



PRACTICE

COMPETE

JOBS

LEADERBOARD

Search



Facebook\_EIR ▾

[All Contests](#) > [ABCS 18: Trees](#) > [Tree: Preorder Traversal](#)

# Tree: Preorder Traversal

locked

by [shashank21j](#)

Problem

Submissions

Leaderboard

Discussions

Editorial

Tested by [AllisonP](#)

## Statistics

Difficulty: Easy

Publish Date: May 04 2015

Problem Tester's code :

```
void preOrder(Node root) {  
    if(root != null){  
        System.out.print(root.data + " ");  
        preOrder(root.left);  
        preOrder(root.right);  
    }  
}
```

```
def preOrder(root):  
    if root != None:  
        print(root.data, end=' ')  
        preOrder(root.left)  
        preOrder(root.right)
```

[Contest Calendar](#) | [Interview Prep](#) | [Blog](#) | [Scoring](#) | [Environment](#) | [FAQ](#) | [About Us](#) | [Support](#) | [Careers](#) | [Terms Of Service](#) | [Privacy Policy](#) | [Request a Feature](#)



# Tree: Height of a Binary Tree

locked

by [vatsalchanana](#)

Problem

Submissions

Leaderboard

Discussions

Editorial

 Editorial by [AllisonP](#)

We are using the following basic recursive algorithm for calculating the height of a binary tree:

```
if t is empty,
    getHeight(t) = -1
else
    getHeight(t) = 1 + MAX( getHeight( leftSubtree(t), rightSubtree(t) ) )
```

**Note:** The height of binary tree with single node is taken to be **0** and the height of the empty binary tree is taken to be **-1**.

 Tested by [AllisonP](#)

Problem Tester's code :

C++

```
int getHeight(Node* root){
    if ( root == NULL ){
        return -1;
    }
    else{
        return 1 + max( getHeight(root->left), getHeight(root->right) );
    }
}
```

Java

```
public static int getHeight(Node root){
    if (root == null){
        return -1;
    }
    else{
        return 1 + Math.max( getHeight(root.left), getHeight(root.right) );
    }
}
```

JavaScript

```
if( root === null ){
    return -1;
}
```

## Statistics

Difficulty: Easy

Publish Date: Apr 06 2016

```

else{
    return 1 + Math.max( this.getHeight(root.left), this.getHeight(root.right) );
}

```

## PHP

```

public function getHeight($root){
    if($root == null) {
        return -1;
    }

    return 1 + max($this->getHeight($root->left), $this->getHeight($root->right));
}

```

## Python

```

def getHeight(self, root):
    if root == None:
        return -1
    else:
        return 1 + max( self.getHeight(root.left), self.getHeight(root.right) )

```

## Ruby

```

def getHeight(root)
    if root == nil
        return -1
    else
        return 1 + [self.getHeight(root.left), self.getHeight(root.right)].max
    end
end

```

## Swift

```

return (root == nil) ? -1 : 1 + max( getHeight(root?.left), getHeight(root?.right) );

```



# Binary Search Tree : Insertion

locked

by [vatsalchanana](#)

Problem

Submissions

Leaderboard

Discussions

Editorial

Editorial by [Shafaet](#)

## Statistics

Difficulty: Easy

Time  $O(\text{Depth})$ 

Complexity: Required

Knowledge: Binary Search Tree

Publish Date: Aug 07 2016

Case 1: The binary tree is empty

The new node will be the root.

Case 2: The value is greater than current root node, and right subtree is not empty.

Go to the right subtree.

Case 3: The value is greater than current root node, and right subtree is empty.

Add the new node in root->right.

Case 2: The value is smaller or equal to the current root node and left subtree is not empty.

Go to the left subtree.

Case 3: The value is smaller or equal to the current root node and left subtree is empty.

Add the new node in root->left.

C++

```
node *insert(node *root, int value) {
    node *new_node = new node();
    new_node->data = value;
    new_node->left = NULL;
    new_node->right = NULL;

    if (!root) { // Case 1
        root = new_node;
        return root;
    }
    node *current_root = root;
    while (1) {
        if (current_root->data > value) {
            if (current_root->left) // case 2
                current_root = current_root->left;
            else { // case 3
                current_root->left = new_node;
                break;
            }
        } else {
            if (current_root->right) // case 4
                current_root = current_root->right;
            else { // case 5
```

```
        current_root->right = new_node;
        break;
    }
}
}
return root;
}
```

---

[Contest Calendar](#) | [Interview Prep](#) | [Blog](#) | [Scoring](#) | [Environment](#) | [FAQ](#) | [About Us](#) | [Support](#) | [Careers](#) | [Terms Of Service](#) | [Privacy Policy](#) | [Request a Feature](#)

[All Contests](#) > [ABCS 18: Trees](#) > [Tree: Level Order Traversal](#)

# Tree: Level Order Traversal

locked

by [vatsalchanana](#)

Problem

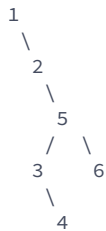
Submissions

Leaderboard

Discussions

Editorial

You are given a pointer to the root of a binary tree. You need to print the level order traversal of this tree. In level order traversal, we visit the nodes level by level from left to right. You only have to complete the function. For example:



For the above tree, the level order traversal is 1 -> 2 -> 5 -> 3 -> 6 -> 4.

## Input Format

You are given a function,

```
void levelOrder(Node * root) {  
  
}
```

## Constraints

$1 \leq \text{Nodes in the tree} \leq 500$

## Output Format

Print the values in a single line separated by a space.

## Sample Input



## Sample Output

1 2 5 3 6 4

## Explanation

We need to print the nodes level by level. We process each level from left to right.

Level Order Traversal: 1 -> 2 -> 5 -> 3 -> 6 -> 4.

[f](#) [t](#) [in](#)

Submissions: 39

Max Score: 20

Difficulty: Easy


Rate This Challenge:

☆☆☆☆☆

[More](#)Current Buffer (saved locally, editable)  

C



```
1 ▶ #include 
5
6 ▼ struct node {
7
8     int data;
9     struct node *left;
10    struct node *right;
11
12 };
13
14 ▼ struct node* insert( struct node* root, int data ) {
15
16     if(root == NULL) {
17
18         struct node* node = (struct node*)malloc(sizeof(struct node));
19
20         node->data = data;
21
22         node->left = NULL;
23         node->right = NULL;
24         return node;
25
26     } else {
27
28         struct node* cur;
29
30         if(data <= root->data) {
31             cur = insert(root->left, data);
32             root->left = cur;
33         } else {
34             cur = insert(root->right, data);
35             root->right = cur;
36         }
37
38         return root;
39     }
40 }
41
42 /* you only have to complete the function given below.
43 node is defined as
44 struct node {
45
46     int data;
47     struct node *left;
48     struct node *right;
```

```
49
50 };
51
52 */
53 void levelOrder( struct node *root) {
54
55 }
56
57 int main() {
58
59     struct node* root = NULL;
60
61     int t;
62     int data;
63
64     scanf("%d", &t);
65
66     while(t-- > 0) {
67         scanf("%d", &data);
68         root = insert(root, data);
69     }
70
71     levelOrder(root);
72     return 0;
73 }
74
```

Line: 17 Col: 1

 [Upload Code as File](#) ☐ Test against custom input

Run Code

Submit Code





# Tree: Huffman Decoding

locked



by vatsalchanana

Problem

Submissions

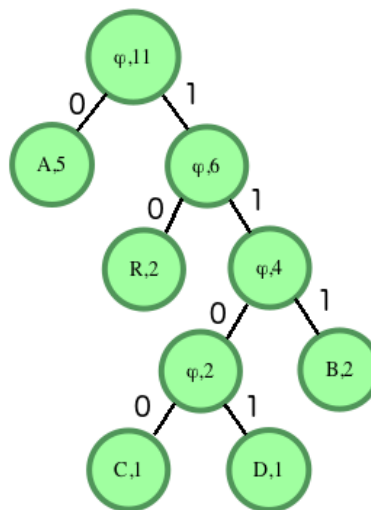
Leaderboard

Discussions

Editorial

**Huffman coding** assigns variable length codewords to fixed length input characters based on their frequencies. More frequent characters are assigned shorter codewords and less frequent characters are assigned longer codewords. All edges along the path to a character contain a code digit. If they are on the left side of the tree, they will be a *0* (zero). If on the right, they'll be a *1* (one). Only the leaves will contain a letter and its frequency count. All other nodes will contain a null instead of a character, and the count of the frequency of all of it and its descendant characters.

For instance, consider the string *ABRACADABRA*. There are a total of **11** characters in the string. This number should match the count in the ultimately determined root of the tree. Our frequencies are **A = 5, B = 2, R = 2, C = 1** and **D = 1**. The two smallest frequencies are for **C** and **D**, both equal to **1**, so we'll create a tree with them. The root node will contain the sum of the counts of its descendants, in this case **1 + 1 = 2**. The left node will be the first character encountered, **C**, and the right will contain **D**. Next we have **3** items with a character count of **2**: the tree we just created, the character **B** and the character **R**. The tree came first, so it will go on the left of our new root node. **B** will go on the right. Repeat until the tree is complete, then fill in the **1**'s and **0**'s for the edges. The finished graph looks like:



Input characters are only present in the leaves. Internal nodes have a character value of  $\phi$  (NULL). We can determine that our values for characters are:

```
A - 0
B - 111
C - 1100
D - 1101
R - 10
```

Our Huffman encoded string is:

```

A B   R A C   A D   A B   R A
0 111 10 0 1100 0 1101 0 111 10 0
or
01111001100011010111100

```

To avoid ambiguity, Huffman encoding is a prefix free encoding technique. No codeword appears as a prefix of any other codeword.

To decode the encoded string, follow the zeros and ones to a leaf and return the character there.

You are given pointer to the root of the Huffman tree and a binary coded string to decode. You need to print the decoded string.

### Function Description

Complete the function `decode_huff` in the editor below. It must return the decoded string.

`decode_huff` has the following parameters:

- *root*: a reference to the root node of the Huffman tree
- *s*: a Huffman encoded string

### Input Format

There is one line of input containing the plain string, *s*. Background code creates the Huffman tree then passes the head node and the encoded string to the function.

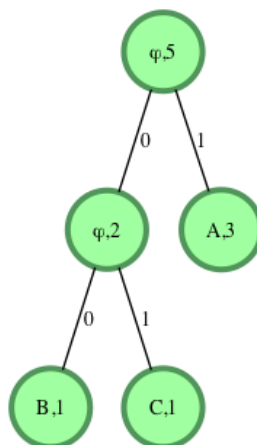
### Constraints

$$1 \leq |s| \leq 25$$

### Output Format

Output the decoded string on a single line.

### Sample Input



```
s="1001011"
```

### Sample Output

```
ABACA
```

### Explanation

```

S="1001011"
Processing the string from left to right.

```

S[0]='1' : we move to the right child of the root. We encounter a leaf node with value 'A'. We add 'A' to the decoded string.  
We move back to the root.

S[1]='0' : we move to the left child.  
S[2]='0' : we move to the left child. We encounter a leaf node with value 'B'. We add 'B' to the decoded string.  
We move back to the root.

S[3] = '1' : we move to the right child of the root. We encounter a leaf node with value 'A'. We add 'A' to the decoded string.  
We move back to the root.

S[4]='0' : we move to the left child.  
S[5]='1' : we move to the right child. We encounter a leaf node with value C'. We add 'C' to the decoded string.  
We move back to the root.

S[6] = '1' : we move to the right child of the root. We encounter a leaf node with value 'A'. We add 'A' to the decoded string.  
We move back to the root.

Decoded String = "ABACA"

[f](#) [t](#) [in](#)


Submissions: 24

Max Score: 20

Difficulty: Medium

Rate This Challenge:

☆☆☆☆☆

[More](#)Current Buffer (saved locally, editable)  

C++



```
1 //
2 // main.cpp
3 // Huffman
4 //
5 // Created by Vatsal Chanana
6
7 #include<bits/stdc++.h>
8 using namespace std;
9
10 typedef struct node {
11     int freq;
12     char data;
13     node * left;
14     node * right;
15 } node;
16
17 struct deref:public binary_function<node*, node*, bool> {
18     bool operator()(const node * a, const node * b) const {
19         return a->freq > b->freq;
20     }
21 };
22
23 typedef priority_queue<node *, vector<node*>, deref> spq;
24
25 node * huffman_hidden(string s) {
26
27     spq pq;
28     vector<int> count(256,0);
29
30     for(int i = 0; i < s.length(); i++ ) {
31         count[s[i]]++;
32     }
```

```

33
34 ▼ for(int i=0; i < 256; i++) {
35
36     node * n_node = new node;
37     n_node->left = NULL;
38     n_node->right = NULL;
39     n_node->data = (char)i;
40 ▼     n_node->freq = count[i];
41
42 ▼     if( count[i] != 0 )
43         pq.push(n_node);
44
45 }
46
47 ▼ while( pq.size() != 1 ) {
48
49     node * left = pq.top();
50     pq.pop();
51     node * right = pq.top();
52     pq.pop();
53     node * comb = new node;
54     comb->freq = left->freq + right->freq;
55     comb->data = '\0';
56     comb->left = left;
57     comb->right = right;
58     pq.push(comb);
59
60 }
61
62     return pq.top();
63 }
64 }
65
66 ▼ void print_codes_hidden(node * root, string code, map<char, string>&mp) {
67
68     if(root == NULL)
69         return;
70
71 ▼     if(root->data != '\0') {
72 ▼         mp[root->data] = code;
73     }
74
75     print_codes_hidden(root->left, code+'0', mp);
76     print_codes_hidden(root->right, code+'1', mp);
77
78 }
79
80 /*
81 The structure of the node is
82
83 typedef struct node {
84     int freq;
85     char data;
86     node * left;
87     node * right;
88
89 } node;
90
91 */
92
93
94 ▼ void decode_huff(node * root, string s) {
95
96 }
97
98 int main() {↵}

```



# Swap Nodes [Algo]

locked



by abhiranjan

Problem

Submissions

Leaderboard

Discussions

Editorial

A binary tree is a tree which is characterized by one of the following properties:

- It can be empty (null).
- It contains a root node only.
- It contains a root node with a left subtree, a right subtree, or both. These subtrees are also binary trees.

*In-order* traversal is performed as

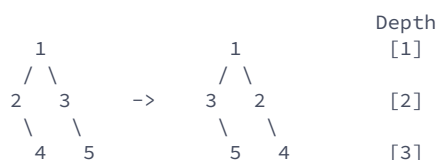
1. Traverse the left subtree.
2. Visit root.
3. Traverse the right subtree.

For this in-order traversal, start from the left child of the root node and keep exploring the left subtree until you reach a leaf. When you reach a leaf, back up to its parent, check for a right child and visit it if there is one. If there is not a child, you've explored its left and right subtrees fully. If there is a right child, traverse its left subtree then its right in the same manner. Keep doing this until you have traversed the entire tree. You will only store the values of a node as you visit when one of the following is true:

- it is the first node visited, the first time visited
- it is a leaf, should only be visited once
- all of its subtrees have been explored, should only be visited once while this is true
- it is the root of the tree, the first time visited

**Swapping:** Swapping subtrees of a node means that if initially node has left subtree  $L$  and right subtree  $R$ , then after swapping, the left subtree will be  $R$  and the right subtree,  $L$ .

For example, in the following tree, we swap children of node 1.



In-order traversal of left tree is 2 4 1 3 5 and of right tree is 3 5 1 2 4.

**Swap operation:**

We define depth of a node as follows:

- The root node is at depth 1.
- If the depth of the parent node is  $d$ , then the depth of current node will be  $d+1$ .

Given a tree and an integer,  $k$ , in one operation, we need to swap the subtrees of all the nodes at each depth  $h$ , where  $h \in [k, 2k, 3k, \dots]$ . In other words, if  $h$  is a multiple of  $k$ , swap the left and right subtrees of that level.

You are given a tree of  $n$  nodes where nodes are indexed from  $[1..n]$  and it is rooted at 1. You have to perform  $t$  swap operations on it, and after each swap operation print the in-order traversal of the current state of the tree.

### Function Description

Complete the `swapNodes` function in the editor below. It should return a two-dimensional array where each element is an array of integers representing the node indices of an in-order traversal after a swap operation.

`swapNodes` has the following parameter(s):

- *indexes*: an array of integers representing index values of each `node[i]`, beginning with `node[1]`, the first element, as the root.
- *queries*: an array of integers, each representing a  $k$  value.

### Input Format

The first line contains  $n$ , number of nodes in the tree.

Each of the next  $n$  lines contains two integers,  $a$   $b$ , where  $a$  is the index of left child, and  $b$  is the index of right child of  $i^{th}$  node.

**Note:**  $-1$  is used to represent a null node.

The next line contains an integer,  $t$ , the size of *queries*.

Each of the next  $t$  lines contains an integer `queries[i]`, each being a value  $k$ .

### Output Format

For each  $k$ , perform the swap operation and store the indices of your in-order traversal to your result array. After all swap operations have been performed, return your result array for printing.

### Constraints

- $1 \leq n \leq 1024$
- $1 \leq t \leq 100$
- $1 \leq k \leq n$
- Either  $a = -1$  or  $2 \leq a \leq n$
- Either  $b = -1$  or  $2 \leq b \leq n$
- The index of a non-null child will always be greater than that of its parent.

### Sample Input 0

```
3
2 3
-1 -1
-1 -1
2
1
1
```

### Sample Output 0

```
3 1 2
2 1 3
```

### Explanation 0

As nodes 2 and 3 have no children, swapping will not have any effect on them. We only have to swap the child nodes of the root node.



**Note:** [s] indicates that a swap operation is done at this depth.

### Sample Input 1

```

5
2 3
-1 4
-1 5
-1 -1
-1 -1
1
2

```

### Sample Output 1

```

4 2 1 5 3

```

### Explanation 1

Swapping child nodes of node 2 and 3 we get



### Sample Input 2

```

11
2 3
4 -1
5 -1
6 -1
7 8
-1 9
-1 -1
10 11
-1 -1
-1 -1
-1 -1
2
2
4

```

### Sample Output 2

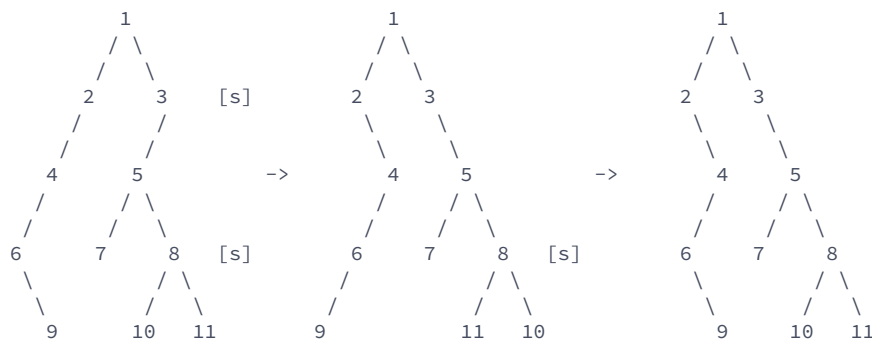
```

2 9 6 4 1 3 7 5 11 8 10
2 6 9 4 1 3 7 5 10 8 11

```

### Explanation 2

Here we perform swap operations at the nodes whose depth is either 2 or 4 for  $K = 2$  and then at nodes whose depth is 4 for  $K = 4$ .


[f](#) [t](#) [in](#)

Submissions: 4

Max Score: 40

Difficulty: Medium

Rate This Challenge:

☆☆☆☆☆

[More](#)

Current Buffer (saved locally, editable)

C



```

1 #include <assert.h>
2 #include <limits.h>
3 #include <math.h>
4 #include <stdbool.h>
5 #include <stdio.h>
6 #include <stdlib.h>
7 #include <string.h>
8
9 char* readline();
10 char** split_string(char*);
11
12 /*
13  * Complete the swapNodes function below.
14  */
15
16 /*
17  * Please store the row size of the 2D integer array to be returned in result_rows pointer and
18  * column size in result_columns pointer. For example,
19  * int a[2][3] = {
20  *     {1, 2, 3},
21  *     {4, 5, 6}
22  * };
23  * *result_rows = 2;
24  * *result_columns = 3;
25  * return a;
26  */
27
28
29 int** swapNodes(int indexes_rows, int indexes_columns, int** indexes, int queries_count, int*
queries, int* result_rows, int* result_columns) {
30     /*
31      * Write your code here.
32      */
33

```



```
34 }
35
36 int main()
37 {
38     FILE* fptr = fopen(getenv("OUTPUT_PATH"), "w");
39
40     char* n_endptr;
41     char* n_str = readline();
42     int n = strtol(n_str, &n_endptr, 10);
43
44     if (n_endptr == n_str || *n_endptr != '\0') { exit(EXIT_FAILURE); }
45
46     int** indexes = malloc(n * sizeof(int*));
47
48     for (int indexes_row_itr = 0; indexes_row_itr < n; indexes_row_itr++) {
49         indexes[indexes_row_itr] = malloc(2 * (sizeof(int)));
50
51         char** indexes_item_temp = split_string(readline());
52
53         for (int indexes_column_itr = 0; indexes_column_itr < 2; indexes_column_itr++) {
54             char* indexes_item_endptr;
55             char* indexes_item_str = indexes_item_temp[indexes_column_itr];
56             int indexes_item = strtol(indexes_item_str, &indexes_item_endptr, 10);
57
58             if (indexes_item_endptr == indexes_item_str || *indexes_item_endptr != '\0') {
59                 exit(EXIT_FAILURE); }
60
61             indexes[indexes_row_itr][indexes_column_itr] = indexes_item;
62         }
63
64         char* queries_count_endptr;
65         char* queries_count_str = readline();
66         int queries_count = strtol(queries_count_str, &queries_count_endptr, 10);
67
68         if (queries_count_endptr == queries_count_str || *queries_count_endptr != '\0') {
69             exit(EXIT_FAILURE); }
70
71         int queries[queries_count];
72
73         for (int queries_itr = 0; queries_itr < queries_count; queries_itr++) {
74             char* queries_item_endptr;
75             char* queries_item_str = readline();
76             int queries_item = strtol(queries_item_str, &queries_item_endptr, 10);
77
78             if (queries_item_endptr == queries_item_str || *queries_item_endptr != '\0') {
79                 exit(EXIT_FAILURE); }
80
81             queries[queries_itr] = queries_item;
82         }
83
84         int result_rows;
85         int result_columns;
86         int** result = swapNodes(indexes_rows, indexes_columns, indexes, queries_count, queries,
87                                 &result_rows, &result_columns);
88
89         for (int result_row_itr = 0; result_row_itr < result_rows; result_row_itr++) {
90             for (int result_column_itr = 0; result_column_itr < result_columns; result_column_itr++) {
91                 fprintf(fptr, "%d", result[result_row_itr][result_column_itr]);
92
93                 if (result_column_itr != result_columns - 1) {
94                     fprintf(fptr, " ");
95                 }
96
97                 if (result_row_itr != result_rows - 1) {
```

```
96         fprintf(fp, "\n");
97     }
98 }
99
100 fprintf(fp, "\n");
101
102 fclose(fp);
103
104 return 0;
105 }
106
107 char* readline() {
108     size_t alloc_length = 1024;
109     size_t data_length = 0;
110     char* data = malloc(alloc_length);
111
112     while (true) {
113         char* cursor = data + data_length;
114         char* line = fgets(cursor, alloc_length - data_length, stdin);
115
116         if (!line) { break; }
117
118         data_length += strlen(cursor);
119
120         if (data_length < alloc_length - 1 || data[data_length - 1] == '\n') { break; }
121
122         size_t new_length = alloc_length << 1;
123         data = realloc(data, new_length);
124
125         if (!data) { break; }
126
127         alloc_length = new_length;
128     }
129
130     if (data[data_length - 1] == '\n') {
131         data[data_length - 1] = '\0';
132     }
133
134     data = realloc(data, data_length);
135
136     return data;
137 }
138
139 char** split_string(char* str) {
140     char** splits = NULL;
141     char* token = strtok(str, " ");
142
143     int spaces = 0;
144
145     while (token) {
146         splits = realloc(splits, sizeof(char*) * ++spaces);
147         if (!splits) {
148             return splits;
149         }
150
151         splits[spaces - 1] = token;
152
153         token = strtok(NULL, " ");
154     }
155
156     return splits;
157 }
158
```

Line: 1 Col: 1