight, n, prefix) << endl;

    return 0;
}
```

Java

```
// Java program to find the maximum weight
// with given prefix.

public class GFG{
    static final int MAX = 1000;

    // Structure of a trie node
    static class TrieNode
    {
        // children
        TrieNode[] children = new TrieNode[26];

        // To store weight of string.
        int weight;
    }
}
```

```

// constructor
public TrieNode() {
    weight = Integer.MIN_VALUE;
    for (int i = 0; i < 26; i++)
        children[i] = null;
}
//static TrieNode root;

// Inserting the node in the Trie.
static TrieNode insert(String str, int wt, int idx, TrieNode root)
{
    int cur = str.charAt(idx) - 'a';

    if (root.children[cur] == null)
        root.children[cur] = new TrieNode();

    // Assigning the maximum weight
    root.children[cur].weight =
        Math.max(root.children[cur].weight, wt);

    if (idx + 1 != str.length())
        root.children[cur] =
            insert(str, wt, idx + 1, root.children[cur]);

    return root;
}

// Search and return the maximum weight.
static int searchMaximum(String prefix, TrieNode root)
{
    int idx = 0, ans = -1;
    int n = prefix.length();

    // Searching the prefix in TRie.
    while (idx < n)
    {
        int cur = prefix.charAt(idx) - 'a';

        // If prefix not found return -1.
        if (root.children[cur] == null)
            return -1;

        ans = root.children[cur].weight;
        root = root.children[cur];
        ++idx;
    }
}

```

```
        return ans;
    }

// Return the maximum weight of string having given prefix.
static int maxWeight(String str[], int weight[], int n,
                      String prefix)
{
    TrieNode root = new TrieNode();

    // Inserting all string in the Trie.
    for (int i = 0; i < n; i++)
        root = insert(str[i], weight[i], 0, root);

    return searchMaximum(prefix, root);
}

// Driven Program
public static void main(String args[])
{
    int n = 3;
    String str[] = { "geeks", "geeksfor", "geeksforgeeks" };
    int weight[] = {15, 30, 45};
    String prefix = "geek";

    System.out.println(maxWeight(str, weight, n, prefix));
}
}

//This code is contributed by Sumit Ghosh
```

45

Improved By : [vt_m](#)

Source

<https://www.geeksforgeeks.org/weighted-prefix-search/>

Chapter 217

Word formation using concatenation of two dictionary words

Word formation using concatenation of two dictionary words - GeeksforGeeks

Given a dictionary find out if given word can be made by two words in the dictionary.

Note: Words in the dictionary must be unique and the word to be formed should not be a repetition of same words that are present in the Trie.

Examples:

```
Input : dictionary[] = {"news", "abcd", "tree",
                       "geeks", "paper"}
        word = "newspaper"
Output : Yes
We can form "newspaper" using "news" and "paper"

Input : dictionary[] = {"geeks", "code", "xyz",
                       "forgeeks", "paper"}
        word = "geeksforgeeks"
Output : Yes

Input : dictionary[] = {"geek", "code", "xyz",
                       "forgeeks", "paper"}
        word = "geeksforgeeks"
Output : No
```

The idea is store all words of dictionary in a [Trie](#). We do prefix search for given word. Once we find a prefix, we search for rest of the word.

Algorithm :

- 1- Store all the words of the dictionary in a Trie.
- 2- Start searching for the given word in Trie.
If it partially matched then split it into two parts and then search for the second part in the Trie.
- 3- If both found, then return true.
- 4- Otherwise return false.

Below is the implementation of above idea.

C++

```
// C++ program to check if a string can be
// formed by concatenating two words
#include<bits/stdc++.h>
using namespace std;

// Converts key current character into index
// use only 'a' through 'z'
#define char_int(c) ((int)c - (int)'a')

// Alphabet size
#define SIZE (26)

// Trie Node
struct TrieNode
{
    TrieNode *children[SIZE];

    // isLeaf is true if the node represents
    // end of a word
    bool isLeaf;
};

// Returns new trie node (initialized to NULLs)
TrieNode *getNode()
{
    TrieNode *newNode = new TrieNode;
    newNode->isLeaf = false;
    for (int i = 0 ; i < SIZE ; i++)
        newNode->children[i] = NULL;
    return newNode;
}

// If not present, inserts key into Trie
```

```
// If the key is prefix of trie node, just
// mark leaf node
void insert(TrieNode *root, string Key)
{
    int n = Key.length();
    TrieNode * pCrawl = root;

    for (int i=0; i<n; i++)
    {
        int index = char_int(Key[i]);

        if (pCrawl->children[index] == NULL)
            pCrawl->children[index] = getNode();

        pCrawl = pCrawl->children[index];
    }

    // make last node as leaf node
    pCrawl->isLeaf = true;
}

// Searches a prefix of key. If prefix is present,
// returns its ending position in string. Else
// returns -1.
int findPrefix(struct TrieNode *root, string key)
{
    int pos = -1, level;
    struct TrieNode *pCrawl = root;

    for (level = 0; level < key.length(); level++)
    {
        int index = char_int(key[level]);
        if (pCrawl->isLeaf == true)
            pos = level;
        if (!pCrawl->children[index])
            return pos;

        pCrawl = pCrawl->children[index];
    }
    if (pCrawl != NULL && pCrawl->isLeaf)
        return level;
}

// Function to check if word formation is possible
// or not
bool isPossible(struct TrieNode* root, string word)
{
    // Search for the word in the trie and
```

```
// store its position upto which it is matched
int len = findPrefix(root, word);

// print not possible if len = -1 i.e. not
// matched in trie
if (len == -1)
    return false;

// If word is partially matched in the dictionary
// as another word
// search for the word made after splitting
// the given word up to the length it is
// already matched
string split_word(word, len, word.length()-(len));
int split_len = findPrefix(root, split_word);

// check if word formation is possible or not
return (len + split_len == word.length());
}

// Driver program to test above function
int main()
{
    // Let the given dictionary be following
    vector<string> dictionary = {"geeks", "forgeeks",
                                  "quiz", "geek"};

    string word = "geeksquiz"; //word to be formed

    // root Node of trie
    TrieNode *root = getNode();

    // insert all words of dictionary into trie
    for (int i=0; i<dictionary.size(); i++)
        insert(root, dictionary[i]);

    if(isPossible(root, word) ? cout << "Yes":
       cout << "No");

    return 0;
}
```

Java

```
// Java program to check if a string can be
// formed by concatenating two words
public class GFG {
```

```
// Alphabet size
final static int SIZE = 26;

// Trie Node
static class TrieNode
{
    TrieNode[] children = new TrieNode[SIZE];

    // isLeaf is true if the node represents
    // end of a word
    boolean isLeaf;

    // constructor
    public TrieNode() {
        isLeaf = false;
        for (int i = 0 ; i < SIZE ; i++)
            children[i] = null;
    }
}

static TrieNode root;

// If not present, inserts key into Trie
// If the key is prefix of trie node, just
// mark leaf node
static void insert(TrieNode root, String Key)
{
    int n = Key.length();
    TrieNode pCrawl = root;

    for (int i=0; i < n; i++)
    {
        int index = Key.charAt(i) - 'a';

        if (pCrawl.children[index] == null)
            pCrawl.children[index] = new TrieNode();

        pCrawl = pCrawl.children[index];
    }

    // make last node as leaf node
    pCrawl.isLeaf = true;
}

// Searches a prefix of key. If prefix is present,
// returns its ending position in string. Else
// returns -1.
```

```
static int findPrefix(TrieNode root, String key)
{
    int pos = -1, level;
    TrieNode pCrawl = root;

    for (level = 0; level < key.length(); level++)
    {
        int index = key.charAt(level) - 'a';
        if (pCrawl.isLeaf == true)
            pos = level;
        if (pCrawl.children[index] == null)
            return pos;

        pCrawl = pCrawl.children[index];
    }
    if (pCrawl != null && pCrawl.isLeaf)
        return level;

    return -1;
}

// Function to check if word formation is possible
// or not
static boolean isPossible(TrieNode root, String word)
{
    // Search for the word in the trie and
    // store its position upto which it is matched
    int len = findPrefix(root, word);

    // print not possible if len = -1 i.e. not
    // matched in trie
    if (len == -1)
        return false;

    // If word is partially matched in the dictionary
    // as another word
    // search for the word made after splitting
    // the given word up to the length it is
    // already matched
    // string split_word(word, len, word.length()-(len));
    String split_word = word.substring(len, word.length());
    int split_len = findPrefix(root, split_word);

    // check if word formation is possible or not
    return (len + split_len == word.length());
}

// Driver program to test above function
```

```
public static void main(String args[])
{
    // Let the given dictionary be following
    String[] dictionary = {"geeks", "forgeeks", "quiz",
                           "geek"};

    String word = "geeksquiz"; //word to be formed

    // root Node of trie
    root = new TrieNode();

    // insert all words of dictionary into trie
    for (int i=0; i<dictionary.length; i++)
        insert(root, dictionary[i]);

    if(isPossible(root, word))
        System.out.println( "Yes");
    else
        System.out.println("No");
}
}

// This code is contributed by Sumit Ghosh
```

Output:

Yes

Exercise :

A generalized version of the problem is to check if a given word can be formed using concatenation of 1 or more dictionary words. Write code for the generalized version.

Source

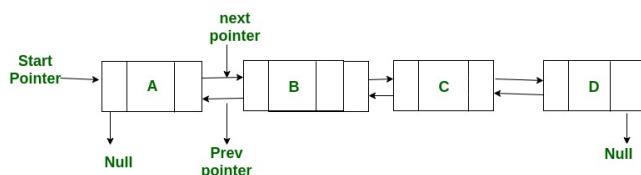
<https://www.geeksforgeeks.org/word-formation-using-concatenation-of-two-dictionary-words/>

Chapter 218

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://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 219

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/>

Chapter 220

XOR of numbers that appeared even number of times in given Range

XOR of numbers that appeared even number of times in given Range - GeeksforGeeks

Given an array of numbers of size N and Q queries. Each query or a range can be represented by L (LeftIndex) and R(RightIndex). Find the XOR-sum of the numbers that appeared even number of times in the given range.

Prerequisite : [Queries for number of distinct numbers in given range. Segment Tree for range query](#)

Examples :

```
Input : arr[] = { 1, 2, 1, 3, 3, 2, 3 }
        Q = 5
        L = 3,  R = 6
        L = 3,  R = 4
        L = 0,  R = 2
        L = 0,  R = 6
        L = 0,  R = 4
Output : 0
         3
         1
         3
         2
```

Explanation of above example:

In Query 1, there are no numbers which appeared even number of times.
Hence the XOR-sum is 0.

In Query 2, {3} appeared even number of times. XOR-sum is 3.

In Query 3, {1} appeared even number of times. XOR-sum is 1.

In Query 4, {1, 2} appeared even number of times. XOR-sum is 1 xor 2 = 3.

In Query 5, {1, 3} appeared even number of times. XOR-sum is 1 xor 3 = 2.

Segment Trees or Binary Indexed Trees can be used to solve this problem efficiently.

Approach :

Firstly, it is easy to note that the answer for the query is the XOR-sum of **all** elements in the query range xor-ed with XOR-sum of **distinct** elements in the query range (since taking XOR of an element with itself results into a null value). Find the XOR-sum of all numbers in query range using prefix XOR-sums.

To find the XOR-sum of distinct elements in range : Number of distinct elements in a subarray of given range.

Now, returning back to our main problem, just change the assignment $\text{BIT}[i] = 1$ to $\text{BIT}[i] = \text{arr}_i$ and count the XOR-sum instead of sum.

Below is the implementation using Binary Indexed Trees in CPP

```
// CPP Program to Find the XOR-sum
// of elements that appeared even
// number of times within a range
#include <bits/stdc++.h>
using namespace std;

/* structure to store queries
   L --> Left Bound of Query
   R --> Right Bound of Query
   idx --> Query Number */
struct que {
    int L, R, idx;
};

// cmp function to sort queries
// according to R
bool cmp(que a, que b)
{
    if (a.R != b.R)
        return a.R < b.R;
    else
        return a.L < b.L;
}

/* N --> Number of elements present in
   input array. BIT[0..N] --> Array that
   represents Binary Indexed Tree*/

// Returns XOR-sum of arr[0..index]. This
// function assumes that the array is
```

```
// preprocessed and partial sums of array
// elements are stored in BIT[] .
int getSum(int BIT[], int index)
{
    // Initialize result
    int xorSum = 0;

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

    // Traverse ancestors of BIT[index]
    while (index > 0)
    {
        // Take XOR of current element
        // of BIT to xorSum
        xorSum ^= BIT[index];

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

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

    // Traverse all ancestors and
    // take xor with 'val'
    while (index <= N)
    {
        // Take xor with 'val' to
        // current node of BIT
        BIT[index] ^= val;

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

```
}  
  
// Constructs and returns a Binary Indexed  
// Tree for given array of size N.  
int* constructBITree(int arr[], int N)  
{  
    // Create and initialize BITree[] as 0  
    int* BIT = new int[N + 1];  
  
    for (int i = 1; i <= N; i++)  
        BIT[i] = 0;  
  
    return BIT;  
}  
  
// Function to answer the Queries  
void answeringQueries(int arr[], int N,  
                      que queries[], int Q, int BIT[])  
{  
    // Creating an array to calculate  
    // prefix XOR sums  
    int* prefixXOR = new int[N + 1];  
  
    // map for coordinate compression  
    // as numbers can be very large but we  
    // have limited space  
    map<int, int> mp;  
  
    for (int i = 0; i < N; i++) {  
  
        // If A[i] has not appeared yet  
        if (!mp[arr[i]])  
            mp[arr[i]] = i;  
  
        // calculate prefixXOR sums  
        if (i == 0)  
            prefixXOR[i] = arr[i];  
        else  
            prefixXOR[i] =  
                prefixXOR[i - 1] ^ arr[i];  
    }  
  
    // Creating an array to store the  
    // last occurrence of arr[i]  
    int lastOcc[10000001];  
    memset(lastOcc, -1, sizeof(lastOcc));  
  
    // sort the queries according to comparator
```

```

sort(queries, queries + Q, cmp);

// answer for each query
int res[Q];

// Query Counter
int j = 0;

for (int i = 0; i < Q; i++)
{
    while (j <= queries[i].R)
    {
        // If last visit is not -1 update
        // arr[j] to set null by taking
        // xor with itself at the idx
        // equal lastOcc[mp[arr[j]]]
        if (lastOcc[mp[arr[j]]] != -1)
            updateBIT(BIT, N,
                      lastOcc[mp[arr[j]]], arr[j]);

        // Setting lastOcc[mp[arr[j]]] as j and
        // updating the BIT array accordingly
        updateBIT(BIT, N, j, arr[j]);
        lastOcc[mp[arr[j]]] = j;
        j++;
    }

    // get the XOR-sum of all elements within
    // range using precomputed prefix XORsums
    int allXOR = prefixXOR[queries[i].R] ^
                  prefixXOR[queries[i].L - 1];

    // get the XOR-sum of distinct elements
    // within range using BIT query function
    int distinctXOR = getSum(BIT, queries[i].R) ^
                      getSum(BIT, queries[i].L - 1);

    // store the final answer at the numbered query
    res[queries[i].idx] = allXOR ^ distinctXOR;
}

// Output the result
for (int i = 0; i < Q; i++)
    cout << res[i] << endl;
}

// Driver program to test above functions
int main()

```

```
{  
    int arr[] = { 1, 2, 1, 3, 3, 2, 3 };  
    int N = sizeof(arr) / sizeof(arr[0]);  
  
    int* BIT = constructBITree(arr, N);  
  
    // structure of array for queries  
    que queries[5];  
  
    // Intializing values (L, R, idx) to queries  
    queries[0].L = 3;  
    queries[0].R = 6, queries[0].idx = 0;  
    queries[1].L = 3;  
    queries[1].R = 4, queries[1].idx = 1;  
    queries[2].L = 0;  
    queries[2].R = 2, queries[2].idx = 2;  
    queries[3].L = 0;  
    queries[3].R = 6, queries[3].idx = 3;  
    queries[4].L = 0;  
    queries[4].R = 4, queries[4].idx = 4;  
  
    int Q = sizeof(queries) / sizeof(queries[0]);  
  
    // answer Queries  
    answeringQueries(arr, N, queries, Q, BIT);  
  
    return 0;  
}
```

Output:

```
0  
3  
1  
3  
2
```

Time Complexity: $O(Q * \log(N))$, where N is the size of array, Q is the total number of queries.

Source

<https://www.geeksforgeeks.org/xor-numbers-appeared-even-number-times-given-range/>

Chapter 221

proto van Emde Boas Trees Set 1 (Background and Introduction)

proto van Emde Boas Trees Set 1 (Background and Introduction) - GeeksforGeeks

Let us consider the below problem statement and think of different solutions for it.

Given a set **S** of elements such that the elements are taken from universe $\{0, 1, \dots, u-1\}$, perform following operations efficiently.

- **insert(x)** : Adds an item x to the set + S.
- **isEmpty()** : Returns true if S is empty, else false.
- **find(x)** : Returns true if x is present in S, else false.
- **insert(x)** : Inserts an item x to S.
- **delete(x)** : Delete an item x from S.
- **max()** : Returns maximum value from S.
- **min()** : Returns minimum value from S.
- **successor(x)** : Returns the smallest value in S which is greater than x.
- **predecessor(x)** : Returns the largest value in S which is smaller than x.

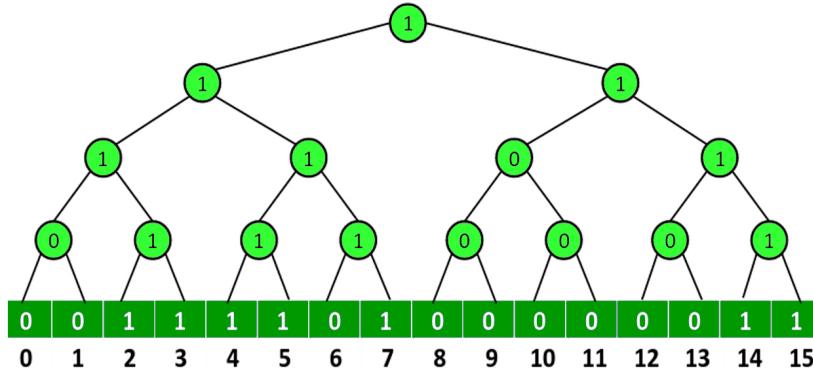
Different Solutions

Below are different solutions for the above problem.

- One solution to solve above problem is to use a **self-balancing Binary Search Tree** like [Red-Black Tree](#), [AVL Tree](#), etc. With this solution, we can perform all above operations in $O(\log n)$ time.
- Another solution is to use **Binary Array (or Bitvector)**. We create an array of size u and mark presence and absence of an element as 1 or 0 respectively. This solution supports `insert()`, `delete()` and `find()` in $O(1)$ time, but other operations may take $O(u)$ time in worst case.
- **Van Emde Boas tree (or vEB tree)** supports `insert()`, `delete`, `find()`, `successor()` and `predecessor()` operations in $O(\log \log u)$ time, and `max()` and `min()` in $O(1)$ time.
Note : In BST solution, we have time complexity in terms of n , here we have time complexity in terms of u . So Van Emde Boas tree may not be suitable when u is much larger than n .

Background (Superimposing a Binary Tree Structure on Binary Array solution)

The time complexities of `max()`, `min()`, `successor()` and `predecessor()` are high in case of Binary Array solution. The idea is to reduce time complexities of these operations by superimposing a binary tree structure over it.



The set represented is {2,3,4,5,7,14,15}

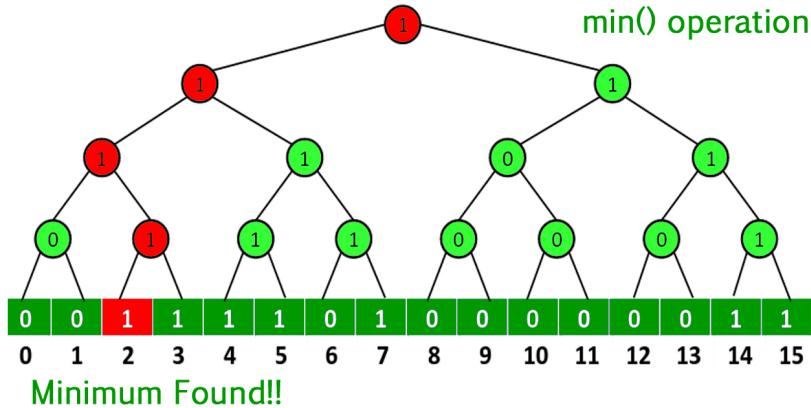
Explanation of above structure:

1. Leaves of binary tree represent entries of binary array.
2. An internal node has value 1 if any of its children has value 1, i.e., value of an internal node is bitwise OR of all values of its children.

With above structure, we have optimized `max()`, `min()`, `successor()` and `predecessor()` to time complexity $O(\log u)$.

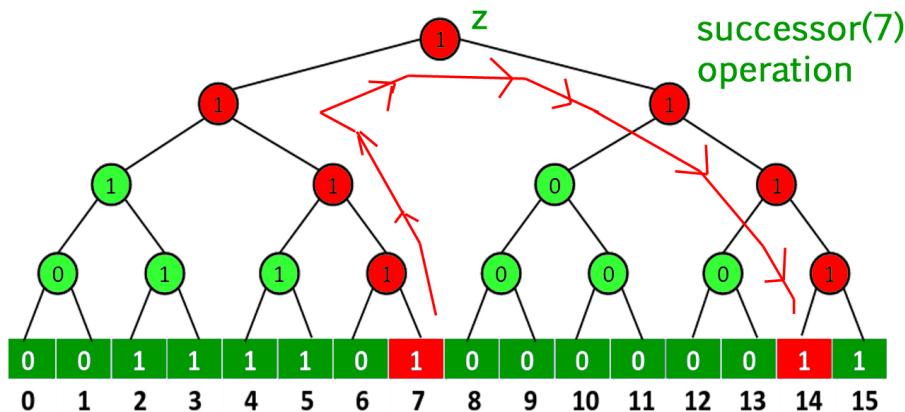
1. **min()** : Start with root and traverse to a leaf using following rules. While traversing, always choose the leftmost child, i.e., see if left child is 1, go to left child, else go to

right child. The leaf node we reach this way is minimum.



Since we travel across height of binary tree with u leaves, time complexity is reduced to $O(\log u)$

2. **max()** : Similar to min(). Instead of left child, we prefer right child.
3. **successor(x)** : Start with leaf node indexed with x and travel to root until we reach node z through its left child. Stop at z and travel down to a leaf following the leftmost node with value 1.



4. **predecessor()** : This operation is similar to successor. Here we replace left with right and right with left in successor().
5. **find()** is still $O(1)$ as we still have binary array as leaves. **insert()** and **delete()** are now $O(\log u)$ as we need to update internal nodes. In case of insert, we mark the corresponding leaf as 1, we traverse up and keep updating ancestors to 1 if they were 0.

proto van Emde Boas Tree

We have seen that superimposing a binary tree over binary array reduces time complexity of $\max()$, $\min()$, $\text{successor}()$ and $\text{predecessor}()$ to $O(\log u)$. Can we reduce this time complexity further to $O(\log \log u)$?

The idea is to have varying degree at different levels. The root node (first level) covers whole universe. Every node of second level (next to root) covers $u^{1/2}$ elements of universe. Every node of third level covers $u^{1/4}$ elements and so on.

With above recursive structure, we get time complexities of operations using below recursion.

$$T(u) = T(\sqrt{u}) + O(1)$$

Solution of this recurrence is,

$$T(u) = O(\log \log u)$$

Refer this for detailed steps to
get the above result.

Recursive definition of proto van Emde Boas Tree:

Let $u = 2^{2^k}$ be the size of universe for some $k \geq 0$.

1. If $u = 2$, then it is a base size tree contains only a binary array of size 2.
2. Otherwise split the universe into $\Theta(u^{1/2})$ blocks of size $\Theta(u^{1/2})$ each and add a summary structure to the top.

We perform all queries as using the approach described in background.

In this post, we have introduced the idea that is to superimpose tree structure on Binary Array such that nodes of different levels of the tree have varying degrees. We will soon be discussing following in coming sets.

- 1) Detailed representation.
- 2) How to optimize $\max()$ and $\min()$ to work in $O(1)$?
- 3) Implementation of the above operations.

Sources:

<http://www-di.inf.puc-rio.br/~laber/vanEmdeBoas.pdf>
<http://web.stanford.edu/class/cs166/lectures/14/Small14.pdf>

Source

<https://www.geeksforgeeks.org/proto-van-emde-boas-trees-set-1-background-introduction/>

Chapter 222

sklearn.Binarizer() in Python

sklearn.Binarizer() in Python - GeeksforGeeks

sklearn.preprocessing.Binarizer() is a method which belongs to preprocessing module. It plays a key role in the discretization of continuous feature values.

Example #1:

A continuous data of pixels values of an 8-bit grayscale image have values ranging between 0 (black) and 255 (white) and one needs it to be black and white. So, using **Binarizer()** one can set a threshold converting pixel values from 0 – 127 to 0 and 128 – 255 as 1.

Example #2:

One has a machine record having “Sucess Percentage” as a feature. These values are continuous ranging from 10% to 99% but a researcher simply wants to use this data for prediction of pass or fail status for the machine based on other given parameters.

Syntax :

```
sklearn.preprocessing.Binarizer(threshold, copy)
```

Parameters :

threshold :[float, optional] Values less than or equal to threshold is mapped to 0, else to 1. By default threshold value is 0.0.

copy :[boolean, optional] If set to False, it avoids a copy. By default it is True.

Return :

Binarized Feature values

	A	B	C	D
1	Country	Age	Salary	Purchased
2	France	44	72000	0
3	Spain	27	48000	1
4	Germany	30	54000	0
5	Spain	38	61000	0
6	Germany	40	1000	1
7	France	35	58000	1
8	Spain	78	52000	0
9	France	48	79000	1
10	Germany	50	83000	0
11	France	37	67000	1

Download the dataset:

Go to the link and download [Data.csv](#)

Below is the Python code explaining `sklearn.Binarizer()`

```
# Python code explaining how
# to Binarize feature values

""" PART 1
    Importing Libraries """

import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

# Sklearn library
from sklearn import preprocessing

""" PART 2
    Importing Data """

data_set = pd.read_csv(
    'C:\\\\Users\\\\dell\\\\Desktop\\\\Data_for_Feature_Scaling.csv')
data_set.head()

# here Features - Age and Salary columns
# are taken using slicing
# to binarize values
age = data_set.iloc[:, 1].values
salary = data_set.iloc[:, 2].values
print ("\nOriginal age data values : \n", age)
print ("\nOriginal salary data values : \n", salary)
```

```
""" PART 4
    Binarizing values """

from sklearn.preprocessing import Binarizer

x = age
x = x.reshape(1, -1)
y = salary
y = y.reshape(1, -1)

# For age, let threshold be 35
# For salary, let threshold be 61000
binarizer_1 = Binarizer(35)
binarizer_2 = Binarizer(61000)

# Transformed feature
print ("\nBinarized age : \n", binarizer_1.fit_transform(x))

print ("\nBinarized salary : \n", binarizer_2.fit_transform(y))
```

Output :

	Country	Age	Salary	Purchased
0	France	44	72000	0
1	Spain	27	48000	1
2	Germany	30	54000	0
3	Spain	38	61000	0
4	Germany	40	1000	1

Original age data values :
[44 27 30 38 40 35 78 48 50 37]

Original salary data values :
[72000 48000 54000 61000 1000 58000 52000 79000 83000 67000]

Binarized age :
[[1 0 0 1 1 0 1 1 1 1]]

Binarized salary :
[[1 0 0 0 0 0 0 1 1 1]]

Source

<https://www.geeksforgeeks.org/sklearn-binarizer-in-python/>

Chapter 223

kasai's Algorithm for Construction of LCP array from Suffix Array

kasai's Algorithm for Construction of LCP array from Suffix Array - GeeksforGeeks

Background

Suffix Array : A suffix array is a sorted array of all suffixes of a given string.
Let the given string be “banana”.

0 banana	Sort the Suffixes	5 a
1 anana	----->	3 ana
2 nana		1 anana
3 ana	alphabetically	0 banana
4 na		4 na
5 a		2 nana

The suffix array for “banana” :

`suffix[] = {5, 3, 1, 0, 4, 2}`

We have discussed [Suffix Array and its O\(nLogn\) construction](#).

Once Suffix array is built, we can use it to efficiently search a pattern in a text. For example, we can use Binary Search to find a pattern (Complete code for the same is discussed [here](#))

LCP Array

The Binary Search based solution discussed [here](#) takes $O(m * \log n)$ time where m is length of the pattern to be searched and n is length of the text. With the help of LCP array, we can search a pattern in $O(m + \log n)$ time. For example, if our task is to search “ana” in “banana”, $m = 3$, $n = 5$.

LCP Array is an array of size n (like Suffix Array). A value $lcp[i]$ indicates length of the longest common prefix of the suffixes indexed by $\text{suffix}[i]$ and $\text{suffix}[i+1]$. $\text{suffix}[n-1]$ is not defined as there is no suffix after it.

```
txt[0..n-1] = "banana"
suffix[]   = {5, 3, 1, 0, 4, 2|
lcp[]      = {1, 3, 0, 0, 2, 0}
```

Suffixes represented by suffix array in order are:
{"a", "ana", "anana", "banana", "na", "nana"}

```
lcp[0] = Longest Common Prefix of "a" and "ana"      = 1
lcp[1] = Longest Common Prefix of "ana" and "anana" = 3
lcp[2] = Longest Common Prefix of "anana" and "banana" = 0
lcp[3] = Longest Common Prefix of "banana" and "na" = 0
lcp[4] = Longest Common Prefix of "na" and "nana" = 2
lcp[5] = Longest Common Prefix of "nana" and None = 0
```

How to construct LCP array?

LCP array construction is done two ways:

- 1) Compute the LCP array as a byproduct to the suffix array (Manber & Myers Algorithm)
- 2) Use an already constructed suffix array in order to compute the LCP values. (Kasai Algorithm).

There exist algorithms that can construct Suffix Array in $O(n)$ time and therefore we can always construct LCP array in $O(n)$ time. But in the below implementation, a $O(n \log n)$ algorithm is discussed.

kasai's Algorithm

In this article kasai's Algorithm is discussed. The algorithm constructs LCP array from suffix array and input text in $O(n)$ time. The idea is based on below fact:

Let lcp of suffix beginning at $\text{txt}[i]$ be k . If k is greater than 0, then lcp for suffix beginning at $\text{txt}[i+1]$ will be at-least $k-1$. The reason is, relative order of characters remain same. If we delete the first character from both suffixes, we know that at least k characters will match. For example for substring "ana", lcp is 3, so for string "na" lcp will be at-least 2. Refer [this](#) for proof.

Below is C++ implementation of Kasai's algorithm.

```
// C++ program for building LCP array for given text
#include <bits/stdc++.h>
using namespace std;

// Structure to store information of a suffix
struct suffix
{
    int index; // To store original index
```

```
int rank[2]; // To store ranks and next rank pair
};

// A comparison function used by sort() to compare two suffixes
// Compares two pairs, returns 1 if first pair is smaller
int cmp(struct suffix a, struct suffix b)
{
    return (a.rank[0] == b.rank[0])? (a.rank[1] < b.rank[1] ?1: 0):
        (a.rank[0] < b.rank[0] ?1: 0);
}

// This is the main function that takes a string 'txt' of size n as an
// argument, builds and return the suffix array for the given string
vector<int> buildSuffixArray(string txt, int n)
{
    // A structure to store suffixes and their indexes
    struct suffix suffixes[n];

    // Store suffixes and their indexes in an array of structures.
    // The structure is needed to sort the suffixes alphabetically
    // and maintain their old indexes while sorting
    for (int i = 0; i < n; i++)
    {
        suffixes[i].index = i;
        suffixes[i].rank[0] = txt[i] - 'a';
        suffixes[i].rank[1] = ((i+1) < n)? (txt[i + 1] - 'a'): -1;
    }

    // Sort the suffixes using the comparison function
    // defined above.
    sort(suffixes, suffixes+n, cmp);

    // At this point, all suffixes are sorted according to first
    // 2 characters. Let us sort suffixes according to first 4
    // characters, then first 8 and so on
    int ind[n]; // This array is needed to get the index in suffixes[]
    // from original index. This mapping is needed to get
    // next suffix.
    for (int k = 4; k < 2*n; k = k*2)
    {
        // Assigning rank and index values to first suffix
        int rank = 0;
        int prev_rank = suffixes[0].rank[0];
        suffixes[0].rank[0] = rank;
        ind[suffixes[0].index] = 0;

        // Assigning rank to suffixes
        for (int i = 1; i < n; i++)
    }
```

```

{
    // If first rank and next ranks are same as that of previous
    // suffix in array, assign the same new rank to this suffix
    if (suffixes[i].rank[0] == prev_rank &&
        suffixes[i].rank[1] == suffixes[i-1].rank[1])
    {
        prev_rank = suffixes[i].rank[0];
        suffixes[i].rank[0] = rank;
    }
    else // Otherwise increment rank and assign
    {
        prev_rank = suffixes[i].rank[0];
        suffixes[i].rank[0] = ++rank;
    }
    ind[suffixes[i].index] = i;
}

// Assign next rank to every suffix
for (int i = 0; i < n; i++)
{
    int nextindex = suffixes[i].index + k/2;
    suffixes[i].rank[1] = (nextindex < n)?
                           suffixes[ind[nextindex]].rank[0]: -1;
}

// Sort the suffixes according to first k characters
sort(suffixes, suffixes+n, cmp);
}

// Store indexes of all sorted suffixes in the suffix array
vector<int>suffixArr;
for (int i = 0; i < n; i++)
    suffixArr.push_back(suffixes[i].index);

// Return the suffix array
return suffixArr;
}

/* To construct and return LCP */
vector<int> kasai(string txt, vector<int> suffixArr)
{
    int n = suffixArr.size();

    // To store LCP array
    vector<int> lcp(n, 0);

    // An auxiliary array to store inverse of suffix array
    // elements. For example if suffixArr[0] is 5, the

```

```
// invSuff[5] would store 0. This is used to get next
// suffix string from suffix array.
vector<int> invSuff(n, 0);

// Fill values in invSuff[]
for (int i=0; i < n; i++)
    invSuff[suffixArr[i]] = i;

// Initialize length of previous LCP
int k = 0;

// Process all suffixes one by one starting from
// first suffix in txt[]
for (int i=0; i<n; i++)
{
    /* If the current suffix is at n-1, then we don't
       have next substring to consider. So lcp is not
       defined for this substring, we put zero. */
    if (invSuff[i] == n-1)
    {
        k = 0;
        continue;
    }

    /* j contains index of the next substring to
       be considered to compare with the present
       substring, i.e., next string in suffix array */
    int j = suffixArr[invSuff[i]+1];

    // Directly start matching from k'th index as
    // at-least k-1 characters will match
    while (i+k<n && j+k<n && txt[i+k]==txt[j+k])
        k++;

    lcp[invSuff[i]] = k; // lcp for the present suffix.

    // Deleting the starting character from the string.
    if (k>0)
        k--;
}

// return the constructed lcp array
return lcp;
}

// Utility function to print an array
void printArr(vector<int>arr, int n)
{
```

```

        for (int i = 0; i < n; i++)
            cout << arr[i] << " ";
        cout << endl;
    }

// Driver program
int main()
{
    string str = "banana";

    vector<int>suffixArr = buildSuffixArray(str, str.length());
    int n = suffixArr.size();

    cout << "Suffix Array : \n";
    printArr(suffixArr, n);

    vector<int>lcp = kasai(str, suffixArr);

    cout << "\nLCP Array : \n";
    printArr(lcp, n);
    return 0;
}

```

Output:

```

Suffix Array :
5 3 1 0 4 2

LCP Array :
1 3 0 0 2 0

```

Illustration:

```
txt[]      = "banana",  suffix[]  = {5, 3, 1, 0, 4, 2|
```

```
Suffix array represents
{"a", "ana", "anana", "banana", "na", "nana"}
```

```
Inverse Suffix Array would be
invSuff[] = {3, 2, 5, 1, 4, 0}
```

LCP values are evaluated in below order

We first compute LCP of first suffix in text which is “banana”. We need next suffix in suffix array to compute LCP (Remember $lcp[i]$ is defined as Longest Common Prefix of suffix[i] and suffix[i+1]). To find the next suffix in $\text{suffixArr}[]$, we use $\text{SuffInv}[]$. The next suffix is “na”. Since there is no common prefix between “banana” and “na”, the value of LCP for “banana” is 0 and it is at index 3 in suffix array, so we fill $\text{lcp}[3]$ as 0.

Next we compute LCP of second suffix which “**anana**”. Next suffix of “anana” in suffix array is “banana”. Since there is no common prefix, the value of LCP for “anana” is 0 and it is at index 2 in suffix array, so we fill **lcp[2]** as 0.

Next we compute LCP of third suffix which “**nana**”. Since there is no next suffix, the value of LCP for “nana” is not defined. We fill **lcp[5]** as 0.

Next suffix in text is “ana”. Next suffix of “**ana**” in suffix array is “anana”. Since there is a common prefix of length 3, the value of LCP for “ana” is 3. We fill **lcp[1]** as 3.

Now we lcp for next suffix in text which is “**na**”. This is where Kasai’s algorithm uses the trick that LCP value must be at least 2 because previous LCP value was 3. Since there is no character after “na”, final value of LCP is 2. We fill **lcp[4]** as 2.

Next suffix in text is “**a**”. LCP value must be at least 1 because previous value was 2. Since there is no character after “a”, final value of LCP is 1. We fill **lcp[0]** as 1.

We will soon be discussing implementation of search with the help of LCP array and how LCP array helps in reducing time complexity to $O(m + \log n)$.

References:

- <http://web.stanford.edu/class/cs97si/suffix-array.pdf>
- <http://www.mi.fu-berlin.de/wiki/pub/ABI/RnaSeqP4/suffix-array.pdf>
- <http://codeforces.com/blog/entry/12796>

This article is contributed by **Prakhar Agrawal**. 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/%c2%ad%c2%adkasais-algorithm-for-construction-of-lcp-array-from-suffix-array/>