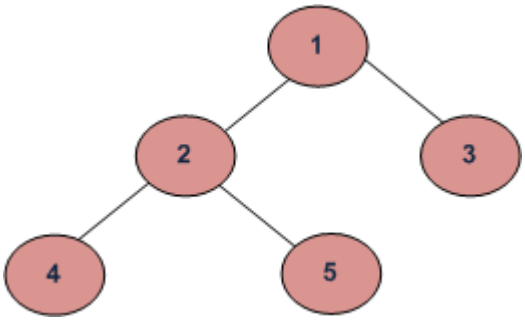
1. Write a program to Calculate Size of a tree | Recursion

Size of a tree is the number of elements present in the tree. Size of the below tree is 5.



Example Tree

Size() function recursively calculates the size of a tree. It works as follows:

Size of a tree = Size of left subtree + 1 + Size of right subtree.

Algorithm:

```
size(tree)

1. If tree is empty then return 0

2. Else

    (a) Get the size of left subtree recursively  i.e., call

        size( tree->left-subtree)

    (a) Get the size of right subtree recursively  i.e., call

        size( tree->right-subtree)

    (c) Calculate size of the tree as following:

          tree_size  =  size(left-subtree) + size(right-

                              subtree) + 1

    (d) Return tree_size
```

C++

```
// A recursive C++ program to
// calculate the size of the tree
```

```cpp
#include <bits/stdc++.h>
using namespace std;

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

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

    return(Node);
}

/* Computes the number of nodes in a tree. */
int size(node* node)
{
    if (node == NULL)
        return 0;
    else
        return(size(node->left) + 1 + size(node->right));
}

/* Driver code*/
int main()
{
    node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);

    cout << "Size of the tree is " << size(root);
    return 0;
}

// This code is contributed by rathbhupendra
```

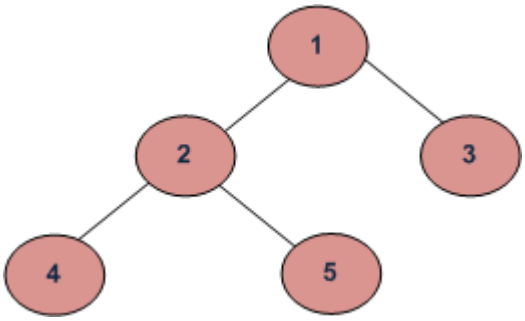
Output:

```
Size of the tree is 5
```

Time & Space Complexities: Since this program is similar to traversal of tree, time and space complexities will be same as Tree traversa

2. Iterative program to Calculate Size of a tree

Size of a tree is the number of elements present in the tree. Size of the below tree is 5.



Approach

The idea is to use Level Order Traversing

```
1) Create an empty queue q

2) temp_node = root /*start from root*/

3) Loop while temp_node is not NULL

    a) Enqueue temp_node's children (first left then right children) to
q

    b) Increase count with every enqueuing.

    c) Dequeue a node from q and assign it's value to temp_node
```

```
// C++ program to print size of tree in iterative
#include<iostream>
#include<queue>
using namespace std;

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

// A utility function to
// create a new Binary Tree Node
Node *newNode(int data)
{
```

```
    Node *temp = new Node;
    temp->data = data;
    temp->left = NULL;
    temp->right = NULL;

    return temp;
}

// return size of tree
int sizeoftree(Node *root)
{

    // if tree is empty it will
    // return 0
    if(root == NULL)
        return 0;


    // Using level order Traversal.
    queue<Node *> q;
    int count = 1;
    q.push(root);

    while(!q.empty())
    {
        Node *temp = q.front();

        if(temp->left)
        {
            // Enqueue left child
            q.push(temp->left);

            // Increment count
            count++;
        }

        if(temp->right)
        {
            // Enqueue right child
            q.push(temp->right);

            // Increment count
            count++;
        }
        q.pop();
    }

    return count;
}
```

```
// Driver Code
int main()
{
    Node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);

    cout << "Size of the tree is " <<
        sizeoftree(root) << endl;
    return 0;
}

// This code is contributed by SHAKEELMOHAMMAD
```

Output:

```
Size of the tree is 5
```

Time Complexity: O(n)

Auxiliary Space : O(level_max) where level max is maximum number of node in any leve

3. Write Code to Determine if Two Trees are Identical

Two trees are identical when they have same data and arrangement of data is also same.

To identify if two trees are identical, we need to traverse both trees simultaneously, and while traversing we need to compare data and children of the trees.

Algorithm:

```
sameTree(tree1, tree2)

1. If both trees are empty then return 1.

2. Else If both trees are non -empty

    (a) Check data of the root nodes (tree1->data ==  tree2->data)

    (b) Check left subtrees recursively  i.e., call sameTree(

        tree1->left_subtree, tree2->left_subtree)

    (c) Check right subtrees recursively  i.e., call sameTree(

        tree1->right_subtree, tree2->right_subtree)

    (d) If a,b and c are true then return 1.

3  Else return 0 (one is empty and other is not)
```

```
// C++ program to see if two trees are identical
#include <bits/stdc++.h>
using namespace std;

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

/* Helper function that allocates a new node with the
given data and NULL left and right pointers. */
```

```
node* newNode(int data)
{
    node* Node = new node();
    Node->data = data;
    Node->left = NULL;
    Node->right = NULL;

    return(Node);
}

/* Given two trees, return true if they are
structurally identical */
int identicalTrees(node* a, node* b)
{
    /*1. both empty */
    if (a == NULL && b == NULL)
        return 1;

    /* 2. both non-empty -> compare them */
    if (a != NULL && b != NULL)
    {
        return
        (
            a->data == b->data &&
            identicalTrees(a->left, b->left) &&
            identicalTrees(a->right, b->right)
        );
    }

    /* 3. one empty, one not -> false */
    return 0;
}

/* Driver code*/
int main()
{
    node *root1 = newNode(1);
    node *root2 = newNode(1);
    root1->left = newNode(2);
    root1->right = newNode(3);
    root1->left->left = newNode(4);
    root1->left->right = newNode(5);

    root2->left = newNode(2);
    root2->right = newNode(3);
    root2->left->left = newNode(4);
    root2->left->right = newNode(5);

    if(identicalTrees(root1, root2))
```

```
        cout << "Both tree are identical.";
    else
        cout << "Trees are not identical.";

return 0;
}

// This code is contributed by rathbhupendra
```

Output:

```
Both trees are identical
```

Time Complexity:

Complexity of the identicalTree() will be according to the tree with lesser number of nodes. Let number of nodes in two trees be m and n then complexity of sameTree() is O(m) where m < n

4. Iterative function to check if two trees are identical

Two trees are identical when they have same data and arrangement of data is also same. To identify if two trees are identical, we need to traverse both trees simultaneously, and while traversing we need to compare data and children of the trees.

Examples:

```
Input : Roots of below trees

   10           10

  /   \         /

 5     6       5

Output : false


Input : Roots of below trees

   10            10

  /   \         /   \

 5     6       5     6

Output : true
```

We have discussed recursive solution here. In this article iterative solution is discussed.

The idea is to use level order traversal. We traverse both trees simultaneously and compare the data whenever we dequeue and item from queue. Below is the implementation of the idea.

```
/* Iterative C++ program to check if two */
#include <bits/stdc++.h>
using namespace std;

// A Binary Tree Node
struct Node
{
    int data;
    struct Node *left, *right;
```

```
};

// Iterative method to find height of Binary Tree
bool areIdentical(Node *root1, Node *root2)
{
    // Return true if both trees are empty
    if (root1  && root2) return true;

    // Return false if one is empty and other is not
    if (root1 || root2)  return false;

    // Create an empty queues for simultaneous traversals
    queue<Node *> q1, q2;

    // Enqueue Roots of trees in respective queues
    q1.push(root1);
    q2.push(root2);

    while (!q1.empty() && !q2.empty())
    {
        // Get front nodes and compare them
        Node *n1 = q1.front();
        Node *n2 = q2.front();

        if (n1->data != n2->data)
           return false;

        // Remove front nodes from queues
        q1.pop(), q2.pop();

        /* Enqueue left children of both nodes */
        if (n1->left && n2->left)
        {
            q1.push(n1->left);
            q2.push(n2->left);
        }

        // If one left child is empty and other is not
        else if (n1->left || n2->left)
            return false;

        // Right child code (Similar to left child code)
        if (n1->right && n2->right)
        {
            q1.push(n1->right);
            q2.push(n2->right);
        }
        else if (n1->right || n2->right)
            return false;
```

```
    }

    return true;
}

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

// Driver program to test above functions
int main()
{
    Node *root1 = newNode(1);
    root1->left = newNode(2);
    root1->right = newNode(3);
    root1->left->left = newNode(4);
    root1->left->right = newNode(5);

    Node *root2 = newNode(1);
    root2->left = newNode(2);
    root2->right = newNode(3);
    root2->left->left = newNode(4);
    root2->left->right = newNode(5);

    areIdentical(root1, root2)? cout << "Yes"
                              : cout << "No";
    return 0;
}
```

Output:

```
Yes
```

Time complexity of above solution is O(n + m) where m and n are number of nodes in two trees

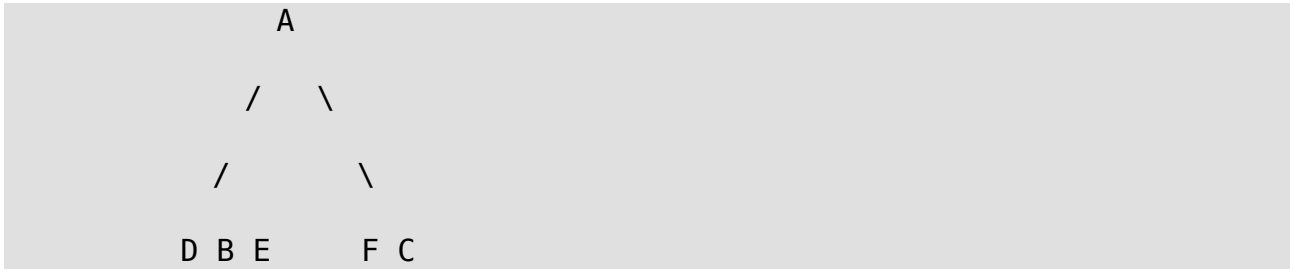
5. Construct Tree from given Inorder and Preorder traversals

Let us consider the below traversals:

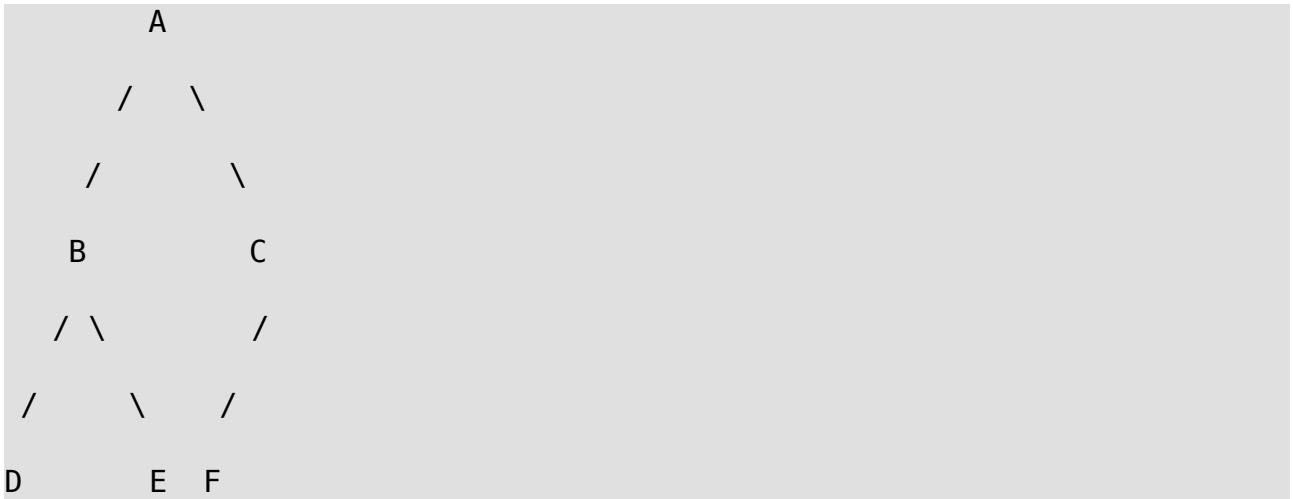
Inorder sequence: D B E A F C

Preorder sequence: A B D E C F

In a Preorder sequence, leftmost element is the root of the tree. So we know ‘A’ is root for given sequences. By searching ‘A’ in Inorder sequence, we can find out all elements on left side of ‘A’ are in left subtree and elements on right are in right subtree. So we know below structure now.



We recursively follow above steps and get the following tree.



Algorithm: buildTree()

1) Pick an element from Preorder. Increment a Preorder Index Variable (preIndex in below code) to pick next element in next recursive call.

2) Create a new tree node tNode with the data as picked element.

3) Find the picked element’s index in Inorder. Let the index be inIndex.

4) Call buildTree for elements before inIndex and make the built tree as left subtree of tNode.

5) Call buildTree for elements after inIndex and make the built tree as right subtree of tNode.

6) return tNode.

Thanks to Rohini and Tushar for suggesting the code.

```
/* C++ program to construct tree using
inorder and preorder traversals */
#include <bits/stdc++.h>
using namespace std;

/* A binary tree node has data, pointer to left child
and a pointer to right child */
class node
{
    public:
    char data;
    node* left;
    node* right;
};

/* Prototypes for utility functions */
int search(char arr[], int strt, int end, char value);
node* newNode(char data);

/* Recursive function to construct binary
of size len from Inorder traversal in[]
and Preorder traversal pre[]. Initial values
of inStrt and inEnd should be 0 and len -1.
The function doesn't do any error checking
for cases where inorder and preorder do not
form a tree */
node* buildTree(char in[], char pre[], int inStrt, int inEnd)
{
    static int preIndex = 0;

    if (inStrt > inEnd)
        return NULL;

    /* Pick current node from Preorder
    traversal using preIndex
    and increment preIndex */
```

```
    node* tNode = newNode(pre[preIndex++]);

    /* If this node has no children then return */
    if (inStrt == inEnd)
        return tNode;

    /* Else find the index of this node in Inorder traversal */
    int inIndex = search(in, inStrt, inEnd, tNode->data);

    /* Using index in Inorder traversal, construct left and
    right subtress */
    tNode->left = buildTree(in, pre, inStrt, inIndex - 1);
    tNode->right = buildTree(in, pre, inIndex + 1, inEnd);

    return tNode;
}

/* UTILITY FUNCTIONS */
/* Function to find index of value in arr[start...end]
The function assumes that value is present in in[] */
int search(char arr[], int strt, int end, char value)
{
    int i;
    for (i = strt; i <= end; i++)
    {
        if (arr[i] == value)
            return i;
    }
}

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

    return (Node);
}

/* This funtcion is here just to test buildTree() */
void printInorder(node* node)
{
    if (node == NULL)
        return;

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

```
    /* then print the data of node */
    cout<<node->data<<" ";

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

/* Driver code */
int main()
{
    char in[] = { 'D', 'B', 'E', 'A', 'F', 'C' };
    char pre[] = { 'A', 'B', 'D', 'E', 'C', 'F' };
    int len = sizeof(in) / sizeof(in[0]);
    node* root = buildTree(in, pre, 0, len - 1);

    /* Let us test the built tree by
    printing Insorder traversal */
    cout << "Inorder traversal of the constructed tree is \n";
    printInorder(root);
}

// This is code is contributed by rathbhupendra
```

Output:

```
Inorder traversal of the constructed tree is

D B E A F C
```

Time Complexity: O(n^2). Worst case occurs when tree is left skewed. Example Preorder and Inorder traversals for worst case are {A, B, C, D} and {D, C, B, A}.

Efficient Approach :

We can optimize the above solution using hashing (unordered_map in C++ or HashMap in Java). We store indexes of inorder traversal in a hash table. So that search can be done O(1) time.

```
/* C++ program to construct tree using inorder
   and preorder traversals */
#include <bits/stdc++.h>
using namespace std;

/* A binary tree node has data, pointer to left child
and a pointer to right child */
```

```
struct Node {
    char data;
    struct Node* left;
    struct Node* right;
};

struct Node* newNode(char data)
{
    struct Node* node = new Node;
    node->data = data;
    node->left = node->right = NULL;
    return (node);
}

/* Recursive function to construct binary of size
len from Inorder traversal in[] and Preorder traversal
pre[]. Initial values of inStrt and inEnd should be
0 and len -1. The function doesn't do any error
checking for cases where inorder and preorder
do not form a tree */
struct Node* buildTree(char in[], char pre[], int inStrt,
                       int inEnd, unordered_map<char, int>& mp)
{
    static int preIndex = 0;

    if (inStrt > inEnd)
        return NULL;

    /* Pick current node from Preorder traversal using preIndex
    and increment preIndex */
    char curr = pre[preIndex++];
    struct Node* tNode = newNode(curr);

    /* If this node has no children then return */
    if (inStrt == inEnd)
        return tNode;

    /* Else find the index of this node in Inorder traversal */
    int inIndex = mp[curr];

    /* Using index in Inorder traversal, construct left and
    right subtress */
    tNode->left = buildTree(in, pre, inStrt, inIndex - 1, mp);
    tNode->right = buildTree(in, pre, inIndex + 1, inEnd, mp);

    return tNode;
}

// This function mainly creates an unordered_map, then
// calls buildTree()
```

```
struct Node* buldTreeWrap(char in[], char pre[], int len)
{
    // Store indexes of all items so that we
    // we can quickly find later
    unordered_map<char, int> mp;
    for (int i = 0; i < len; i++)
        mp[in[i]] = i;

    return buildTree(in, pre, 0, len - 1, mp);
}

/* This funtcion is here just to test buildTree() */
void printInorder(struct Node* node)
{
    if (node == NULL)
        return;
    printInorder(node->left);
    printf("%c ", node->data);
    printInorder(node->right);
}

/* Driver program to test above functions */
int main()
{
    char in[] = { 'D', 'B', 'E', 'A', 'F', 'C' };
    char pre[] = { 'A', 'B', 'D', 'E', 'C', 'F' };
    int len = sizeof(in) / sizeof(in[0]);

    struct Node* root = buldTreeWrap(in, pre, len);

    /* Let us test the built tree by printing
      Insorder traversal */
    printf("Inorder traversal of the constructed tree is \n");
    printInorder(root);
}
```

Output:

```
Inorder traversal of the constructed tree is

D B E A F C
```

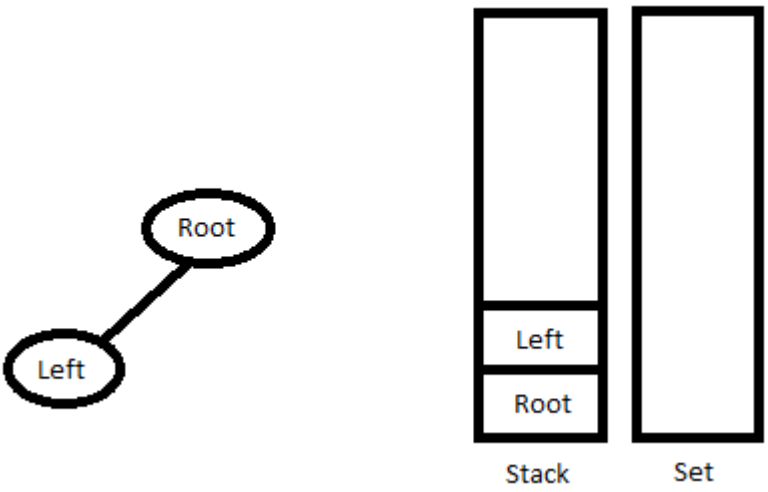
Time Complexity : O(n)

Another approach :

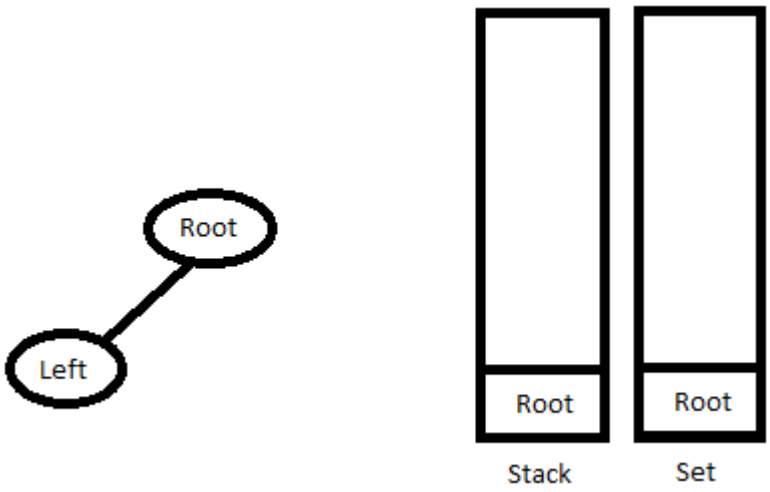
Use the fact that InOrder traversal is Left-Root-Right and PreOrder traversal is

Root-Left-Right. Also, first node in the PreOrder traversal is always the root node and the first node in the InOrder traversal is the leftmost node in the tree. Maintain two data-structures : Stack (to store the path we visited while traversing PreOrder array) and Set (to maintain the node in which the next right subtree is expected).

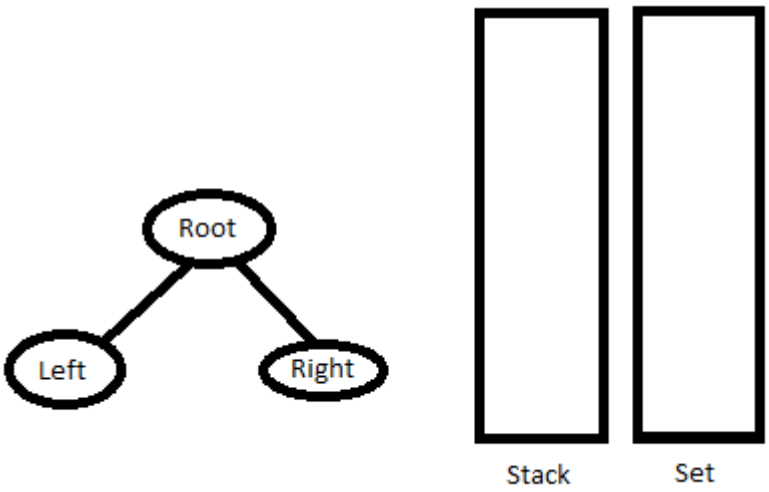
1. Do below until you reach the leftmost node.
Keep creating the nodes from PreOrder traversal
If the stack’s topmost element is not in the set, link the created node to the left child of stack’s topmost element (if any), without popping the element.
Else link the created node to the right child of stack’s topmost element.
Remove the stack’s topmost element from the set and the stack.
Push the node to a stack.



2. Keep popping the nodes from the stack until either the stack is empty, or the topmost element of stack compares to the current element of InOrder traversal. Once the loop is over, push the last node back into the stack and into the set.



3. Goto Step 1.




```
// C++ program to construct a tree using
// inorder and preorder traversal
#include<bits/stdc++.h>
using namespace std;

class TreeNode
{
    public:
    int val;
    TreeNode* left;
    TreeNode* right;
    TreeNode(int x) { val = x; }
};

set<TreeNode*> s;
stack<TreeNode*> st;
```

```
// Function to build tree using given traversal
TreeNode* buildTree(int preorder[], int inorder[],int n)
{

    TreeNode* root = NULL;
    for (int pre = 0, in = 0; pre < n;)
    {

        TreeNode* node = NULL;
        do
        {
            node = new TreeNode(preorder[pre]);
            if (root == NULL)
            {
                root = node;
            }
            if (st.size() > 0)
            {
                if (s.find(st.top()) != s.end())
                {
                    s.erase(st.top());
                    st.top()->right = node;
                    st.pop();
                }
                else
                {
                    st.top()->left = node;
                }
            }
            st.push(node);
        } while (preorder[pre++] != inorder[in] && pre < n);

        node = NULL;
        while (st.size() > 0 && in < n &&
                st.top()->val == inorder[in])
        {
            node = st.top();
            st.pop();
            in++;
        }

        if (node != NULL)
        {
            s.insert(node);
            st.push(node);
        }
    }

    return root;
```

```
}

// Function to print tree in Inorder
void printInorder(TreeNode* node)
{
    if (node == NULL)
        return;

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

    /* then print the data of node */
    cout << node->val << " ";

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

// Driver code
int main()
{
    int in[] = { 9, 8, 4, 2, 10, 5, 10, 1, 6, 3, 13, 12, 7 };
    int pre[] = { 1, 2, 4, 8, 9, 5, 10, 10, 3, 6, 7, 12, 13 };
    int len = sizeof(in)/sizeof(int);

    TreeNode *root = buildTree(pre, in, len);

    printInorder(root);
    return 0;
}

// This code is contributed by Arnab Kundu
```

Output:

```
9 8 4 2 10 5 10 1 6 3 13 12 7
```

6. Construct a Binary Tree from Postorder and Inorder

Given Postorder and Inorder traversals, construct the tree.

Examples:

```
Input :

in[]   = {2, 1, 3}

post[] = {2, 3, 1}


Output : Root of below tree

     1

   /   \

  2     3




Input :

in[]   = {4, 8, 2, 5, 1, 6, 3, 7}

post[] = {8, 4, 5, 2, 6, 7, 3, 1}


Output : Root of below tree

         1

      /     \

    2         3

  /    \    /   \

 4     5   6    7

   \

     8
```

We have already discussed construction of tree from Inorder and Preorder traversals. The idea is similar.

Let us see the process of constructing tree from in[] = {4, 8, 2, 5, 1, 6, 3, 7} and post[] = {8, 4, 5, 2, 6, 7, 3, 1}

1) We first find the last node in post[]. The last node is “1”, we know this value is root as root always appear in the end of postorder traversal.
2) We search “1” in in[] to find left and right subtrees of root. Everything on left of “1” in in[] is in left subtree and everything on right is in right subtree.

```
        1

      /    \

[4, 8, 2, 5]   [6, 3, 7]
```

3) We recur the above process for following two.
....b) Recur for in[] = {6, 3, 7} and post[] = {6, 7, 3}
.....Make the created tree as right child of root.
....a) Recur for in[] = {4, 8, 2, 5} and post[] = {8, 4, 5, 2}.
.....Make the created tree as left child of root.
Below is the implementation of above idea. One important observation is, we recursively call for right subtree before left subtree as we decrease index of postorder index whenever we create a new node.

```
/* C++ program to construct tree using inorder and
   postorder traversals */
#include <bits/stdc++.h>

using namespace std;

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

// Utility function to create a new node
Node* newNode(int data);
```

```c
/* Prototypes for utility functions */
int search(int arr[], int strt, int end, int value);

/* Recursive function to construct binary of size n
   from  Inorder traversal in[] and Postorder traversal
   post[].  Initial values of inStrt and inEnd should
   be 0 and n -1.  The function doesn't do any error
   checking for cases where inorder and postorder
   do not form a tree */
Node* buildUtil(int in[], int post[], int inStrt,
                int inEnd, int* pIndex)
{
    // Base case
    if (inStrt > inEnd)
        return NULL;

    /* Pick current node from Postorder traversal using
       postIndex and decrement postIndex */
    Node* node = newNode(post[*pIndex]);
    (*pIndex)--;

    /* If this node has no children then return */
    if (inStrt == inEnd)
        return node;

    /* Else find the index of this node in Inorder
       traversal */
    int iIndex = search(in, inStrt, inEnd, node->data);

    /* Using index in Inorder traversal, construct left and
       right subtress */
    node->right = buildUtil(in, post, iIndex + 1, inEnd, pIndex);
    node->left = buildUtil(in, post, inStrt, iIndex - 1, pIndex);

    return node;
}

// This function mainly initializes index of root
// and calls buildUtil()
Node* buildTree(int in[], int post[], int n)
{
    int pIndex = n - 1;
    return buildUtil(in, post, 0, n - 1, &pIndex);
}

/* Function to find index of value in arr[start...end]
   The function assumes that value is postsent in in[] */
int search(int arr[], int strt, int end, int value)
{
```

```
    int i;
    for (i = strt; i <= end; i++) {
        if (arr[i] == value)
            break;
    }
    return i;
}

/* Helper function that allocates a new node */
Node* newNode(int data)
{
    Node* node = (Node*)malloc(sizeof(Node));
    node->data = data;
    node->left = node->right = NULL;
    return (node);
}

/* This funtcion is here just to test  */
void preOrder(Node* node)
{
    if (node == NULL)
        return;
    printf("%d ", node->data);
    preOrder(node->left);
    preOrder(node->right);
}

// Driver code
int main()
{
    int in[] = { 4, 8, 2, 5, 1, 6, 3, 7 };
    int post[] = { 8, 4, 5, 2, 6, 7, 3, 1 };
    int n = sizeof(in) / sizeof(in[0]);

    Node* root = buildTree(in, post, n);

    cout << "Preorder of the constructed tree : \n";
    preOrder(root);

    return 0;
}
```

Output :

```
Preorder of the constructed tree :

1 2 4 8 5 3 6 7
```

Time Complexity : $O(n^2)$

Optimized approach: We can optimize the above solution using hashing (unordered_map in C++ or HashMap in Java). We store indexes of inorder traversal in a hash table. So that search can be done O(1) time.

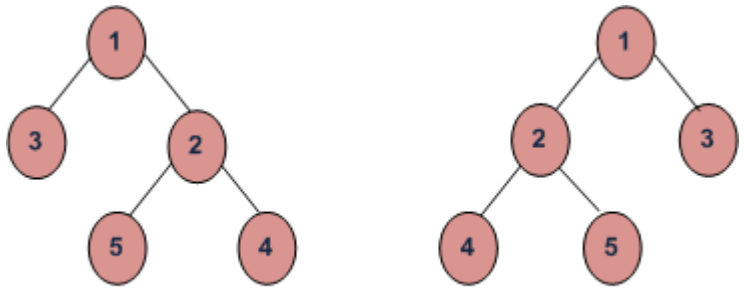
```
/* C++ program to construct tree using inorder and
postorder traversals */
#include <bits/stdc++.h>

using namespace std;

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

// Utility function to create a new node
Node* newNode(int data);

/* Recursive function to construct binary of size n
from Inorder traversal in[] and Postorder traversal
post[]. Initial values of inStrt and inEnd should
be 0 and n -1. The function doesn't do any error
checking for cases where inorder and postorder
do not form a tree */
Node* buildUtil(int in[], int post[], int inStrt,
    int inEnd, int* pIndex, unordered_map<int, int>& mp)
{
    // Base case
    if (inStrt > inEnd)
        return NULL;

    /* Pick current node from Postorder traversal
    using postIndex and decrement postIndex */
    int curr = post[*pIndex];
    Node* node = newNode(curr);
    (*pIndex)--;

    /* If this node has no children then return */
    if (inStrt == inEnd)
        return node;

    /* Else find the index of this node in Inorder
    traversal */
    int iIndex = mp[curr];

    /* Using index in Inorder traversal, construct
    left and right subtress */
```

```
    node->right = buildUtil(in, post, iIndex + 1,
                            inEnd, pIndex, mp);
    node->left = buildUtil(in, post, inStrt,
                           iIndex - 1, pIndex, mp);

    return node;
}

// This function mainly creates an unordered_map, then
// calls buildTreeUtil()
struct Node* buildTree(int in[], int post[], int len)
{
    // Store indexes of all items so that we
    // we can quickly find later
    unordered_map<int, int> mp;
    for (int i = 0; i < len; i++)
        mp[in[i]] = i;

    int index = len - 1; // Index in postorder
    return buildUtil(in, post, 0, len - 1,
                     &index, mp);
}

/* Helper function that allocates a new node */
Node* newNode(int data)
{
    Node* node = (Node*)malloc(sizeof(Node));
    node->data = data;
    node->left = node->right = NULL;
    return (node);
}

/* This funtcion is here just to test */
void preOrder(Node* node)
{
    if (node == NULL)
        return;
    printf("%d ", node->data);
    preOrder(node->left);
    preOrder(node->right);
}

// Driver code
int main()
{
    int in[] = { 4, 8, 2, 5, 1, 6, 3, 7 };
    int post[] = { 8, 4, 5, 2, 6, 7, 3, 1 };
    int n = sizeof(in) / sizeof(in[0]);

    Node* root = buildTree(in, post, n);
```

```
    cout << "Preorder of the constructed tree : \n";
    preOrder(root);

    return 0;
}
```
Time Complexity : O(n)

7. Convert a Binary Tree into its Mirror Tree

Mirror of a Tree: Mirror of a Binary Tree T is another Binary Tree M(T) with left and right children of all non-leaf nodes interchanged.



Mirror Trees

Trees in the above figure are mirror of each other

Method 1 (Recursive)

Algorithm – Mirror(tree):

```
(1)  Call Mirror for left-subtree    i.e., Mirror(left-subtree)

(2)  Call Mirror for right-subtree  i.e., Mirror(right-subtree)

(3)  Swap left and right subtrees.

         temp = left-subtree

         left-subtree = right-subtree

         right-subtree = temp
```

```
// C++ program to convert a binary tree
// to its mirror
```

```cpp
#include<bits/stdc++.h>
using namespace std;

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

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

    return(node);
}


/* Change a tree so that the roles of the left and
    right pointers are swapped at every node.

So the tree...
    4
    / \
    2 5
    / \
1 3

is changed to...
    4
    / \
    5 2
        / \
    3 1
*/
void mirror(struct Node* node)
{
    if (node == NULL)
        return;
    else
    {
        struct Node* temp;
```

```
        /* do the subtrees */
        mirror(node->left);
        mirror(node->right);

        /* swap the pointers in this node */
        temp     = node->left;
        node->left = node->right;
        node->right = temp;
    }
}

/* Helper function to print
Inorder traversal.*/
void inOrder(struct Node* node)
{
    if (node == NULL)
        return;

    inOrder(node->left);
    cout << node->data << " ";
    inOrder(node->right);
}


// Driver Code
int main()
{
    struct Node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);

    /* Print inorder traversal of the input tree */
    cout << "Inorder traversal of the constructed"
         << " tree is" << endl;
    inOrder(root);

    /* Convert tree to its mirror */
    mirror(root);

    /* Print inorder traversal of the mirror tree */
    cout << "\nInorder traversal of the mirror tree"
         << " is \n";
    inOrder(root);

    return 0;
}
```

```
// This code is contributed by Akanksha Rai
```

Output :

```
Inorder traversal of the constructed tree is

4 2 5 1 3

Inorder traversal of the mirror tree is

3 1 5 2 4
```

Time & Space Complexities: This program is similar to traversal of tree space and time complexities will be same as Tree traversal (Please see our Tree Traversal post for details)

Method 2 (Iterative)

The idea is to do queue based level order traversal. While doing traversal, swap left and right children of every node.

```
// Iterative CPP program to convert a Binary
// Tree to its mirror
#include<bits/stdc++.h>
using namespace std;

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

/* Helper function that allocates a new node
   with the given data and NULL left and right
   pointers. */
struct Node* newNode(int data)

{
    struct Node* node = new Node;
    node->data = data;
    node->left = node->right = NULL;
    return(node);
}

/* Change a tree so that the roles of the  left and
```

```
    right pointers are swapped at every node.
 So the tree...
       4
      / \
     2   5
    / \
   1   3

 is changed to...
       4
      / \
     5   2
        / \
       3   1
*/
void mirror(Node* root)
{
    if (root == NULL)
        return;

    queue<Node*> q;
    q.push(root);

    // Do BFS. While doing BFS, keep swapping
    // left and right children
    while (!q.empty())
    {
        // pop top node from queue
        Node* curr = q.front();
        q.pop();

        // swap left child with right child
        swap(curr->left, curr->right);

        // push left and right children
        if (curr->left)
            q.push(curr->left);
        if (curr->right)
            q.push(curr->right);
    }
}


/* Helper function to print Inorder traversal.*/
void inOrder(struct Node* node)
{
    if (node == NULL)
        return;
    inOrder(node->left);
```

```
    cout << node->data << " ";
    inOrder(node->right);
}


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

    /* Print inorder traversal of the input tree */
    cout << "\n Inorder traversal of the"
            " constructed tree is \n";
    inOrder(root);

    /* Convert tree to its mirror */
    mirror(root);

    /* Print inorder traversal of the mirror tree */
    cout << "\n Inorder traversal of the "
           "mirror tree is \n";
    inOrder(root);

    return 0;
}
```

Output:

```
 Inorder traversal of the constructed tree is

4 2 5 1 3

 Inorder traversal of the mirror tree is

3 1 5 2 4
```

8. Create a mirror tree from the given binary tree

Given a binary tree, the task is to create a new binary tree which is a mirror image of the given binary tree.

Examples:

```
Input:
       5
      / \
     3   6
    / \
   2   4
Output:
Inorder of original tree: 2 3 4 5 6
Inorder of mirror tree: 6 5 4 3 2
Mirror tree will be:
  5
 / \
6   3
   / \
  4   2


Input:
       2
      / \
     1   8
    /     \
   12      9
```

```
Output:
Inorder of original tree: 12 1 2 8 9

Inorder of mirror tree: 9 8 2 1 12
```

Approach: Write a recursive function that will take two nodes as the argument, one of the original tree and the other of the newly created tree. Now, for every passed node of the original tree, create a corresponding node in the mirror tree and then recursively call the same method for the child nodes but passing the left child of the original tree node with the right child of the mirror tree node and the right child of the original tree node with the left child of the mirror tree node.

Below is the implementation of the above approach:

```c
// C implementation of the approach
#include <malloc.h>
#include <stdio.h>

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

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

// Helper function to print Inorder traversal
void inorder(node* root)
{
    if (root == NULL)
        return;
```

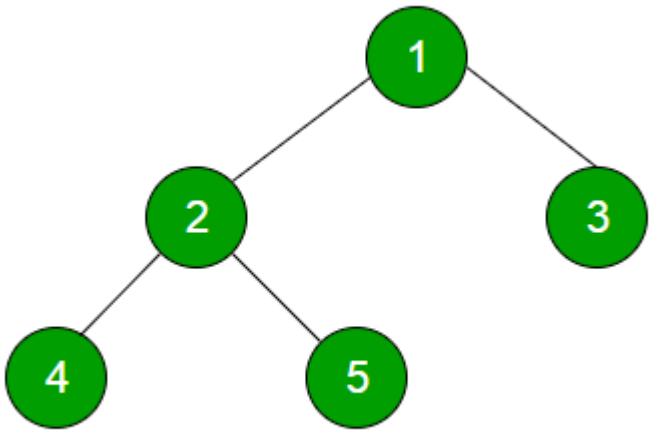
```
    inorder(root->left);
    printf("%d ", root->val);
    inorder(root->right);
}

// mirrorify function takes two trees,
// original tree and a mirror tree
// It recurses on both the trees,
// but when original tree recurses on left,
// mirror tree recurses on right and
// vice-versa
void mirrorify(node* root, node** mirror)
{
    if (root == NULL) {
        mirror = NULL;
        return;
    }

    // Create new mirror node from original tree node
    *mirror = createNode(root->val);
    mirrorify(root->left, &((*mirror)->right));
    mirrorify(root->right, &((*mirror)->left));
}

// Driver code
int main()
{

    node* tree = createNode(5);
    tree->left = createNode(3);
    tree->right = createNode(6);
    tree->left->left = createNode(2);
    tree->left->right = createNode(4);

    // Print inorder traversal of the input tree
    printf("Inorder of original tree: ");
    inorder(tree);
    node* mirror = NULL;
    mirrorify(tree, &mirror);

    // Print inorder traversal of the mirror tree
    printf("\nInorder of mirror tree: ");
    inorder(mirror);

    return 0;
}
```

Output:

```
Inorder of original tree: 2 3 4 5 6

Inorder of mirror tree: 6 5 4 3
```

9. Level Order Tree Traversal

Level order traversal of a tree is breadth first traversal for the tree.



Level order traversal of the above tree is 1 2 3 4 5

Method 1 (Use function to print a given level)

Algorithm:

There are basically two functions in this method. One is to print all nodes at a given level (printGivenLevel), and other is to print level order traversal of the tree (printLevelorder). printLevelorder makes use of printGivenLevel to print nodes at all levels one by one starting from root.

```
/*Function to print level order traversal of tree*/

printLevelorder(tree)
for d = 1 to height(tree)

   printGivenLevel(tree, d);


/*Function to print all nodes at a given level*/

printGivenLevel(tree, level)
if tree is NULL then return;

if level is 1, then

    print(tree->data);
```

```
else if level greater than 1, then

    printGivenLevel(tree->left, level-1);

    printGivenLevel(tree->right, level-1);
```

Implementation:

```
// Recursive CPP program for level
// order traversal of Binary Tree
#include <bits/stdc++.h>
using namespace std;

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

/* Function protoypes */
void printGivenLevel(node* root, int level);
int height(node* node);
node* newNode(int data);

/* Function to print level
order traversal a tree*/
void printLevelOrder(node* root)
{
    int h = height(root);
    int i;
    for (i = 1; i <= h; i++)
        printGivenLevel(root, i);
}

/* Print nodes at a given level */
void printGivenLevel(node* root, int level)
{
    if (root == NULL)
        return;
    if (level == 1)
        cout << root->data << " ";
    else if (level > 1)
    {
        printGivenLevel(root->left, level-1);
        printGivenLevel(root->right, level-1);
    }
}
```

```
/* Compute the "height" of a tree -- the number of
    nodes along the longest path from the root node
    down to the farthest leaf node.*/
int height(node* node)
{
    if (node == NULL)
        return 0;
    else
    {
        /* compute the height of each subtree */
        int lheight = height(node->left);
        int rheight = height(node->right);

        /* use the larger one */
        if (lheight > rheight)
            return(lheight + 1);
        else return(rheight + 1);
    }
}

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

    return(Node);
}

/* Driver code*/
int main()
{
    node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);

    cout << "Level Order traversal of binary tree is \n";
    printLevelOrder(root);

    return 0;
}

// This code is contributed by rathbhupendra
```

Output:

```
Level order traversal of binary tree is -

1 2 3 4 5
```

Time Complexity: O(n^2) in worst case. For a skewed tree, printGivenLevel() takes O(n) time where n is the number of nodes in the skewed tree. So time complexity of printLevelOrder() is O(n) + O(n-1) + O(n-2) + .. + O(1) which is O(n^2).

Space Complexity: O(n) in worst case. For a skewed tree, printGivenLevel() uses O(n) space for call stack. For a Balanced tree, call stack uses O(log n) space, (i.e., height of the balanced tree).

Method 2 (Using queue)

Algorithm:

For each node, first the node is visited and then it’s child nodes are put in a FIFO queue.

```
printLevelorder(tree)

1) Create an empty queue q

2) temp_node = root /*start from root*/

3) Loop while temp_node is not NULL

    a) print temp_node->data.

    b) Enqueue temp_node’s children (first left then right children) to
q

    c) Dequeue a node from q and assign it’s value to temp_node
```

Implementation:

Here is a simple implementation of the above algorithm. Queue is implemented using an array with maximum size of 500. We can implement queue as linked list also.

```
/* C++ program to print level order traversal using STL */
```

```cpp
#include <bits/stdc++.h>
using namespace std;

// A Binary Tree Node
struct Node
{
    int data;
    struct Node *left, *right;
};

// Iterative method to find height of Binary Tree
void printLevelOrder(Node *root)
{
    // Base Case
    if (root == NULL)  return;

    // Create an empty queue for level order tarversal
    queue<Node *> q;

    // Enqueue Root and initialize height
    q.push(root);

    while (q.empty() == false)
    {
        // Print front of queue and remove it from queue
        Node *node = q.front();
        cout << node->data << " ";
        q.pop();

        /* Enqueue left child */
        if (node->left != NULL)
            q.push(node->left);

        /*Enqueue right child */
        if (node->right != NULL)
            q.push(node->right);
    }
}

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

// Driver program to test above functions
int main()
```

```
{
    // Let us create binary tree shown in above diagram
    Node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);

    cout << "Level Order traversal of binary tree is \n";
    printLevelOrder(root);
    return 0;
}
```

Output:

```
Level order traversal of binary tree is -

1 2 3 4 5
```

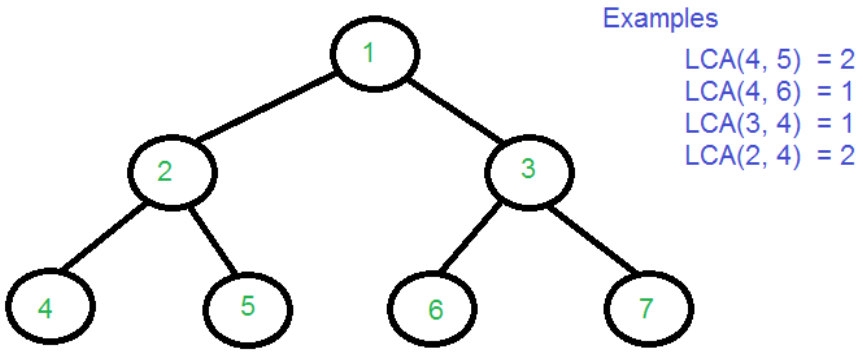
Time Complexity: O(n) where n is number of nodes in the binary tree

Space Complexity: O(n) where n is number of nodes in the binary tree

10. Lowest Common Ancestor in a Binary Tree | Set 1

Given a binary tree (not a binary search tree) and two values say n1 and n2, write a program to find the least common ancestor.

Following is definition of LCA from Wikipedia:
Let T be a rooted tree. The lowest common ancestor between two nodes n1 and n2 is defined as the lowest node in T that has both n1 and n2 as descendants (where we allow a node to be a descendant of itself).
The LCA of n1 and n2 in T is the shared ancestor of n1 and n2 that is located farthest from the root. Computation of lowest common ancestors may be useful, for instance, as part of a procedure for determining the distance between pairs of nodes in a tree: the distance from n1 to n2 can be computed as the distance from the root to n1, plus the distance from the root to n2, minus twice the distance from the root to their lowest common ancestor.
(Source Wiki)



We have discussed an efficient solution to find LCA in Binary Search Tree. In Binary Search Tree, using BST properties, we can find LCA in O(h) time where h is height of tree. Such an implementation is not possible in Binary Tree as keys Binary Tree nodes don't follow any order. Following are different approaches to find LCA in Binary Tree.
Method 1 (By Storing root to n1 and root to n2 paths):
Following is simple O(n) algorithm to find LCA of n1 and n2.

1) Find path from root to n1 and store it in a vector or array.

2) Find path from root to n2 and store it in another vector or array.

3) Traverse both paths till the values in arrays are same. Return the common element just before the mismatch.

Following is the implementation of above algorithm.

```
// C++ Program for Lowest Common Ancestor in a Binary Tree
// A O(n) solution to find LCA of two given values n1 and n2
#include <iostream>
#include <vector>

using namespace std;

// A Binary Tree node
struct Node
{
    int key;
    struct Node *left, *right;
};

// Utility function creates a new binary tree node with given key
Node * newNode(int k)
{
    Node *temp = new Node;
    temp->key = k;
    temp->left = temp->right = NULL;
    return temp;
}

// Finds the path from root node to given root of the tree, Stores
the
// path in a vector path[], returns true if path exists otherwise
false
bool findPath(Node *root, vector<int> &path, int k)
{
    // base case
    if (root == NULL) return false;

    // Store this node in path vector. The node will be removed if
    // not in path from root to k
    path.push_back(root->key);

    // See if the k is same as root's key
    if (root->key == k)
        return true;

    // Check if k is found in left or right sub-tree
```

```
    if ( (root->left && findPath(root->left, path, k)) ||
         (root->right && findPath(root->right, path, k)) )
        return true;

    // If not present in subtree rooted with root, remove root
from
    // path[] and return false
    path.pop_back();
    return false;
}

// Returns LCA if node n1, n2 are present in the given binary
tree,
// otherwise return -1
int findLCA(Node *root, int n1, int n2)
{
    // to store paths to n1 and n2 from the root
    vector<int> path1, path2;

    // Find paths from root to n1 and root to n1. If either n1 or
n2
    // is not present, return -1
    if ( !findPath(root, path1, n1) || !findPath(root, path2, n2))
          return -1;

    /* Compare the paths to get the first different value */
    int i;
    for (i = 0; i < path1.size() && i < path2.size() ; i++)
        if (path1[i] != path2[i])
            break;
    return path1[i-1];
}

// Driver program to test above functions
int main()
{
    // Let us create the Binary Tree shown in above diagram.
    Node * root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->left = newNode(6);
    root->right->right = newNode(7);
    cout << "LCA(4, 5) = " << findLCA(root, 4, 5);
    cout << "nLCA(4, 6) = " << findLCA(root, 4, 6);
    cout << "nLCA(3, 4) = " << findLCA(root, 3, 4);
    cout << "nLCA(2, 4) = " << findLCA(root, 2, 4);
    return 0;
}
```

Output:

```
LCA(4, 5) = 2

LCA(4, 6) = 1

LCA(3, 4) = 1

LCA(2, 4) = 2
```

Time Complexity: Time complexity of the above solution is O(n). The tree is traversed twice, and then path arrays are compared.

Thanks to Ravi Chandra Enaganti for suggesting the initial solution based on this method.

Method 2 (Using Single Traversal)

The method 1 finds LCA in O(n) time, but requires three tree traversals plus extra spaces for path arrays. If we assume that the keys n1 and n2 are present in Binary Tree, we can find LCA using single traversal of Binary Tree and without extra storage for path arrays.

The idea is to traverse the tree starting from root. If any of the given keys (n1 and n2) matches with root, then root is LCA (assuming that both keys are present). If root doesn't match with any of the keys, we recur for left and right subtree. The node which has one key present in its left subtree and the other key present in right subtree is the LCA. If both keys lie in left subtree, then left subtree has LCA also, otherwise LCA lies in right subtree.

```
/* C++ Program to find LCA of n1 and n2 using one traversal of
Binary Tree */
#include <iostream>

using namespace std;

// A Binary Tree Node
struct Node
{
    struct Node *left, *right;
    int key;
};
```

```
// Utility function to create a new tree Node
Node* newNode(int key)
{
    Node *temp = new Node;
    temp->key = key;
    temp->left = temp->right = NULL;
    return temp;
}

// This function returns pointer to LCA of two given values n1 and
n2.
// This function assumes that n1 and n2 are present in Binary Tree
struct Node *findLCA(struct Node* root, int n1, int n2)
{
    // Base case
    if (root == NULL) return NULL;

    // If either n1 or n2 matches with root's key, report
    // the presence by returning root (Note that if a key is
    // ancestor of other, then the ancestor key becomes LCA
    if (root->key == n1 || root->key == n2)
        return root;

    // Look for keys in left and right subtrees
    Node *left_lca  = findLCA(root->left, n1, n2);
    Node *right_lca = findLCA(root->right, n1, n2);

    // If both of the above calls return Non-NULL, then one key
    // is present in once subtree and other is present in other,
    // So this node is the LCA
    if (left_lca && right_lca)  return root;

    // Otherwise check if left subtree or right subtree is LCA
    return (left_lca != NULL)? left_lca: right_lca;
}

// Driver program to test above functions
int main()
{
    // Let us create binary tree given in the above example
    Node * root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->left = newNode(6);
    root->right->right = newNode(7);
    cout << "LCA(4, 5) = " << findLCA(root, 4, 5)->key;
    cout << "nLCA(4, 6) = " << findLCA(root, 4, 6)->key;
```

```
    cout << "nLCA(3, 4) = " << findLCA(root, 3, 4)->key;
    cout << "nLCA(2, 4) = " << findLCA(root, 2, 4)->key;
    return 0;
}
```

Output:

```
LCA(4, 5) = 2

LCA(4, 6) = 1

LCA(3, 4) = 1

LCA(2, 4) = 2
```

Thanks to Atul Singh for suggesting this solution.

Time Complexity: Time complexity of the above solution is O(n) as the method does a simple tree traversal in bottom up fashion.

Note that the above method assumes that keys are present in Binary Tree. If one key is present and other is absent, then it returns the present key as LCA (Ideally should have returned NULL).

We can extend this method to handle all cases by passing two boolean variables v1 and v2. v1 is set as true when n1 is present in tree and v2 is set as true if n2 is present in tree.

```
/* C++ program to find LCA of n1 and n2 using one traversal of
Binary Tree.
   It handles all cases even when n1 or n2 is not there in Binary
Tree */
#include <iostream>
using namespace std;

// A Binary Tree Node
struct Node
{
    struct Node *left, *right;
    int key;
};

// Utility function to create a new tree Node
Node* newNode(int key)
{
    Node *temp = new Node;
    temp->key = key;
    temp->left = temp->right = NULL;
    return temp;
```

```
}

// This function returns pointer to LCA of two given values n1 and
n2.
// v1 is set as true by this function if n1 is found
// v2 is set as true by this function if n2 is found
struct Node *findLCAUtil(struct Node* root, int n1, int n2, bool
&v1, bool &v2)
{
    // Base case
    if (root == NULL) return NULL;

    // If either n1 or n2 matches with root's key, report the
presence
    // by setting v1 or v2 as true and return root (Note that if a
key
    // is ancestor of other, then the ancestor key becomes LCA)
    if (root->key == n1)
    {
        v1 = true;
        return root;
    }
    if (root->key == n2)
    {
        v2 = true;
        return root;
    }

    // Look for keys in left and right subtrees
    Node *left_lca  = findLCAUtil(root->left, n1, n2, v1, v2);
    Node *right_lca = findLCAUtil(root->right, n1, n2, v1, v2);

    // If both of the above calls return Non-NULL, then one key
    // is present in once subtree and other is present in other,
    // So this node is the LCA
    if (left_lca && right_lca)  return root;

    // Otherwise check if left subtree or right subtree is LCA
    return (left_lca != NULL)? left_lca: right_lca;
}

// Returns true if key k is present in tree rooted with root
bool find(Node *root, int k)
{
    // Base Case
    if (root == NULL)
        return false;

    // If key is present at root, or in left subtree or right
subtree,
```

```
    // return true;
    if (root->key == k || find(root->left, k) ||  find(root-
>right, k))
        return true;

    // Else return false
    return false;
}

// This function returns LCA of n1 and n2 only if both n1 and n2
are present
// in tree, otherwise returns NULL;
Node *findLCA(Node *root, int n1, int n2)
{
    // Initialize n1 and n2 as not visited
    bool v1 = false, v2 = false;

    // Find lca of n1 and n2 using the technique discussed above
    Node *lca = findLCAUtil(root, n1, n2, v1, v2);

    // Return LCA only if both n1 and n2 are present in tree
    if (v1 && v2 || v1 && find(lca, n2) || v2 && find(lca, n1))
        return lca;

    // Else return NULL
    return NULL;
}

// Driver program to test above functions
int main()
{
    // Let us create binary tree given in the above example
    Node * root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->left = newNode(6);
    root->right->right = newNode(7);
    Node *lca =  findLCA(root, 4, 5);
    if (lca != NULL)
       cout << "LCA(4, 5) = " << lca->key;
    else
       cout << "Keys are not present ";

    lca =  findLCA(root, 4, 10);
    if (lca != NULL)
       cout << "nLCA(4, 10) = " << lca->key;
    else
```

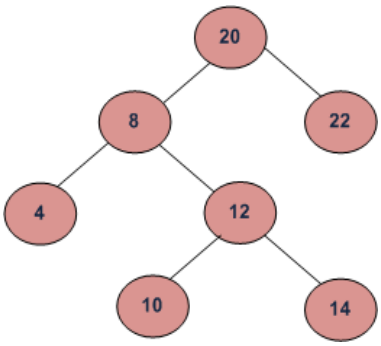
```
        cout << "nKeys are not present ";

    return 0;
}
```

Output:

```
LCA(4, 5) = 2

Keys are not present
```

11. Lowest Common Ancestor in a Binary Tree | Set 2 (Using Parent Pointer)

Given values of two nodes in a Binary Tree, find the Lowest Common Ancestor (LCA). It may be assumed that both nodes exist in the tree.



For example, consider the Binary Tree in diagram, LCA of 10 and 14 is 12 and LCA of 8 and 14 is 8.

Let T be a rooted tree. The lowest common ancestor between two nodes n1 and n2 is defined as the lowest node in T that has both n1 and n2 as descendants (where we allow a node to be a descendant of itself). Source : Wikipedia.

We have discussed different approaches to find LCA in set 1. Finding LCA becomes easy when parent pointer is given as we can easily find all ancestors of a node using parent pointer.

Below are steps to find LCA.

1. Create an empty hash table.
2. Insert n1 and all of its ancestors in hash table.
3. Check if n2 or any of its ancestors exist in hash table, if yes return the first existing ancestor.

Below is the implementation of above steps.

```
// C++ program to find lowest common ancestor using parent pointer
#include <bits/stdc++.h>
using namespace std;

// A Tree Node
struct Node
{
    Node *left, *right, *parent;
    int key;
};
```

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

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

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

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

// To find LCA of nodes n1 and n2 in Binary Tree
Node *LCA(Node *n1, Node *n2)
{
   // Creata a map to store ancestors of n1
   map <Node *, bool> ancestors;

   // Insert n1 and all its ancestors in map
   while (n1 != NULL)
   {
       ancestors[n1] = true;
       n1 = n1->parent;
   }

   // Check if n2 or any of its ancestors is in
   // map.
   while (n2 != NULL)
   {
```

```
        if (ancestors.find(n2) != ancestors.end())
            return n2;
        n2 = n2->parent;
    }

    return NULL;
}

// Driver method to test above functions
int main(void)
{
    Node * root = NULL;

    root = insert(root, 20);
    root = insert(root, 8);
    root = insert(root, 22);
    root = insert(root, 4);
    root = insert(root, 12);
    root = insert(root, 10);
    root = insert(root, 14);

    Node *n1 = root->left->right->left;
    Node *n2 = root->left;
    Node *lca = LCA(n1, n2);

    printf("LCA of %d and %d is %d \n", n1->key, n2->key, lca-
>key);

    return 0;
}
```

Output:

```
LCA of 10 and 8 is 8
```

Note : The above implementation uses insert of Binary Search Tree to create a Binary Tree, but the function LCA is for any Binary Tree (not necessarily a Binary Search Tree).

Time Complexity : O(h) where h is height of Binary Tree if we use hash table to implement the solution (Note that the above solution uses map which takes O(Log h) time to insert and find). So the time complexity of above implementation is O(h Log h).

Auxiliary Space : O(h)

A O(h) time and O(1) Extra Space Solution:

The above solution requires extra space because we need to use a hash table to store visited ancestors. We can solve the problem in O(1) extra space using following fact : If both nodes are at same level and if we traverse up using parent pointers of both nodes, the first common node in the path to root is lca. The idea is to find depths of given nodes and move up the deeper node pointer by the difference between depths. Once both nodes reach same level, traverse them up and return the first common node.

Thanks to Mysterious Mind for suggesting this approach.

```
// C++ program to find lowest common ancestor using parent pointer
#include <bits/stdc++.h>
using namespace std;

// A Tree Node
struct Node
{
    Node *left, *right, *parent;
    int key;
};

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

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

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

```
    }
    else if (key > node->key)
    {
        node->right = insert(node->right, key);
        node->right->parent = node;
    }

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

// A utility function to find depth of a node
// (distance of it from root)
int depth(Node *node)
{
    int d = -1;
    while (node)
    {
        ++d;
        node = node->parent;
    }
    return d;
}

// To find LCA of nodes n1 and n2 in Binary Tree
Node *LCA(Node *n1, Node *n2)
{
    // Find depths of two nodes and differences
    int d1 = depth(n1), d2 = depth(n2);
    int diff = d1 - d2;

    // If n2 is deeper, swap n1 and n2
    if (diff < 0)
    {
        Node * temp = n1;
        n1 = n2;
        n2 = temp;
        diff = -diff;
    }

    // Move n1 up until it reaches the same level as n2
    while (diff--)
        n1 = n1->parent;

    // Now n1 and n2 are at same levels
    while (n1 && n2)
    {
        if (n1 == n2)
            return n1;
```

```
        n1 = n1->parent;
        n2 = n2->parent;
    }

    return NULL;
}

// Driver method to test above functions
int main(void)
{
    Node * root = NULL;

    root = insert(root, 20);
    root = insert(root, 8);
    root = insert(root, 22);
    root = insert(root, 4);
    root = insert(root, 12);
    root = insert(root, 10);
    root = insert(root, 14);

    Node *n1 = root->left->right->left;
    Node *n2 = root->right;

    Node *lca = LCA(n1, n2);
    printf("LCA of %d and %d is %d \n", n1->key, n2->key, lca-
>key);

    return 0;
}
```

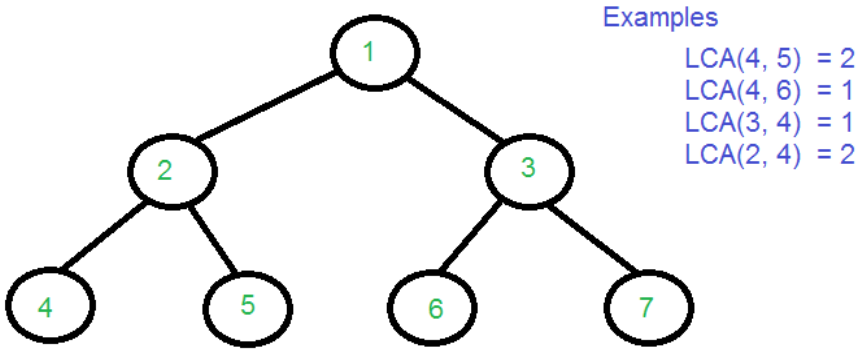
Output :

```
LCA of 10 and 22 is 20
```

12. Lowest Common Ancestor in a Binary Tree | Set 3 (Using RMQ)

Given a rooted tree, and two nodes which are in the tree, find the Lowest common ancestor of both the nodes. The LCA for two nodes u and v is defined as the farthest node from the root that is the ancestor to both u and v.

Prerequisites : LCA | SET 1



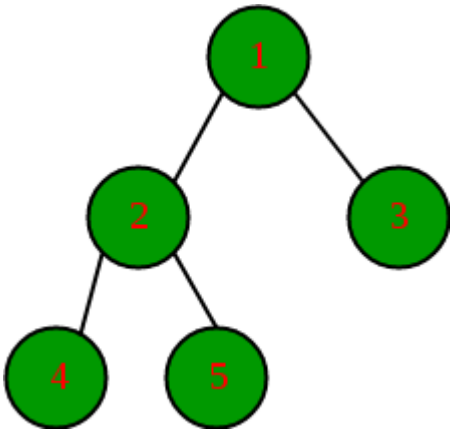
Example for above figure :

```
Input : 4 5

Output : 2


Input : 4 7

Output : 1
```

Converting LCA to RMQ(Range Minimum Query):

Take an array named E[], which stores the order of dfs traversal i.e. the order in

which the nodes are covered during the dfs traversal. For example,



The tree given above has dfs traversal in the order: 1-2-4-2-5-2-1-3.

Take another array L[], in which L[i] is the level of node E[i].

And the array H[], which stores the index of the first occurrence of ith node in the array E[].

So, for the above tree,
E[] = {1, 2, 4, 2, 5, 2, 1, 3}
L[] = {1, 2, 3, 2, 3, 2, 1, 2}
H[] = {0, 1, 7, 2, 4}
Note that the arrays E and L are with one-based indexing but the array H has zero-based indexing.

Now, to find the LCA(4, 3), first, use the array H and find the indices at which 4 and 3 are found in E i.e. H[4] and H[3]. So, the indices come out to be 2 and 7. Now, look at the subarray L[2 : 7], and find the minimum in this subarray which is 1 (at the 6th index), and the corresponding element in the array E i.e. E[6] is the LCA(4, 3).

To understand why this works, take LCA(4, 3) again. The path by which one can reach node 3 from node 4 is the subarray E[2 : 7]. And, if there is a node with the lowest level in this path, then it can simply be claimed to be the LCA(4, 3).

Now, the problem is to find the minimum in the subarray E[H[u]....H[v]] (assuming that H[u] >= H[v]). And, that could be done using a segment tree or sparse table. Below is the code using the segment tree.

```
// CPP code to find LCA of given
// two nodes in a tree
#include <bits/stdc++.h>

#define sz(x) x.size()
#define pb push_back
#define left 2 * i + 1
#define right 2 * i + 2
using namespace std;

const int maxn = 100005;

// the graph
vector<vector<int>> g(maxn);

// level of each node
int level[maxn];

vector<int> e;
vector<int> l;
int h[maxn];

// the segment tree
int st[5 * maxn];

// adding edges to the graph(tree)
void add_edge(int u, int v) {
  g[u].pb(v);
  g[v].pb(u);
}

// assigning level to nodes
void leveling(int src) {
  for (int i = 0; i < sz(g[src]); i++) {
    int des = g[src][i];
    if (!level[des]) {
      level[des] = level[src] + 1;
      leveling(des);
    }
```

```
  }
}

bool visited[maxn];

// storing the dfs traversal
// in the array e
void dfs(int src) {
  e.pb(src);
  visited[src] = 1;
  for (int i = 0; i < sz(g[src]); i++) {
    int des = g[src][i];
    if (!visited[des]) {
      dfs(des);
      e.pb(src);
    }
  }
}

// making the array l
void setting_l(int n) {
  for (int i = 0; i < sz(e); i++)
    l.pb(level[e[i]]);
}

// making the array h
void setting_h(int n) {
  for (int i = 0; i <= n; i++)
    h[i] = -1;
  for (int i = 0; i < sz(e); i++) {
    // if is already stored
    if (h[e[i]] == -1)
      h[e[i]] = i;
  }
}

// Range minimum query to return the index
// of minimum in the subarray L[qs:qe]
int RMQ(int ss, int se, int qs, int qe, int i) {
  if (ss > se)
    return -1;

  // out of range
  if (se < qs || qe < ss)
    return -1;

  // in the range
  if (qs <= ss && se <= qe)
    return st[i];
```

```
  int mid = (ss + se) >> 1;
  int st = RMQ(ss, mid, qs, qe, left);
  int en = RMQ(mid + 1, se, qs, qe, right);

  if (st != -1 && en != -1) {
    if (l[st] < l[en])
      return st;
    return en;
  } else if (st != -1)
    return st;
  else if (en != -1)
    return en;
}

// constructs the segment tree
void SegmentTreeConstruction(int ss, int se, int i) {
  if (ss > se)
    return;
  if (ss == se) // leaf
  {
    st[i] = ss;
    return;
  }
  int mid = (ss + se) >> 1;

  SegmentTreeConstruction(ss, mid, left);
  SegmentTreeConstruction(mid + 1, se, right);

  if (l[st[left]] < l[st[right]])
    st[i] = st[left];
  else
    st[i] = st[right];
}

// Funtion to get LCA
int LCA(int x, int y) {
  if (h[x] > h[y])
    swap(x, y);
  return e[RMQ(0, sz(l) - 1, h[x], h[y], 0)];
}

// Driver code
int main() {
  ios::sync_with_stdio(0);

  // n=number of nodes in the tree
  // q=number of queries to answer
  int n = 15, q = 5;
```

```
  // making the tree
  /*
                    1
                  / | \
                 2  3  4
                    |   \
                    5    6
                  / |  \
                 8  7    9 (right of 5)
                   / | \   | \
                  10 11 12 13 14
                        |
                        15
  */
  add_edge(1, 2);
  add_edge(1, 3);
  add_edge(1, 4);
  add_edge(3, 5);
  add_edge(4, 6);
  add_edge(5, 7);
  add_edge(5, 8);
  add_edge(5, 9);
  add_edge(7, 10);
  add_edge(7, 11);
  add_edge(7, 12);
  add_edge(9, 13);
  add_edge(9, 14);
  add_edge(12, 15);

  level[1] = 1;
  leveling(1);

  dfs(1);

  setting_l(n);

  setting_h(n);

  SegmentTreeConstruction(0, sz(l) - 1, 0);

  cout << LCA(10, 15) << endl;
  cout << LCA(11, 14) << endl;

  return 0;
}
```

Output:

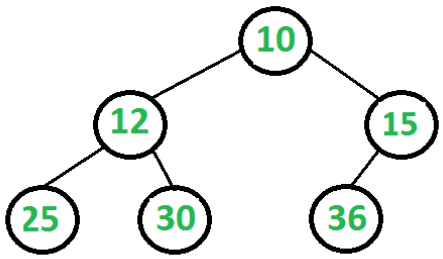
```
7

5
```

Time Complexity :

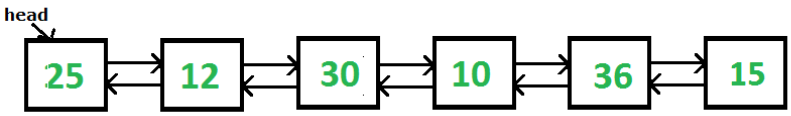
The arrays defined are stored in O(n). The segment tree construction also takes O(n) time. The LCA function calls the function RMQ which takes O(logn) per query (as it uses the segment tree). So overall time complexity is O(n + q * logn)

13. Convert a given Binary Tree to Doubly Linked List | Set 1

Given a Binary Tree (Bt), convert it to a Doubly Linked List(DLL). The left and right pointers in nodes are to be used as previous and next pointers respectively in converted DLL. The order of nodes in DLL must be same as Inorder of the given Binary Tree. The first node of Inorder traversal (left most node in BT) must be head node of the DLL.



The above tree should be in-place converted to following Doubly Linked List(DLL).



I came across this interview during one of my interviews. A similar problem is discussed in this post. The problem here is simpler as we don't need to create circular DLL, but a simple DLL. The idea behind its solution is quite simple and straight.

1. If left subtree exists, process the left subtree

.....1.a) Recursively convert the left subtree to DLL.

.....1.b) Then find inorder predecessor of root in left subtree (inorder predecessor is rightmost node in left subtree).

.....1.c) Make inorder predecessor as previous of root and root as next of inorder predecessor.

2. If right subtree exists, process the right subtree (Below 3 steps are similar to left subtree).

.....2.a) Recursively convert the right subtree to DLL.

.....2.b) Then find inorder successor of root in right subtree (inorder successor is leftmost node in right subtree).

.....2.c) Make inorder successor as next of root and root as previous of inorder successor.

3. Find the leftmost node and return it (the leftmost node is always head of converted DLL).

Below is the source code for above algorithm.

```
// A C++ program for in-place
// conversion of Binary Tree to DLL
#include <bits/stdc++.h>
using namespace std;

/* A binary tree node has data,
and left and right pointers */
class node
{
    public:
    int data;
    node* left;
    node* right;
};

/* This is the core function to convert
Tree to list. This function follows
steps 1 and 2 of the above algorithm */
node* bintree2listUtil(node* root)
{
    // Base case
    if (root == NULL)
        return root;

    // Convert the left subtree and link to root
    if (root->left != NULL)
    {
        // Convert the left subtree
        node* left = bintree2listUtil(root->left);

        // Find inorder predecessor. After this loop, left
        // will point to the inorder predecessor
        for (; left->right != NULL; left = left->right);

        // Make root as next of the predecessor
        left->right = root;
```

```
        // Make predecssor as previous of root
        root->left = left;
    }

    // Convert the right subtree and link to root
    if (root->right != NULL)
    {
        // Convert the right subtree
        node* right = bintree2listUtil(root->right);

        // Find inorder successor. After this loop, right
        // will point to the inorder successor
        for (; right->left != NULL; right = right->left);

        // Make root as previous of successor
        right->left = root;

        // Make successor as next of root
        root->right = right;
    }

    return root;
}

// The main function that first calls
// bintree2listUtil(), then follows step 3
// of the above algorithm
node* bintree2list(node *root)
{
    // Base case
    if (root == NULL)
        return root;

    // Convert to DLL using bintree2listUtil()
    root = bintree2listUtil(root);

    // bintree2listUtil() returns root node of the converted
    // DLL. We need pointer to the leftmost node which is
    // head of the constructed DLL, so move to the leftmost node
    while (root->left != NULL)
        root = root->left;

    return (root);
}

/* Helper function that allocates a new node with the
given data and NULL left and right pointers. */
node* newNode(int data)
```

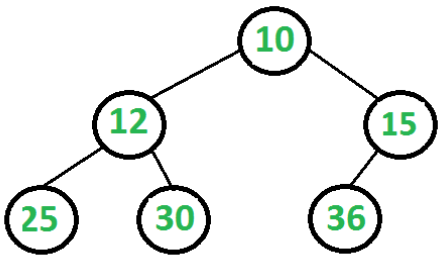
```
{
    node* new_node = new node();
    new_node->data = data;
    new_node->left = new_node->right = NULL;
    return (new_node);
}

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

/* Driver code*/
int main()
{
    // Let us create the tree shown in above diagram
    node *root = newNode(10);
    root->left = newNode(12);
    root->right = newNode(15);
    root->left->left = newNode(25);
    root->left->right = newNode(30);
    root->right->left = newNode(36);

    // Convert to DLL
    node *head = bintree2list(root);

    // Print the converted list
    printList(head);

    return 0;
}

// This code is contributed by rathbhupendra
```

Output:

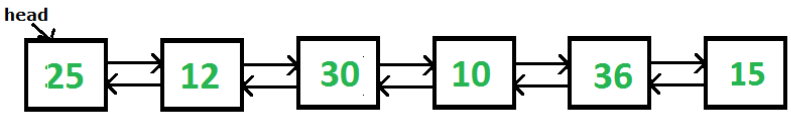
```
25 12 30 10 36 15
```

14. Convert a given Binary Tree to Doubly Linked List | Set 2

Given a Binary Tree (BT), convert it to a Doubly Linked List(DLL). The left and right pointers in nodes are to be used as previous and next pointers respectively in converted DLL. The order of nodes in DLL must be same as Inorder of the given Binary Tree. The first node of Inorder traversal (left most node in BT) must be head node of the DLL.



The above tree should be in-place converted to following Doubly Linked List(DLL).



A solution to this problem is discussed in this post.
In this post, another simple and efficient solution is discussed. The solution discussed here has two simple steps.

1) Fix Left Pointers: In this step, we change left pointers to point to previous nodes in DLL. The idea is simple, we do inorder traversal of tree. In inorder traversal, we keep track of previous visited node and change left pointer to the previous node. See fixPrevPtr() in below implementation.
2) Fix Right Pointers: The above is intuitive and simple. How to change right pointers to point to next node in DLL? The idea is to use left pointers fixed in step 1. We start from the rightmost node in Binary Tree (BT). The rightmost node is the last node in DLL. Since left pointers are changed to point to previous node in DLL, we can linearly traverse the complete DLL using these

pointers. The traversal would be from last to first node. While traversing the DLL, we keep track of the previously visited node and change the right pointer to the previous node. See fixNextPtr() in below implementation.

```
// A simple inorder traversal based
// program to convert a Binary Tree to DLL
#include <bits/stdc++.h>
using namespace std;

// A tree node
class node
{
    public:
    int data;
    node *left, *right;
};

// A utility function to create
// a new tree node
node *newNode(int data)
{
    node *Node = new node();
    Node->data = data;
    Node->left = Node->right = NULL;
    return(Node);
}

// Standard Inorder traversal of tree
void inorder(node *root)
{
    if (root != NULL)
    {
        inorder(root->left);
        cout << "\t" << root->data;
        inorder(root->right);
    }
}

// Changes left pointers to work as
// previous pointers in converted DLL
// The function simply does inorder
// traversal of Binary Tree and updates
// left pointer using previously visited node
void fixPrevPtr(node *root)
{
    static node *pre = NULL;

    if (root != NULL)
    {
```

```
        fixPrevPtr(root->left);
        root->left = pre;
        pre = root;
        fixPrevPtr(root->right);
    }
}

// Changes right pointers to work
// as next pointers in converted DLL
node *fixNextPtr(node *root)
{
    node *prev = NULL;

    // Find the right most node
    // in BT or last node in DLL
    while (root && root->right != NULL)
        root = root->right;

    // Start from the rightmost node,
    // traverse back using left pointers.
    // While traversing, change right pointer of nodes.
    while (root && root->left != NULL)
    {
        prev = root;
        root = root->left;
        root->right = prev;
    }

    // The leftmost node is head
    // of linked list, return it
    return (root);
}

// The main function that converts
// BST to DLL and returns head of DLL
node *BTToDLL(node *root)
{
    // Set the previous pointer
    fixPrevPtr(root);

    // Set the next pointer and return head of DLL
    return fixNextPtr(root);
}

// Traverses the DLL from left tor right
void printList(node *root)
{
    while (root != NULL)
    {
        cout<<"\t"<<root->data;
```

```
        root = root->right;
    }
}

// Driver code
int main(void)
{
    // Let us create the tree
    // shown in above diagram
    node *root = newNode(10);
    root->left = newNode(12);
    root->right = newNode(15);
    root->left->left = newNode(25);
    root->left->right = newNode(30);
    root->right->left = newNode(36);

    cout<<"\n\t\tInorder Tree Traversal\n\n";
    inorder(root);

    node *head = BTToDLL(root);

    cout << "\n\n\t\tDLL Traversal\n\n";
    printList(head);
    return 0;
}

// This code is contributed by rathbhupendra
```

Output:

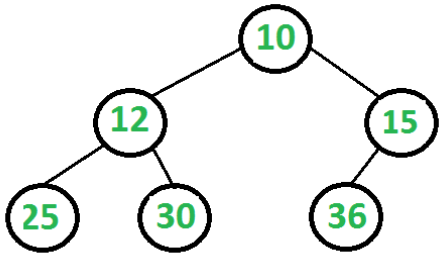
```
              Inorder Tree Traversal


       25      12      30      10      36      15


              DLL Traversal


       25      12      30      10      36      15
```

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

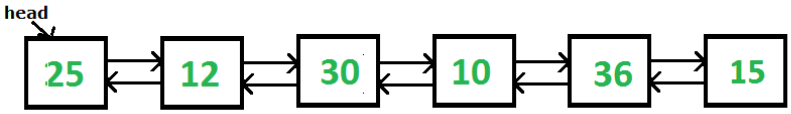
The solution simply does two traversals of all Binary Tree nodes

15. Convert a given Binary Tree to Doubly Linked List | Set 3

Given a Binary Tree (BT), convert it to a Doubly Linked List(DLL) In-Place. The left and right pointers in nodes are to be used as previous and next pointers respectively in converted DLL. The order of nodes in DLL must be same as Inorder of the given Binary Tree. The first node of Inorder traversal (left most node in BT) must be head node of the DLL.



The above tree should be in-place converted to following Doubly Linked List(DLL).



Following two different solutions have been discussed for this problem.

Convert a given Binary Tree to Doubly Linked List | Set 1

Convert a given Binary Tree to Doubly Linked List | Set 2

In this post, a third solution is discussed which seems to be the simplest of all. The idea is to do inorder traversal of the binary tree. While doing inorder traversal, keep track of the previously visited node in a variable say prev. For every visited node, make it next of prev and previous of this node as prev. Thanks to rahul, wishall and all other readers for their useful comments on the above two posts.

Following is the implementation of this solution.

```
// A C++ program for in-place conversion of Binary Tree to DLL
#include <iostream>
```

```
using namespace std;

/* A binary tree node has data, and left and right pointers */
struct node
{
    int data;
    node* left;
    node* right;
};

// A simple recursive function to convert a given Binary tree to
Doubly
// Linked List
// root --> Root of Binary Tree
// head --> Pointer to head node of created doubly linked list
void BinaryTree2DoubleLinkedList(node *root, node **head)
{
    // Base case
    if (root == NULL) return;

    // Initialize previously visited node as NULL. This is
    // static so that the same value is accessible in all
recursive
    // calls
    static node* prev = NULL;

    // Recursively convert left subtree
    BinaryTree2DoubleLinkedList(root->left, head);

    // Now convert this node
    if (prev == NULL)
        *head = root;
    else
    {
        root->left = prev;
        prev->right = root;
    }
    prev = root;

    // Finally convert right subtree
    BinaryTree2DoubleLinkedList(root->right, head);
}

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

```
    return (new_node);
}

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

/* Driver program to test above functions*/
int main()
{
    // Let us create the tree shown in above diagram
    node *root        = newNode(10);
    root->left        = newNode(12);
    root->right       = newNode(15);
    root->left->left  = newNode(25);
    root->left->right = newNode(30);
    root->right->left = newNode(36);

    // Convert to DLL
    node *head = NULL;
    BinaryTree2DoubleLinkedList(root, &head);

    // Print the converted list
    printList(head);

    return 0;
}
```

Output:

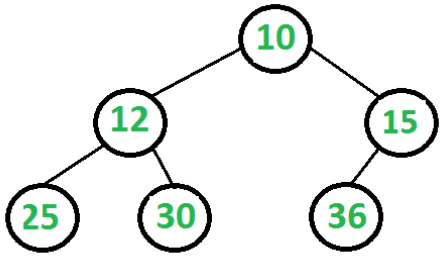
```
25 12 30 10 36 15
```

Note that use of static variables like above is not a recommended practice (we have used static for simplicity). Imagine a situation where same function is called for two or more trees, the old value of prev would be used in next call for a different tree. To avoid such problems, we can use double pointer or reference to a pointer.

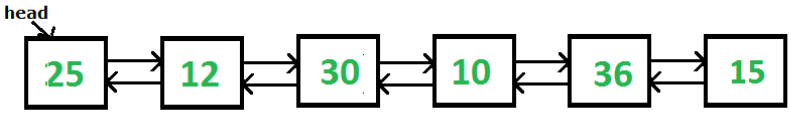
Time Complexity: The above program does a simple inorder traversal, so time complexity is O(n) where n is the number of nodes in given binary tree

16. Convert a given Binary Tree to Doubly Linked List | Set 4

Given a Binary Tree (BT), convert it to a Doubly Linked List(DLL) In-Place. The left and right pointers in nodes are to be used as previous and next pointers respectively in converted DLL. The order of nodes in DLL must be same as Inorder of the given Binary Tree. The first node of Inorder traversal (left most node in BT) must be head node of the DLL.



The above tree should be in-place converted to following Doubly Linked List(DLL).



Below three different solutions have been discussed for this problem.

In the following implementation, we traverse the tree in inorder fashion. We add nodes at the beginning of current linked list and update head of the list using pointer to head pointer. Since we insert at the beginning, we need to process leaves in reverse order. For reverse order, we first traverse the right subtree before the left subtree. i.e. do a reverse inorder traversal.

```
// C++ program to convert a given Binary
// Tree to Doubly Linked List
#include <bits/stdc++.h>

// Structure for tree and linked list
struct Node
{
    int data;
    Node *left, *right;
```

```
};

// A simple recursive function to convert a given
// Binary tree to Doubly Linked List
// root     --> Root of Binary Tree
// head_ref --> Pointer to head node of created
//              doubly linked list
void BToDLL(Node* root, Node** head_ref)
{
    // Base cases
    if (root == NULL)
        return;

    // Recursively convert right subtree
    BToDLL(root->right, head_ref);

    // insert root into DLL
    root->right = *head_ref;

    // Change left pointer of previous head
    if (*head_ref != NULL)
        (*head_ref)->left = root;

    // Change head of Doubly linked list
    *head_ref = root;

    // Recursively convert left subtree
    BToDLL(root->left, head_ref);
}

// Utility function for allocating node for Binary
// Tree.
Node* newNode(int data)
{
    Node* node = new Node;
    node->data = data;
    node->left = node->right = NULL;
    return node;
}

// Utility function for printing double linked list.
void printList(Node* head)
{
    printf("Extracted Double Linked list is:\n");
    while (head)
    {
        printf("%d ", head->data);
        head = head->right;
    }
}
```

```
// Driver program to test above function
int main()
{
    /* Constructing below tree
                5
              /   \
             3     6
            / \     \
           1   4     8
          / \       / \
         0   2     7   9  */
    Node* root = newNode(5);
    root->left = newNode(3);
    root->right = newNode(6);
    root->left->left = newNode(1);
    root->left->right = newNode(4);
    root->right->right = newNode(8);
    root->left->left->left = newNode(0);
    root->left->left->right = newNode(2);
    root->right->right->left = newNode(7);
    root->right->right->right = newNode(9);

    Node* head = NULL;
    BToDLL(root, &head);

    printList(head);

    return 0;
}
```

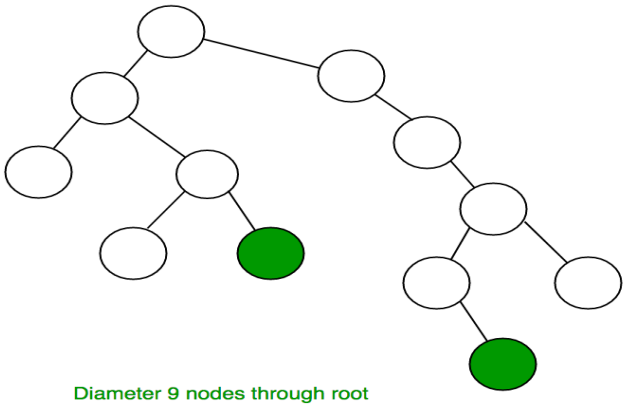
Output :

```
Extracted Double Linked list is:

0 1 2 3 4 5 6 7 8 9
```

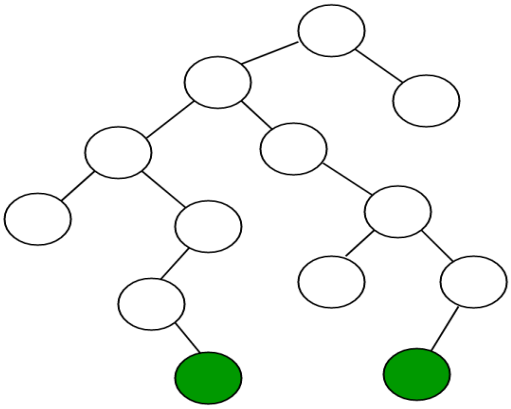
Time Complexity: O(n), as the solution does a single traversal of given Binary Tree

17. Diameter of a Binary Tree

The diameter of a tree (sometimes called the width) is the number of nodes on the longest path between two end nodes. The diagram below shows two trees each with diameter nine, the leaves that form the ends of a longest path are shaded (note that there is more than one path in each tree of length nine, but no path longer than nine nodes).



Diameter 9 nodes through root



Diameter 9 nodes not through root

The diameter of a tree T is the largest of the following quantities:

* the diameter of T’s left subtree

* the diameter of T’s right subtree

* the longest path between leaves that goes through the root of T (this can be computed from the heights of the subtrees of T)

Implementation:

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

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

```c
/* function to create a new node of tree and returns pointer */
struct node* newNode(int data);

/* returns max of two integers */
int max(int a, int b);

/* function to Compute height of a tree. */
int height(struct node* node);

/* Function to get diameter of a binary tree */
int diameter(struct node * tree)
{
   /* base case where tree is empty */
   if (tree == NULL)
     return 0;

  /* get the height of left and right sub-trees */
  int lheight = height(tree->left);
  int rheight = height(tree->right);

  /* get the diameter of left and right sub-trees */
  int ldiameter = diameter(tree->left);
  int rdiameter = diameter(tree->right);

  /* Return max of following three
   1) Diameter of left subtree
   2) Diameter of right subtree
   3) Height of left subtree + height of right subtree + 1 */
  return max(lheight + rheight + 1, max(ldiameter, rdiameter));
}

/* UTILITY FUNCTIONS TO TEST diameter() FUNCTION */

/*  The function Compute the "height" of a tree. Height is the
    number f nodes along the longest path from the root node
    down to the farthest leaf node.*/
int height(struct node* node)
{
   /* base case tree is empty */
   if(node == NULL)
       return 0;

   /* If tree is not empty then height = 1 + max of left
      height and right heights */
   return 1 + max(height(node->left), height(node->right));
}

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newNode(int data)
```

```
{
  struct node* node = (struct node*)
                       malloc(sizeof(struct node));
  node->data = data;
  node->left = NULL;
  node->right = NULL;

  return(node);
}

/* returns maximum of two integers */
int max(int a, int b)
{
  return (a >= b)? a: b;
}

/* Driver program to test above functions*/
int main()
{

  /* Constructed binary tree is
            1
          /   \
        2      3
      /  \
    4     5
  */
  struct node *root = newNode(1);
  root->left        = newNode(2);
  root->right       = newNode(3);
  root->left->left  = newNode(4);
  root->left->right = newNode(5);

  printf("Diameter of the given binary tree is %d\n",
diameter(root));

  getchar();
  return 0;
}
```

Time Complexity: O(n^2)
Output:

```
Diameter of the given binary tree is 4
```

Optimized implementation: The above implementation can be optimized by

calculating the height in the same recursion rather than calling a height() separately. Thanks to Amar for suggesting this optimized version. This optimization reduces time complexity to O(n).

```
/*The second parameter is to store the height of tree.
   Initially, we need to pass a pointer to a location with value
   as 0. So, function should be used as follows:

   int height = 0;
   struct node *root = SomeFunctionToMakeTree();
   int diameter = diameterOpt(root, &height); */
int diameterOpt(struct node *root, int* height)
{
  /* lh --> Height of left subtree
     rh --> Height of right subtree */
  int lh = 0, rh = 0;

  /* ldiameter  --> diameter of left subtree
     rdiameter  --> Diameter of right subtree */
  int ldiameter = 0, rdiameter = 0;

  if(root == NULL)
  {
    *height = 0;
     return 0; /* diameter is also 0 */
  }

  /* Get the heights of left and right subtrees in lh and rh
    And store the returned values in ldiameter and ldiameter */
  ldiameter = diameterOpt(root->left, &lh);
  rdiameter = diameterOpt(root->right, &rh);

  /* Height of current node is max of heights of left and
     right subtrees plus 1*/
  *height = max(lh, rh) + 1;

  return max(lh + rh + 1, max(ldiameter, rdiameter));
}
```

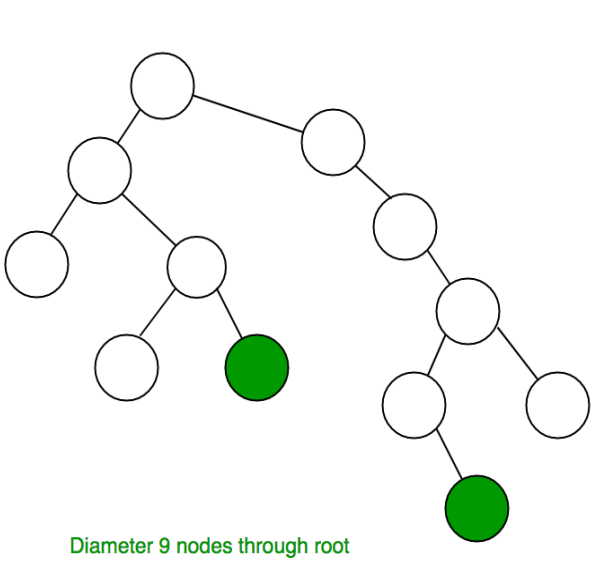
Time Complexity: O(n)

Output:

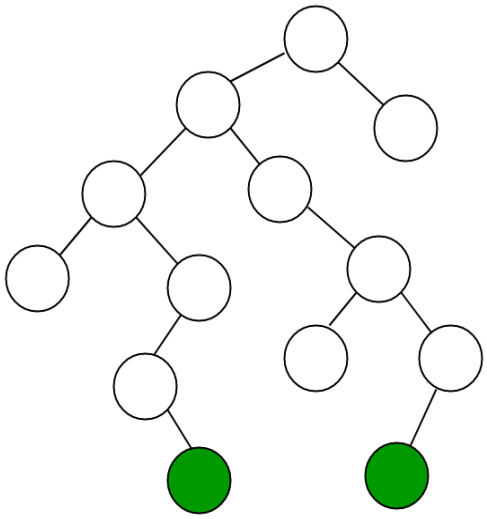
```
4
```

18. Diameter of a Binary Tree in O(n) [A new method]

The diameter of a tree is the number of nodes on the longest path between two leaves in the tree. The diagram below shows two trees each with diameter nine, the leaves that form the ends of a longest path are colored (note that there may be more than one path in tree of same diameter).



Diameter 9 nodes through root



Diameter 9 nodes not through root

Examples:

```
Input :     1
          /   \
        2      3
      /  \
    4     5


Output : 4


Input :     1
          /   \
```

```
       2      3

     /  \ .    \

   4     5 .    6


Output : 5
```

We have discussed a solution in below post.

Diameter of a Binary Tree

In this post a new simple O(n) method is discussed. Diameter of a tree can be calculated by only using the height function, because the diameter of a tree is nothing but maximum value of (left_height + right_height + 1) for each node. So we need to calculate this value (left_height + right_height + 1) for each node and update the result. Time complexity – O(n)

```
// Simple C++ program to find diameter
// of a binary tree.
#include <bits/stdc++.h>
using namespace std;

/* Tree node structure used in the program */
struct Node {
    int data;
    Node* left, *right;
};

/* Function to find height of a tree */
int height(Node* root, int& ans)
{
    if (root == NULL)
        return 0;

    int left_height = height(root->left, ans);

    int right_height = height(root->right, ans);

    // update the answer, because diameter of a
    // tree is nothing but maximum value of
    // (left_height + right_height + 1) for each node
    ans = max(ans, 1 + left_height + right_height);

    return 1 + max(left_height, right_height);
}
```

```
/* Computes the diameter of binary tree with given root. */
int diameter(Node* root)
{
    if (root == NULL)
        return 0;
    int ans = INT_MIN; // This will store the final answer
    int height_of_tree = height(root, ans);
    return ans;
}

struct Node* newNode(int data)
{
    struct Node* node = new Node;
    node->data = data;
    node->left = node->right = NULL;

    return (node);
}

// Driver code
int main()
{
    struct Node* root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);

    printf("Diameter is %d\n", diameter(root));

    return 0;
}
```

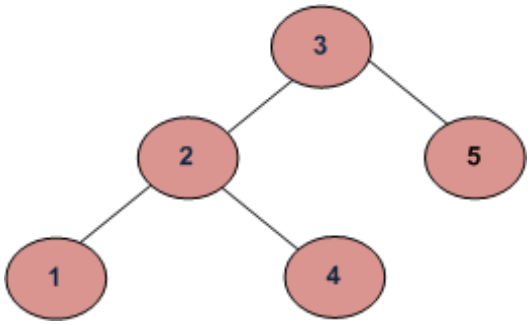
Output:

```
Diameter is 4
```

19. Get Level of a node in a Binary Tree

Given a Binary Tree and a key, write a function that returns level of the key.

For example, consider the following tree. If the input key is 3, then your function should return 1. If the input key is 4, then your function should return 3. And for key which is not present in key, then your function should return 0.



The idea is to start from the root and level as 1. If the key matches with root's data, return level. Else recursively call for left and right subtrees with level as level + 1.

```
// C++ program to Get Level of a
// node in a Binary Tree
#include<bits/stdc++.h>
using namespace std;

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

/* Helper function for getLevel().
It returns level of the data if data is
present in tree, otherwise returns 0.*/
int getLevelUtil(struct node *node,
                 int data, int level)
{
    if (node == NULL)
        return 0;

    if (node -> data == data)
        return level;
```

```
    int downlevel = getLevelUtil(node -> left,
                                 data, level + 1);
    if (downlevel != 0)
        return downlevel;

    downlevel = getLevelUtil(node->right,
                             data, level + 1);
    return downlevel;
}

/* Returns level of given data value */
int getLevel(struct node *node, int data)
{
    return getLevelUtil(node, data, 1);
}

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

    return temp;
}

// Driver Code
int main()
{
    struct node *root = new struct node;
    int x;

    /* Constructing tree given in
    the above figure */
    root = newNode(3);
    root->left = newNode(2);
    root->right = newNode(5);
    root->left->left = newNode(1);
    root->left->right = newNode(4);

    for (x = 1; x <= 5; x++)
    {
        int level = getLevel(root, x);
        if (level)
            cout << "Level of "<< x << " is "
                 << getLevel(root, x) << endl;
        else
```

```
            cout << x << "is not present in tree"
                      << endl;
    }

    getchar();
    return 0;
}

// This code is contributed
// by Akanksha Rai
```

Output:

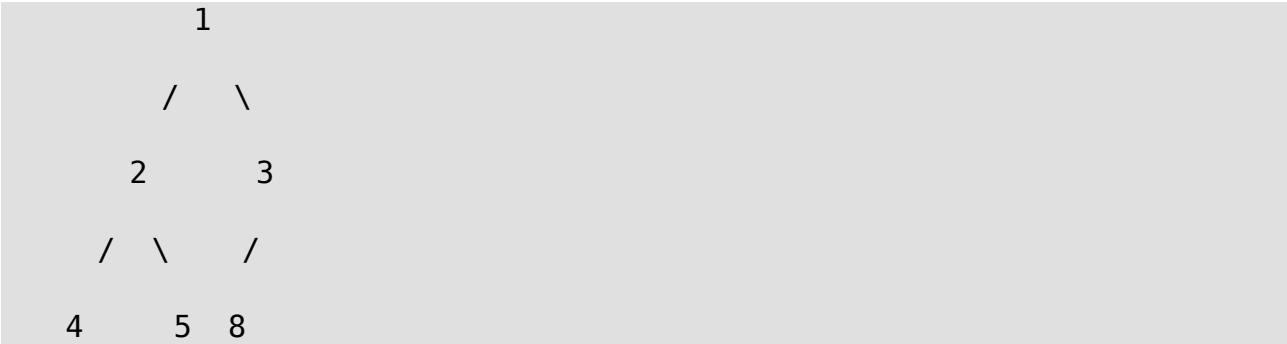
```
Level of 1 is 3

Level of 2 is 2

Level of 3 is 1

Level of 4 is 3

Level of 5 is 2
```

Time Complexity of getLevel() is O(n) where n is the number of nodes in the given Binary Tree.

20. Print nodes at k distance from root

Given a root of a tree, and an integer k. Print all the nodes which are at k distance from root.

For example, in the below tree, 4, 5 & 8 are at distance 2 from root.



The problem can be solved using recursion. Thanks to eldho for suggesting the solution.

```
#include<bits/stdc++.h>

using namespace std;

/* A binary tree node has data,
pointer to left child and
a pointer to right child */
class node
{
    public:
    int data;
    node* left;
    node* right;

    /* Constructor that allocates a new node with the
    given data and NULL left and right pointers. */
    node(int data)
    {
        this->data = data;
        this->left = NULL;
        this->right = NULL;
    }
};

void printKDistant(node *root , int k)
{
    if(root == NULL)
```

```
        return;
    if( k == 0 )
    {
        cout << root->data << " ";
        return ;
    }
    else
    {
        printKDistant( root->left, k - 1 ) ;
        printKDistant( root->right, k - 1 ) ;
    }
}


/* Driver code*/
int main()
{

    /* Constructed binary tree is
            1
            / \
        2     3
        / \     /
        4 5 8
    */
    node *root = new node(1);
    root->left = new node(2);
    root->right = new node(3);
    root->left->left = new node(4);
    root->left->right = new node(5);
    root->right->left = new node(8);

    printKDistant(root, 2);
    return 0;
}

// This code is contributed by rathbhupendra
```

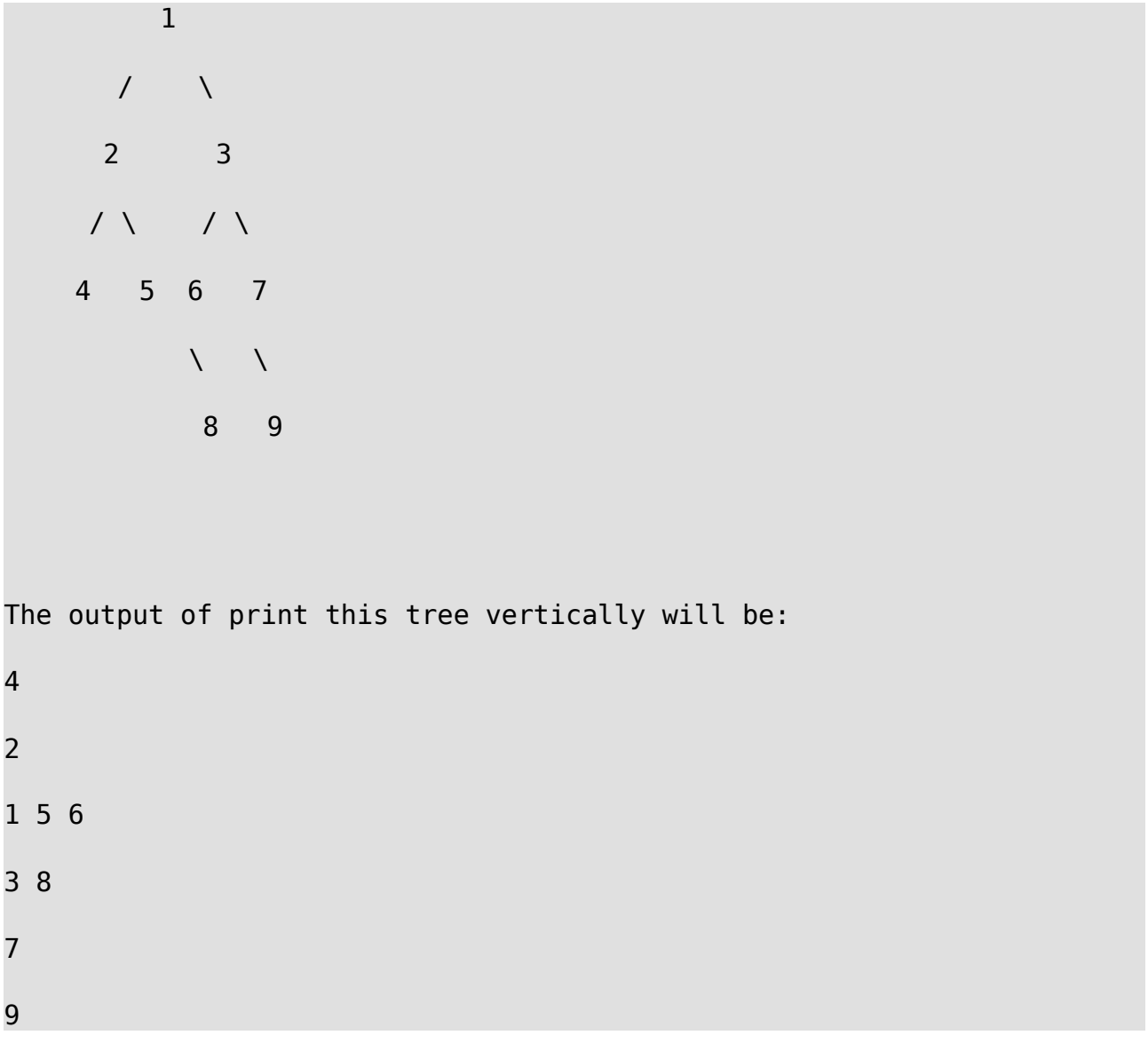
Output:

```
4 5 8
```

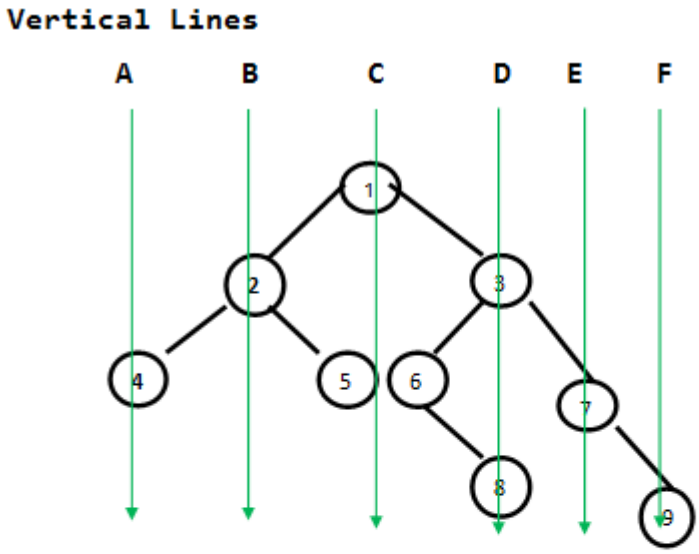
Time Complexity: O(n) where n is number of nodes in the given binary tree.

21. Print a Binary Tree in Vertical Order | Set 1

Given a binary tree, print it vertically. The following example illustrates vertical order traversal.




```
The output of print this tree vertically will be:
4
2
1 5 6
3 8
7
9
```



Vertical order traversal is:

A- 4

B- 2

C- 1 5 6

D- 3 8

E- 7

F- 9

The idea is to traverse the tree once and get the minimum and maximum horizontal distance with respect to root. For the tree shown above, minimum distance is -2 (for node with value 4) and maximum distance is 3 (For node with value 9).

Once we have maximum and minimum distances from root, we iterate for each vertical line at distance minimum to maximum from root, and for each vertical line traverse the tree and print the nodes which lie on that vertical line.

Algorithm:

```
// min --> Minimum horizontal distance from root

// max --> Maximum horizontal distance from root

// hd  --> Horizontal distance of current node from root
```

```
findMinMax(tree, min, max, hd)
    if tree is NULL then return;


    if hd is less than min then

         min = hd;

    else if hd is greater than max then

        *max = hd;


    findMinMax(tree->left, min, max, hd-1);

    findMinMax(tree->right, min, max, hd+1);




printVerticalLine(tree, line_no, hd)
    if tree is NULL then return;


    if hd is equal to line_no, then

         print(tree->data);

    printVerticalLine(tree->left, line_no, hd-1);

    printVerticalLine(tree->right, line_no, hd+1);
```

Implementation:

Following is the implementation of above algorithm.

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

// A node of binary tree
struct Node
{
    int data;
    struct Node *left, *right;
};

// A utility function to create a new Binary Tree node
```

```
Node* newNode(int data)
{
    Node *temp = new Node;
    temp->data = data;
    temp->left = temp->right = NULL;
    return temp;
}

// A utility function to find min and max distances with respect
// to root.
void findMinMax(Node *node, int *min, int *max, int hd)
{
    // Base case
    if (node == NULL) return;

    // Update min and max
    if (hd < *min)  *min = hd;
    else if (hd > *max) *max = hd;

    // Recur for left and right subtrees
    findMinMax(node->left, min, max, hd-1);
    findMinMax(node->right, min, max, hd+1);
}

// A utility function to print all nodes on a given line_no.
// hd is horizontal distance of current node with respect to root.
void printVerticalLine(Node *node, int line_no, int hd)
{
    // Base case
    if (node == NULL) return;

    // If this node is on the given line number
    if (hd == line_no)
        cout << node->data << " ";

    // Recur for left and right subtrees
    printVerticalLine(node->left, line_no, hd-1);
    printVerticalLine(node->right, line_no, hd+1);
}

// The main function that prints a given binary tree in
// vertical order
void verticalOrder(Node *root)
{
    // Find min and max distances with resepect to root
    int min = 0, max = 0;
    findMinMax(root, &min, &max, 0);

    // Iterate through all possible vertical lines starting
    // from the leftmost line and print nodes line by line
```

```
    for (int line_no = min; line_no <= max; line_no++)
    {
        printVerticalLine(root, line_no, 0);
        cout << endl;
    }
}

// Driver program to test above functions
int main()
{
    // Create binary tree shown in above figure
    Node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->left = newNode(6);
    root->right->right = newNode(7);
    root->right->left->right = newNode(8);
    root->right->right->right = newNode(9);

    cout << "Vertical order traversal is \n";
    verticalOrder(root);

    return 0;
}
```

Output:

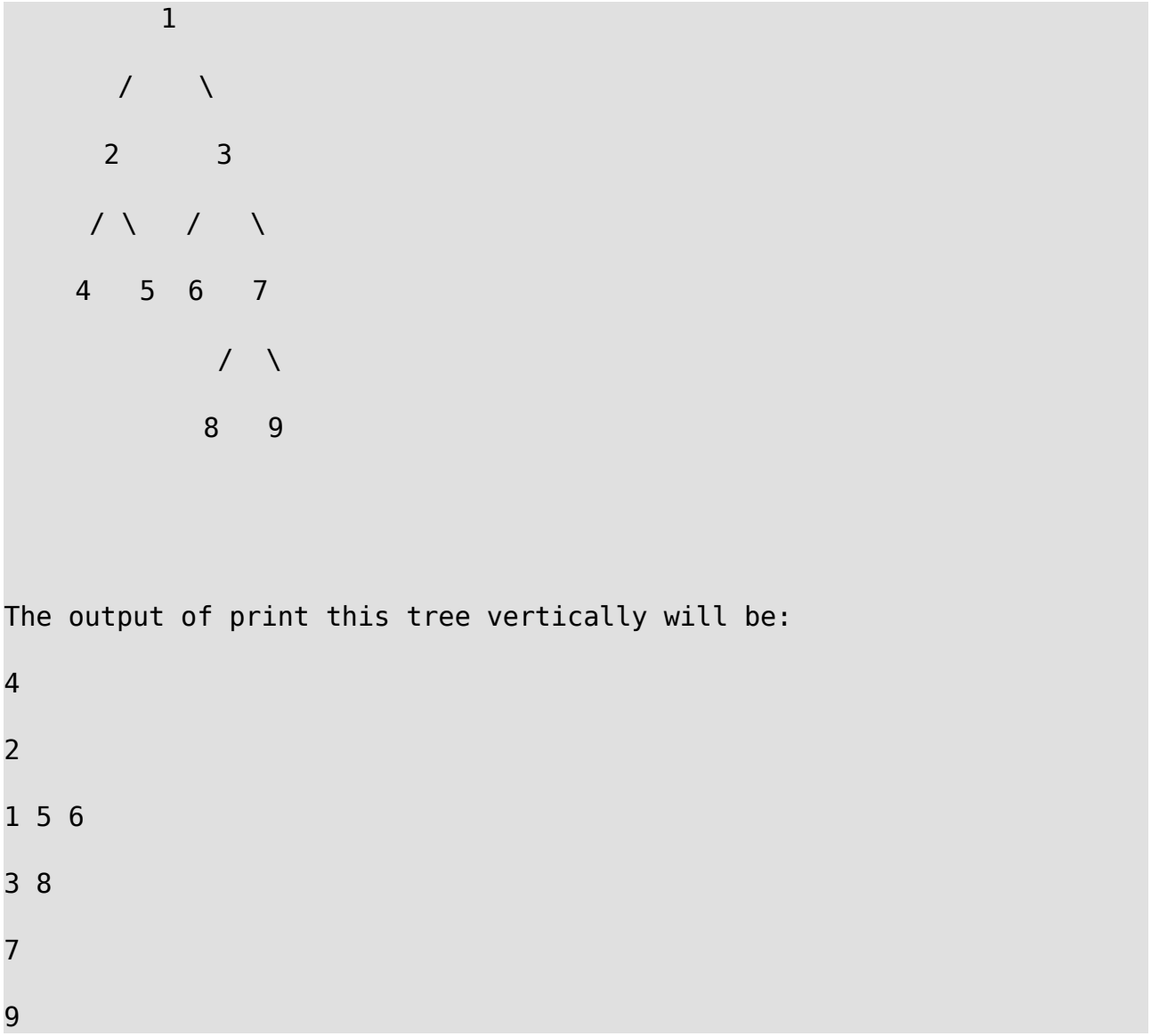
```
Vertical order traversal is

4

2

1 5 6

3 8

7

9
```

Time Complexity: Time complexity of above algorithm is O(w*n) where w is width of Binary Tree and n is number of nodes in Binary Tree. In worst case, the value of w can be O(n) (consider a complete tree for example) and time complexity can become $O(n^2)$.

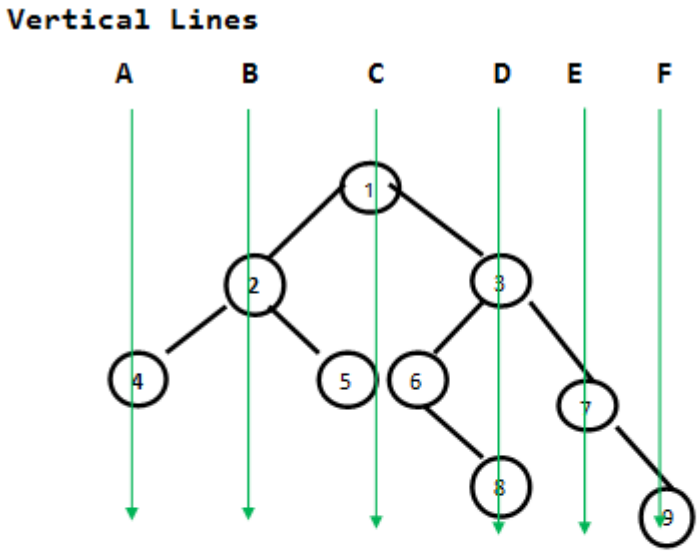
This problem can be solved more efficiently using the technique discussed in this post. We will soon be discussing complete algorithm and implementation of more efficient method.

22. Print a Binary Tree in Vertical Order | Set 2 (Map based Method)

Given a binary tree, print it vertically. The following example illustrates vertical order traversal.




```
The output of print this tree vertically will be:
4
2
1 5 6
3 8
7
9
```



Vertical order traversal is:

A- 4

B- 2

C- 1 5 6

D- 3 8

E- 7

F- 9

We have discussed a O(n2) solution in the previous post. In this post, an efficient solution based on the hash map is discussed. We need to check the Horizontal Distances from the root for all nodes. If two nodes have the same Horizontal Distance (HD), then they are on the same vertical line. The idea of HD is simple. HD for root is 0, a right edge (edge connecting to right subtree) is considered as +1 horizontal distance and a left edge is considered as -1 horizontal distance. For example, in the above tree, HD for Node 4 is at -2, HD for Node 2 is -1, HD for 5 and 6 is 0 and HD for node 7 is +2.

We can do preorder traversal of the given Binary Tree. While traversing the tree, we can recursively calculate HDs. We initially pass the horizontal distance as 0 for root. For left subtree, we pass the Horizontal Distance as Horizontal distance of root minus 1. For right subtree, we pass the Horizontal Distance as Horizontal Distance of root plus 1. For every HD value, we maintain a list of

nodes in a hash map. Whenever we see a node in traversal, we go to the hash map entry and add the node to the hash map using HD as a key in a map.

Following is the C++ implementation of the above method. Thanks to Chirag for providing the below C++ implementation.

```
// C++ program for printing vertical order of a given binary tree
#include <iostream>
#include <vector>
#include <map>
using namespace std;

// Structure for a binary tree node
struct Node
{
    int key;
    Node *left, *right;
};

// A utility function to create a new node
struct Node* newNode(int key)
{
    struct Node* node = new Node;
    node->key = key;
    node->left = node->right = NULL;
    return node;
}

// Utility function to store vertical order in map 'm'
// 'hd' is horigontal distance of current node from root.
// 'hd' is initially passed as 0
void getVerticalOrder(Node* root, int hd, map<int, vector<int>>
&m)
{
    // Base case
    if (root == NULL)
        return;

    // Store current node in map 'm'
    m[hd].push_back(root->key);

    // Store nodes in left subtree
    getVerticalOrder(root->left, hd-1, m);

    // Store nodes in right subtree
    getVerticalOrder(root->right, hd+1, m);
```

```
}

// The main function to print vertical order of a binary tree
// with the given root
void printVerticalOrder(Node* root)
{
    // Create a map and store vertical order in map using
    // function getVerticalOrder()
    map < int,vector<int> > m;
    int hd = 0;
    getVerticalOrder(root, hd,m);

    // Traverse the map and print nodes at every horigontal
    // distance (hd)
    map< int,vector<int> > :: iterator it;
    for (it=m.begin(); it!=m.end(); it++)
    {
        for (int i=0; i<it->second.size(); ++i)
            cout << it->second[i] << " ";
        cout << endl;
    }
}

// Driver program to test above functions
int main()
{
    Node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->left = newNode(6);
    root->right->right = newNode(7);
    root->right->left->right = newNode(8);
    root->right->right->right = newNode(9);
    cout << "Vertical order traversal is n";
    printVerticalOrder(root);
    return 0;
}
```

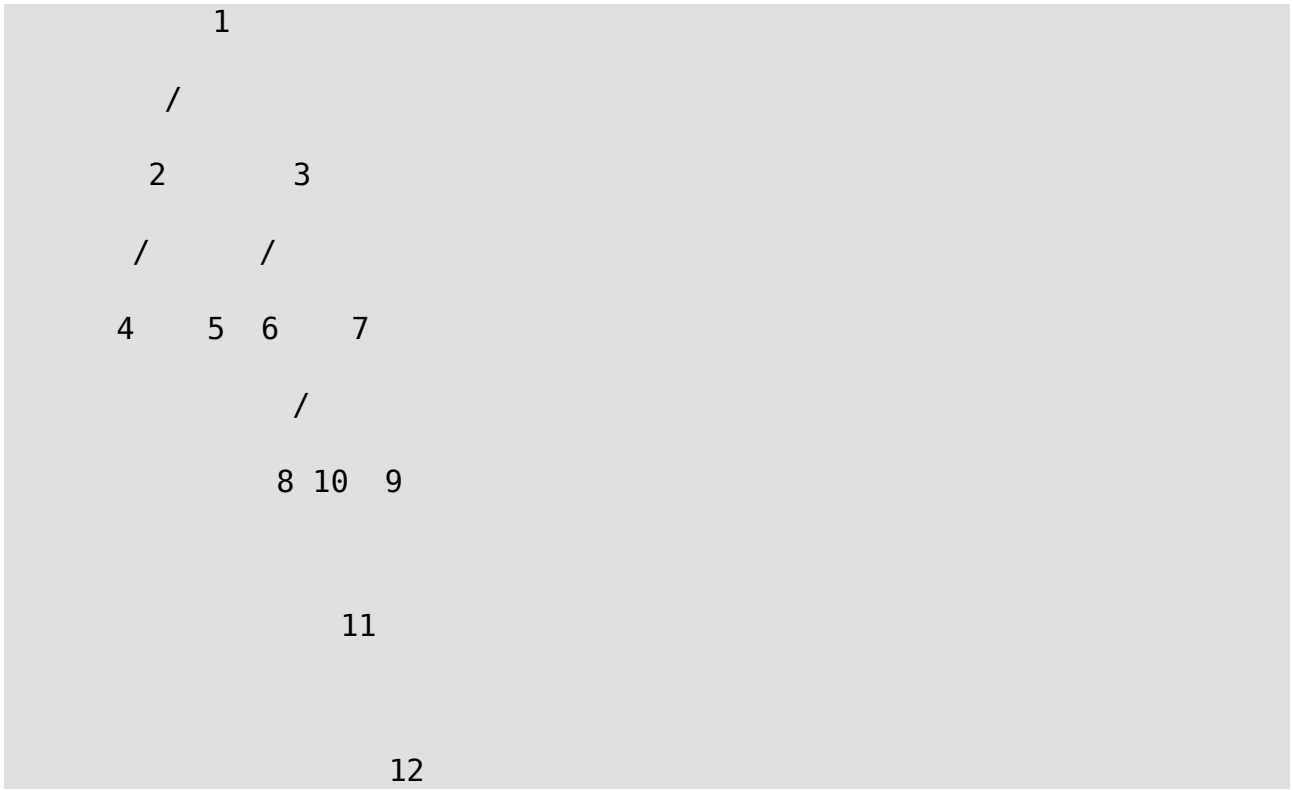
Output:

```
Vertical order traversal is

4

2

1 5 6

3 8
```

```
7

9
```

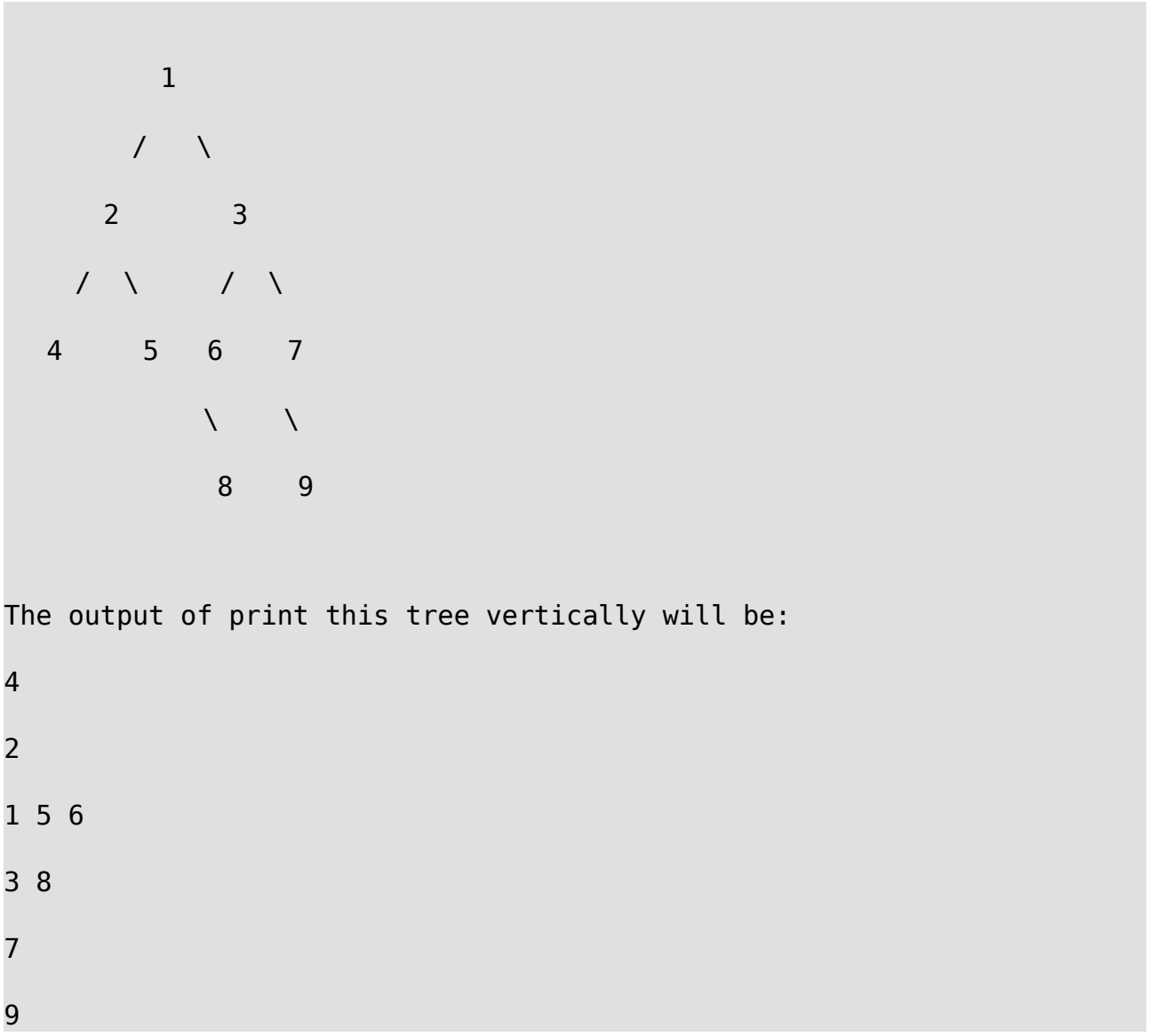
Time Complexity of hashing based solution can be considered as O(n) under the assumption that we have good hashing function that allows insertion and retrieval operations in O(1) time. In the above C++ implementation, map of STL is used. map in STL is typically implemented using a Self-Balancing Binary Search Tree where all operations take O(Logn) time. Therefore time complexity of the above implementation is O(nLogn).

Note that the above solution may print nodes in same vertical order as they appear in tree. For example, the above program prints 12 before 9. See this for a sample run.



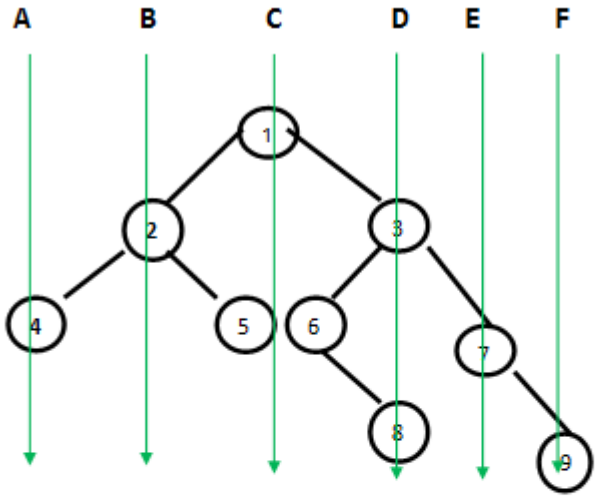
23. Print a Binary Tree in Vertical Order | Set 3 (Using Level Order Traversal)

Given a binary tree, print it vertically. The following example illustrates vertical order traversal.




```
The output of print this tree vertically will be:
4
2
1 5 6
3 8
7
9
```

Vertical Lines



Vertical order traversal is:

A- 4

B- 2

C- 1 5 6

D- 3 8

E- 7

F- 9

We have discussed an efficient approach in below post.

Print a Binary Tree in Vertical Order | Set 2 (Hashmap based Method)

The above solution uses preorder traversal and Hashmap to store nodes according to horizontal distances. Since above approach uses preorder traversal, nodes in a vertical line may not be prined in same order as they appear in tree. For example, the above solution prints 12 before 9 in below tree. See this for a sample run.

```
            1
          /    \
        2       3
      /  \    /  \
```

```
     4    5  6    7
              \  /  \
               8 10  9
                   \
                   11
                     \
                      12
```

If we use level order traversal, we can make sure that if a node like 12 comes below in same vertical line, it is printed after a node like 9 which comes above in vertical line.

```
1. To maintain a hash for the branch of each node.
2. Traverse the tree in level order fashion.
3. In level order traversal, maintain a queue
   which holds, node and its vertical branch.
    * pop from queue.
    * add this node's data in vector corresponding
      to its branch in the hash.
    * if this node hash left child, insert in the
      queue, left with branch - 1.
    * if this node hash right child, insert in the
      queue, right with branch + 1.
```

```cpp
// C++ program for printing vertical order
// of a given binary tree usin BFS.
#include<bits/stdc++.h>

using namespace std;

// Structure for a binary tree node
struct Node
{
```

```
    int key;
    Node *left, *right;
};

// A utility function to create a new node
Node* newNode(int key)
{
    Node* node = new Node;
    node->key = key;
    node->left = node->right = NULL;
    return node;
}

// The main function to print vertical oder of a
// binary tree with given root
void printVerticalOrder(Node* root)
{
    // Base case
    if (!root)
        return;

    // Create a map and store vertical oder in
    // map using function getVerticalOrder()
    map < int,vector<int> > m;
    int hd = 0;

    // Create queue to do level order traversal.
    // Every item of queue contains node and
    // horizontal distance.
    queue<pair<Node*, int> > que;
    que.push(make_pair(root, hd));

     while (!que.empty())
     {
        // pop from queue front
        pair<Node *,int> temp = que.front();
        que.pop();
        hd = temp.second;
        Node* node = temp.first;

        // insert this node's data in vector of hash
        m[hd].push_back(node->key);

        if (node->left != NULL)
            que.push(make_pair(node->left, hd-1));
        if (node->right != NULL)
            que.push(make_pair(node->right, hd+1));
    }
```

```
    // Traverse the map and print nodes at
    // every horigontal distance (hd)
    map< int,vector<int> > :: iterator it;
    for (it=m.begin(); it!=m.end(); it++)
    {
        for (int i=0; i<it->second.size(); ++i)
            cout << it->second[i] << " ";
        cout << endl;
    }
}

// Driver program to test above functions
int main()
{
    Node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->left = newNode(6);
    root->right->right = newNode(7);
    root->right->left->right = newNode(8);
    root->right->right->right = newNode(9);
    root->right->right->left= newNode(10);
    root->right->right->left->right= newNode(11);
    root->right->right->left->right->right= newNode(12);
    cout << "Vertical order traversal is \n";
    printVerticalOrder(root);
    return 0;
}
```

Output:

```
Vertical order traversal is

4

2

1  5  6

3  8  10

7  11

9  12
```

Time Complexity of above implementation is O(n Log n). Note that above implementation uses map which is implemented using self-balancing BST.

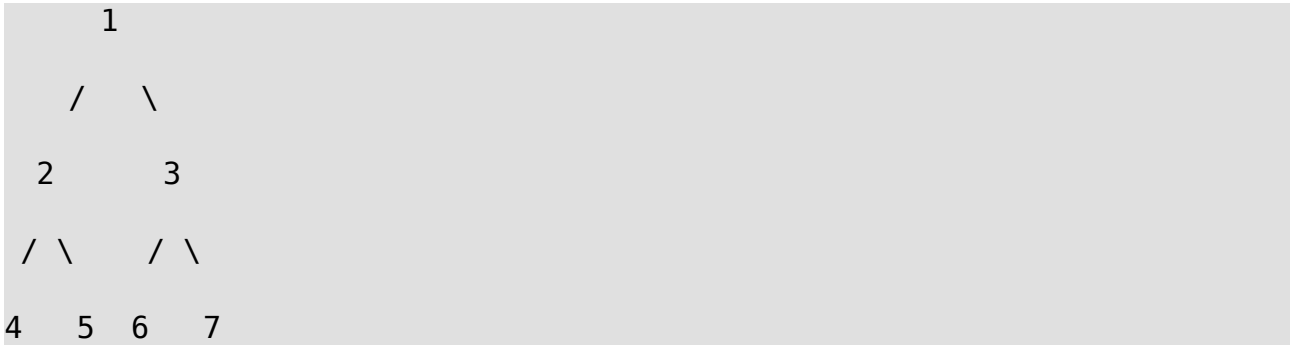
We can reduce time complexity to O(n) using unordered_map. To print nodes in desired order, we can have 2 variables denoting min and max horizontal distance. We can simply iterate from min to max horizontal distance and get corresponding values from Map. So it is O(n)

Auxiliary Space : O(n)

24. Vertical Sum in a given Binary Tree | Set 1

Given a Binary Tree, find the vertical sum of the nodes that are in the same vertical line. Print all sums through different vertical lines.

Examples:



The tree has 5 vertical lines

Vertical-Line-1 has only one node 4 => vertical sum is 4
Vertical-Line-2: has only one node 2=> vertical sum is 2
Vertical-Line-3: has three nodes: 1,5,6 => vertical sum is 1+5+6 = 12
Vertical-Line-4: has only one node 3 => vertical sum is 3
Vertical-Line-5: has only one node 7 => vertical sum is 7

So expected output is 4, 2, 12, 3 and 7

We need to check the Horizontal Distances from the root for all nodes. If two nodes have the same Horizontal Distance (HD), then they are on the same vertical line. The idea of HD is simple. HD for root is 0, a right edge (edge connecting to right subtree) is considered as +1 horizontal distance and a left edge is considered as -1 horizontal distance. For example, in the above tree, HD for Node 4 is at -2, HD for Node 2 is -1, HD for 5 and 6 is 0 and HD for node 7 is +2.
We can do an in-order traversal of the given Binary Tree. While traversing the tree, we can recursively calculate HDs. We initially pass the horizontal distance

as 0 for root. For left subtree, we pass the Horizontal Distance as Horizontal distance of root minus 1. For right subtree, we pass the Horizontal Distance as Horizontal Distance of root plus 1.

Following is Java implementation for the same. HashMap is used to store the vertical sums for different horizontal distances. Thanks to Nages for suggesting this method.

```
// C++ program to find Vertical Sum in
// a given Binary Tree
#include<bits/stdc++.h>
using namespace std;

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

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

// Traverses the tree in in-order form and
// populates a hashMap that contains the
// vertical sum
void verticalSumUtil(Node *node, int hd,
                     map<int, int> &Map)
{
    // Base case
    if (node == NULL) return;

    // Recur for left subtree
    verticalSumUtil(node->left, hd-1, Map);

    // Add val of current node to
    // map entry of corresponding hd
    Map[hd] += node->data;

    // Recur for right subtree
    verticalSumUtil(node->right, hd+1, Map);
```

```
}

// Function to find vertical sum
void verticalSum(Node *root)
{
    // a map to store sum of nodes for each
    // horizontal distance
    map < int, int> Map;
    map < int, int> :: iterator it;

    // populate the map
    verticalSumUtil(root, 0, Map);

    // Prints the values stored by VerticalSumUtil()
    for (it = Map.begin(); it != Map.end(); ++it)
    {
        cout << it->first << ": "
             << it->second << endl;
    }
}

// Driver program to test above functions
int main()
{
    // Create binary tree as shown in above figure
    Node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->left = newNode(6);
    root->right->right = newNode(7);
    root->right->left->right = newNode(8);
    root->right->right->right = newNode(9);

    cout << "Following are the values of vertical"
            " sums with the positions of the "
            "columns with respect to root\n";
    verticalSum(root);

    return 0;
}
// This code is contributed by Aditi Sharma
```

Vertical Sum in Binary Tree | Set 2 (Space Optimized)

Time Complexity: O(n)

25. Vertical Sum in Binary Tree | Set 2 (Space Optimized)

Given a Binary Tree, find vertical sum of the nodes that are in same vertical line. Print all sums through different vertical lines.

Examples:

```
      1
    /   \
  2       3
 / \     / \
4   5  6   7
```

The tree has 5 vertical lines

Vertical-Line-1 has only one node 4 => vertical sum is 4

Vertical-Line-2: has only one node 2=> vertical sum is 2

Vertical-Line-3: has three nodes: 1,5,6 => vertical sum is 1+5+6 = 12

Vertical-Line-4: has only one node 3 => vertical sum is 3

Vertical-Line-5: has only one node 7 => vertical sum is 7

So expected output is 4, 2, 12, 3 and 7

We have discussed Hashing Based Solution in Set 1. Hashing based solution requires a Hash Table to be maintained. We know that hashing requires more space than the number of entries in it. In this post, Doubly Linked List based solution is discussed. The solution discussed here requires only n nodes of linked list where n is total number of vertical lines in binary tree. Below is algorithm.

```
verticalSumDLL(root)
1)  Create a node of doubly linked list node
    with value 0. Let the node be llnode.
```

```
2)  verticalSumDLL(root, llnode)


verticalSumDLL(tnode, llnode)
1) Add current node's data to its vertical line
       llnode.data = llnode.data + tnode.data;

2) Recursively process left subtree
   // If left child is not empty

   if (tnode.left != null)

   {

      if (llnode.prev == null)

      {

         llnode.prev = new LLNode(0);

         llnode.prev.next = llnode;

      }

      verticalSumDLLUtil(tnode.left, llnode.prev);
   }

3)  Recursively process right subtree
   if (tnode.right != null)

   {

      if (llnode.next == null)

      {

         llnode.next = new LLNode(0);

         llnode.next.prev = llnode;

      }

      verticalSumDLLUtil(tnode.right, llnode.next);
   }
```

```
/// C++ program of space optimized solution
/// to find vertical sum of binary tree.
```

```cpp
#include <bits/stdc++.h>

using namespace std;

/// Tree node structure
struct TNode{
    int key;
    struct TNode *left, *right;
};

/// Function to create new tree node
TNode* newTNode(int key)
{
    TNode* temp = new TNode;
    temp->key = key;
    temp->left = temp->right = NULL;
    return temp;
}

/// Doubly linked list structure
struct LLNode{
    int key;
    struct LLNode *prev, *next;
};

/// Function to create new Linked List Node
LLNode* newLLNode(int key)
{
    LLNode* temp = new LLNode;
    temp->key = key;
    temp->prev = temp->next = NULL;
    return temp;
}

/// Function that creates Linked list and store
/// vertical sum in it.
void verticalSumDLLUtil(TNode *root, LLNode *sumNode)
{
    /// Update sum of current line by adding value
    /// of current tree node.
    sumNode->key = sumNode->key + root->key;

    /// Recursive call to left subtree.
    if(root->left)
    {
        if(sumNode->prev == NULL)
        {
            sumNode->prev = newLLNode(0);
            sumNode->prev->next = sumNode;
        }
```

```
        verticalSumDLLUtil(root->left, sumNode->prev);
    }

    /// Recursive call to right subtree.
    if(root->right)
    {
        if(sumNode->next == NULL)
        {
            sumNode->next = newLLNode(0);
            sumNode->next->prev = sumNode;
        }
        verticalSumDLLUtil(root->right, sumNode->next);
    }
}

/// Function to print vertical sum of Tree.
/// It uses verticalSumDLLUtil() to calculate sum.
void verticalSumDLL(TNode* root)
{
    /// Create Linked list node for
    /// line passing through root.
    LLNode* sumNode = newLLNode(0);

    /// Compute vertical sum of different lines.
    verticalSumDLLUtil(root, sumNode);

    /// Make doubly linked list pointer point
    /// to first node in list.
    while(sumNode->prev != NULL){
        sumNode = sumNode->prev;
    }

    /// Print vertical sum of different lines
    /// of binary tree.
    while(sumNode != NULL){
        cout << sumNode->key <<" ";
        sumNode = sumNode->next;
    }
}

int main()
{
    /*
                  1
                 / \
                /   \
               2     3
              / \   / \
             /   \ /   \
```

```
            4    5 6    7
    */
    TNode *root = newTNode(1);
    root->left = newTNode(2);
    root->right = newTNode(3);
    root->left->left = newTNode(4);
    root->left->right = newTNode(5);
    root->right->left = newTNode(6);
    root->right->right = newTNode(7);

    cout << "Vertical Sums are\n";
    verticalSumDLL(root);
    return 0;
}

// This code is contributed by <b>Rahul Titare</b>
```

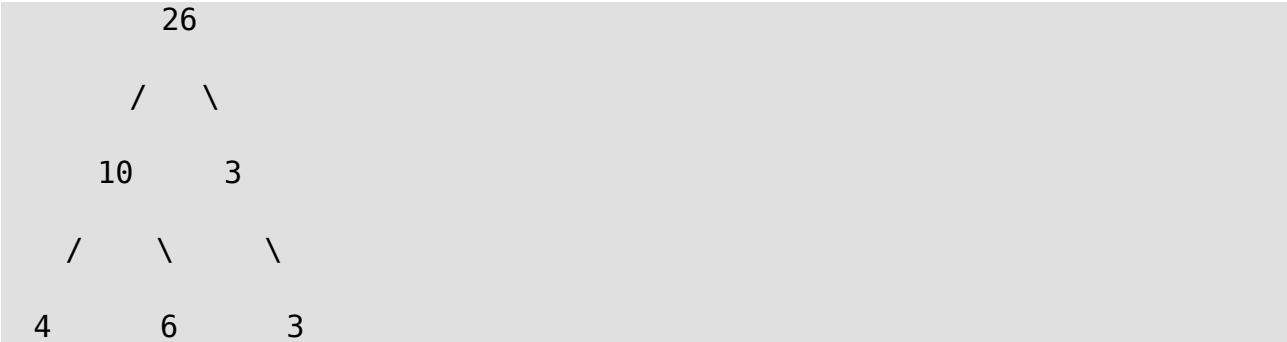
Output :

```
Vertical Sums are

4 2 12 3 7
```

26. Check if a given Binary Tree is SumTree

Write a function that returns true if the given Binary Tree is SumTree else false. A SumTree is a Binary Tree where the value of a node is equal to sum of the nodes present in its left subtree and right subtree. An empty tree is SumTree and sum of an empty tree can be considered as 0. A leaf node is also considered as SumTree.

Following is an example of SumTree.



Method 1 (Simple)

Get the sum of nodes in left subtree and right subtree. Check if the sum calculated is equal to root's data. Also, recursively check if the left and right subtrees are SumTrees.

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

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

/* A utility function to get the sum of values in tree with root
  as root */
int sum(struct node *root)
{
   if(root == NULL)
     return 0;
   return sum(root->left) + root->data + sum(root->right);
}
```

```c
/* returns 1 if sum property holds for the given
    node and both of its children */
int isSumTree(struct node* node)
{
    int ls, rs;

    /* If node is NULL or it's a leaf node then
       return true */
    if(node == NULL ||
            (node->left == NULL && node->right == NULL))
        return 1;

   /* Get sum of nodes in left and right subtrees */
   ls = sum(node->left);
   rs = sum(node->right);

   /* if the node and both of its children satisfy the
       property return 1 else 0*/
    if((node->data == ls + rs)&&
            isSumTree(node->left) &&
            isSumTree(node->right))
        return 1;

   return 0;
}

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

/* Driver program to test above function */
int main()
{
    struct node *root  = newNode(26);
    root->left         = newNode(10);
    root->right        = newNode(3);
    root->left->left   = newNode(4);
    root->left->right  = newNode(6);
```

```
    root->right->right = newNode(3);
    if(isSumTree(root))
        printf("The given tree is a SumTree ");
    else
        printf("The given tree is not a SumTree ");

    getchar();
    return 0;
}
```

Output:

```
The given tree is a SumTree
```

Time Complexity: O(n^2) in worst case. Worst case occurs for a skewed tree.

Method 2 (Tricky)

The Method 1 uses sum() to get the sum of nodes in left and right subtrees.

The method 2 uses following rules to get the sum directly.

1) If the node is a leaf node then sum of subtree rooted with this node is equal to value of this node.

2) If the node is not a leaf node then sum of subtree rooted with this node is twice the value of this node (Assuming that the tree rooted with this node is SumTree).

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

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

/* Utillity function to check if the given node is leaf or not */
int isLeaf(struct node *node)
{
    if(node == NULL)
        return 0;
    if(node->left == NULL && node->right == NULL)
        return 1;
    return 0;
}

/* returns 1 if SumTree property holds for the given
```

```
    tree */
int isSumTree(struct node* node)
{
    int ls; // for sum of nodes in left subtree
    int rs; // for sum of nodes in right subtree

    /* If node is NULL or it's a leaf node then
       return true */
    if(node == NULL || isLeaf(node))
        return 1;

    if( isSumTree(node->left) && isSumTree(node->right))
    {
        // Get the sum of nodes in left subtree
        if(node->left == NULL)
            ls = 0;
        else if(isLeaf(node->left))
            ls = node->left->data;
        else
            ls = 2*(node->left->data);

        // Get the sum of nodes in right subtree
        if(node->right == NULL)
            rs = 0;
        else if(isLeaf(node->right))
            rs = node->right->data;
        else
            rs = 2*(node->right->data);

        /* If root's data is equal to sum of nodes in left
           and right subtrees then return 1 else return 0*/
        return(node->data == ls + rs);
    }

    return 0;
}

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

```
}

/* Driver program to test above function */
int main()
{
    struct node *root  = newNode(26);
    root->left         = newNode(10);
    root->right        = newNode(3);
    root->left->left   = newNode(4);
    root->left->right  = newNode(6);
    root->right->right = newNode(3);
    if(isSumTree(root))
        printf("The given tree is a SumTree ");
    else
        printf("The given tree is not a SumTree ");

    getchar();
    return 0;
}
```

Output:

```
The given tree is a sum tree
```

Time Complexity: O(n)

27. Print Nodes in Top View of Binary Tree

Top view of a binary tree is the set of nodes visible when the tree is viewed from the top. Given a binary tree, print the top view of it. The output nodes can be printed in any order. Expected time complexity is O(n)

A node x is there in output if x is the topmost node at its horizontal distance. Horizontal distance of left child of a node x is equal to horizontal distance of x minus 1, and that of right child is horizontal distance of x plus 1.

```
      1
   /     \
  2       3
 /  \    / \
4    5  6   7
Top view of the above binary tree is
4 2 1 3 7


       1
     /   \
   2       3
     \
       4
         \
           5
             \
               6
Top view of the above binary tree is
2 1 3 6
```

The idea is to do something similar to vertical Order Traversal. Like vertical Order Traversal, we need to put nodes of same horizontal distance together. We do a level order traversal so that the topmost node at a horizontal node is visited before any other node of same horizontal distance below it. Hashing is used to check if a node at given horizontal distance is seen or not.

```
// C++ program to print top
// view of binary tree

#include <bits/stdc++.h>
using namespace std;

// Structure of binary tree
struct Node
{
    Node * left;
    Node* right;
    int hd;
    int data;
};

// function to create a new node
Node* newNode(int key)
{
    Node* node=new Node();
    node->left = node->right = NULL;
    node->data=key;
    return node;
}

// function should print the topView of
// the binary tree
void topview(Node* root)
{
    if(root==NULL)
       return;
     queue<Node*>q;
     map<int,int> m;
     int hd=0;
     root->hd=hd;

     // push node and horizontal distance to queue
    q.push(root);

    cout<< "The top view of the tree is : \n";

    while(q.size())
    {
```

```
        hd=root->hd;

        // count function returns 1 if the container
        // contains an element whose key is equivalent
        // to hd, or returns zero otherwise.
        if(m.count(hd)==0)
        m[hd]=root->data;
        if(root->left)
        {
            root->left->hd=hd-1;
            q.push(root->left);
        }
        if(root->right)
        {
            root->right->hd=hd+1;
            q.push(root->right);
        }
        q.pop();
        root=q.front();

    }



     for(auto i=m.begin();i!=m.end();i++)
    {
        cout<<i->second<<" ";
    }

}

// Driver Program to test above functions
int main()
{
    /* Create following Binary Tree
            1
        / \
        2 3
        \
            4
            \
            5
            \
                6*/
   Node* root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->right = newNode(4);
    root->left->right->right = newNode(5);
```

```
    root->left->right->right->right = newNode(6);
    cout<<"Following are nodes in top view of Binary Tree\n";
    topview(root);
    return 0;
}
/* This code is contributed by Niteesh Kumar */
```

Output:

```
Following are nodes in top view of Binary Tree

2136
```

Another approach:

This approach does not require a queue. Here we use the two variables, one for vertical distance of current node from the root and another for the depth of the current node from the root. We use the vertical distance for indexing. If one node with the same vertical distance comes again, we check if depth of new node is lower or higher with respect to the current node with same vertical distance in the map. If depth of new node is lower, then we replace it.

```
#include<bits/stdc++.h>
using namespace std;

// Structure of binary tree
struct Node{
    Node * left;
    Node* right;
    int data;
};

// function to create a new node
Node* newNode(int key){
    Node* node=new Node();
    node->left = node->right = NULL;
    node->data=key;
    return node;
}

// function to fill the map
void fillMap(Node* root,int d,int l,map<int,pair<int,int>> &m){
    if(root==NULL) return;

    if(m.count(d)==0){
        m[d] = make_pair(root->data,l);
    }else if(m[d].second>l){
        m[d] = make_pair(root->data,l);
```

```
    }

    fillMap(root->left,d-1,l+1,m);
    fillMap(root->right,d+1,l+1,m);
}

// function should print the topView of
// the binary tree
void topView(struct Node *root){

    //map to store the pair of node value and its level
    //with respect to the vertical distance from root.
    map<int,pair<int,int>> m;

    //fillmap(root,vectical_distance_from_root,level_of_node,map)
    fillMap(root,0,0,m);

    for(auto it=m.begin();it!=m.end();it++){
        cout << it->second.first << " ";
    }
}
// Driver Program to test above functions
int main(){
    Node* root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->right = newNode(4);
    root->left->right->right = newNode(5);
    root->left->right->right->right = newNode(6);
    cout<<"Following are nodes in top view of Binary Tree\n";
    topView(root);
    return 0;
}
/* This code is contributed by Akash Debnath */
```

Output:

```
Following are nodes in top view of Binary Tree

2 1 3 6
```

28. Print nodes in top view of Binary Tree | Set 2

Top view of a binary tree is the set of nodes visible when the tree is viewed from the top. Given a binary tree, print the top view of it. The output nodes should be printed from left to right.

Note: A node x is there in output if x is the topmost node at its horizontal distance. Horizontal distance of the left child of a node x is equal to the horizontal distance of x minus 1, and that of right child is the horizontal distance of x plus 1.

```
Input:
      1
   /     \
  2       3
 /  \    / \
4    5  6   7
Output: Top view: 4 2 1 3 7

Input:
       1
     /   \
   2       3
     \
       4
         \
           5
             \
               6
Output: Top view: 2 1 3 6
```

The idea is to do something similar to Vertical Order Traversal. Like Vertical Order Traversal, we need to group nodes of same horizontal distance together. We do a level order traversal so that the topmost node at a horizontal node is visited before any other node of same horizontal distance below it. A Map is used to map the horizontal distance of the node with the node's Data and vertical distance of the node.

Below is the implementation of the above approach:

```
// C++ Program to print Top View of Binary Tree
// using hashmap and recursion
#include <bits/stdc++.h>
using namespace std;

// Node structure
struct Node {
    // Data of the node
    int data;

    // Horizontal Distance of the node
    int hd;

    // Reference to left node
    struct Node* left;

    // Reference to right node
    struct Node* right;
};

// Initialising node
struct Node* newNode(int data)
{
    struct Node* node = new Node;
    node->data = data;
    node->hd = INT_MAX;
    node->left = NULL;
    node->right = NULL;
    return node;
}

void printTopViewUtil(Node* root, int height,
    int hd, map<int, pair<int, int> >& m)
{
    // Base Case
```

```
    if (root == NULL)
        return;

    // If the node for particular horizontal distance
    // is not present in the map, add it.
    // For top view, we consider the first element
    // at horizontal distance in level order traversal
    if (m.find(hd) == m.end()) {
        m[hd] = make_pair(root->data, height);
    }
    else{
        pair<int, int> p = (m.find(hd))->second;

        if (p.second > height) {
            m.erase(hd);
            m[hd] = make_pair(root->data, height);
        }
    }

    // Recur for left and right subtree
    printTopViewUtil(root->left, height + 1, hd - 1, m);
    printTopViewUtil(root->right, height + 1, hd + 1, m);
}

void printTopView(Node* root)
{
    // Map to store horizontal distance,
    // height and node's data
    map<int, pair<int, int> > m;
    printTopViewUtil(root, 0, 0, m);

    // Print the node's value stored by printTopViewUtil()
    for (map<int, pair<int, int> >::iterator it = m.begin();
                                        it != m.end(); it++) {
        pair<int, int> p = it->second;
        cout << p.first << " ";
    }
}

int main()
{
    Node* root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->right = newNode(4);
    root->left->right->right = newNode(5);
    root->left->right->right->right = newNode(6);

    cout << "Top View : ";
```

```
    printTopView(root);

    return 0;
}
```

Output:

```
Top View : 2 1 3 6
```

29. Print nodes in the Top View of Binary Tree | Set 3

Top view of a binary tree is the set of nodes visible when the tree is viewed from the top. Given a binary tree, print the top view of it. The output nodes can be printed in any order. Expected time complexity is O(n)

A node x is there in output if x is the topmost node at its horizontal distance. Horizontal distance of left child of a node x is equal to horizontal distance of x minus 1, and that of right child is horizontal distance of x plus 1.
Example :

```
      1
   /     \
  2       3
 /  \    / \
4    5  6   7
Top view of the above binary tree is
4 2 1 3 7


       1
     /   \
   2       3
     \
       4
         \
           5
            \
```

```
             6

Top view of the above binary tree is

2 1 3 6
```

Approach:

- The idea here is to observe that, if we try to see a tree from its top, then only the nodes which are at top in vertical order will be seen.
- Start BFS from root. Maintain a queue of pairs comprising of node(Node *) type and vertical distance of node from root. Also, maintain a map which should store the node at a particular vertical distance.
- While processing a node, just check if any node is there in the map at that vertical distance.
- If any node is there, it means the node can’t be seen from top, do not consider it. Else, if there is no node at that vertical distance, store that in map and consider for top view.

Below is the implementation based on above approach:

```cpp
// C++ program to print top
// view of binary tree
#include <bits/stdc++.h>
using namespace std;

// Structure of binary tree
struct Node {
    Node* left;
    Node* right;
    int data;
};

// function to create a new node
Node* newNode(int key)
{
    Node* node = new Node();
    node->left = node->right = NULL;
    node->data = key;
    return node;
}

// function should print the topView of
```

```
// the binary tree
void topView(struct Node* root)
{
    // Base case
    if (root == NULL) {
        return;
    }

    // Take a temporary node
    Node* temp = NULL;

    // Queue to do BFS
    queue<pair<Node*, int> > q;

    // map to store node at each vartical distance
    map<int, int> mp;

    q.push({ root, 0 });

    // BFS
    while (!q.empty()) {

        temp = q.front().first;
        int d = q.front().second;
        q.pop();

        // If any node is not at that vertical distance
        // just insert that node in map and print it
        if (mp.find(d) == mp.end()) {
            cout << temp->data << " ";
            mp[d] = temp->data;
        }

        // Continue for left node
        if (temp->left) {
            q.push({ temp->left, d - 1 });
        }

        // Continue for right node
        if (temp->right) {
            q.push({ temp->right, d + 1 });
        }
    }
}

// Driver Program to test above functions
int main()
{
    /* Create following Binary Tree
```

```
          1
        / \
        2 3
        \
         4
          \
           5
            \
             6*/
    Node* root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->right = newNode(4);
    root->left->right->right = newNode(5);
    root->left->right->right->right = newNode(6);
    cout << "Following are nodes in top view of Binary Tree\n";
    topView(root);
    return 0;
}
```

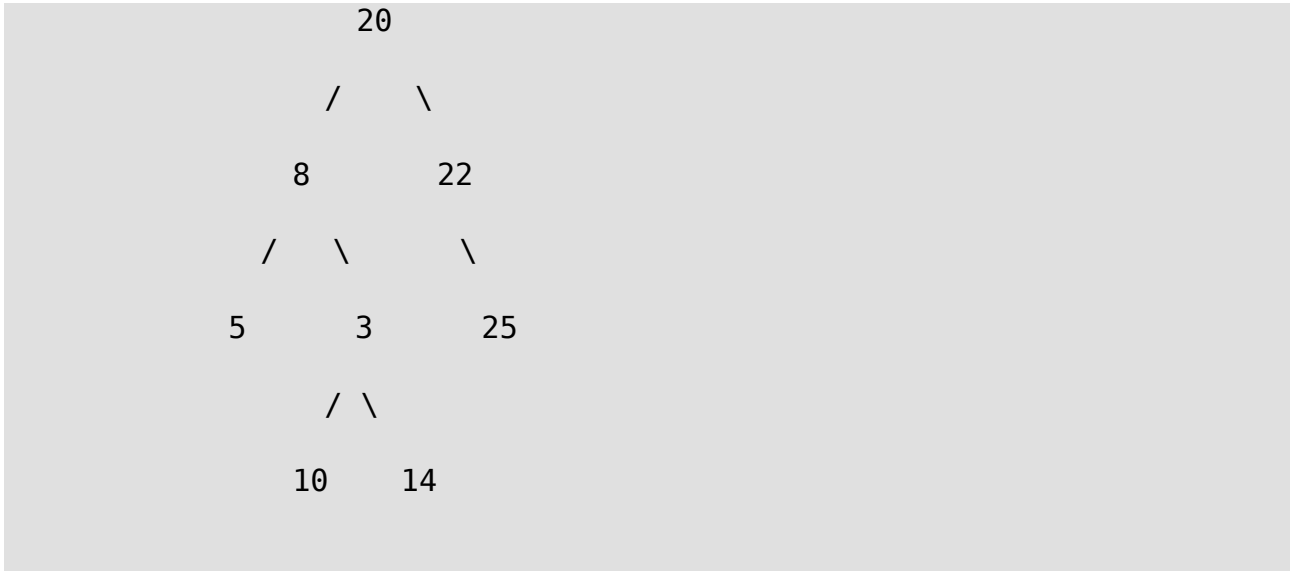
Output:

```
Following are nodes in top view of Binary Tree

2 1 3
```

30. Bottom View of a Binary Tree

Given a Binary Tree, we need to print the bottom view from left to right. A node x is there in output if x is the bottommost node at its horizontal distance. Horizontal distance of left child of a node x is equal to horizontal distance of x minus 1, and that of right child is horizontal distance of x plus 1.

Examples:



For the above tree the output should be 5, 10, 3, 14, 25.

If there are multiple bottom-most nodes for a horizontal distance from root, then print the later one in level traversal. For example, in the below diagram, 3 and 4 are both the bottom-most nodes at horizontal distance 0, we need to print 4.

```
            5      3 4     25

                  / \

                10    14
```

For the above tree the output should be 5, 10, 4, 14, 25.

Method 1 – Using Queue

The following are steps to print Bottom View of Binary Tree.

1. We put tree nodes in a queue for the level order traversal.

2. Start with the horizontal distance(hd) 0 of the root node, keep on adding left child to queue along with the horizontal distance as hd-1 and right child as hd+1.

3. Also, use a TreeMap which stores key value pair sorted on key.

4. Every time, we encounter a new horizontal distance or an existing horizontal distance put the node data for the horizontal distance as key. For the first time it will add to the map, next time it will replace the value. This will make sure that the bottom most element for that horizontal distance is present in the map and if you see the tree from beneath that you will see that element.

A Java based implementation is below :

```
// C++ Program to print Bottom View of Binary Tree
#include<bits/stdc++.h>
using namespace std;

// Tree node class
struct Node
{
    int data; //data of the node
    int hd; //horizontal distance of the node
    Node *left, *right; //left and right references

    // Constructor of tree node
    Node(int key)
    {
        data = key;
        hd = INT_MAX;
        left = right = NULL;
    }
};
```

```
// Method that prints the bottom view.
void bottomView(Node *root)
{
    if (root == NULL)
        return;

    // Initialize a variable 'hd' with 0
    // for the root element.
    int hd = 0;

    // TreeMap which stores key value pair
    // sorted on key value
    map<int, int> m;

    // Queue to store tree nodes in level
    // order traversal
    queue<Node *> q;

    // Assign initialized horizontal distance
    // value to root node and add it to the queue.
    root->hd = hd;
    q.push(root);  // In STL, push() is used enqueue an item

    // Loop until the queue is empty (standard
    // level order loop)
    while (!q.empty())
    {
        Node *temp = q.front();
        q.pop();   // In STL, pop() is used dequeue an item

        // Extract the horizontal distance value
        // from the dequeued tree node.
        hd = temp->hd;

        // Put the dequeued tree node to TreeMap
        // having key as horizontal distance. Every
        // time we find a node having same horizontal
        // distance we need to replace the data in
        // the map.
        m[hd] = temp->data;

        // If the dequeued node has a left child, add
        // it to the queue with a horizontal distance hd-1.
        if (temp->left != NULL)
        {
            temp->left->hd = hd-1;
            q.push(temp->left);
        }

```

```
        // If the dequeued node has a right child, add
        // it to the queue with a horizontal distance
        // hd+1.
        if (temp->right != NULL)
        {
            temp->right->hd = hd+1;
            q.push(temp->right);
        }
    }

    // Traverse the map elements using the iterator.
    for (auto i = m.begin(); i != m.end(); ++i)
        cout << i->second << " ";
}

// Driver Code
int main()
{
    Node *root = new Node(20);
    root->left = new Node(8);
    root->right = new Node(22);
    root->left->left = new Node(5);
    root->left->right = new Node(3);
    root->right->left = new Node(4);
    root->right->right = new Node(25);
    root->left->right->left = new Node(10);
    root->left->right->right = new Node(14);
    cout << "Bottom view of the given binary tree :\n"
    bottomView(root);
    return 0;
}
```

Output:

```
Bottom view of the given binary tree:

5 10 4 14 25
```

Method 2- Using HashMap()

This method is contributed by Ekta Goel.

Approach:

Create a map like, map where key is the horizontal distance and value is a pair(a, b) where a is the value of the node and b is the height of the node.

Perform a pre-order traversal of the tree. If the current node at a horizontal distance of h is the first we've seen, insert it in the map. Otherwise, compare

the node with the existing one in map and if the height of the new node is greater, update in the Map.

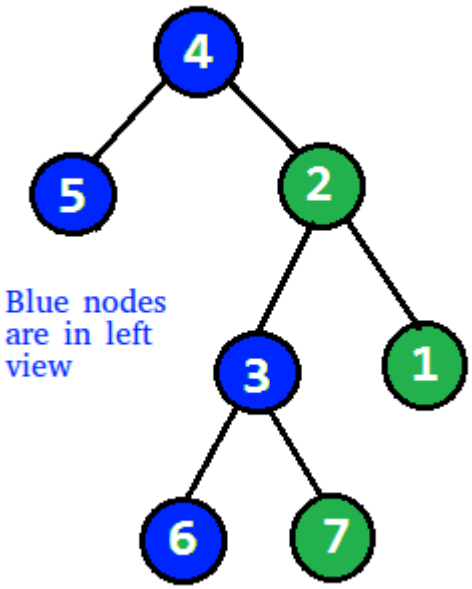
Below is the implementation of the above:

```
// C++ Program to print Bottom View of Binary Tree
#include < bits / stdc++.h >
#include < map >
using namespace std;

// Tree node class
struct Node
{
    // data of the node
    int data;

    // horizontal distance of the node
    int hd;

    //left and right references
    Node * left, * right;

    // Constructor of tree node
    Node(int key)
    {
        data = key;
        hd = INT_MAX;
        left = right = NULL;
    }
};

void printBottomViewUtil(Node * root, int curr, int hd, map <int,
pair <int, int>> & m)
{
    // Base case
    if (root == NULL)
        return;

    // If node for a particular
    // horizontal distance is not
    // present, add to the map.
    if (m.find(hd) == m.end())
    {
        m[hd] = make_pair(root -> data, curr);
    }
    // Compare height for already
    // present node at similar horizontal
    // distance
    else
```

```
    {
        pair < int, int > p = m[hd];
        if (p.second <= curr)
        {
            m[hd].second = curr;
            m[hd].first = root -> data;
        }
    }

    // Recur for left subtree
    printBottomViewUtil(root -> left, curr + 1, hd - 1, m);

    // Recur for right subtree
    printBottomViewUtil(root -> right, curr + 1, hd + 1, m);
}

void printBottomView(Node * root)
{

    // Map to store Horizontal Distance,
    // Height and Data.
    map < int, pair < int, int > > m;

    printBottomViewUtil(root, 0, 0, m);

     // Prints the values stored by printBottomViewUtil()
    map < int, pair < int, int > > ::iterator it;
    for (it = m.begin(); it != m.end(); ++it)
    {
        pair < int, int > p = it -> second;
        cout << p.first << " ";
    }
}

int main()
{
    Node * root = new Node(20);
    root -> left = new Node(8);
    root -> right = new Node(22);
    root -> left -> left = new Node(5);
    root -> left -> right = new Node(3);
    root -> right -> left = new Node(4);
    root -> right -> right = new Node(25);
    root -> left -> right -> left = new Node(10);
    root -> left -> right -> right = new Node(14);
    cout << "Bottom view of the given binary tree :\n";
    printBottomView(root);
    return 0;
```

31. Print Left View of a Binary Tree

Given a Binary Tree, print left view of it. Left view of a Binary Tree is set of nodes visible when tree is visited from left side.



Examples:

```
Input :
                1
              /   \
             2     3
           / \      \
          4   5      6
Output : 1 2 4


Input :
       1
```

```
     /   \
   2       3
     \
       4
         \
           5
            \
              6
Output :1 2 4 5 6
```

The left view contains all nodes that are first nodes in their levels. A simple solution is to do level order traversal and print the first node in every level. The problem can also be solved using simple recursive traversal. We can keep track of the level of a node by passing a parameter to all recursive calls. The idea is to keep track of the maximum level also. Whenever we see a node whose level is more than maximum level so far, we print the node because this is the first node in its level (Note that we traverse the left subtree before right subtree). Following is the implementation-

```
// C++ program to print left
// view of Binary Tree
#include <bits/stdc++.h>
using namespace std;

class node {
public:
    int data;
    node *left, *right;
};

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

```
// Recursive function to print left view of a binary tree.
void leftViewUtil(node* root, int level, int* max_level)
{
    // Base Case
    if (root == NULL)
        return;

    // If this is the first node of its level
    if (*max_level < level) {
        cout << root->data << "\t";
        *max_level = level;
    }

    // Recur for left and right subtrees
    leftViewUtil(root->left, level + 1, max_level);
    leftViewUtil(root->right, level + 1, max_level);
}

// A wrapper over leftViewUtil()
void leftView(node* root)
{
    int max_level = 0;
    leftViewUtil(root, 1, &max_level);
}

// Driver code
int main()
{
    node* root = newNode(12);
    root->left = newNode(10);
    root->right = newNode(30);
    root->right->left = newNode(25);
    root->right->right = newNode(40);

    leftView(root);

    return 0;
}

// This code is contributed by rathbhupendra
```

Output:

```
12      10      25
```

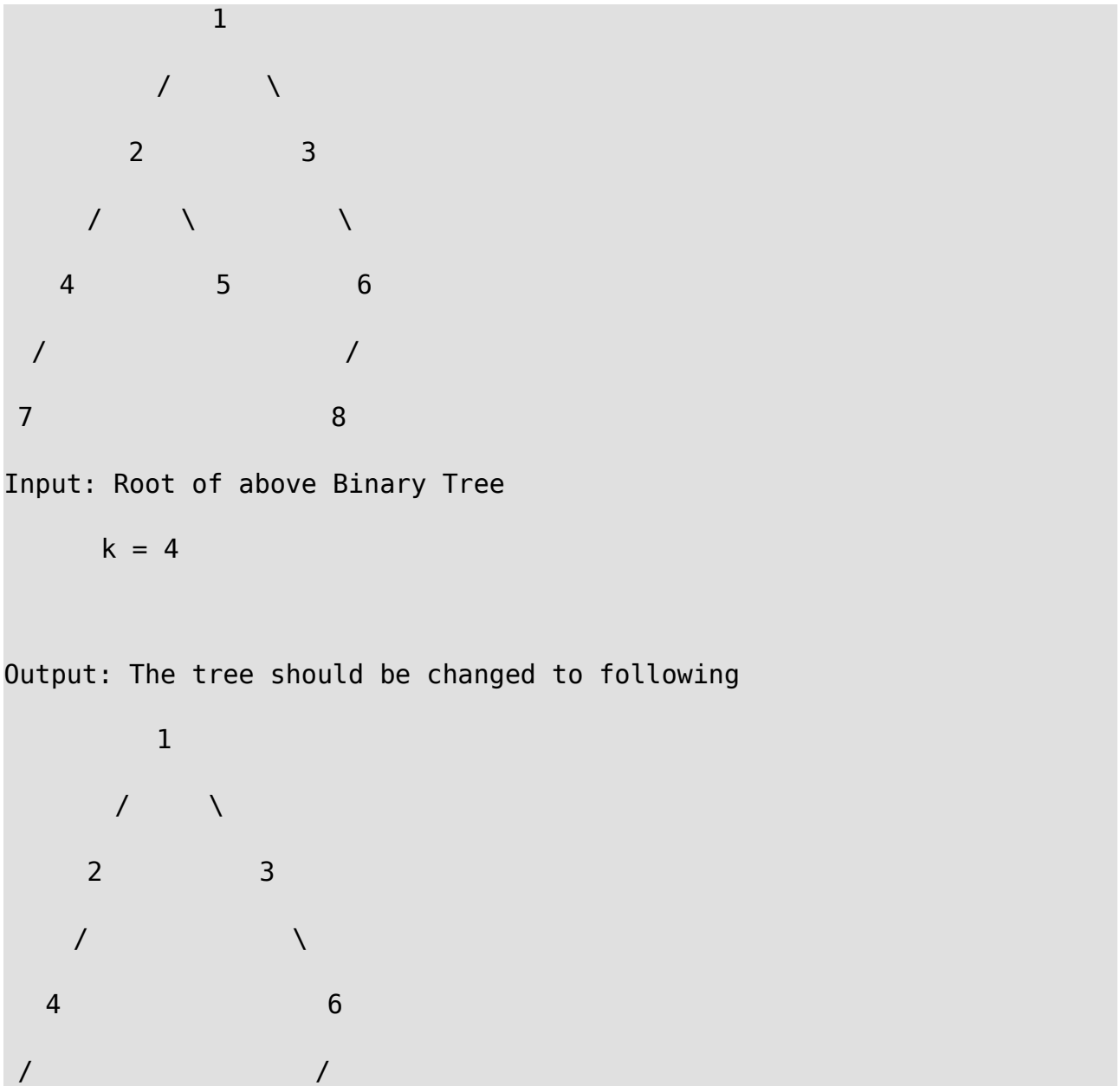
Time Complexity: The function does a simple traversal of the tree, so the complexity is O(n).

Auxiliary Space: O(n), due to the stack space during recursive call.

32. Remove nodes on root to leaf paths of length < K

Given a Binary Tree and a number k, remove all nodes that lie only on root to leaf path(s) of length smaller than k. If a node X lies on multiple root-to-leaf paths and if any of the paths has path length >= k, then X is not deleted from Binary Tree. In other words a node is deleted if all paths going through it have lengths smaller than k.

Consider the following example Binary Tree




```
Input: Root of above Binary Tree
      k = 4


Output: The tree should be changed to following
          1
       /     \
     2          3
    /             \
  4                 6
 /                  /
```

```
7                  8

There are 3 paths

i) 1->2->4->7      path length = 4

ii) 1->2->5        path length = 3

iii) 1->3->6->8    path length = 4

There is only one path " 1->2->5 " of length smaller than 4.

The node 5 is the only node that lies only on this path, so

node 5 is removed.

Nodes 2 and 1 are not removed as they are parts of other paths

of length 4 as well.


If k is 5 or greater than 5, then whole tree is deleted.


If k is 3 or less than 3, then nothing is deleted.
```

We strongly recommend to minimize your browser and try this yourself first

The idea here is to use post order traversal of the tree. Before removing a node we need to check that all the children of that node in the shorter path are already removed.

There are 2 cases:

i) This node becomes a leaf node in which case it needs to be deleted.

ii) This node has other child on a path with path length >= k. In that case it needs not to be deleted.

The implementation of above approach is as below :

```
// C++ program to remove nodes on root to leaf paths of length < K
#include<bits/stdc++.h>
using namespace std;
```

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

//New node of a tree
Node *newNode(int data)
{
    Node *node = new Node;
    node->data = data;
    node->left = node->right = NULL;
    return node;
}

// Utility method that actually removes the nodes which are not
// on the pathLen >= k. This method can change the root as well.
Node *removeShortPathNodesUtil(Node *root, int level, int k)
{
    //Base condition
    if (root == NULL)
        return NULL;

    // Traverse the tree in postorder fashion so that if a leaf
    // node path length is shorter than k, then that node and
    // all of its descendants till the node which are not
    // on some other path are removed.
    root->left = removeShortPathNodesUtil(root->left, level + 1,
k);
    root->right = removeShortPathNodesUtil(root->right, level + 1,
k);

    // If root is a leaf node and it's level is less than k then
    // remove this node.
    // This goes up and check for the ancestor nodes also for the
    // same condition till it finds a node which is a part of
other
    // path(s) too.
    if (root->left == NULL && root->right == NULL && level < k)
    {
        delete root;
        return NULL;
    }

    // Return root;
    return root;
}

// Method which calls the utitlity method to remove the short path
// nodes.
```

```
Node *removeShortPathNodes(Node *root, int k)
{
    int pathLen = 0;
    return removeShortPathNodesUtil(root, 1, k);
}

//Method to print the tree in inorder fashion.
void printInorder(Node *root)
{
    if (root)
    {
        printInorder(root->left);
        cout << root->data << " ";
        printInorder(root->right);
    }
}

// Driver method.
int main()
{
    int k = 4;
    Node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->left->left->left = newNode(7);
    root->right->right = newNode(6);
    root->right->right->left = newNode(8);
    cout << "Inorder Traversal of Original tree" << endl;
    printInorder(root);
    cout << endl;
    cout << "Inorder Traversal of Modified tree" << endl;
    Node *res = removeShortPathNodes(root, k);
    printInorder(res);
    return 0;
}
```

Output:

```
Inorder Traversal of Original tree

7 4 2 5 1 3 8 6

Inorder Traversal of Modified tree

7 4 2 1 3 8 6
```

Time complexity of the above solution is O(n) where n is number of nodes in given Binary Tree.

33. Check if a binary tree is subtree of another binary tree | Set 1

Given two binary trees, check if the first tree is subtree of the second one. A subtree of a tree T is a tree S consisting of a node in T and all of its descendants in T. The subtree corresponding to the root node is the entire tree; the subtree corresponding to any other node is called a proper subtree.

For example, in the following case, tree S is a subtree of tree T.

```
      Tree 2
        10
      /    \
    4       6
     \
      30


      Tree 1
           26
         /   \
        10     3
      /    \     \
    4       6      3
     \
      30
```

Solution: Traverse the tree T in preorder fashion. For every visited node in the traversal, see if the subtree rooted with this node is identical to S.

Following is the implementation for this.

```
// C++ program to check if binary tree
// is subtree of another binary tree
#include<bits/stdc++.h>
using namespace std;

/* A binary tree node has data,
left child and right child */
class node
{
    public:
    int data;
    node* left;
    node* right;
};

/* A utility function to check
whether trees with roots as root1 and
root2 are identical or not */
bool areIdentical(node * root1, node *root2)
{
    /* base cases */
    if (root1 == NULL && root2 == NULL)
        return true;

    if (root1 == NULL || root2 == NULL)
        return false;

    /* Check if the data of both roots is
    same and data of left and right
    subtrees are also same */
    return (root1->data == root2->data &&
            areIdentical(root1->left, root2->left) &&
            areIdentical(root1->right, root2->right) );
}


/* This function returns true if S
is a subtree of T, otherwise false */
bool isSubtree(node *T, node *S)
{
    /* base cases */
    if (S == NULL)
        return true;

    if (T == NULL)
        return false;

    /* Check the tree with root as current node */
```

```
    if (areIdentical(T, S))
        return true;

    /* If the tree with root as current
    node doesn't match then try left
    and right subtrees one by one */
    return isSubtree(T->left, S) ||
        isSubtree(T->right, S);
}


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

/* Driver code*/
int main()
{
    // TREE 1
    /* Construct the following tree
            26
            / \
        10 3
        / \ \
    4 6 3
    \
        30
    */
    node *T = newNode(26);
    T->right         = newNode(3);
    T->right->right = newNode(3);
    T->left         = newNode(10);
    T->left->left     = newNode(4);
    T->left->left->right = newNode(30);
    T->left->right     = newNode(6);

    // TREE 2
    /* Construct the following tree
        10
        / \
    4 6
```

```
    \
        30
    */
    node *S = newNode(10);
    S->right     = newNode(6);
    S->left     = newNode(4);
    S->left->right = newNode(30);


    if (isSubtree(T, S))
        cout << "Tree 2 is subtree of Tree 1";
    else
        cout << "Tree 2 is not a subtree of Tree 1";

    return 0;
}

// This code is contributed by rathbhupendra
```

Output:

```
Tree 2 is subtree of Tree 1
```

Time Complexity: Time worst case complexity of above solution is O(mn) where m and n are number of nodes in given two trees.

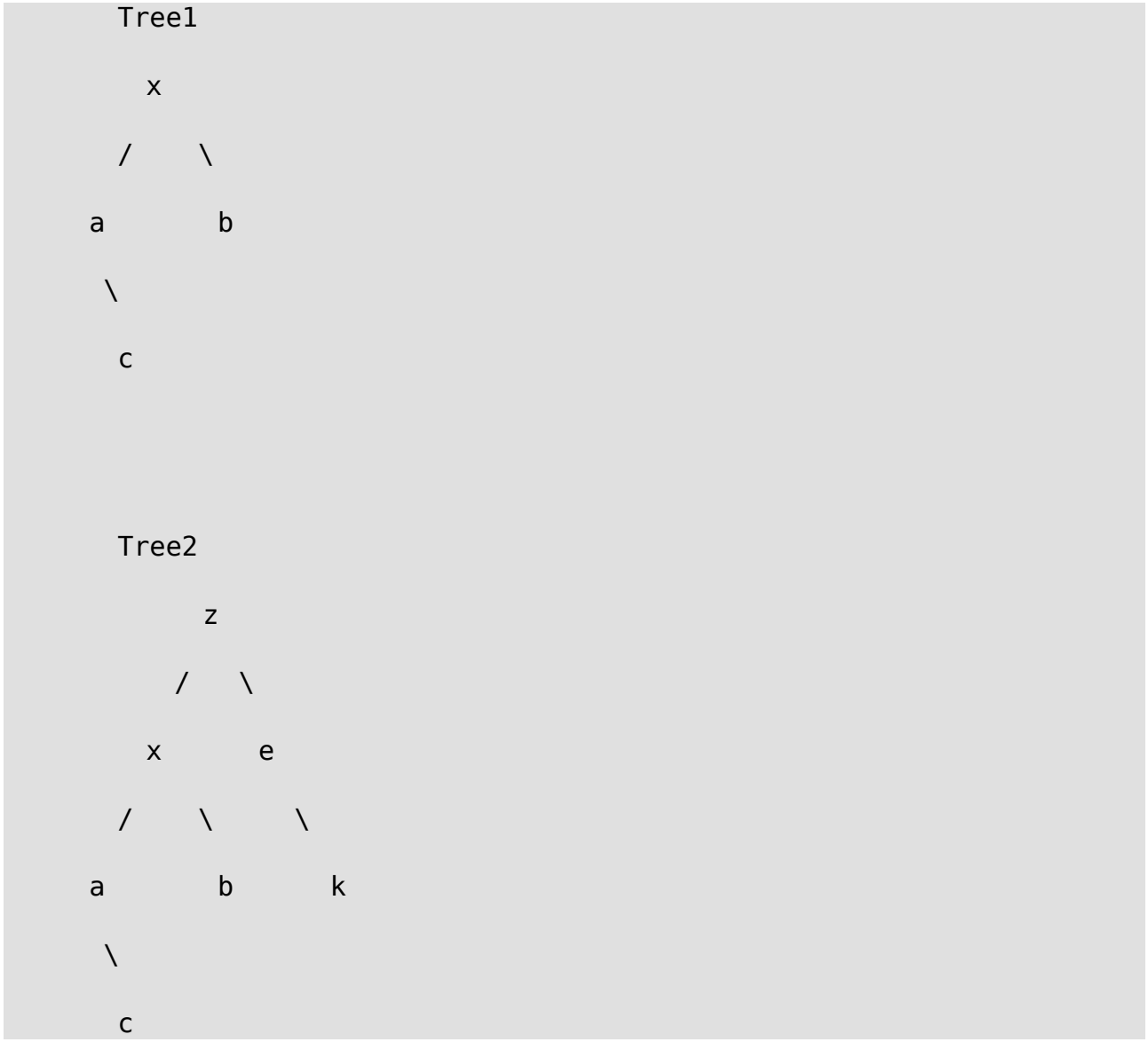
We can solve the above problem in O(n) time.

34. Check if a binary tree is subtree of another binary tree | Set 2

Given two binary trees, check if the first tree is subtree of the second one. A subtree of a tree T is a tree S consisting of a node in T and all of its descendants in T.

The subtree corresponding to the root node is the entire tree; the subtree corresponding to any other node is called a proper subtree.

For example, in the following case, Tree1 is a subtree of Tree2.



We have discussed a O(n2) solution for this problem. In this post a O(n) solution is discussed. The idea is based on the fact that inorder and preorder/postorder uniquely identify a binary tree. Tree S is a subtree of T if both inorder and preorder traversals of S arew substrings of inorder and preorder traversals of T respectively.
Following are detailed steps.

1) Find inorder and preorder traversals of T, store them in two auxiliary arrays inT[] and preT[].
2) Find inorder and preorder traversals of S, store them in two auxiliary arrays inS[] and preS[].
3) If inS[] is a subarray of inT[] and preS[] is a subarray preT[], then S is a subtree of T. Else not.
We can also use postorder traversal in place of preorder in the above algorithm.

Let us consider the above example

```
Inorder and Preorder traversals of the big tree are.

inT[]   =  {a, c, x, b, z, e, k}

preT[]  =  {z, x, a, c, b, e, k}


Inorder and Preorder traversals of small tree are

inS[]  = {a, c, x, b}

preS[] = {x, a, c, b}


We can easily figure out that inS[] is a subarray of

inT[] and preS[] is a subarray of preT[].
```

EDIT

```
The above algorithm doesn't work for cases where a tree is present

in another tree, but not as a subtree. Consider the following example.
```

```
      Tree1
        x
      /    \
     a       b
    /
   c



      Tree2
        x
      /    \
     a       b
    /         \
   c            d

Inorder and Preorder traversals of the big tree or Tree2 are.

Inorder and Preorder traversals of small tree or Tree1 are

The Tree2 is not a subtree of Tree1, but inS[] and preS[] are
subarrays of inT[] and preT[] respectively.
```

The above algorithm can be extended to handle such cases by adding a special character whenever we encounter NULL in inorder and preorder traversals. Thanks to Shivam Goel for suggesting this extension.

Following is the implementation of the above algorithm.

```cpp
#include <cstring>
#include <iostream>
using namespace std;
#define MAX 100

// Structure of a tree node
struct Node {
    char key;
    struct Node *left, *right;
};

// A utility function to create a new BST node
Node* newNode(char item)
{
    Node* temp = new Node;
    temp->key = item;
    temp->left = temp->right = NULL;
    return temp;
}

// A utility function to store inorder traversal of tree rooted
// with root in an array arr[]. Note that i is passed as reference
void storeInorder(Node* root, char arr[], int& i)
{
    if (root == NULL) {
        arr[i++] = '$';
        return;
    }
    storeInorder(root->left, arr, i);
    arr[i++] = root->key;
    storeInorder(root->right, arr, i);
}

// A utility function to store preorder traversal of tree rooted
// with root in an array arr[]. Note that i is passed as reference
void storePreOrder(Node* root, char arr[], int& i)
{
    if (root == NULL) {
        arr[i++] = '$';
        return;
    }
    arr[i++] = root->key;
    storePreOrder(root->left, arr, i);
    storePreOrder(root->right, arr, i);
}

/* This function returns true if S is a subtree of T, otherwise
false */
bool isSubtree(Node* T, Node* S)
{
```

```
    /* base cases */
    if (S == NULL)
        return true;
    if (T == NULL)
        return false;

    // Store Inorder traversals of T and S in inT[0..m-1]
    // and inS[0..n-1] respectively
    int m = 0, n = 0;
    char inT[MAX], inS[MAX];
    storeInorder(T, inT, m);
    storeInorder(S, inS, n);
    inT[m] = '\0', inS[n] = '\0';

    // If inS[] is not a substring of preS[], return false
    if (strstr(inT, inS) == NULL)
        return false;

    // Store Preorder traversals of T and S in inT[0..m-1]
    // and inS[0..n-1] respectively
    m = 0, n = 0;
    char preT[MAX], preS[MAX];
    storePreOrder(T, preT, m);
    storePreOrder(S, preS, n);
    preT[m] = '\0', preS[n] = '\0';

    // If inS[] is not a substring of preS[], return false
    // Else return true
    return (strstr(preT, preS) != NULL);
}

// Driver program to test above function
int main()
{
    Node* T = newNode('a');
    T->left = newNode('b');
    T->right = newNode('d');
    T->left->left = newNode('c');
    T->right->right = newNode('e');

    Node* S = newNode('a');
    S->left = newNode('b');
    S->left->left = newNode('c');
    S->right = newNode('d');

    if (isSubtree(T, S))
        cout << "Yes: S is a subtree of T";
    else
        cout << "No: S is NOT a subtree of T";
```

```
    return 0;
}
```

Output:

```
No: S is NOT a subtree of T
```

Time Complexity: Inorder and Preorder traversals of Binary Tree take O(n) time.

The function strstr() can also be implemented in O(n) time using KMP string matching algorithm.

Auxiliary Space: O(n)

35. Check if two nodes are cousins in a Binary Tree

Given the binary Tree and the two nodes say ‘a’ and ‘b’, determine whether the two nodes are cousins of each other or not.

Two nodes are cousins of each other if they are at same level and have different parents.

Example:

```
    6

  /   \

 3     5

/ \   / \

7   8 1   3

Say two node be 7 and 1, result is TRUE.

Say two nodes are 3 and 5, result is FALSE.

Say two nodes are 7 and 5, result is FALSE.
```

The idea is to find level of one of the nodes. Using the found level, check if ‘a’ and ‘b’ are at this level. If ‘a’ and ‘b’ are at given level, then finally check if they are not children of same parent.

Following is the implementation of the above approach.

```c
// C program to check if two Nodes in a binary tree are cousins
#include <stdio.h>
#include <stdlib.h>

// A Binary Tree Node
struct Node
{
    int data;
    struct Node *left, *right;
};
```

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

// Recursive function to check if two Nodes are siblings
int isSibling(struct Node *root, struct Node *a, struct Node *b)
{
    // Base case
    if (root==NULL)  return 0;

    return ((root->left==a && root->right==b)||
            (root->left==b && root->right==a)||
            isSibling(root->left, a, b)||
            isSibling(root->right, a, b));
}

// Recursive function to find level of Node 'ptr' in a binary tree
int level(struct Node *root, struct Node *ptr, int lev)
{
    // base cases
    if (root == NULL) return 0;
    if (root == ptr)  return lev;

    // Return level if Node is present in left subtree
    int l = level(root->left, ptr, lev+1);
    if (l != 0)  return l;

    // Else search in right subtree
    return level(root->right, ptr, lev+1);
}


// Returns 1 if a and b are cousins, otherwise 0
int isCousin(struct Node *root, struct Node *a, struct Node *b)
{
    //1. The two Nodes should be on the same level in the binary
tree.
    //2. The two Nodes should not be siblings (means that they
should
    // not have the same parent Node).
    if ((level(root,a,1) == level(root,b,1)) && !
(isSibling(root,a,b)))
        return 1;
    else return 0;
```

```
}

// Driver Program to test above functions
int main()
{
    struct Node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->left->right->right = newNode(15);
    root->right->left = newNode(6);
    root->right->right = newNode(7);
    root->right->left->right = newNode(8);

    struct Node *Node1,*Node2;
    Node1 = root->left->left;
    Node2 = root->right->right;

    isCousin(root,Node1,Node2)? puts("Yes"): puts("No");

    return 0;
}
```

Output:

```
 Yes
```

Time Complexity of the above solution is O(n) as it does at most three traversals of binary tree.

36. Check if two nodes are cousins in a Binary Tree | Set-2

Given a binary tree and the two nodes say ‘a’ and ‘b’, determine whether two given nodes are cousins of each other or not.

Two nodes are cousins of each other if they are at same level and have different parents.

Example:

```
    6
   /   \
  3     5
 / \   / \
7   8 1   3
Say two node be 7 and 1, result is TRUE.
Say two nodes are 3 and 5, result is FALSE.
Say two nodes are 7 and 5, result is FALSE.
```

A solution in Set-1 that finds whether given nodes are cousins or not by performing three traversals of binary tree has been discussed. The problem can be solved by performing level order traversal. The idea is to use a queue to perform level order traversal, in which each queue element is a pair of node and parent of that node. For each node visited in level order traversal, check if that node is either first given node or second given node. If any node is found store parent of that node. While performing level order traversal, one level is traversed at a time. If both nodes are found in given level, then their parent values are compared to check if they are siblings or not. If one node is found in given level and another is not found, then given nodes are not cousins.
Below is the implementation of above approach:

```cpp
// CPP program to check if two Nodes in
// a binary tree are cousins
// using level-order traversals
#include <bits/stdc++.h>
using namespace std;

// A Binary Tree Node
struct Node {
    int data;
    struct Node *left, *right;
};

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

// Returns true if a and b are cousins,
// otherwise false.
bool isCousin(Node* root, Node* a, Node* b)
{
    if (root == NULL)
        return false;

    // To store parent of node a.
    Node* parA = NULL;

    // To store parent of node b.
    Node* parB = NULL;

    // queue to perform level order
    // traversal. Each element of
    // queue is a pair of node and
    // its parent.
    queue<pair<Node*, Node*> > q;

    // Dummy node to act like parent
    // of root node.
    Node* tmp = newNode(-1);

    // To store front element of queue.
    pair<Node*, Node*> ele;

    // Push root to queue.
    q.push(make_pair(root, tmp));
```

```
int levSize;

while (!q.empty()) {

    // find number of elements in
    // current level.
    levSize = q.size();
    while (levSize) {

        ele = q.front();
        q.pop();

        // check if current node is node a
        // or node b or not.
        if (ele.first->data == a->data) {
            parA = ele.second;
        }

        if (ele.first->data == b->data) {
            parB = ele.second;
        }

        // push children of current node
        // to queue.
        if (ele.first->left) {
            q.push(make_pair(ele.first->left, ele.first));
        }

        if (ele.first->right) {
            q.push(make_pair(ele.first->right, ele.first));
        }

        levSize--;

        // If both nodes are found in
        // current level then no need
        // to traverse current level further.
        if (parA && parB)
            break;
    }

    // Check if both nodes are siblings
    // or not.
    if (parA && parB) {
        return parA != parB;
    }

    // If one node is found in current level
    // and another is not found, then
```

```
        // both nodes are not cousins.
        if ((parA && !parB) || (parB && !parA)) {
            return false;
        }
    }

    return false;
}
// Driver Code
int main()
{
    /*
            1
           /  \
          2    3
         / \  / \
        4   5 6  7
             \ \
             15 8
    */

    struct Node* root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->left->right->right = newNode(15);
    root->right->left = newNode(6);
    root->right->right = newNode(7);
    root->right->left->right = newNode(8);

    struct Node *Node1, *Node2;
    Node1 = root->left->left;
    Node2 = root->right->right;

    isCousin(root, Node1, Node2) ? puts("Yes") : puts("No");

    return 0;
}
```

Output:

```
Yes
```

Time Complexity: O(n)

Auxiliary Space: O(n

37. Sorted Array to Balanced BST

Given a sorted array. Write a function that creates a Balanced Binary Search Tree using array elements.

Examples:

```
Input:  Array {1, 2, 3}

Output: A Balanced BST

    2

  /  \

 1    3


Input: Array {1, 2, 3, 4}

Output: A Balanced BST

     3

   /  \

  2    4

 /

1
```

Algorithm

In the previous post, we discussed construction of BST from sorted Linked List. Constructing from sorted array in O(n) time is simpler as we can get the middle element in O(1) time. Following is a simple algorithm where we first find the middle node of list and make it root of the tree to be constructed.

```
1) Get the Middle of the array and make it root.

2) Recursively do same for left half and right half.

     a) Get the middle of left half and make it left child of the root
```

```
        created in step 1.

    b) Get the middle of right half and make it right child of the

        root created in step 1.
```

Following is the implementation of the above algorithm. The main code which creates Balanced BST is highlighted.

```
// C++ program to print BST in given range
#include<bits/stdc++.h>
using namespace std;

/* A Binary Tree node */
class TNode
{
    public:
    int data;
    TNode* left;
    TNode* right;
};

TNode* newNode(int data);

/* A function that constructs Balanced
Binary Search Tree from a sorted array */
TNode* sortedArrayToBST(int arr[],
                        int start, int end)
{
    /* Base Case */
    if (start > end)
    return NULL;

    /* Get the middle element and make it root */
    int mid = (start + end)/2;
    TNode *root = newNode(arr[mid]);

    /* Recursively construct the left subtree
    and make it left child of root */
    root->left = sortedArrayToBST(arr, start,
                                  mid - 1);

    /* Recursively construct the right subtree
    and make it right child of root */
    root->right = sortedArrayToBST(arr, mid + 1, end);

    return root;
}
```

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

    return node;
}

/* A utility function to print
preorder traversal of BST */
void preOrder(TNode* node)
{
    if (node == NULL)
        return;
    cout << node->data << " ";
    preOrder(node->left);
    preOrder(node->right);
}

// Driver Code
int main()
{
    int arr[] = {1, 2, 3, 4, 5, 6, 7};
    int n = sizeof(arr) / sizeof(arr[0]);

    /* Convert List to BST */
    TNode *root = sortedArrayToBST(arr, 0, n-1);
    cout << "PreOrder Traversal of constructed BST \n";
    preOrder(root);

    return 0;
}

// This code is contributed by rathbhupendra
```

Output:

```
Preorder traversal of constructed BST

4 2 1 3 6 5 7
```

Time Complexity: O(n)

Following is the recurrance relation for sortedArrayToBST().

```
  T(n) = 2T(n/2) + C
```

```
T(n) -->  Time taken for an array of size n

 C   -->  Constant (Finding middle of array and linking root to left
                  and right subtrees take constant time)
```

38. Convert a normal BST to Balanced BST

Given a BST (Binary Search Tree) that may be unbalanced, convert it into a balanced BST that has minimum possible height.

Examples :

```
Input:
      30
     /
    20
   /
  10
Output:
    20
   /   \
 10     30



Input:
        4
       /
      3
     /
    2
   /
  1
Output:
```

```
     3            3           2
   /  \         /  \        /  \
  1    4   OR  2    4  OR  1    3   OR ..
   \          /                  \
    2        1                    4


Input:
        4
       /   \
      3     5
     /       \
    2         6
   /           \
  1             7
Output:
      4
   /    \
  2      6
 /  \    /  \
1    3  5    7
```

A Simple Solution is to traverse nodes in Inorder and one by one insert into a self-balancing BST like AVL tree. Time complexity of this solution is O(n Log n) and this solution doesn’t guarantee

An Efficient Solution can construct balanced BST in O(n) time with minimum possible height. Below are steps.

1. Traverse given BST in inorder and store result in an array. This step takes O(n) time. Note that this array would be sorted as inorder traversal of BST always produces sorted sequence.
2. Build a balanced BST from the above created sorted array using the recursive approach discussed here. This step also takes O(n) time as we traverse every element exactly once and processing an element takes O(1) time.

Below is the implementation of above steps.

```
// C++ program to convert a left unbalanced BST to
// a balanced BST
#include <bits/stdc++.h>
using namespace std;

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

/* This function traverse the skewed binary tree and
   stores its nodes pointers in vector nodes[] */
void storeBSTNodes(Node* root, vector<Node*> &nodes)
{
    // Base case
    if (root==NULL)
        return;

    // Store nodes in Inorder (which is sorted
    // order for BST)
    storeBSTNodes(root->left, nodes);
    nodes.push_back(root);
    storeBSTNodes(root->right, nodes);
}

/* Recursive function to construct binary tree */
Node* buildTreeUtil(vector<Node*> &nodes, int start,
                    int end)
{
    // base case
    if (start > end)
        return NULL;

    /* Get the middle element and make it root */
    int mid = (start + end)/2;
    Node *root = nodes[mid];
```

```
    /* Using index in Inorder traversal, construct
       left and right subtress */
    root->left  = buildTreeUtil(nodes, start, mid-1);
    root->right = buildTreeUtil(nodes, mid+1, end);

    return root;
}

// This functions converts an unbalanced BST to
// a balanced BST
Node* buildTree(Node* root)
{
    // Store nodes of given BST in sorted order
    vector<Node *> nodes;
    storeBSTNodes(root, nodes);

    // Constucts BST from nodes[]
    int n = nodes.size();
    return buildTreeUtil(nodes, 0, n-1);
}

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

/* Function to do preorder traversal of tree */
void preOrder(Node* node)
{
    if (node == NULL)
        return;
    printf("%d ", node->data);
    preOrder(node->left);
    preOrder(node->right);
}

// Driver program
int main()
{
    /* Constructed skewed binary tree is
                10
               /
              8
             /
            7
```

```
          /
        6
      /
    5   */

    Node* root = newNode(10);
    root->left = newNode(8);
    root->left->left = newNode(7);
    root->left->left->left = newNode(6);
    root->left->left->left->left = newNode(5);

    root = buildTree(root);

    printf("Preorder traversal of balanced "
           "BST is : \n");
    preOrder(root);

    return 0;
}
```

Output :

```
Preorder traversal of balanced BST is :

7 5 6 8 10
```

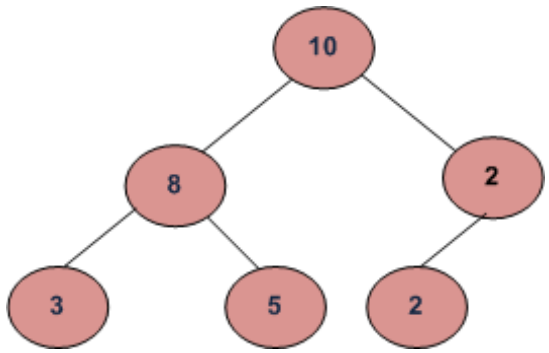
39. Given a binary tree, print all root-to-leaf paths

For the below example tree, all root-to-leaf paths are:

10 –> 8 –> 3

10 –> 8 –> 5

10 –> 2 –> 2



Algorithm:

Use a path array path[] to store current root to leaf path. Traverse from root to all leaves in top-down fashion. While traversing, store data of all nodes in current path in array path[]. When we reach a leaf node, print the path array.

```
#include <bits/stdc++.h>
using namespace std;

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

/* Prototypes for funtions needed in printPaths() */
void printPathsRecur(node* node, int path[], int pathLen);
void printArray(int ints[], int len);

/*Given a binary tree, print out all of its root-to-leaf
paths, one per line. Uses a recursive helper to do the work.*/
void printPaths(node* node)
{
    int path[1000];
    printPathsRecur(node, path, 0);
```

```
}

/* Recursive helper function -- given a node,
and an array containing the path from the root
node up to but not including this node,
print out all the root-leaf paths.*/
void printPathsRecur(node* node, int path[], int pathLen)
{
    if (node == NULL)
        return;

    /* append this node to the path array */
    path[pathLen] = node->data;
    pathLen++;

    /* it's a leaf, so print the path that led to here */
    if (node->left == NULL && node->right == NULL)
    {
        printArray(path, pathLen);
    }
    else
    {
        /* otherwise try both subtrees */
        printPathsRecur(node->left, path, pathLen);
        printPathsRecur(node->right, path, pathLen);
    }
}


/* UTILITY FUNCTIONS */
/* Utility that prints out an array on a line. */
void printArray(int ints[], int len)
{
    int i;
    for (i = 0; i < len; i++)
    {
        cout << ints[i] << " ";
    }
    cout<<endl;
}

/* utility that allocates a new node with the
given data and NULL left and right pointers. */
node* newnode(int data)
{
    node* Node = new node();
    Node->data = data;
    Node->left = NULL;
    Node->right = NULL;
```

```
    return(Node);
}

/* Driver code*/
int main()
{

    /* Constructed binary tree is
                10
            / \
            8 2
        / \ /
        3 5 2
    */
    node *root = newnode(10);
    root->left = newnode(8);
    root->right = newnode(2);
    root->left->left = newnode(3);
    root->left->right = newnode(5);
    root->right->left = newnode(2);

    printPaths(root);
    return 0;
}

// This code is contributed by rathbhupendra
```

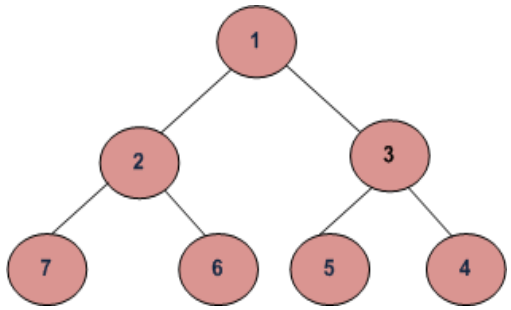
Output :

```
10 8 3

10 8 5

10 2 2
```

Time Complexity: O(n^2) where n is number of nodes

40. Level order traversal in spiral form

Write a function to print spiral order traversal of a tree. For below tree, function should print 1, 2, 3, 4, 5, 6, 7.



Method 1 (Recursive)

This problem can be seen as an extension of the level order traversal post.

To print the nodes in spiral order, nodes at different levels should be printed in alternating order. An additional Boolean variable ltr is used to change printing order of levels. If ltr is 1 then printGivenLevel() prints nodes from left to right else from right to left. Value of ltr is flipped in each iteration to change the order.

Function to print level order traversal of tree

```
printSpiral(tree)
  bool ltr = 0;

  for d = 1 to height(tree)

     printGivenLevel(tree, d, ltr);

     ltr ~= ltr /*flip ltr*/
```

Function to print all nodes at a given level

```
printGivenLevel(tree, level, ltr)
if tree is NULL then return;

if level is 1, then
```

```
    print(tree->data);

else if level greater than 1, then

    if(ltr)

        printGivenLevel(tree->left, level-1, ltr);

        printGivenLevel(tree->right, level-1, ltr);

    else

        printGivenLevel(tree->right, level-1, ltr);

        printGivenLevel(tree->left, level-1, ltr);
```

Following is the implementation of above algorithm.

```
// C program for recursive level order traversal in spiral form
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>

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

/* Function protoypes */
void printGivenLevel(struct node* root, int level, int ltr);
int height(struct node* node);
struct node* newNode(int data);

/* Function to print spiral traversal of a tree*/
void printSpiral(struct node* root)
{
    int h = height(root);
    int i;

    /*ltr -> Left to Right. If this variable is set,
      then the given level is traversed from left to right. */
    bool ltr = false;
    for (i = 1; i <= h; i++) {
        printGivenLevel(root, i, ltr);

        /*Revert ltr to traverse next level in opposite order*/
        ltr = !ltr;
```

```
    }
}

/* Print nodes at a given level */
void printGivenLevel(struct node* root, int level, int ltr)
{
    if (root == NULL)
        return;
    if (level == 1)
        printf("%d ", root->data);
    else if (level > 1) {
        if (ltr) {
            printGivenLevel(root->left, level - 1, ltr);
            printGivenLevel(root->right, level - 1, ltr);
        }
        else {
            printGivenLevel(root->right, level - 1, ltr);
            printGivenLevel(root->left, level - 1, ltr);
        }
    }
}

/* Compute the "height" of a tree -- the number of
    nodes along the longest path from the root node
    down to the farthest leaf node.*/
int height(struct node* node)
{
    if (node == NULL)
        return 0;
    else {
        /* compute the height of each subtree */
        int lheight = height(node->left);
        int rheight = height(node->right);

        /* use the larger one */
        if (lheight > rheight)
            return (lheight + 1);
        else
            return (rheight + 1);
    }
}

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

```
    node->left = NULL;
    node->right = NULL;

    return (node);
}

/* Driver program to test above functions*/
int main()
{
    struct node* root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(7);
    root->left->right = newNode(6);
    root->right->left = newNode(5);
    root->right->right = newNode(4);
    printf("Spiral Order traversal of binary tree is \n");
    printSpiral(root);

    return 0;
}
```

Output:

```
Spiral Order traversal of binary tree is

1 2 3 4 5 6 7
```

Time Complexity: Worst case time complexity of the above method is O(n^2). Worst case occurs in case of skewed trees.

Method 2 (Iterative)

We can print spiral order traversal in O(n) time and O(n) extra space. The idea is to use two stacks. We can use one stack for printing from left to right and other stack for printing from right to left. In every iteration, we have nodes of one level in one of the stacks. We print the nodes, and push nodes of next level in other stack.

```
// C++ implementation of a O(n) time method for spiral order
traversal
#include <iostream>
#include <stack>
using namespace std;
```

```
// Binary Tree node
struct node {
    int data;
    struct node *left, *right;
};

void printSpiral(struct node* root)
{
    if (root == NULL)
        return; // NULL check

    // Create two stacks to store alternate levels
    stack<struct node*> s1; // For levels to be printed from right
to left
    stack<struct node*> s2; // For levels to be printed from left
to right

    // Push first level to first stack 's1'
    s1.push(root);

    // Keep printing while any of the stacks has some nodes
    while (!s1.empty() || !s2.empty()) {
        // Print nodes of current level from s1 and push nodes of
        // next level to s2
        while (!s1.empty()) {
            struct node* temp = s1.top();
            s1.pop();
            cout << temp->data << " ";

            // Note that is right is pushed before left
            if (temp->right)
                s2.push(temp->right);
            if (temp->left)
                s2.push(temp->left);
        }

        // Print nodes of current level from s2 and push nodes of
        // next level to s1
        while (!s2.empty()) {
            struct node* temp = s2.top();
            s2.pop();
            cout << temp->data << " ";

            // Note that is left is pushed before right
            if (temp->left)
                s1.push(temp->left);
            if (temp->right)
                s1.push(temp->right);
        }
```

```
    }
}

// A utility function to create a new node
struct node* newNode(int data)
{
    struct node* node = new struct node;
    node->data = data;
    node->left = NULL;
    node->right = NULL;

    return (node);
}

int main()
{
    struct node* root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(7);
    root->left->right = newNode(6);
    root->right->left = newNode(5);
    root->right->right = newNode(4);
    cout << "Spiral Order traversal of binary tree is \n";
    printSpiral(root);

    return 0;
}
```

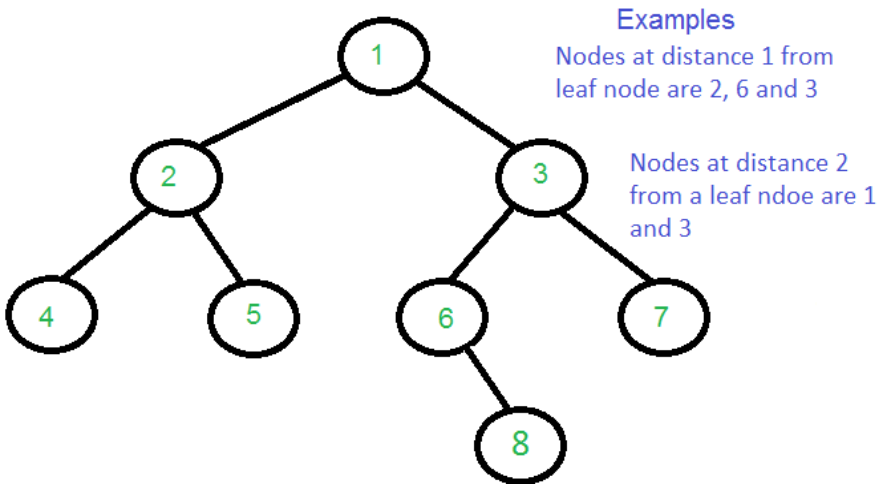
Output:

```
Spiral Order traversal of binary tree is

1 2 3 4 5 6 7
```

41. Print all nodes that are at distance k from a leaf node

Given a Binary Tree and a positive integer k, print all nodes that are distance k from a leaf node.

Here the meaning of distance is different from previous post. Here k distance from a leaf means k levels higher than a leaf node. For example if k is more than height of Binary Tree, then nothing should be printed. Expected time complexity is O(n) where n is the number nodes in the given Binary Tree.



The idea is to traverse the tree. Keep storing all ancestors till we hit a leaf node. When we reach a leaf node, we print the ancestor at distance k. We also need to keep track of nodes that are already printed as output. For that we use a boolean array visited[].

```
/* Program to print all nodes which are at distance k from a leaf
*/
#include <iostream>
using namespace std;
#define MAX_HEIGHT 10000

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

/* utility that allocates a new Node with the given key  */
```

```
Node* newNode(int key)
{
    Node* node = new Node;
    node->key = key;
    node->left = node->right = NULL;
    return (node);
}

/* This function prints all nodes that are distance k from a leaf
node
   path[] --> Store ancestors of a node
   visited[] --> Stores true if a node is printed as output.  A
node may be k
                 distance away from many leaves, we want to print
it once */
void kDistantFromLeafUtil(Node* node, int path[], bool visited[],
                          int pathLen, int k)
{
    // Base case
    if (node == NULL)
        return;

    /* append this Node to the path array */
    path[pathLen] = node->key;
    visited[pathLen] = false;
    pathLen++;

    /* it's a leaf, so print the ancestor at distance k only
       if the ancestor is not already printed  */
    if (node->left == NULL && node->right == NULL && pathLen - k -
1 >= 0 && visited[pathLen - k - 1] == false) {
        cout << path[pathLen - k - 1] << " ";
        visited[pathLen - k - 1] = true;
        return;
    }

    /* If not leaf node, recur for left and right subtrees */
    kDistantFromLeafUtil(node->left, path, visited, pathLen, k);
    kDistantFromLeafUtil(node->right, path, visited, pathLen, k);
}

/* Given a binary tree and a nuber k, print all nodes that are k
   distant from a leaf*/
void printKDistantfromLeaf(Node* node, int k)
{
    int path[MAX_HEIGHT];
    bool visited[MAX_HEIGHT] = { false };
    kDistantFromLeafUtil(node, path, visited, 0, k);
}
```

```
/* Driver program to test above functions*/
int main()
{
    // Let us create binary tree given in the above example
    Node* root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->left = newNode(6);
    root->right->right = newNode(7);
    root->right->left->right = newNode(8);

    cout << "Nodes at distance 2 are: ";
    printKDistantfromLeaf(root, 2);

    return 0;
}
```

Output:

```
Nodes at distance 2 are: 1 3
```

Time Complexity: Time Complexity of above code is O(n) as the code does a simple tree traversal.

Space Optimized Solution :

```
// Java program to print all nodes at a distance k from leaf
// A binary tree node
class Node {
    int data;
    Node left, right;

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

class BinaryTree {
    Node root;

    /* Given a binary tree and a nuber k, print all nodes that are
k
    distant from a leaf*/
    int printKDistantfromLeaf(Node node, int k)
    {
```

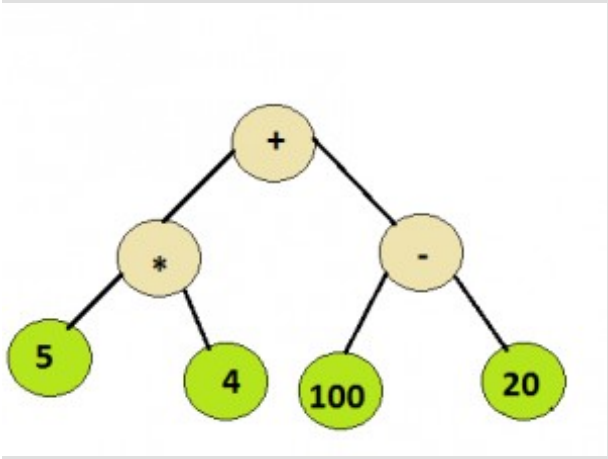
```
        if (node == null)
            return -1;
        int lk = printKDistantfromLeaf(node.left, k);
        int rk = printKDistantfromLeaf(node.right, k);
        boolean isLeaf = lk == -1 && lk == rk;
        if (lk == 0 || rk == 0 || (isLeaf && k == 0))
            System.out.print(" " + node.data);
        if (isLeaf && k > 0)
            return k - 1; // leaf node
        if (lk > 0 && lk < k)
            return lk - 1; // parent of left leaf
        if (rk > 0 && rk < k)
            return rk - 1; // parent of right leaf

        return -2;
    }

    // Driver program to test the above functions
    public static void main(String args[])
    {
        BinaryTree tree = new BinaryTree();

        /* Let us construct the tree shown in above diagram */
        tree.root = new Node(1);
        tree.root.left = new Node(2);
        tree.root.right = new Node(3);
        tree.root.left.left = new Node(4);
        tree.root.left.right = new Node(5);
        tree.root.right.left = new Node(6);
        tree.root.right.right = new Node(7);
        tree.root.right.left.right = new Node(8);

        System.out.println(" Nodes at distance 2 are :");
        tree.printKDistantfromLeaf(tree.root, 2);
    }
}

// This code has been contributed by Vijayan Annamalai
```

42. Evaluation of Expression Tree

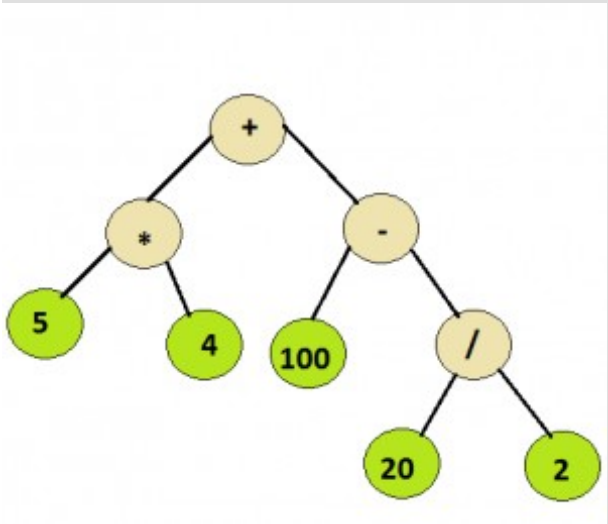
Given a simple expression tree, consisting of basic binary operators i.e., + , – ,* and / and some integers, evaluate the expression tree.

Examples:

```
Input :

Root node of the below tree
```




```
Output :

100


Input :

Root node of the below tree
```



```
Output :

110
```

As all the operators in the tree are binary hence each node will have either 0 or 2 children. As it can be inferred from the examples above , the integer values would appear at the leaf nodes , while the interior nodes represent the operators.

To evaluate the syntax tree , a recursive approach can be followed .

```
Algorithm :

Let t be the syntax tree

If  t is not null then

     If t.info is operand then

        Return  t.info

     Else

        A = solve(t.left)

        B = solve(t.right)

        return A operator B

        where operator is the info contained in t
```

The time complexity would be O(n), as each node is visited once. Below is a C++ program for the same:

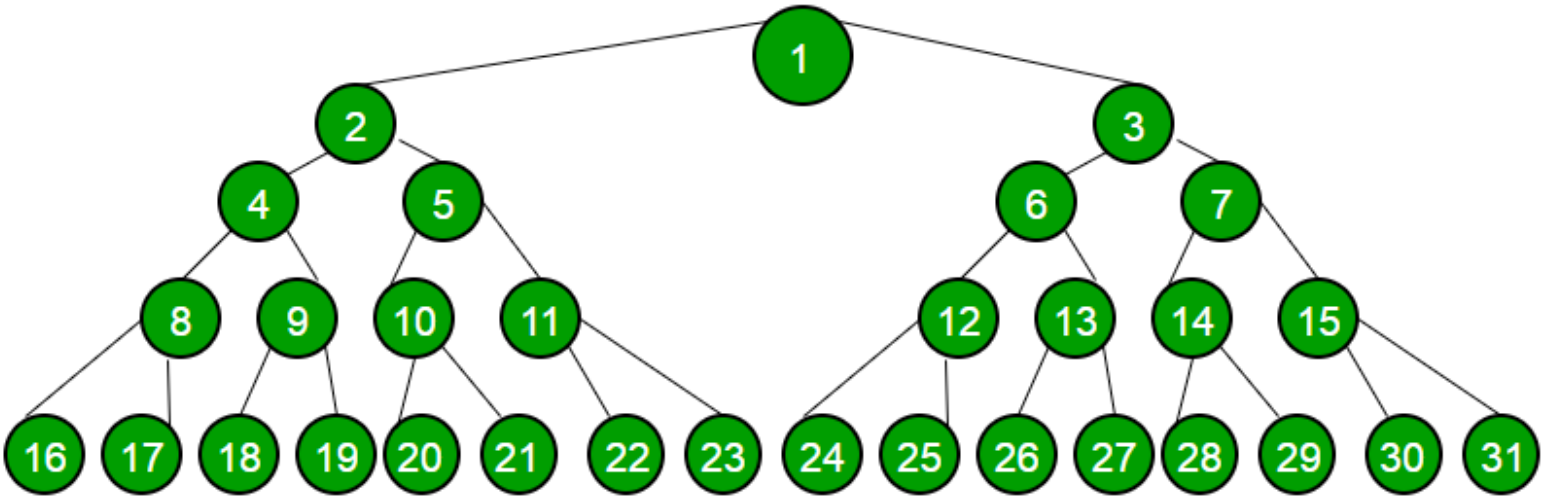
```
// C++ program to evaluate an expression tree
#include <bits/stdc++.h>
using namespace std;

// Class to represent the nodes of syntax tree
class node
{
public:
    string info;
```

```
    node *left = NULL, *right = NULL;
    node(string x)
    {
        info = x;
    }
};

// Utility function to return the integer value
// of a given string
int toInt(string s)
{
    int num = 0;

      // Check if the integral value is
      // negative or not
      // If it is not negative, generate the number
      // normally
      if(s[0]!='-')
        for (int i=0; i<s.length(); i++)
            num = num*10 + (int(s[i])-48);
    // If it is negative, calculate the +ve number
    // first ignoring the sign and invert the
    // sign at the end
      else
        for (int i=1; i<s.length(); i++)
        {
            num = num*10 + (int(s[i])-48);
            num = num*-1;
        }

    return num;
}

// This function receives a node of the syntax tree
// and recursively evaluates it
int eval(node* root)
{
    // empty tree
    if (!root)
        return 0;

    // leaf node i.e, an integer
    if (!root->left && !root->right)
        return toInt(root->info);

    // Evaluate left subtree
    int l_val = eval(root->left);

    // Evaluate right subtree
```

```
    int r_val = eval(root->right);

    // Check which operator to apply
    if (root->info=="+")
        return l_val+r_val;

    if (root->info=="-")
        return l_val-r_val;

    if (root->info=="*")
        return l_val*r_val;

    return l_val/r_val;
}

//driver function to check the above program
int main()
{
    // create a syntax tree
    node *root = new node("+");
    root->left = new node("*");
    root->left->left = new node("5");
    root->left->right = new node("-4");
    root->right = new node("-");
    root->right->left = new node("100");
    root->right->right = new node("20");
    cout << eval(root) << endl;

    delete(root);

    root = new node("+");
    root->left = new node("*");
    root->left->left = new node("5");
    root->left->right = new node("4");
    root->right = new node("-");
    root->right->left = new node("100");
    root->right->right = new node("/");
    root->right->right->left = new node("20");
    root->right->right->right = new node("2");

    cout << eval(root);
    return 0;
}
```

Output:

```
100

110
```



43. Print extreme nodes of each level of Binary Tree in alternate order

Given a binary tree, print nodes of extreme corners of each level but in alternate order.

Example:

For above tree, the output can be

1 2 7 8 31

– print rightmost node of 1st level

– print leftmost node of 2nd level

– print rightmost node of 3rd level

– print leftmost node of 4th level

– print rightmost node of 5th level

OR

1 3 4 15 16

– print leftmost node of 1st level

– print rightmost node of 2nd level

– print leftmost node of 3rd level

– print rightmost node of 4th level

– print leftmost node of 5th level

The idea is to traverse tree level by level. For each level, we count number of nodes in it and print its leftmost or the rightmost node based on value of a Boolean flag. We dequeue all nodes of current level and enqueue all nodes of next level and invert value of Boolean flag when switching levels.

Below is the implementation of above idea –

```
/* C++ program to print nodes of extreme corners
of each level in alternate order */
#include <bits/stdc++.h>
using namespace std;

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

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

/* Function to print nodes of extreme corners
of each level in alternate order */
void printExtremeNodes(Node* root)
{
    if (root == NULL)
        return;

    // Create a queue and enqueue left and right
    // children of root
    queue<Node*> q;
    q.push(root);

    // flag to indicate whether leftmost node or
    // the rightmost node has to be printed
```

```
    bool flag = false;
    while (!q.empty())
    {
        // nodeCount indicates number of nodes
        // at current level.
        int nodeCount = q.size();
        int n = nodeCount;

        // Dequeue all nodes of current level
        // and Enqueue all nodes of next level
        while (n--)
        {
            Node* curr = q.front();

            // Enqueue left child
            if (curr->left)
                q.push(curr->left);

            // Enqueue right child
            if (curr->right)
                q.push(curr->right);

            // Dequeue node
            q.pop();

            // if flag is true, print leftmost node
            if (flag && n == nodeCount - 1)
                cout << curr->data << " ";

            // if flag is false, print rightmost node
            if (!flag && n == 0)
                cout << curr->data << " ";
        }
        // invert flag for next level
        flag = !flag;
    }
}

/* Driver program to test above functions */
int main()
{
    // Binary Tree of Height 4
    Node* root = newNode(1);

    root->left = newNode(2);
    root->right = newNode(3);

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

```
    root->right->right = newNode(7);

    root->left->left->left  = newNode(8);
    root->left->left->right  = newNode(9);
    root->left->right->left  = newNode(10);
    root->left->right->right  = newNode(11);
    root->right->right->left  = newNode(14);
    root->right->right->right  = newNode(15);

    root->left->left->left->left  = newNode(16);
    root->left->left->left->right  = newNode(17);
    root->right->right->right->right  = newNode(31);

    printExtremeNodes(root);

    return 0;
}
```

Output:

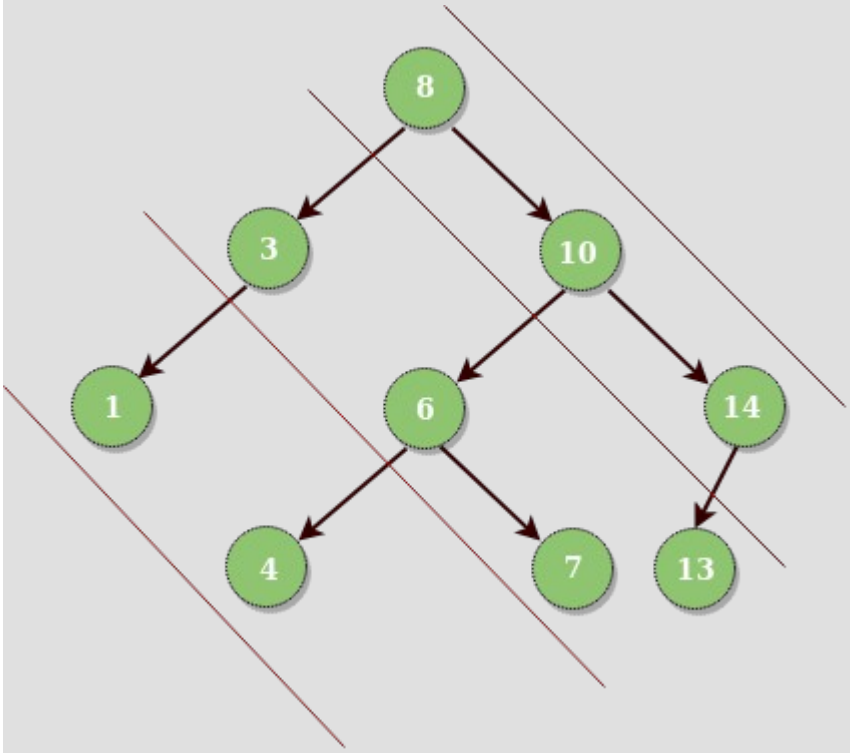
```
1 2 7 8 31
```

Time complexity of above solution is O(n) where n is total number of nodes in given binary tree

44. Diagonal Traversal of Binary Tree

Consider lines of slope -1 passing between nodes. Given a Binary Tree, print all diagonal elements in a binary tree belonging to same line.

```
Input : Root of below tree
```




```
Output :

Diagonal Traversal of binary tree :
 8 10 14
 3 6 7 13
 1 4
```

The idea is to use map. We use different slope distances and use them as key in map. Value in map is vector (or dynamic array) of nodes. We traverse the tree to store values in map. Once map is built, we print contents of it.

Below is implementation of above idea.

```
// C++ program for diagnoal traversal of Binary Tree
```

```cpp
#include <bits/stdc++.h>
using namespace std;

// Tree node
struct Node
{
    int data;
    Node *left, *right;
};

/* root - root of the binary tree
   d -  distance of current line from rightmost
        -topmost slope.
   diagonalPrint - multimap to store Diagonal
                   elements (Passed by Reference) */
void diagonalPrintUtil(Node* root, int d,
                       map<int, vector<int>> &diagonalPrint)
{
    // Base case
    if (!root)
        return;

    // Store all nodes of same line together as a vector
    diagonalPrint[d].push_back(root->data);

    // Increase the vertical distance if left child
    diagonalPrintUtil(root->left, d + 1, diagonalPrint);

    // Vertical distance remains same for right child
    diagonalPrintUtil(root->right, d, diagonalPrint);
}

// Print diagonal traversal of given binary tree
void diagonalPrint(Node* root)
{
    // create a map of vectors to store Diagonal elements
    map<int, vector<int> > diagonalPrint;
    diagonalPrintUtil(root, 0, diagonalPrint);

    cout << "Diagonal Traversal of binary tree : n";
    for (auto it = diagonalPrint.begin();
         it != diagonalPrint.end(); ++it)
    {
        for (auto itr = it->second.begin();
             itr != it->second.end(); ++itr)
            cout << *itr  << ' ';

        cout << 'n';
    }
}
```

```
// Utility method to create a new node
Node* newNode(int data)
{
    Node* node = new Node;
    node->data = data;
    node->left = node->right = NULL;
    return node;
}

// Driver program
int main()
{
    Node* root = newNode(8);
    root->left = newNode(3);
    root->right = newNode(10);
    root->left->left = newNode(1);
    root->left->right = newNode(6);
    root->right->right = newNode(14);
    root->right->right->left = newNode(13);
    root->left->right->left = newNode(4);
    root->left->right->right = newNode(7);

    /*  Node* root = newNode(1);
        root->left = newNode(2);
        root->right = newNode(3);
        root->left->left = newNode(9);
        root->left->right = newNode(6);
        root->right->left = newNode(4);
        root->right->right = newNode(5);
        root->right->left->right = newNode(7);
        root->right->left->left = newNode(12);
        root->left->right->left = newNode(11);
        root->left->left->right = newNode(10);*/

    diagonalPrint(root);

    return 0;
}
```

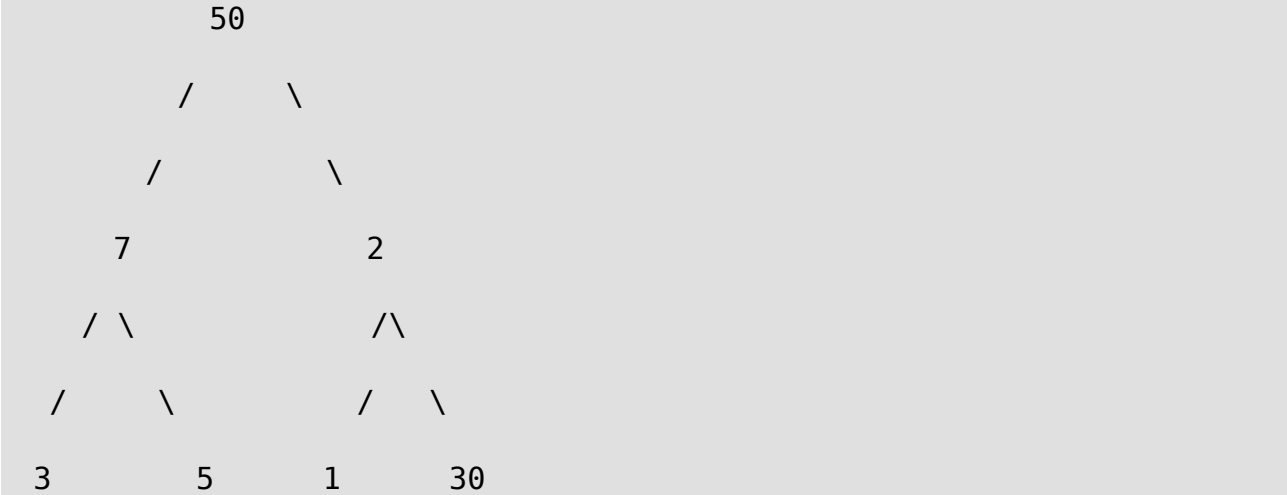
Output :

```
Diagonal Traversal of binary tree :

 8 10 14

 3 6 7 13

 1 4
```

45. Convert an arbitrary Binary Tree to a tree that holds Children Sum Property

Question: Given an arbitrary binary tree, convert it to a binary tree that holds Children Sum Property. You can only increment data values in any node (You cannot change the structure of the tree and cannot decrement the value of any node).

For example, the below tree doesn't hold the children sum property, convert it to a tree that holds the property.



Algorithm:

Traverse the given tree in post order to convert it, i.e., first change left and right children to hold the children sum property then change the parent node.

Let difference between node's data and children sum be diff.

```
    diff = node's children sum - node's data
```

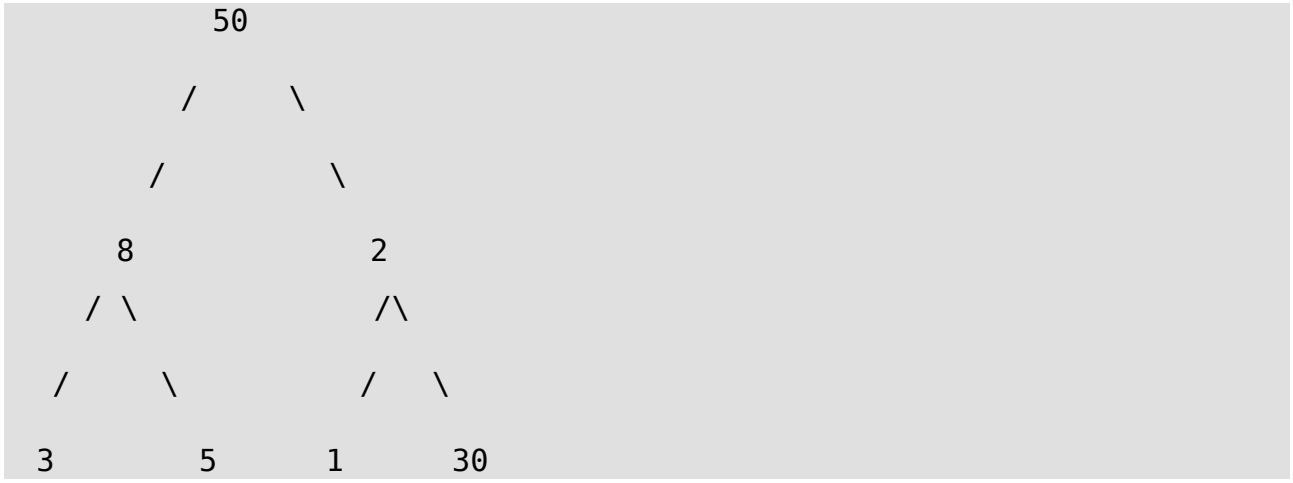
If diff is 0 then nothing needs to be done.

If diff > 0 (node's data is smaller than node's children sum) increment the node's data by diff.

If diff < 0 (node’s data is greater than the node's children sum) then increment one child’s data. We can choose to increment either left or right child if they both are not NULL. Let us always first increment the left child. Incrementing a child changes the subtree’s children sum property so we need to change left subtree also. So we recursively increment the left child. If left child is empty then we recursively call increment() for right child.

Let us run the algorithm for the given example.

First convert the left subtree (increment 7 to 8).



Then convert the right subtree (increment 2 to 31)

Now convert the root, we have to increment left subtree for converting the root.

```
       /    \
     /        \
   19           31
  / \           /  \
 /     \       /     \
14      5     1       30
```

Please note the last step – we have incremented 8 to 19, and to fix the subtree we have incremented 3 to 14.

Implementation:

```
/* C++ Program to convert an aribitary
binary tree to a tree that hold
children sum property */
#include<bits/stdc++.h>
using namespace std;

class node
{
    public:
    int data;
    node* left;
    node* right;

    /* Constructor that allocates a new node
    with the given data and NULL left and right
    pointers. */
    node(int data)
    {
        this->data = data;
        this->left = NULL;
        this->right = NULL;
    }
};

/* This function is used
to increment left subtree */
void increment(node* node, int diff);

/* This function changes a tree
to hold children sum property */
void convertTree(node* node)
{
    int left_data = 0, right_data = 0, diff;
```

```
    /* If tree is empty or it's a leaf
        node then return true */
    if (node == NULL || (node->left == NULL &&
                         node->right == NULL))
        return;
    else
    {
        /* convert left and right subtrees */
        convertTree(node->left);
        convertTree(node->right);

        /* If left child is not present then 0 is used
        as data of left child */
        if (node->left != NULL)
        left_data = node->left->data;

        /* If right child is not present then 0 is used
        as data of right child */
        if (node->right != NULL)
        right_data = node->right->data;

        /* get the diff of node's data and children sum */
        diff = left_data + right_data - node->data;

        /* If node's children sum is
        greater than the node's data */
        if (diff > 0)
        node->data = node->data + diff;

        /* THIS IS TRICKY --> If node's data
        is greater than children sum,
        then increment subtree by diff */
        if (diff < 0)
        increment(node, -diff); // -diff is used to make diff
positive
    }
}

/* This function is used
to increment subtree by diff */
void increment(node* node, int diff)
{
    /* IF left child is not
    NULL then increment it */
    if(node->left != NULL)
    {
        node->left->data = node->left->data + diff;

        // Recursively call to fix
```

```
        // the descendants of node->left
        increment(node->left, diff);
    }
    else if (node->right != NULL) // Else increment right child
    {
        node->right->data = node->right->data + diff;

        // Recursively call to fix
        // the descendants of node->right
        increment(node->right, diff);
    }
}

/* Given a binary tree,
printInorder() prints out its
inorder traversal*/
void printInorder(node* node)
{
    if (node == NULL)
        return;

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

    /* then print the data of node */
    cout<<node->data<<" ";

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

/* Driver code */
int main()
{
    node *root = new node(50);
    root->left = new node(7);
    root->right = new node(2);
    root->left->left = new node(3);
    root->left->right = new node(5);
    root->right->left = new node(1);
    root->right->right = new node(30);

    cout << "\nInorder traversal before conversion: " << endl;
    printInorder(root);

    convertTree(root);

    cout << "\nInorder traversal after conversion: " << endl;
    printInorder(root);
```

```
    return 0;
}

// This code is contributed by rathbhupendra
```

Output :

```
Inorder traversal before conversion is :

3 7 5 50 1 2 30

Inorder traversal after conversion is :

14 19 5 50 1 31 30
```

Time Complexity: O(n^2), Worst case complexity is for a skewed tree such that nodes are in decreasing order from root to leaf.

46. Find multiplication of sums of data of leaves at same levels

Given a Binary Tree, return following value for it.

1) For every level, compute sum of all leaves if there are leaves at this level. Otherwise ignore it.

2) Return multiplication of all sums.

Examples:

```
Input: Root of below tree
        2
      /   \
     7     5
            \
             9
Output: 63
First levels doesn't have leaves. Second level
has one leaf 7 and third level also has one
leaf 9.  Therefore result is 7*9 = 63



Input: Root of below tree
        2
      /   \
    7      5
   / \      \
  8   6      9
```

```
     / \    /  \

    1  11  4    10


Output: 208

First two levels don't have leaves. Third

level has single leaf 8. Last level has four

leaves 1, 11, 4 and 10. Therefore result is

8 * (1 + 11 + 4 + 10)
```

One Simple Solution is to recursively compute leaf sum for all level starting from top to bottom. Then multiply sums of levels which have leaves. Time complexity of this solution would be $O(n^2)$.

An Efficient Solution is to use Queue based level order traversal. While doing the traversal, process all different levels separately. For every processed level, check if it has a leaves. If it has then compute sum of leaf nodes. Finally return product of all sums.

```
/* Iterative C++ program to find sum of data of all leaves
   of a binary tree on same level and then multiply sums
   obtained of all levels. */
#include <bits/stdc++.h>
using namespace std;

// A Binary Tree Node
struct Node {
    int data;
    struct Node *left, *right;
};

// helper function to check if a Node is leaf of tree
bool isLeaf(Node* root)
{
    return (!root->left && !root->right);
}

/* Calculate sum of all leaf Nodes at each level and returns
   multiplication of sums */
int sumAndMultiplyLevelData(Node* root)
```

```
{
    // Tree is empty
    if (!root)
        return 0;

    int mul = 1; /* To store result */

    // Create an empty queue for level order tarversal
    queue<Node*> q;

    // Enqueue Root and initialize height
    q.push(root);

    // Do level order traversal of tree
    while (1) {
        // NodeCount (queue size) indicates number of Nodes
        // at current lelvel.
        int NodeCount = q.size();

        // If there are no Nodes at current level, we are done
        if (NodeCount == 0)
            break;

        // Initialize leaf sum for current level
        int levelSum = 0;

        // A boolean variable to indicate if found a leaf
        // Node at current level or not
        bool leafFound = false;

        // Dequeue all Nodes of current level and Enqueue all
        // Nodes of next level
        while (NodeCount > 0) {
            // Process next Node  of current level
            Node* Node = q.front();

            /* if Node is a leaf, update sum at the level */
            if (isLeaf(Node)) {
                leafFound = true;
                levelSum += Node->data;
            }
            q.pop();

            // Add children of Node
            if (Node->left != NULL)
                q.push(Node->left);
            if (Node->right != NULL)
                q.push(Node->right);
            NodeCount--;
```

```
        }

        // If we found at least one leaf, we multiply
        // result with level sum.
        if (leafFound)
            mul *= levelSum;
    }

    return mul; // Return result
}

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

// Driver program to test above functions
int main()
{
    Node* root = newNode(2);
    root->left = newNode(7);
    root->right = newNode(5);
    root->left->right = newNode(6);
    root->left->left = newNode(8);
    root->left->right->left = newNode(1);
    root->left->right->right = newNode(11);
    root->right->right = newNode(9);
    root->right->right->left = newNode(4);
    root->right->right->right = newNode(10);

    cout << "Final product value = "
         << sumAndMultiplyLevelData(root) << endl;

    return 0;
}
```

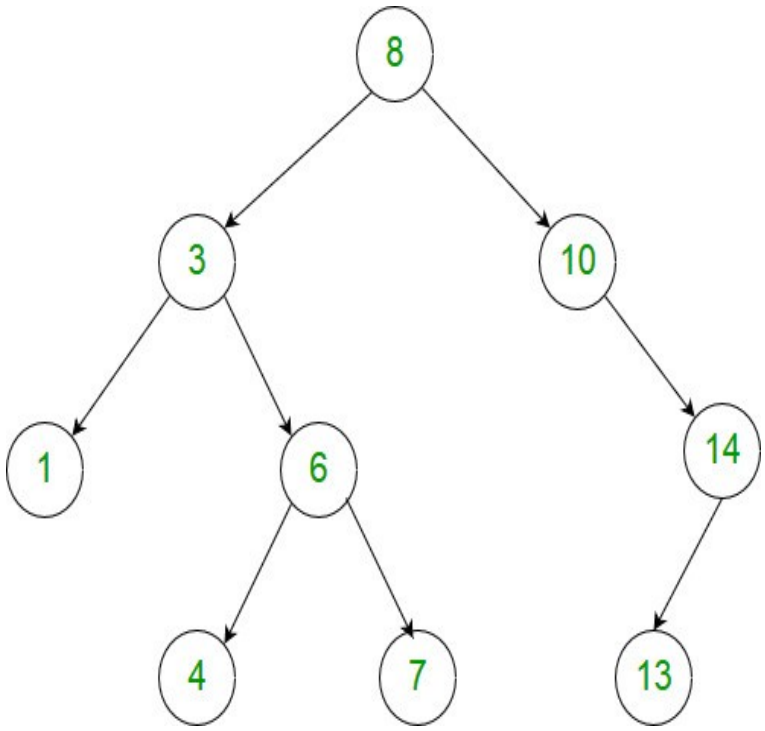
Output:

```
Final product value = 208
```

47. Maximum difference between node and its ancestor in Binary Tree

Given a binary tree, we need to find maximum value we can get by subtracting value of node B from value of node A, where A and B are two nodes of the binary tree and A is an ancestor of B. Expected time complexity is O(n).

For example, consider below binary tree



We can have various ancestor-node difference, some of which are given below :

8 – 3 = 5

3 – 7 = -4

8 – 1 = 7

10 – 13 = -3

. . . .

But among all those differences maximum value is 7 obtained by subtracting 1 from 8, which we need to return as result.

As we are given a binary tree, there is no relationship between node values so we need to traverse whole binary tree to get max difference and we can obtain the result in one traversal only by following below steps :
If we are at leaf node then just return its value because it can't be ancestor of any node. Then at each internal node we will try to get minimum value from left subtree and right subtree and calculate the difference between node value and this minimum value and according to that we will update the result.
As we are calculating minimum value while retuning in recurrence we will check all optimal possibilities (checking node value with minimum subtree value only) of differences and hence calculate the result in one traversal only.

Below is the implementation of above idea.

```
// C++ program to find maximum difference between node
// and its ancestor
#include <bits/stdc++.h>
using namespace std;

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

/* To create a newNode of tree and return pointer */
struct Node* newNode(int key)
{
    Node* temp = new Node;
    temp->key = key;
    temp->left = temp->right = NULL;
    return (temp);
}

/* Recursive function to calculate maximum ancestor-node
   difference in  binary tree. It updates value at 'res'
   to store the result.  The returned value of this function
   is minimum value in subtree rooted with 't' */
int maxDiffUtil(Node* t, int *res)
{
    /* Returning Maximum int value if node is not
       there (one child case)  */
    if (t == NULL)
```

```
        return INT_MAX;

    /* If leaf node then just return node's value  */
    if (t->left == NULL && t->right == NULL)
        return t->key;

    /* Recursively calling left and right subtree
       for minimum value  */
    int val = min(maxDiffUtil(t->left, res),
                  maxDiffUtil(t->right, res));

    /* Updating res if (node value - minimum value
       from subtree) is bigger than res  */
    *res = max(*res, t->key - val);

    /* Returning minimum value got so far */
    return min(val, t->key);
}

/* This function mainly calls maxDiffUtil() */
int maxDiff(Node *root)
{
    // Initialising result with minimum int value
    int res = INT_MIN;

    maxDiffUtil(root, &res);

    return res;
}

/* Helper function to print inorder traversal of
  binary tree   */
void inorder(Node* root)
{
    if (root)
    {
        inorder(root->left);
        printf("%d ", root->key);
        inorder(root->right);
    }
}

// Driver program to test above functions
int main()
{
    // Making above given diagram's binary tree
    Node* root;
    root = newNode(8);
    root->left = newNode(3);
```

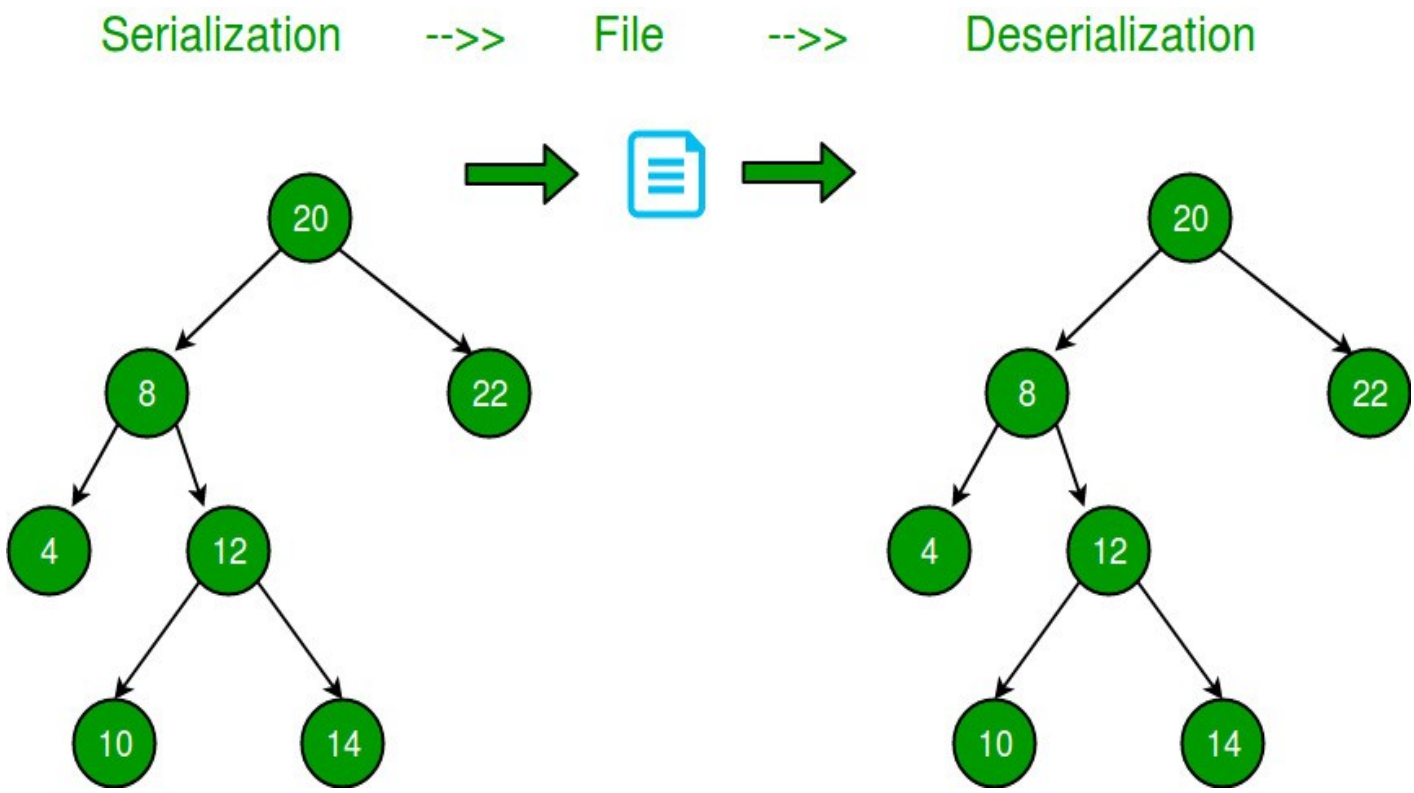
```
    root->left->left = newNode(1);
    root->left->right = newNode(6);
    root->left->right->left = newNode(4);
    root->left->right->right = newNode(7);

    root->right = newNode(10);
    root->right->right = newNode(14);
    root->right->right->left = newNode(13);

    printf("Maximum difference between a node and"
           " its ancestor is : %d\n", maxDiff(root));
}
```

Output :

```
Maximum difference between a node and its ancestor is : 7
```

48. Serialize and Deserialize a Binary Tree

Serialization is to store tree in a file so that it can be later restored. The structure of tree must be maintained. Deserialization is reading tree back from file.



Following are some simpler versions of the problem:

If given Tree is Binary Search Tree?

If the given Binary Tree is Binary Search Tree, we can store it by either storing preorder or postorder traversal. In case of Binary Search Trees, only preorder or postorder traversal is sufficient to store structure information.

If given Binary Tree is Complete Tree?

A Binary Tree is complete if all levels are completely filled except possibly the last level and all nodes of last level are as left as possible (Binary Heaps are complete Binary Tree). For a complete Binary Tree, level order traversal is

sufficient to store the tree. We know that the first node is root, next two nodes are nodes of next level, next four nodes are nodes of 2nd level and so on.

If given Binary Tree is Full Tree?

A full Binary is a Binary Tree where every node has either 0 or 2 children. It is easy to serialize such trees as every internal node has 2 children. We can simply store preorder traversal and store a bit with every node to indicate whether the node is an internal node or a leaf node.

How to store a general Binary Tree?

A simple solution is to store both Inorder and Preorder traversals. This solution requires requires space twice the size of Binary Tree.

We can save space by storing Preorder traversal and a marker for NULL pointers.

```
Let the marker for NULL pointers be '-1'

Input:

    12

   /

  13

Output: 12 13 -1 -1 -1


Input:

     20

   /   \

  8     22

Output: 20 8 -1 -1 22 -1 -1


Input:

        20

      /
```

```
     8
    / \
   4  12
     /  \
    10  14
```

Output: 20 8 4 -1 -1 12 10 -1 -1 14 -1 -1 -1

Input:

```
         20
        /
       8
     /
   10
   /
  5
```

Output: 20 8 10 5 -1 -1 -1 -1 -1

Input:

```
         20
           \
            8
             \
              10
                \
                 5
```

Output: 20 -1 8 -1 10 -1 5 -1 -1

Deserialization can be done by simply reading data from file one by one.

Following is C++ implementation of the above idea.

```
// A C++ program to demonstrate serialization and deserialiation
of
// Binary Tree
#include <stdio.h>
#define MARKER -1

/* A binary tree Node has key, pointer to left and right children
*/
struct Node
{
    int key;
    struct Node* left, *right;
};

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

// This function stores a tree in a file pointed by fp
void serialize(Node *root, FILE *fp)
{
    // If current node is NULL, store marker
    if (root == NULL)
    {
        fprintf(fp, "%d ", MARKER);
        return;
    }

    // Else, store current node and recur for its children
    fprintf(fp, "%d ", root->key);
    serialize(root->left, fp);
    serialize(root->right, fp);
}

// This function constructs a tree from a file pointed by 'fp'
void deSerialize(Node *&root, FILE *fp)
{
    // Read next item from file. If theere are no more items or
next
    // item is marker, then return
```

```
    int val;
    if ( !fscanf(fp, "%d ", &val) || val == MARKER)
       return;

    // Else create node with this item and recur for children
    root = newNode(val);
    deSerialize(root->left, fp);
    deSerialize(root->right, fp);
}

// A simple inorder traversal used for testing the constructed
tree
void inorder(Node *root)
{
    if (root)
    {
        inorder(root->left);
        printf("%d ", root->key);
        inorder(root->right);
    }
}

/* Driver program to test above functions*/
int main()
{
    // Let us construct a tree shown in the above figure
    struct Node *root        = newNode(20);
    root->left               = newNode(8);
    root->right              = newNode(22);
    root->left->left         = newNode(4);
    root->left->right        = newNode(12);
    root->left->right->left  = newNode(10);
    root->left->right->right = newNode(14);

    // Let us open a file and serialize the tree into the file
    FILE *fp = fopen("tree.txt", "w");
    if (fp == NULL)
    {
        puts("Could not open file");
        return 0;
    }
    serialize(root, fp);
    fclose(fp);

    // Let us deserialize the storeed tree into root1
    Node *root1 = NULL;
    fp = fopen("tree.txt", "r");
    deSerialize(root1, fp);
```

```
    printf("Inorder Traversal of the tree constructed from file:\
n");
    inorder(root1);

    return 0;
}
```

Output:

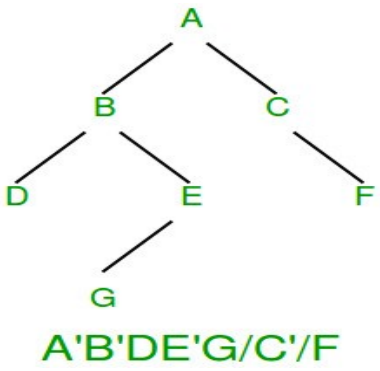
```
Inorder Traversal of the tree constructed from file:

4 8 10 12 14 20 22
```

How much extra space is required in above solution?

If there are n keys, then the above solution requires n+1 markers which may be better than simple solution (storing keys twice) in situations where keys are big or keys have big data items associated with them.

Can we optimize it further?

The above solution can be optimized in many ways. If we take a closer look at above serialized trees, we can observer that all leaf nodes require two markers. One simple optimization is to store a separate bit with every node to indicate that the node is internal or external. This way we don’t have to store two markers with every leaf node as leaves can be identified by extra bit. We still need marker for internal nodes with one child. For example in the following diagram ‘ is used to indicate an internal node set bit, and ‘/’ is used as NULL marker. The diagram is taken from here.

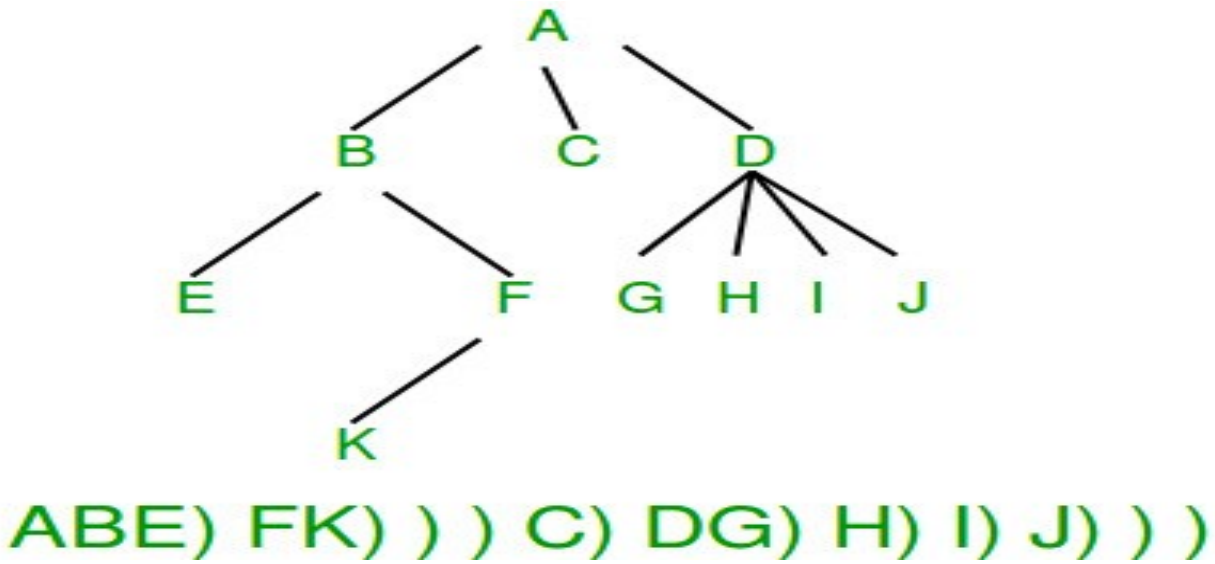


Please note that there are always more leaf nodes than internal nodes in a Binary Tree (Number of leaf nodes is number of internal nodes plus 1, so this optimization makes sense.

How to serialize n-ary tree?

In an n-ary tree, there is no designated left or right child. We can store an ‘end of children’ marker with every node. The following diagram shows serialization where ‘)’ is used as end of children marker. We will soon be covering

implementation for n-ary tree. The diagram is taken from here.



49. Serialize and Deserialize an N-ary Tree

Given an N-ary tree where every node has at-most N children. How to serialize and deserialze it? Serialization is to store tree in a file so that it can be later restored. The structure of tree must be maintained. Deserialization is reading tree back from file.

This post is mainly an extension of below post.

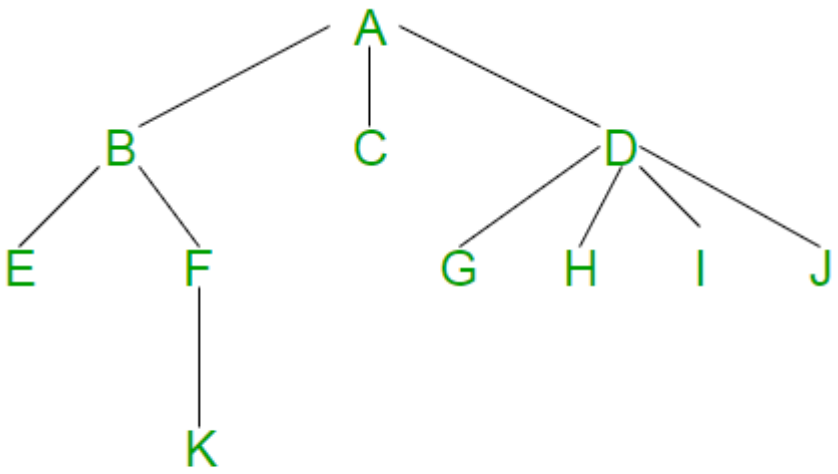
Serialize and Deserialize a Binary Tree

In an N-ary tree, there are no designated left and right children. An N-ary tree is represented by storing an array or list of child pointers with every node.

The idea is to store an 'end of children' marker with every node. The following diagram shows serialization where ')' is used as end of children marker.

Following is C++ implementation of above idea.

```
// A C++ Program serialize and deserialize an N-ary tree
#include<cstdio>
#define MARKER ')'
#define N 5
using namespace std;

// A node of N-ary tree
struct Node {
   char key;
   Node *child[N];  // An array of pointers for N children
};

// A utility function to create a new N-ary tree node
Node *newNode(char key)
{
    Node *temp = new Node;
    temp->key = key;
    for (int i = 0; i < N; i++)
        temp->child[i] = NULL;
    return temp;
}

// This function stores the given N-ary tree in a file pointed by
```



ABE)FK)))C)DG)H)I)J)))

```
fp
void
```

```
serialize(Node *root, FILE *fp)
{
    // Base case
    if (root == NULL) return;

    // Else, store current node and recur for its children
    fprintf(fp, "%c ", root->key);
    for (int i = 0; i < N && root->child[i]; i++)
         serialize(root->child[i],  fp);

    // Store marker at the end of children
    fprintf(fp, "%c ", MARKER);
}

// This function constructs N-ary tree from a file pointed by
'fp'.
// This functionr returns 0 to indicate that the next item is a
valid
// tree key. Else returns 0
int deSerialize(Node *&root, FILE *fp)
{
    // Read next item from file. If theere are no more items or
next
    // item is marker, then return 1 to indicate same
    char val;
    if ( !fscanf(fp, "%c ", &val) || val == MARKER )
       return 1;

    // Else create node with this item and recur for children
    root = newNode(val);
    for (int i = 0; i < N; i++)
      if (deSerialize(root->child[i], fp))
         break;

    // Finally return 0 for successful finish
```

```
    return 0;
}

// A utility function to create a dummy tree shown in above
diagram
Node *createDummyTree()
{
    Node *root = newNode('A');
    root->child[0] = newNode('B');
    root->child[1] = newNode('C');
    root->child[2] = newNode('D');
    root->child[0]->child[0] = newNode('E');
    root->child[0]->child[1] = newNode('F');
    root->child[2]->child[0] = newNode('G');
    root->child[2]->child[1] = newNode('H');
    root->child[2]->child[2] = newNode('I');
    root->child[2]->child[3] = newNode('J');
    root->child[0]->child[1]->child[0] = newNode('K');
    return root;
}

// A utlity function to traverse the constructed N-ary tree
void traverse(Node *root)
{
    if (root)
    {
        printf("%c ", root->key);
        for (int i = 0; i < N; i++)
            traverse(root->child[i]);
    }
}

// Driver program to test above functions
int main()
{
    // Let us create an N-ary tree shown in above diagram
    Node *root = createDummyTree();

    // Let us open a file and serialize the tree into the file
    FILE *fp = fopen("tree.txt", "w");
    if (fp == NULL)
    {
        puts("Could not open file");
        return 0;
    }
    serialize(root, fp);
    fclose(fp);

    // Let us deserialize the storeed tree into root1
    Node *root1 = NULL;
```

```
    fp = fopen("tree.txt", "r");
    deSerialize(root1, fp);

    printf("Constructed N-Ary Tree from file is \n");
    traverse(root1);

    return 0;
}
```

Output:

```
Constructed N-Ary Tree from file is

A B E F K C D G H I J
```

The above implementation can be optimized in many ways for example by using a vector in place of array of pointers. We have kept it this way to keep it simple to read and understand

50. Construct tree from ancestor matrix

Given an ancestor matrix mat[n][n] where Ancestor matrix is defined as below.

```
  mat[i][j] = 1 if i is ancestor of j

  mat[i][j] = 0, otherwise
```

Construct a Binary Tree from given ancestor matrix where all its values of nodes are from 0 to n-1.

1. It may be assumed that the input provided the program is valid and tree can be constructed out of it.
2. Many Binary trees can be constructed from one input. The program will construct any one of them.

Examples:

```
Input: 0 1 1
      0 0 0
      0 0 0
Output: Root of one of the below trees.
   0                0
  /   \     OR     /   \
 1     2          2     1


Input: 0 0 0 0 0 0
      1 0 0 0 1 0
      0 0 0 1 0 0
      0 0 0 0 0 0
      0 0 0 0 0 0
```

```
      1 1 1 1 1 0

Output: Root of one of the below trees.

     5              5              5

   /    \          / \           /   \

  1      2   OR    2   1    OR    1     2  OR ....

 /  \   /        /    /  \       / \    /

0   4  3        3    0    4     4   0  3

There are different possible outputs because ancestor

matrix doesn't store that which child is left and which

is right.
```

This problem is mainly reverse of below problem.

Construct Ancestor Matrix from a Given Binary Tree

We strongly recommend you to minimize your browser and try this yourself first.

Observations used in the solution:

1. The rows that correspond to leaves have all 0’s
2. The row that corresponds to root has maximum number of 1’s.
3. Count of 1’s in i’th row indicates number of descendants of node i.

The idea is to construct the tree in bottom up manner.

1) Create an array of node pointers node[].

2) Store row numbers that correspond to a given count. We have used multimap for this purpose.

3) Process all entries of multimap from smallest count to largest (Note that entries in map and multimap can be traversed in sorted order). Do following for every entry.

…….a) Create a new node for current row number.

…….b) If this node is not a leaf node, consider all those descendants of it whose parent is not set, make current node as its parent.

4) The last processed node (node with maximum sum) is root of tree.

Below is C++ implementation of above idea. Following are steps.

```
// Given an ancestor matrix for binary tree, construct
// the tree.
#include <bits/stdc++.h>
using namespace std;

# define N 6

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

/* Helper function to create a new node */
Node* newNode(int data)
{
    Node* node = new Node;
    node->data = data;
    node->left = node->right = NULL;
    return (node);
}

// Constructs tree from ancestor matrix
Node* ancestorTree(int mat[][N])
{
    // Binary array to determine whether
    // parent is set for node i or not
    int parent[N] = {0};

    // Root will store the root of the constructed tree
    Node* root = NULL;

    // Create a multimap, sum is used as key and row
    // numbers are used as values
    multimap<int, int> mm;

    for (int i = 0; i < N; i++)
    {
        int sum = 0; // Initialize sum of this row
        for (int j = 0; j < N; j++)
            sum += mat[i][j];

        // insert(sum, i) pairs into the multimap
        mm.insert(pair<int, int>(sum, i));
```

```
    }

    // node[i] will store node for i in constructed tree
    Node* node[N];

    // Traverse all entries of multimap.  Note that values
    // are accessed in increasing order of sum
    for (auto it = mm.begin(); it != mm.end(); ++it)
    {
      // create a new node for every value
      node[it->second] = newNode(it->second);

      // To store last processed node. This node will be
      // root after loop terminates
      root = node[it->second];

      // if non-leaf node
      if (it->first != 0)
      {
        // traverse row 'it->second' in the matrix
        for (int i = 0; i < N; i++)
        {
           // if parent is not set and ancestor exits
           if (!parent[i] && mat[it->second][i])
           {
             // check for unoccupied left/right node
             // and set parent of node i
             if (!node[it->second]->left)
               node[it->second]->left = node[i];
             else
               node[it->second]->right = node[i];

             parent[i] = 1;
           }
        }
      }
    }
    return root;
}

/* Given a binary tree, print its nodes in inorder */
void printInorder(Node* node)
{
    if (node == NULL)
        return;
    printInorder(node->left);
    printf("%d ", node->data);
    printInorder(node->right);
}
```

```
// Driver program
int main()
{
    int mat[N][N] = {{ 0, 0, 0, 0, 0, 0 },
        { 1, 0, 0, 0, 1, 0 },
        { 0, 0, 0, 1, 0, 0 },
        { 0, 0, 0, 0, 0, 0 },
        { 0, 0, 0, 0, 0, 0 },
        { 1, 1, 1, 1, 1, 0 }
    };

    Node* root = ancestorTree(mat);

    cout << "Inorder traversal of tree is \n";
    printInorder(root);

    return 0;
}
```

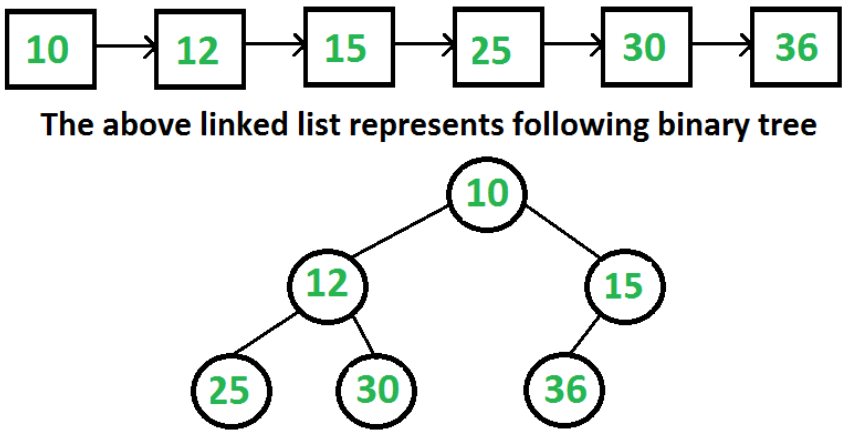
Output:

```
Inorder traversal of tree is

0 1 4 5 3 2
```

51. Construct Complete Binary Tree from its Linked List Representation

Given Linked List Representation of Complete Binary Tree, construct the Binary tree. A complete binary tree can be represented in an array in the following approach.

If root node is stored at index i, its left, and right children are stored at indices 2*i+1, 2*i+2 respectively.
Suppose tree is represented by a linked list in same way, how do we convert this into normal linked representation of binary tree where every node has data, left and right pointers? In the linked list representation, we cannot directly access the children of the current node unless we traverse the list.



We are mainly given level order traversal in sequential access form. We know head of linked list is always is root of the tree. We take the first node as root and we also know that the next two nodes are left and right children of root. So we know partial Binary Tree. The idea is to do Level order traversal of the partially built Binary Tree using queue and traverse the linked list at the same time. At every step, we take the parent node from queue, make next two nodes of linked list as children of the parent node, and enqueue the next two nodes to queue.

1. Create an empty queue.

2. Make the first node of the list as root, and enqueue it to the queue.

3. Until we reach the end of the list, do the following.

.....a. Dequeue one node from the queue. This is the current parent.

.....b. Traverse two nodes in the list, add them as children of the current parent.

.....c. Enqueue the two nodes into the queue.

Below is the code which implements the same in C++.

```
// C++ program to create a Complete Binary tree from its Linked
List
// Representation
#include <iostream>
#include <string>
#include <queue>
using namespace std;

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

// Binary tree node structure
struct BinaryTreeNode
{
    int data;
    BinaryTreeNode *left, *right;
};

// Function to insert a node at the beginning of the Linked List
void push(struct ListNode** head_ref, int new_data)
{
    // allocate node and assign data
    struct ListNode* new_node = new ListNode;
    new_node->data = new_data;

    // link the old list off the new node
    new_node->next = (*head_ref);

    // move the head to point to the new node
    (*head_ref)    = new_node;
}
```

```
// method to create a new binary tree node from the given data
BinaryTreeNode* newBinaryTreeNode(int data)
{
    BinaryTreeNode *temp = new BinaryTreeNode;
    temp->data = data;
    temp->left = temp->right = NULL;
    return temp;
}

// converts a given linked list representing a complete binary tree into the
// linked representation of binary tree.
void convertList2Binary(ListNode *head, BinaryTreeNode* &root)
{
    // queue to store the parent nodes
    queue<BinaryTreeNode *> q;

    // Base Case
    if (head == NULL)
    {
        root = NULL; // Note that root is passed by reference
        return;
    }

    // 1.) The first node is always the root node, and add it to the queue
    root = newBinaryTreeNode(head->data);
    q.push(root);

    // advance the pointer to the next node
    head = head->next;

    // until the end of linked list is reached, do the following steps
    while (head)
    {
        // 2.a) take the parent node from the q and remove it from q
        BinaryTreeNode* parent = q.front();
        q.pop();

        // 2.c) take next two nodes from the linked list. We will add
        // them as children of the current parent node in step 2.b. Push them
        // into the queue so that they will be parents to the future nodes
        BinaryTreeNode *leftChild = NULL, *rightChild = NULL;
        leftChild = newBinaryTreeNode(head->data);
        q.push(leftChild);
```

```
        head = head->next;
        if (head)
        {
            rightChild = newBinaryTreeNode(head->data);
            q.push(rightChild);
            head = head->next;
        }

        // 2.b) assign the left and right children of parent
        parent->left = leftChild;
        parent->right = rightChild;
    }
}

// Utility function to traverse the binary tree after conversion
void inorderTraversal(BinaryTreeNode* root)
{
    if (root)
    {
        inorderTraversal( root->left );
        cout << root->data << " ";
        inorderTraversal( root->right );
    }
}

// Driver program to test above functions
int main()
{
    // create a linked list shown in above diagram
    struct ListNode* head = NULL;
    push(&head, 36);  /* Last node of Linked List */
    push(&head, 30);
    push(&head, 25);
    push(&head, 15);
    push(&head, 12);
    push(&head, 10); /* First node of Linked List */

    BinaryTreeNode *root;
    convertList2Binary(head, root);

    cout << "Inorder Traversal of the constructed Binary Tree
is: \n";
    inorderTraversal(root);
    return 0;
}
```

Output:

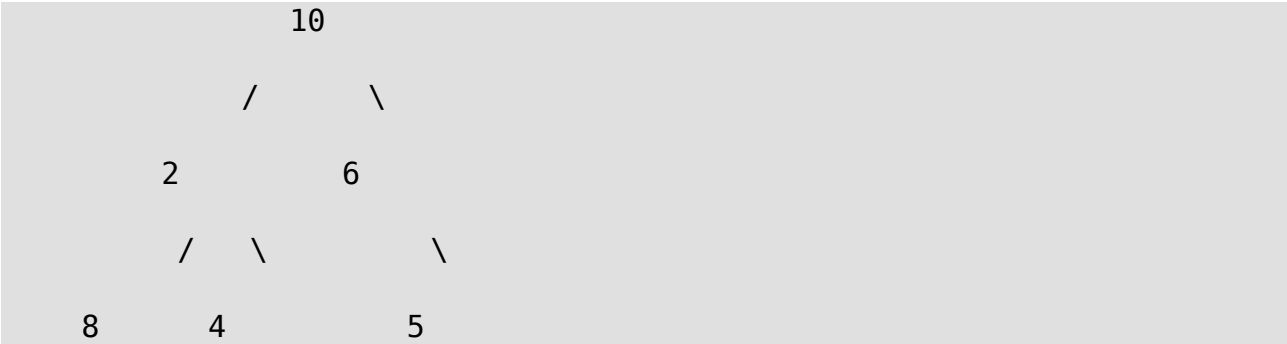
```
Inorder Traversal of the constructed Binary Tree is:

25 12 30 10 36 15
```

Time Complexity: Time complexity of the above solution is O(n) where n is the number of nodes.

52. Find next right node of a given key

Given a Binary tree and a key in the binary tree, find the node right to the given key. If there is no node on right side, then return NULL. Expected time complexity is O(n) where n is the number of nodes in the given binary tree.

For example, consider the following Binary Tree. Output for 2 is 6, output for 4 is 5. Output for 10, 6 and 5 is NULL.



Solution: The idea is to do level order traversal of given Binary Tree. When we find the given key, we just check if the next node in level order traversal is of same level, if yes, we return the next node, otherwise return NULL.

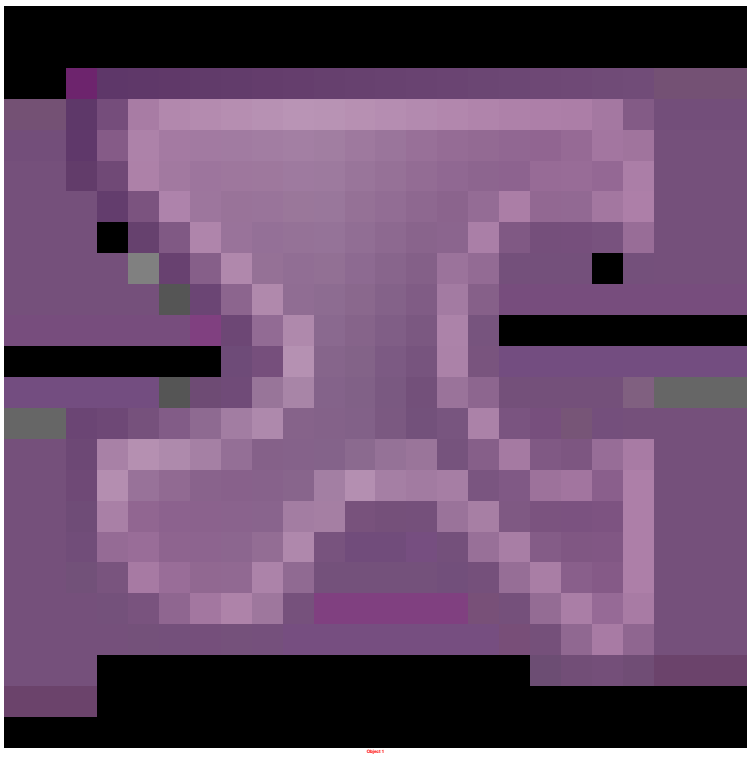
```
/* Program to find next right of a given key */
#include <iostream>
#include <queue>
using namespace std;

// A Binary Tree Node
struct node
{
    struct node *left, *right;
    int key;
};

// Method to find next right of given key k, it returns NULL if k
is
// not present in tree or k is the rightmost node of its level
node* nextRight(node *root, int k)
{
    // Base Case
    if (root == NULL)
        return 0;

    // Create an empty queue for level order tarversal
    queue<node *> qn; // A queue to store node addresses
```

```
    queue<int> ql;   // Another queue to store node levels

    int level = 0;  // Initialize level as 0

    // Enqueue Root and its level
    qn.push(root);
    ql.push(level);

    // A standard BFS loop
    while (qn.size())
    {
        // dequeue an node from qn and its level from ql
        node *node = qn.front();
        level = ql.front();
        qn.pop();
        ql.pop();

        // If the dequeued node has the given key k
        if (node->key == k)
        {
            // If there are no more items in queue or given node
is
            // the rightmost node of its level, then return NULL
            if (ql.size() == 0 || ql.front() != level)
               return NULL;

            // Otherwise return next node from queue of nodes
            return qn.front();
        }

        // Standard BFS steps: enqueue children of this node
        if (node->left != NULL)
        {
            qn.push(node->left);
            ql.push(level+1);
        }
        if (node->right != NULL)
        {
            qn.push(node->right);
            ql.push(level+1);
        }
    }

    // We reach here if given key x doesn't exist in tree
    return NULL;
}

// Utility function to create a new tree node
node* newNode(int key)
```

```
{
    node *temp = new node;
    temp->key = key;
    temp->left = temp->right = NULL;
    return temp;
}

// A utility function to test above functions
void test(node *root, int k)
{
    node *nr = nextRight(root, k);
    if (nr != NULL)
      cout << "Next Right of " << k << " is " << nr->key << endl;
    else
      cout << "No next right node found for " << k << endl;
}

// Driver program to test above functions
int main()
{
    // Let us create binary tree given in the above example
    node *root = newNode(10);
    root->left = newNode(2);
    root->right = newNode(6);
    root->right->right = newNode(5);
    root->left->left = newNode(8);
    root->left->right = newNode(4);

    test(root, 10);
    test(root, 2);
    test(root, 6);
    test(root, 5);
    test(root, 8);
    test(root, 4);
    return 0;
}
```

Output:

```
No next right node found for 10

Next Right of 2 is 6

No next right node found for 6

No next right node found for 5

Next Right of 8 is 4

Next Right of 4 is 5
```



Time Complexity: The above code is a simple BFS traversal code which visits every enqueue and dequeues a node at most once. Therefore, the time complexity is O(n) where n is the number of nodes in the given binary tree

53. Boundary Traversal of binary tree

Given a binary tree, print boundary nodes of the binary tree Anti-Clockwise starting from the root. The boundary includes left boundary, leaves, and right boundary in order without duplicate nodes. (The values of the nodes may still be duplicates.)

The left boundary is defined as the path from the root to the left-most node.

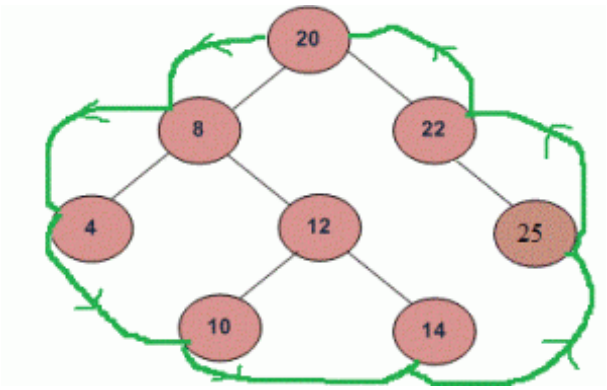
The right boundary is defined as the path from the root to the right-most node.

If the root doesn't have left subtree or right subtree, then the root itself is left boundary or right boundary. Note this definition only applies to the input binary tree, and not apply to any subtrees.

The left-most node is defined as a leaf node you could reach when you always firstly travel to the left subtree if it exists. If not, travel to the right subtree. Repeat until you reach a leaf node.

The right-most node is also defined in the same way with left and right exchanged.

For example, boundary traversal of the following tree is "20 8 4 10 14 25 22"



We break the problem in 3 parts:

1. Print the left boundary in top-down manner.

2. Print all leaf nodes from left to right, which can again be sub-divided into two sub-parts:

.....2.1 Print all leaf nodes of left sub-tree from left to right.

.....2.2 Print all leaf nodes of right subtree from left to right.

3. Print the right boundary in bottom-up manner.

We need to take care of one thing that nodes are not printed again. e.g. The left most node is also the leaf node of the tree.

Based on the above cases, below is the implementation:

```
/* C program for boundary traversal
of a binary tree */

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

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

// A simple function to print leaf nodes of a binary tree
void printLeaves(struct node* root)
{
    if (root == NULL)
        return;

    printLeaves(root->left);

    // Print it if it is a leaf node
    if (!(root->left) && !(root->right))
        printf("%d ", root->data);

    printLeaves(root->right);
}

// A function to print all left boundary nodes, except a leaf
node.
// Print the nodes in TOP DOWN manner
void printBoundaryLeft(struct node* root)
{
    if (root == NULL)
        return;

    if (root->left) {

        // to ensure top down order, print the node
        // before calling itself for left subtree
        printf("%d ", root->data);
```

```
        printBoundaryLeft(root->left);
    }
    else if (root->right) {
        printf("%d ", root->data);
        printBoundaryLeft(root->right);
    }
    // do nothing if it is a leaf node, this way we avoid
    // duplicates in output
}

// A function to print all right boundary nodes, except a leaf
node
// Print the nodes in BOTTOM UP manner
void printBoundaryRight(struct node* root)
{
    if (root == NULL)
        return;

    if (root->right) {
        // to ensure bottom up order, first call for right
        // subtree, then print this node
        printBoundaryRight(root->right);
        printf("%d ", root->data);
    }
    else if (root->left) {
        printBoundaryRight(root->left);
        printf("%d ", root->data);
    }
    // do nothing if it is a leaf node, this way we avoid
    // duplicates in output
}

// A function to do boundary traversal of a given binary tree
void printBoundary(struct node* root)
{
    if (root == NULL)
        return;

    printf("%d ", root->data);

    // Print the left boundary in top-down manner.
    printBoundaryLeft(root->left);

    // Print all leaf nodes
    printLeaves(root->left);
    printLeaves(root->right);

    // Print the right boundary in bottom-up manner
    printBoundaryRight(root->right);
```

```
}

// A utility function to create a node
struct node* newNode(int data)
{
    struct node* temp = (struct node*)malloc(sizeof(struct node));

    temp->data = data;
    temp->left = temp->right = NULL;

    return temp;
}

// Driver program to test above functions
int main()
{
    // Let us construct the tree given in the above diagram
    struct node* root = newNode(20);
    root->left = newNode(8);
    root->left->left = newNode(4);
    root->left->right = newNode(12);
    root->left->right->left = newNode(10);
    root->left->right->right = newNode(14);
    root->right = newNode(22);
    root->right->right = newNode(25);

    printBoundary(root);

    return 0;
}
```

Output:

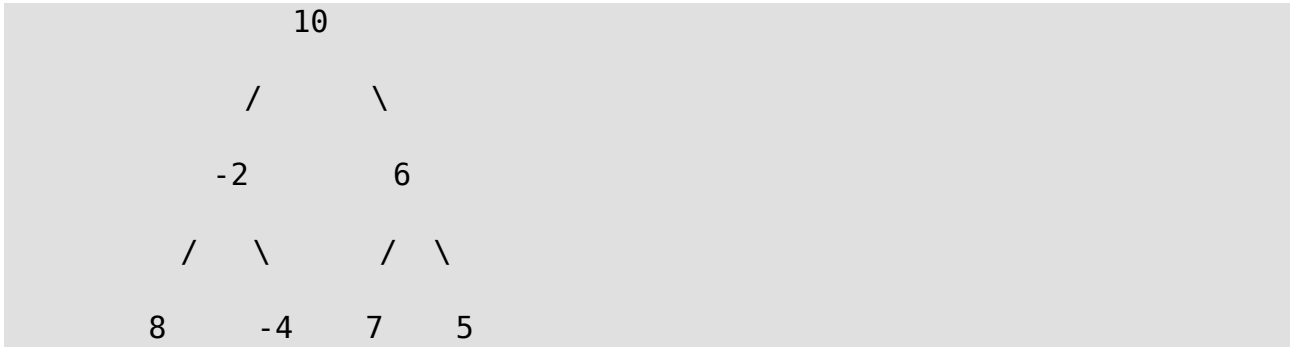
```
20 8 4 10 14 25 22
```

Time Complexity: O(n) where n is the number of nodes in binary tree

54. Convert a given tree to its Sum Tree

Given a Binary Tree where each node has positive and negative values. Convert this to a tree where each node contains the sum of the left and right sub trees in the original tree. The values of leaf nodes are changed to 0.

For example, the following tree



should be changed to

Solution:

Do a traversal of the given tree. In the traversal, store the old value of the current node, recursively call for left and right subtrees and change the value of current node as sum of the values returned by the recursive calls. Finally return the sum of new value and value (which is sum of values in the subtree rooted with this node).

```
// C++ program to convert a tree into its sum tree
#include <bits/stdc++.h>
using namespace std;

/* A tree node structure */
```

```
class node
{
    public:
int data;
node *left;
node *right;
};

// Convert a given tree to a tree where
// every node contains sum of values of
// nodes in left and right subtrees in the original tree
int toSumTree(node *Node)
{
    // Base case
    if(Node == NULL)
    return 0;

    // Store the old value
    int old_val = Node->data;

    // Recursively call for left and
    // right subtrees and store the sum as
    // new value of this node
    Node->data = toSumTree(Node->left) + toSumTree(Node->right);

    // Return the sum of values of nodes
    // in left and right subtrees and
    // old_value of this node
    return Node->data + old_val;
}

// A utility function to print
// inorder traversal of a Binary Tree
void printInorder(node* Node)
{
    if (Node == NULL)
        return;
    printInorder(Node->left);
    cout<<" "<<Node->data;
    printInorder(Node->right);
}

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

```
    return temp;
}

/* Driver code */
int main()
{
    node *root = NULL;
    int x;

    /* Constructing tree given in the above figure */
    root = newNode(10);
    root->left = newNode(-2);
    root->right = newNode(6);
    root->left->left = newNode(8);
    root->left->right = newNode(-4);
    root->right->left = newNode(7);
    root->right->right = newNode(5);

    toSumTree(root);

    // Print inorder traversal of the converted
    // tree to test result of toSumTree()
    cout<<"Inorder Traversal of the resultant tree is: \n";
    printInorder(root);
    return 0;
}

// This code is contributed by rathbhupendra
```

Output:

```
Inorder Traversal of the resultant tree is:

0 4 0 20 0 12 0
```

Time Complexity: The solution involves a simple traversal of the given tree. So the time complexity is O(n) where n is the number of nodes in the given Binary Tree

55. Foldable Binary Trees

Question: Given a binary tree, find out if the tree can be folded or not.

A tree can be folded if left and right subtrees of the tree are structure wise mirror image of each other. An empty tree is considered as foldable.

```
Consider the below trees:

(a) and (b) can be folded.

(c) and (d) cannot be folded.


(a)

      10

    /    \

   7      15

    \    /

     9  11


(b)

       10

      /  \

     7    15

    /      \

   9       11


(c)

       10

      /  \
```

```
     7   15
    /    /
   5   11


(d)


        10
      /   \
     7     15
   /  \    /
  9   10  12
```

Method 1 (Change Left subtree to its Mirror and compare it with Right subtree)
Algorithm: isFoldable(root)

```
1) If tree is empty, then return true.
2) Convert the left subtree to its mirror image
    mirror(root->left); /* See this post */
3) Check if the structure of left subtree and right subtree is same
   and store the result.
    res = isStructSame(root->left, root->right); /*isStructSame()
        recursively compares structures of two subtrees and returns
        true if structures are same */
4) Revert the changes made in step (2) to get the original tree.
    mirror(root->left);
5) Return result res stored in step 2.
```

Thanks to ajaym for suggesting this approach.

```cpp
// C++ program to check foldable binary tree
#include <bits/stdc++.h>
using namespace std;

/* You would want to remove below
3 lines if your compiler supports
bool, true and false */
#define bool int
#define true 1
#define false 0

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

/* converts a tree to its mrror image */
void mirror(node* node);

/* returns true if structure of
two trees a and b is same only
structure is considered for comparison, not data! */
bool isStructSame(node* a, node* b);

/* Returns true if the given tree is foldable */
bool isFoldable(node* root)
{
    bool res;

    /* base case */
    if (root == NULL)
        return true;

    /* convert left subtree to its mirror */
    mirror(root->left);

    /* Compare the structures of the
    right subtree and mirrored
    left subtree */
    res = isStructSame(root->left, root->right);

    /* Get the originial tree back */
    mirror(root->left);

    return res;
```

```
}

bool isStructSame(node* a, node* b)
{
    if (a == NULL && b == NULL) {
        return true;
    }
    if (a != NULL && b != NULL && isStructSame(a->left, b->left)
&& isStructSame(a->right, b->right)) {
        return true;
    }

    return false;
}

/* UTILITY FUNCTIONS */
/* Change a tree so that the roles of the left and
    right pointers are swapped at every node.
    See https:// www.geeksforgeeks.org/?p=662 for details */
void mirror(node* Node)
{
    if (Node == NULL)
        return;
    else {
        node* temp;

        /* do the subtrees */
        mirror(Node->left);
        mirror(Node->right);

        /* swap the pointers in this node */
        temp = Node->left;
        Node->left = Node->right;
        Node->right = temp;
    }
}

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

    return (Node);
}

/* Driver program to test mirror() */
```

```
int main(void)
{
    /* The constructed binary tree is
             1
        / \
        2 3
        \ /
        4 5
    */
    node* root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->right->left = newNode(4);
    root->left->right = newNode(5);

    if (isFoldable(root) == 1) {
        cout << "tree is foldable";
    }
    else {
        cout << "\ntree is not foldable";
    }
    return 0;
}

// This code is contributed by rathbhupendra
```

Output:

```
tree is foldable
```

Time complexity: O(n)

Method 2 (Check if Left and Right subtrees are Mirror)

There are mainly two functions:

// Checks if tree can be folded or not

```
IsFoldable(root)

1) If tree is empty then return true

2) Else check if left and right subtrees are structure wise mirrors of

   each other. Use utility function IsFoldableUtil(root->left,

   root->right) for this.
```

// Checks if n1 and n2 are mirror of each other.

```
IsFoldableUtil(n1, n2)
```

```
1) If both trees are empty then return true.

2) If one of them is empty and other is not then return false.

3) Return true if following conditions are met

   a) n1->left is mirror of n2->right

   b) n1->right is mirror of n2->left
```

```
#include <bits/stdc++.h>
using namespace std;

/* You would want to remove below 3 lines if your compiler
supports bool, true and false */
#define bool int
#define true 1
#define false 0

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

/* A utility function that checks
if trees with roots as n1 and n2
are mirror of each other */
bool IsFoldableUtil(node* n1, node* n2);

/* Returns true if the given tree can be folded */
bool IsFoldable(node* root)
{
    if (root == NULL) {
        return true;
    }

    return IsFoldableUtil(root->left, root->right);
}

/* A utility function that checks
if trees with roots as n1 and n2
are mirror of each other */
bool IsFoldableUtil(node* n1, node* n2)
{
    /* If both left and right subtrees are NULL,
    then return true */
    if (n1 == NULL && n2 == NULL) {
```

```
        return true;
    }

    /* If one of the trees is NULL and other is not,
    then return false */
    if (n1 == NULL || n2 == NULL) {
        return false;
    }

    /* Otherwise check if left and right subtrees are mirrors of
    their counterparts */
    return IsFoldableUtil(n1->left, n2->right) &&
IsFoldableUtil(n1->right, n2->left);
}

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

    return (Node);
}

/* Driver code */
int main(void)
{
    /* The constructed binary tree is
        1
        / \
        2 3
        \ /
        4 5
    */
    node* root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->right = newNode(4);
    root->right->left = newNode(5);

    if (IsFoldable(root) == true) {
        cout << "tree is foldable";
    }
    else {
        cout << "tree is not foldable";
```

```
    }

    return 0;
}

// This is code is contributed by rathbhupendra
```

Output:

```
tree is foldable
```

56. Check if removing an edge can divide a Binary Tree in two halves

Given a Binary Tree, find if there exist edge whose removal creates two trees of equal size.

Examples:

```
Input : root of following tree

          5

        /   \

      1      6

     /      /  \

    3      7    4

Output : true

Removing edge 5-6 creates two trees of equal size



Input : root of following tree

          5

        /   \

      1      6

           /  \

          7    4

        /  \    \

       3    2    8

Output : false

There is no edge whose removal creates two trees
```

of equal size.

Source- Kshitij IIT KGP

Method 1 (Simple)

First count number of nodes in whole tree. Let count of all nodes be n. Now traverse tree and for every node, find size of subtree rooted with this node. Let the subtree size be s. If n-s is equal to s, then return true, else false.

```
// C++ program to check if there exist an edge whose
// removal creates two trees of same size
#include<bits/stdc++.h>
using namespace std;

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

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

// To calculate size of tree with given root
int count(Node* root)
{
    if (root==NULL)
        return 0;
    return count(root->left) + count(root->right) + 1;
}

// This function returns true if there is an edge
// whose removal can divide the tree in two halves
// n is size of tree
bool checkRec(Node* root, int n)
{
    // Base cases
    if (root ==NULL)
       return false;

    // Check for root
    if (count(root) == n-count(root))
```

```
        return true;

    // Check for rest of the nodes
    return checkRec(root->left, n) ||
           checkRec(root->right, n);
}

// This function mainly uses checkRec()
bool check(Node *root)
{
    // Count total nodes in given tree
    int n = count(root);

    // Now recursively check all nodes
    return checkRec(root, n);
}

// Driver code
int main()
{
    struct Node* root = newNode(5);
    root->left = newNode(1);
    root->right = newNode(6);
    root->left->left = newNode(3);
    root->right->left = newNode(7);
    root->right->right = newNode(4);

    check(root)?  printf("YES") : printf("NO");

    return 0;
}
```

Output :

```
YES
```

Time complexity of above solution is $O(n^2)$ where n is number of nodes in given Binary Tree.

Method 2 (Efficient)

We can find the solution in O(n) time. The idea is to traverse tree in bottom up manner and while traversing keep updating size and keep checking if there is a node that follows the required property.

Below is the implementation of above idea.

```cpp
// C++ program to check if there exist an edge whose
// removal creates two trees of same size
#include<bits/stdc++.h>
using namespace std;

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

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

// To calculate size of tree with given root
int count(Node* root)
{
    if (root==NULL)
        return 0;
    return count(root->left) + count(root->right) + 1;
}

// This function returns size of tree rooted with given
// root. It also set "res" as true if there is an edge
// whose removal divides tree in two halves.
// n is size of tree
int checkRec(Node* root, int n, bool &res)
{
    // Base case
    if (root == NULL)
        return 0;

    // Compute sizes of left and right children
    int c = checkRec(root->left, n, res) + 1 +
            checkRec(root->right, n, res);

    // If required property is true for current node
    // set "res" as true
    if (c == n-c)
        res = true;

    // Return size
    return c;
}
```

```
// This function mainly uses checkRec()
bool check(Node *root)
{
    // Count total nodes in given tree
    int n = count(root);

    // Initialize result and recursively check all nodes
    bool res = false;
    checkRec(root, n,  res);

    return res;
}

// Driver code
int main()
{
    struct Node* root = newNode(5);
    root->left = newNode(1);
    root->right = newNode(6);
    root->left->left = newNode(3);
    root->right->left = newNode(7);
    root->right->right = newNode(4);

    check(root)?  printf("YES") : printf("NO");

    return 0;
}
```

Output :

```
YES
```

57. Locking and Unlocking of Resources arranged in the form of n-ary Tree

Given an n-ary tree of resources arranged hierarchically such that height of tree is O(Log N) where N is total number of nodes (or resources). A process needs to lock a resource node in order to use it. But a node cannot be locked if any of its descendant or ancestor is locked.

Following operations are required for a given resource node:

- islock()- returns true if a given node is locked and false if it is not. A node is locked if lock() has successfully executed for the node.
- Lock()- locks the given node if possible and updates lock information. Lock is possible only if ancestors and descendants of current node are not locked.
- unLock()- unlocks the node and updates information.

How design resource nodes and implement above operations such that following time complexities are achieved.

```
islock()  O(1)

Lock()    O(log N)

unLock()  O(log N)
```

It is given that resources need to be stored in the form of n-ary tree. Now the question is, how to augment the tree to achieve above complexity bounds.

Method 1 (Simple)

A Simple Solution is to store a boolean variable isLocked with every resource node. The boolean variable isLocked is true if current node is locked, else false. Let us see how operations work using this Approach.

- isLock(): Check isLocked of the given node.

- Lock(): If isLocked is set, then the node cannot be locked. Else check all descendants and ancestors of the node and if any of them is locked, then also this node cannot be locked. If none of the above conditions is true, then lock this node by setting isLocked as true.
- unLock(): If isLocked of given node is false, nothing to do. Else set isLocked as false.

Time Complexities:

```
isLock()  O(1)

Lock()    O(N)

unLock()  O(1)


Lock is O(N) because there can be O(N) descendants.
```

Method 2 (Time Complexities according to question)

The idea is to augment tree with following three fields:

1. A boolean field isLocked (same as above method).
2. Parent-Pointer to access all ancestors in O(Log n) time.
3. Count-of-locked-descendants. Note that a node can be ancestor of many descendants. We can check if any of the descendants is locked using this count.

Let us see how operations work using this Approach.

- isLock(): Check isLocked of the given node.
- Lock(): Traverse all ancestors. If none of the ancestors is locked, Count-of-locked-descendants is 0 and isLocked is false, set isLocked of current node as true. And increment Count-of-locked-descendants for all ancestors. Time complexity is O(Log N) as there can be at most O(Log N) ancestors.
- unLock(): Traverse all ancestors and decrease Count-of-locked-descendants for all ancestors. Set isLocked of current node as false. Time complexity is O(Log N)

58. Reverse alternate levels of a perfect binary tree

Given a Perfect Binary Tree, reverse the alternate level nodes of the binary tree.

```
Given tree:
             a
          /     \
         b       c
        /  \     /  \
       d    e    f    g
      / \  / \  / \  / \
      h  i j  k l  m  n  o

Modified tree:
            a
          /     \
         c       b
        /  \     /  \
       d    e    f    g
      / \  / \  / \  / \
     o  n m  l k  j  i  h
```

Method 1 (Simple)

A simple solution is to do following steps.

1) Access nodes level by level.

2) If current level is odd, then store nodes of this level in an array.

3) Reverse the array and store elements back in tree.

Method 2 (Using Two Traversals)

Another is to do two inorder traversals. Following are steps to be followed.

1) Traverse the given tree in inorder fashion and store all odd level nodes in an auxiliary array. For the above example given tree, contents of array become {h, i, b, j, k, l, m, c, n, o}

2) Reverse the array. The array now becomes {o, n, c, m, l, k, j, b, i, h}

3) Traverse the tree again inorder fashion. While traversing the tree, one by one take elements from array and store elements from array to every odd level traversed node.

For the above example, we traverse ‘h’ first in above array and replace ‘h’ with ‘o’. Then we traverse ‘i’ and replace it with n.

Following is the implementation of the above algorithm.

```
// C++ program to reverse alternate levels of a binary tree
#include<bits/stdc++.h>
#define MAX 100
using namespace std;

// A Binary Tree node
struct Node
{
    char data;
    struct Node *left, *right;
};

// A utility function to create a new Binary Tree Node
struct Node *newNode(char item)
{
    struct Node *temp =  new Node;
    temp->data = item;
    temp->left = temp->right = NULL;
    return temp;
}

// Function to store nodes of alternate levels in an array
void storeAlternate(Node *root, char arr[], int *index, int l)
{
    // Base case
```

```
    if (root == NULL) return;

    // Store elements of left subtree
    storeAlternate(root->left, arr, index, l+1);

    // Store this node only if this is a odd level node
    if (l%2 != 0)
    {
        arr[*index] = root->data;
        (*index)++;
    }

    // Store elements of right subtree
    storeAlternate(root->right, arr, index, l+1);
}

// Function to modify Binary Tree (All odd level nodes are
// updated by taking elements from array in inorder fashion)
void modifyTree(Node *root, char arr[], int *index, int l)
{
    // Base case
    if (root == NULL) return;

    // Update nodes in left subtree
    modifyTree(root->left, arr, index, l+1);

    // Update this node only if this is an odd level node
    if (l%2 != 0)
    {
        root->data = arr[*index];
        (*index)++;
    }

    // Update nodes in right subtree
    modifyTree(root->right, arr, index, l+1);
}

// A utility function to reverse an array from index
// 0 to n-1
void reverse(char arr[], int n)
{
    int l = 0, r = n-1;
    while (l < r)
    {
        int temp = arr[l];
        arr[l] = arr[r];
        arr[r] = temp;
        l++; r--;
    }
```

```
}

// The main function to reverse alternate nodes of a binary tree
void reverseAlternate(struct Node *root)
{
    // Create an auxiliary array to store nodes of alternate
levels
    char *arr = new char[MAX];
    int index = 0;

    // First store nodes of alternate levels
    storeAlternate(root, arr, &index, 0);

    // Reverse the array
    reverse(arr, index);

    // Update tree by taking elements from array
    index = 0;
    modifyTree(root, arr, &index, 0);
}

// A utility function to print indorder traversal of a
// binary tree
void printInorder(struct Node *root)
{
    if (root == NULL) return;
    printInorder(root->left);
    cout << root->data << " ";
    printInorder(root->right);
}

// Driver Program to test above functions
int main()
{
    struct Node *root = newNode('a');
    root->left = newNode('b');
    root->right = newNode('c');
    root->left->left = newNode('d');
    root->left->right = newNode('e');
    root->right->left = newNode('f');
    root->right->right = newNode('g');
    root->left->left->left = newNode('h');
    root->left->left->right = newNode('i');
    root->left->right->left = newNode('j');
    root->left->right->right = newNode('k');
    root->right->left->left = newNode('l');
    root->right->left->right = newNode('m');
    root->right->right->left = newNode('n');
    root->right->right->right = newNode('o');
```

```
    cout << "Inorder Traversal of given tree\n";
    printInorder(root);

    reverseAlternate(root);

    cout << "\n\nInorder Traversal of modified tree\n";
    printInorder(root);

    return 0;
}
```

Output:

```
Inorder Traversal of given tree

h d i b j e k a l f m c n g o



Inorder Traversal of modified tree

o d n c m e l a k f j b i g h
```

Time complexity of the above solution is O(n) as it does two inorder traversals of binary tree.

Method 3 (Using One Traversal)

```
// C++ program to reverse alternate levels of a tree
#include <bits/stdc++.h>
using namespace std;

struct Node
{
    char key;
    Node *left, *right;
};

void preorder(struct Node *root1, struct Node* root2, int lvl)
{
    // Base cases
    if (root1 == NULL || root2==NULL)
        return;

    // Swap subtrees if level is even
    if (lvl%2 == 0)
        swap(root1->key, root2->key);

    // Recur for left and right subtrees (Note : left of root1
```

```
    // is passed and right of root2 in first call and opposite
    // in second call.
    preorder(root1->left, root2->right, lvl+1);
    preorder(root1->right, root2->left, lvl+1);
}

// This function calls preorder() for left and right children
// of root
void reverseAlternate(struct Node *root)
{
   preorder(root->left, root->right, 0);
}

// Inorder traversal (used to print initial and
// modified trees)
void printInorder(struct Node *root)
{
    if (root == NULL)
       return;
    printInorder(root->left);
    cout << root->key << " ";
    printInorder(root->right);
}

// A utility function to create a new node
Node *newNode(int key)
{
    Node *temp = new Node;
    temp->left = temp->right = NULL;
    temp->key = key;
    return temp;
}

// Driver program to test above functions
int main()
{
    struct Node *root = newNode('a');
    root->left = newNode('b');
    root->right = newNode('c');
    root->left->left = newNode('d');
    root->left->right = newNode('e');
    root->right->left = newNode('f');
    root->right->right = newNode('g');
    root->left->left->left = newNode('h');
    root->left->left->right = newNode('i');
    root->left->right->left = newNode('j');
    root->left->right->right = newNode('k');
    root->right->left->left = newNode('l');
    root->right->left->right = newNode('m');
    root->right->right->left = newNode('n');
```

```
    root->right->right->right = newNode('o');

    cout << "Inorder Traversal of given tree\n";
    printInorder(root);

    reverseAlternate(root);

    cout << "\n\nInorder Traversal of modified tree\n";
    printInorder(root);
    return 0;
}
```

Output :

```
Inorder Traversal of given tree

h d i b j e k a l f m c n g o



Inorder Traversal of modified tree

o d n c m e l a k f j b i g h
```

59. Custom Tree Problem

You are given a set of links, e.g.

```
a ---> b

b ---> c

b ---> d

a ---> e
```

Print the tree that would form when each pair of these links that has the same character as start and end point is joined together. You have to maintain fidelity w.r.t. the height of nodes, i.e. nodes at height n from root should be printed at same row or column. For set of links given above, tree printed should be –

```
-->a

   |-->b

   |    |-->c

   |    |-->d

   |-->e
```

Note that these links need not form a single tree; they could form, ahem, a forest. Consider the following links

```
a ---> b

a ---> g

b ---> c

c ---> d

d ---> e

c ---> f

z ---> y
```

```
y ---> x

x ---> w
```

The output would be following forest.

```
-->a
   |-->b
   |   |-->c
   |   |   |-->d
   |   |   |   |-->e
   |   |   |-->f
   |-->g


-->z
   |-->y
   |   |-->x
   |   |   |-->w
```

You can assume that given links can form a tree or forest of trees only, and there are no duplicates among links.

Solution: The idea is to maintain two arrays, one array for tree nodes and other for trees themselves (we call this array forest). An element of the node array contains the TreeNode object that corresponds to respective character. An element of the forest array contains Tree object that corresponds to respective root of tree.

It should be obvious that the crucial part is creating the forest here, once it is created, printing it out in required format is straightforward. To create the forest, following procedure is used –

```
Do following for each input link,

1. If start of link is not present in node array
```

```
    Create TreeNode objects for start character

    Add entries of start in both arrays.

2. If end of link is not present in node array

    Create TreeNode objects for start character

    Add entry of end in node array.

3. If end of link is present in node array.

    If end of link is present in forest array, then remove it

    from there.

4. Add an edge (in tree) between start and end nodes of link.
```

It should be clear that this procedure runs in linear time in number of nodes as well as of links – it makes only one pass over the links. It also requires linear space in terms of alphabet size.

Following is Java implementation of above algorithm. In the following implementation characters are assumed to be only lower case characters from ‘a’ to ‘z’.

```
// Java program to create a custom tree from a given set of links.

// The main class that represents tree and has main method
public class Tree {

    private TreeNode root;

    /* Returns an array of trees from links input. Links are
assumed to
       be Strings of the form "<s> <e>" where <s> and <e> are
starting
       and ending points for the link. The returned array is of
size 26
       and has non-null values at indexes corresponding to roots
of trees
       in output */
    public Tree[] buildFromLinks(String [] links) {

        // Create two arrays for nodes and forest
        TreeNode[] nodes = new TreeNode[26];
        Tree[] forest = new Tree[26];
```

```
        // Process each link
        for (String link : links) {

            // Find the two ends of current link
            String[] ends = link.split(" ");
            int start = (int) (ends[0].charAt(0) - 'a'); // Start
node
            int end   = (int) (ends[1].charAt(0) - 'a'); // End
node

            // If start of link not seen before, add it two both
arrays
            if (nodes[start] == null)
            {
                nodes[start] = new TreeNode((char) (start +
'a'));

                // Note that it may be removed later when this
character is
                // last character of a link. For example, let we
first see
                // a--->b, then c--->a.  We first add 'a' to array
of trees
                // and when we see link c--->a, we remove it from
trees array.
                forest[start] = new
Tree(nodes[start]);
            }

            // If end of link is not seen before, add it to the
nodes array
            if (nodes[end] == null)
                nodes[end] = new TreeNode((char) (end +
'a'));

            // If end of link is seen before, remove it from
forest if
            // it exists there.
            else forest[end] = null;

            // Establish Parent-Child Relationship between Start
and End
            nodes[start].addChild(nodes[end], end);
        }
        return forest;
    }

    // Constructor
    public Tree(TreeNode root) { this.root = root;  }
```

```
    public static void printForest(String[] links)
    {
        Tree t = new Tree(new TreeNode('\0'));
        for (Tree t1 : t.buildFromLinks(links)) {
           if (t1 != null)
           {
              t1.root.printTreeIdented("");
              System.out.println("");
           }
        }
    }

    // Driver method to test
    public static void main(String[] args) {
        String [] links1 = {"a b", "b c", "b d", "a e"};
        System.out.println("------------ Forest 1
----------------");
        printForest(links1);

        String [] links2 = {"a b", "a g", "b c", "c d", "d e", "c
f",
                            "z y", "y x", "x w"};
        System.out.println("------------ Forest 2
----------------");
        printForest(links2);
    }
}

// Class to represent a tree node
class TreeNode {
    TreeNode []children;
    char c;

    // Adds a child 'n' to this node
    public void addChild(TreeNode n, int index) {
this.children[index] = n;}

    // Constructor
    public TreeNode(char c) { this.c = c;  this.children = new
TreeNode[26];}

    // Recursive method to print indented tree rooted with this
node.
    public void printTreeIdented(String indent) {
            System.out.println(indent + "-->" + c);
            for (TreeNode child : children) {
              if (child != null)
                child.printTreeIdented(indent + "   |");
```

```
            }
    }
}
```

Output:

```
------------ Forest 1 ----------------

-->a

   |-->b

   |   |-->c

   |   |-->d

   |-->e



------------ Forest 2 ----------------

-->a

   |-->b

   |   |-->c

   |   |   |-->d

   |   |   |   |-->e

   |   |   |-->f

   |-->g



-->z

   |-->y

   |   |-->x

   |   |   |-->w
```

60. Threaded Binary Tree

Inorder traversal of a Binary tree can either be done using recursion or with the use of a auxiliary stack. The idea of threaded binary trees is to make inorder traversal faster and do it without stack and without recursion. A binary tree is made threaded by making all right child pointers that would normally be NULL point to the inorder successor of the node (if it exists).

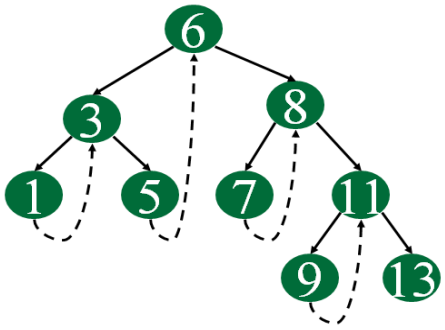
There are two types of threaded binary trees.

Single Threaded: Where a NULL right pointers is made to point to the inorder successor (if successor exists)

Double Threaded: Where both left and right NULL pointers are made to point to inorder predecessor and inorder successor respectively. The predecessor threads are useful for reverse inorder traversal and postorder traversal.

The threads are also useful for fast accessing ancestors of a node.

Following diagram shows an example Single Threaded Binary Tree. The dotted lines represent threads.



C representation of a Threaded Node

Following is C representation of a single threaded node.

```
struct Node
{
```

```
    int data;
    Node *left, *right;
    bool rightThread;
}
```

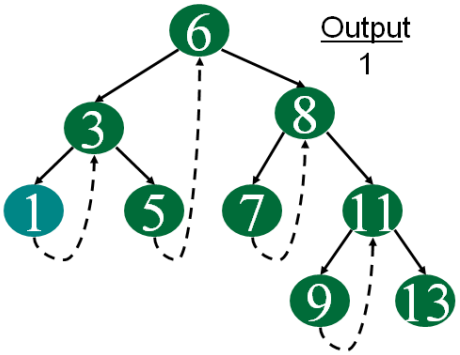
Since right pointer is used for two purposes, the boolean variable rightThread is used to indicate whether right pointer points to right child or inorder successor. Similarly, we can add leftThread for a double threaded binary tree.

Inorder Taversal using Threads

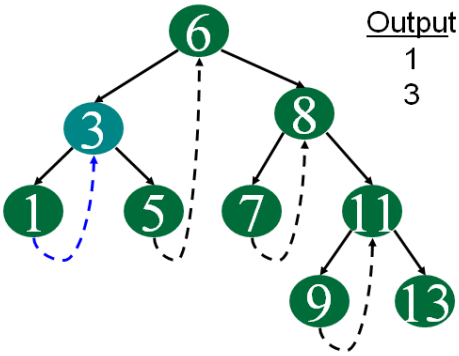
Following is C code for inorder traversal in a threaded binary tree.

```
// Utility function to find leftmost node in a tree rooted with n
struct Node* leftMost(struct Node *n)
{
    if (n == NULL)
       return NULL;

    while (n->left != NULL)
        n = n->left;

    return n;
}

// C code to do inorder traversal in a threaded binary tree
void inOrder(struct Node *root)
{
    struct Node *cur = leftmost(root);
    while (cur != NULL)
    {
        printf("%d ", cur->data);

        // If this node is a thread node, then go to
        // inorder successor
        if (cur->rightThread)
            cur = cur->right;
        else // Else go to the leftmost child in right subtree
            cur = leftmost(cur->right);
    }
}
```

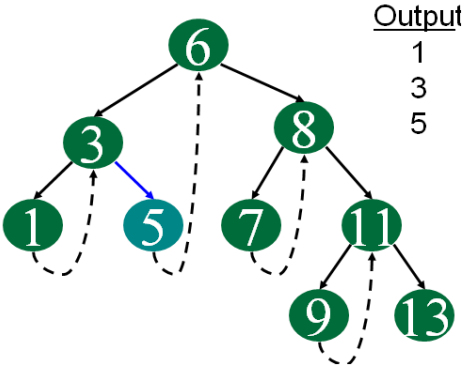
Following diagram demonstrates inorder order traversal using threads.



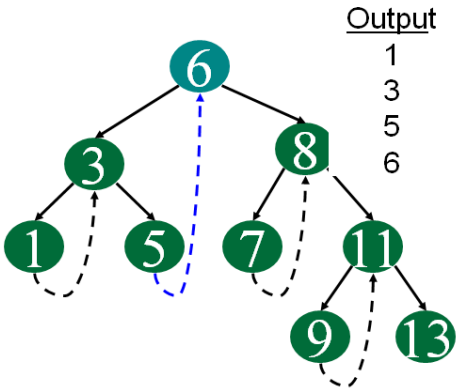
Start at leftmost node, print it



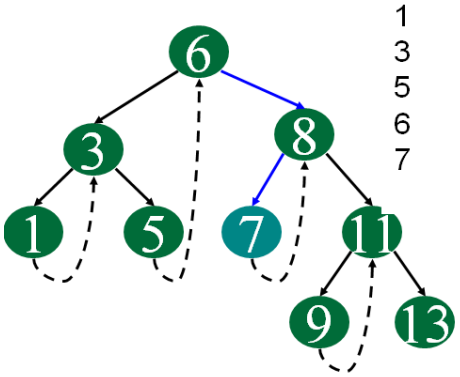
Follow thread to right, print node



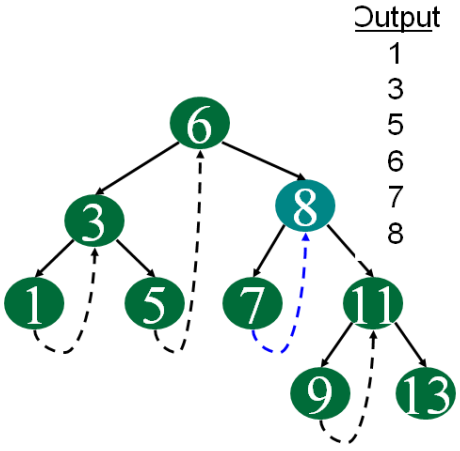
Follow link to right, go to leftmost node and print



Follow thread to right, print node



Follow link to right, go to leftmost node and print



Follow thread to right, print node

continue same way for remaining node.....