**Binary tree**

# 162. Level order traversal

Given a binary tree, find its level order traversal.  
Level order traversal of a tree is [breadth-first traversal f](http://www.geeksforgeeks.org/breadth-first-traversal-for-a-graph/)or the tree.

**Example 1:**

**Input:**

   1

 /   \

 3     2

**Output:**1 3 2

**Example 2:**

**Input:**

        10

    /      \

  20       30

  /   \

40   60

**Output:**10 20 30 40 60

**Your Task:**  
You don't have to take any input. Complete the function **levelOrder()** that takes the root node as input parameter and returns a list of integers containing the level order traversal of the given Binary Tree.

**Expected Time Complexity:**O(N)  
**Expected Auxiliary Space:**O(N)

**Constraints:**  
1 ≤ Number of nodes ≤ 105  
1 ≤ Data of a node ≤ 105

## Solution:

**Method 1 (Use function to print a current level)**

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

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

**printLevelorder(tree)**

for d = 1 to height(tree)

printCurrentLevel(tree, d);

/\*Function to print all nodes at a current level\*/

**printCurrentLevel(tree, level)**

if tree is NULL then return;

if level is 1, then

print(tree->data);

else if level greater than 1, then

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

printCurrentLevel(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 prototypes \*/

void printCurrentLevel(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++)

printCurrentLevel(root, i);

}

/\* Print nodes at a current level \*/

void printCurrentLevel(node\* root, int level)

{

if (root == NULL)

return;

if (level == 1)

cout << root->data << " ";

else if (level > 1) {

printCurrentLevel(root->left, level - 1);

printCurrentLevel(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;

}

**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, the call stack uses O(log n) space, (i.e., the 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.

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

/\* C++ program to print level

order traversal using STL \*/

#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 traversal

queue<Node\*> q;

// Enqueue Root and initialize height

q.push(root);

while (q.empty() == false) {

// Print front of queue and remove it from queue

Node\* node = q.front();

cout << node->data << " ";

q.pop();

/\* Enqueue left child \*/

if (node->left != NULL)

q.push(node->left);

/\*Enqueue right child \*/

if (node->right != NULL)

q.push(node->right);

}

}

// Utility function to create a new tree node

Node\* newNode(int data)

{

Node\* temp = new Node;

temp->data = data;

temp->left = temp->right = NULL;

return temp;

}

// Driver program to test above functions

int main()

{

// Let us create binary tree shown in above diagram

Node\* root = newNode(1);

root->left = newNode(2);

root->right = newNode(3);

root->left->left = newNode(4);

root->left->right = newNode(5);

cout << "Level Order traversal of binary tree is \n";

printLevelOrder(root);

return 0;

}

**Output**

Level Order traversal of binary tree is

1 2 3 4 5

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

# 163. Reverse Level Order traversal

Given a binary tree of size N, find its reverse level order traversal. ie- the traversal must begin from the last level.

**Example 1:**

**Input :**

1

/ \

3 2

**Output:** 3 2 1

**Explanation:**

Traversing level 1 : 3 2

Traversing level 0 : 1

**Example 2:**

**Input :**

10

/ \

20 30

/ \

40 60

**Output:** 40 60 20 30 10

**Explanation:**

Traversing level 2 : 40 60

Traversing level 1 : 20 30

Traversing level 0 : 10

**Your Task:**  
You dont need to read input or print anything. Complete the function **reverseLevelOrder()**which takes the root of the tree as input parameter and returns a list containing the reverse level order traversal of the given tree.

**Expected Time Complexity:** O(N)  
**Expected Auxiliary Space:**O(N)

**Constraints:**  
1 ≤ N ≤ 10^4

## Solution:

**METHOD 1 (Recursive function to print a given level)**   
We can easily modify the method 1 of the normal [level order traversal](https://www.geeksforgeeks.org/level-order-tree-traversal/). In method 1, we have a method printGivenLevel() which prints a given level number. The only thing we need to change is, instead of calling printGivenLevel() from the first level to the last level, we call it from the last level to the first level.

// A recursive C++ program to print

// REVERSE level order traversal

#include <bits/stdc++.h>

using namespace std;

/\* A binary tree node has data,

pointer to left and right child \*/

class node

{

public:

int data;

node\* left;

node\* right;

};

/\*Function prototypes\*/

void printGivenLevel(node\* root, int level);

int height(node\* node);

node\* newNode(int data);

/\* Function to print REVERSE

level order traversal a tree\*/

void reverseLevelOrder(node\* root)

{

int h = height(root);

int i;

for (i=h; i>=1; i--) //THE ONLY LINE DIFFERENT FROM NORMAL LEVEL ORDER

printGivenLevel(root, i);

}

/\* Print nodes at a given level \*/

void printGivenLevel(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";

reverseLevelOrder(root);

return 0;

}

**Output:**

Level Order traversal of binary tree is

4 5 2 3 1

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

**METHOD 2 (Using Queue and Stack)**   
The method 2 of [normal level order traversal](https://www.geeksforgeeks.org/level-order-tree-traversal/) can also be easily modified to print level order traversal in reverse order. The idea is to use a deque(double-ended queue) to get the reverse level order. A deque allows insertion and deletion at both the ends. If we do normal level order traversal and instead of printing a node, push the node to a stack and then print the contents of the deque, we get “5 4 3 2 1” for the above example tree, but the output should be “4 5 2 3 1”. So to get the correct sequence (left to right at every level), we process children of a node in reverse order, we first push the right subtree to the deque, then process the left subtree.

// A C++ program to print REVERSE level order traversal using stack and queue

// This approach is adopted from following link

// http://tech-queries.blogspot.in/2008/12/level-order-tree-traversal-in-reverse.html

#include <bits/stdc++.h>

using namespace std;

/\* A binary tree node has data, pointer to left and right children \*/

struct node

{

int data;

struct node\* left;

struct node\* right;

};

/\* Given a binary tree, print its nodes in reverse level order \*/

void reverseLevelOrder(node\* root)

{

stack <node \*> S;

queue <node \*> Q;

Q.push(root);

// Do something like normal level order traversal order. Following are the

// differences with normal level order traversal

// 1) Instead of printing a node, we push the node to stack

// 2) Right subtree is visited before left subtree

while (Q.empty() == false)

{

/\* Dequeue node and make it root \*/

root = Q.front();

Q.pop();

S.push(root);

/\* Enqueue right child \*/

if (root->right)

Q.push(root->right); // NOTE: RIGHT CHILD IS ENQUEUED BEFORE LEFT

/\* Enqueue left child \*/

if (root->left)

Q.push(root->left);

}

// Now pop all items from stack one by one and print them

while (S.empty() == false)

{

root = S.top();

cout << root->data << " ";

S.pop();

}

}

/\* Helper function that allocates a new node with the

given data and NULL left and right pointers. \*/

node\* newNode(int data)

{

node\* temp = new node;

temp->data = data;

temp->left = NULL;

temp->right = NULL;

return (temp);

}

/\* Driver program to test above functions\*/

int main()

{

struct node \*root = newNode(1);

root->left = newNode(2);

root->right = newNode(3);

root->left->left = newNode(4);

root->left->right = newNode(5);

root->right->left = newNode(6);

root->right->right = newNode(7);

cout << "Level Order traversal of binary tree is \n";

reverseLevelOrder(root);

return 0;

}

**Output:**

Level Order traversal of binary tree is

4 5 6 7 2 3 1

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

Space Complexity: O(n)

# 164. Height of a tree

Given a binary tree, find its height.

**Example 1:**

**Input:**

1

/ \

2 3

**Output:** 2

**Example 2:**

**Input:**

2

\

1

/

3

**Output:** 3

**Your Task:**  
You don't need to read input or print anything. Your task is to complete the function **height()**which takes root node of the tree as input parameter and returns an integer denoting the height of the tree. If the tree is empty, return 0.

**Expected Time Complexity:** O(N)  
**Expected Auxiliary Space:** O(N)

**Constraints:**  
1 <= Number of nodes <= 105  
1 <= Data of a node <= 105

## Solution:

Recursively calculate height of left and right subtrees of a node and assign height to the node as max of the heights of two children plus 1. See below pseudo code and program for details.  
**Algorithm:**

maxDepth()

1. If tree is empty then return -1

2. Else

(a) Get the max depth of left subtree recursively i.e.,

call maxDepth( tree->left-subtree)

(a) Get the max depth of right subtree recursively i.e.,

call maxDepth( tree->right-subtree)

(c) Get the max of max depths of left and right

subtrees and add 1 to it for the current node.

max\_depth = max(max dept of left subtree,

max depth of right subtree)

+ 1

(d) Return max\_depth

**See the below diagram for more clarity about execution of the recursive function maxDepth() for above example tree.**

maxDepth('1') = max(maxDepth('2'), maxDepth('3')) + 1

= 1 + 1

/ \

/ \

/ \

/ \

/ \

maxDepth('2') = 1 maxDepth('3') = 0

= max(maxDepth('4'), maxDepth('5')) + 1

= 1 + 0 = 1

/ \

/ \

/ \

/ \

/ \

maxDepth('4') = 0 maxDepth('5') = 0

**Implementation:**

int height(struct Node\* node){

if(!node)

return 0;

return max(height(node->left), height(node->right))+1;

}

**Time Complexity:**O(n)

**Space Complexity:**O(height of the tree) for recursion call stack.

**Method 2:**Another method to solve this problem is to do **Level Order Traversal.** While doing the level order traversal, while adding Nodes at each level to Queue, we have to add **NULL Node**so that whenever it is encountered, we can increment the value of variable and that level get counted.

**Implementation:**

#include <iostream>

#include <bits/stdc++.h>

using namespace std;

// A Tree node

struct Node

{

int key;

struct Node\* left, \*right;

};

// Utility function to create a new node

Node\* newNode(int key)

{

Node\* temp = new Node;

temp->key = key;

temp->left = temp->right = NULL;

return (temp);

}

/\*Function to find the height(depth) of the tree\*/

int height(struct Node\* root){

//Initialising a variable to count the

//height of tree

int depth = 0;

queue<Node\*>q;

//Pushing first level element along with NULL

q.push(root);

q.push(NULL);

while(!q.empty()){

Node\* temp = q.front();

q.pop();

//When NULL encountered, increment the value

if(temp == NULL){

depth++;

}

//If NULL not encountered, keep moving

if(temp != NULL){

if(temp->left){

q.push(temp->left);

}

if(temp->right){

q.push(temp->right);

}

}

//If queue still have elements left,

//push NULL again to the queue.

else if(!q.empty()){

q.push(NULL);

}

}

return depth;

}

// Driver program

int main()

{

// Let us create Binary Tree shown in above example

Node \*root = newNode(1);

root->left = newNode(12);

root->right = newNode(13);

root->right->left = newNode(14);

root->right->right = newNode(15);

root->right->left->left = newNode(21);

root->right->left->right = newNode(22);

root->right->right->left = newNode(23);

root->right->right->right = newNode(24);

cout<<"Height(Depth) of tree is: "<<height(root);

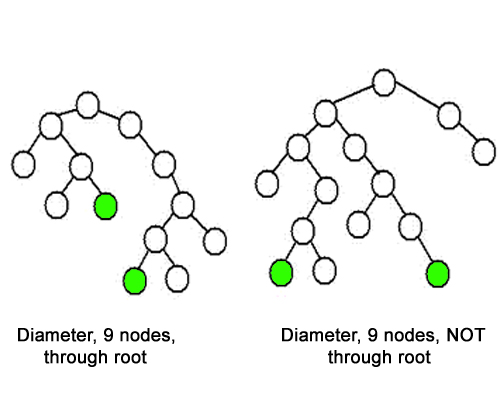
}

**Time Complexity:**O(n)

**Space Complexity:**O(n)

# 165. Diameter of a 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 the longest path are shaded (note that there is more than one path in each tree of length nine, but no path longer than nine nodes).

[](http://geeksforgeeks.org/wp-content/uploads/tree_diameter.GIF)

**Example 1:**

**Input:**

     1

    /  \

   2    3

**Output:** 3

**Example 2:**

**Input:**

         10

       /   \

     20    30

   /   \

   40   60

**Output:** 4

**Your Task:**  
You need to **complete**the **function diameter()**that takes **root**as **parameter**and **returns**the **diameter**.  
  
**Expected Time Complexity:**O(N).  
**Expected Auxiliary Space:**O(Height of the Tree).

**Constraints:**  
1 <= Number of nodes <= 10000  
1 <= Data of a node <= 1000

## Solution:

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:

// Recursive optimized C program to find the diameter of a

// Binary Tree

#include <bits/stdc++.h>

using namespace std;

// A binary tree node has data, pointer to left child

// and a pointer to right child

struct node {

int data;

struct node \*left, \*right;

};

// function to create a new node of tree and returns pointer

struct node\* newNode(int data);

// returns max of two integers

int max(int a, int b) { return (a > b) ? a : 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);

}

// Driver Code

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);

// Function Call

cout << "Diameter of the given binary tree is " <<

diameter(root);

return 0;

}

**Output**

Diameter of the given binary tree is 4

**Time Complexity:** O(n2)  
   
**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).

// Recursive optimized C++ program to find the diameter of a

// Binary Tree

#include <bits/stdc++.h>

using namespace std;

// A binary tree node has data, pointer to left child

// and a pointer to right child

struct node {

int data;

struct node \*left, \*right;

};

// function to create a new node of tree and returns pointer

struct node\* newNode(int data);

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));

}

// 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 Code

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);

int height = 0;

// Function Call

cout << "Diameter of the given binary tree is " << diameterOpt(root, &height);

return 0;

}

**Output**

The diameter of given binary tree is : 4

**Time Complexity:** O(n)

# 166. Mirror of a 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

## Solution:

**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++ implementation of the approach

#include <iostream>

using namespace std;

// 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);

cout <<" "<< 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

cout <<"Inorder of original tree: ";

inorder(tree);

node\* mirror = NULL;

mirrorify(tree, &mirror);

// Print inorder traversal of the mirror tree

cout <<"\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 2

**Approach 2:**  
 In order to change the original tree in its mirror tree, then we simply swap the left and right link of each node. If the node is leaf node then do nothing.

#include <iostream>

using namespace std;

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;

}

// Function to print the inrder traversal

void inorder(node\* root)

{

if (root == NULL)

return;

inorder(root->left);

printf("%d ", root->val);

inorder(root->right);

}

// Function to convert to mirror tree

treenode\* mirrorTree(node\* root)

{

// Base Case

if (root == NULL)

return root;

node\* t = root->left;

root->left = root->right;

root->right = t;

if (root->left)

mirrorTree(root->left);

if (root->right)

mirrorTree(root->right);

return root;

}

// 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);

printf("Inorder of original tree: ");

inorder(tree);

// Function call

mirrorTree(tree);

printf("\nInorder of Mirror tree: ");

inorder(tree);

return 0;

}

**Output**

Inorder of original tree: 2 3 4 5 6

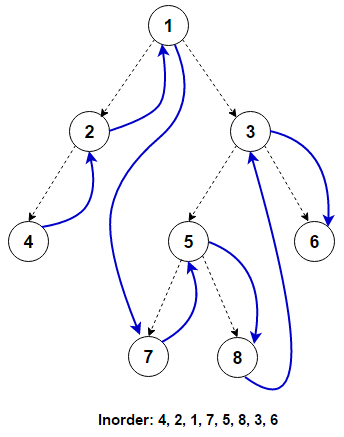
Inorder of Mirror tree: 6 5 4 3 2

# 167. Inorder Traversal of a tree both using recursion and Iteration

For traversing a (non-empty) binary tree in an inorder fashion, we must do these three things for every node n starting from the tree’s root:

**(L)** Recursively traverse its left subtree. When this step is finished, we are back at n again.  
**(N)** Process n itself.  
**(R)** Recursively traverse its right subtree. When this step is finished, we are back at n again.

In normal inorder traversal, we visit the left subtree before the right subtree. If we visit the right subtree before visiting the left subtree, it is referred to as reverse inorder traversal.



## Recursive Implementation

As we can see, before processing any node, the left subtree is processed first, followed by the node, and the right subtree is processed at last. These operations can be defined recursively for each node. The recursive implementation is referred to as a [Depth–first search (DFS)](https://www.techiedelight.com/depth-first-search/), as the search tree is deepened as much as possible on each child before going to the next sibling.

Following is the C++ program that demonstrates it:

#include <iostream>

using namespace std;

// Data structure to store a binary tree node

struct Node

{

int data;

Node \*left, \*right;

Node(int data)

{

this->data = data;

this->left = this->right = nullptr;

}

};

// Recursive function to perform inorder traversal on the tree

void inorder(Node\* root)

{

// return if the current node is empty

if (root == nullptr) {

return;

}

// Traverse the left subtree

inorder(root->left);

// Display the data part of the root (or current node)

cout << root->data << " ";

// Traverse the right subtree

inorder(root->right);

}

int main()

{

/\* Construct the following tree

1

/ \

/ \

2 3

/ / \

/ / \

4 5 6

/ \

/ \

7 8

\*/

Node\* root = new Node(1);

root->left = new Node(2);

root->right = new Node(3);

root->left->left = new Node(4);

root->right->left = new Node(5);

root->right->right = new Node(6);

root->right->left->left = new Node(7);

root->right->left->right = new Node(8);

inorder(root);

return 0;

}

## Iterative Implementation

To convert the above recursive procedure into an iterative one, we need an explicit stack. Following is a simple stack-based iterative algorithm to perform inorder traversal:

**iterativeInorder(node)**  
   
  s —> empty stack  
  while (not s.isEmpty() or node != null)  
    if (node != null)  
      s.push(node)  
      node —> node.left  
    else  
      node —> s.pop()  
      visit(node)  
      node —> node.right

The algorithm can be implemented as follows in C++:

#include <iostream>

#include <stack>

using namespace std;

// Data structure to store a binary tree node

struct Node

{

int data;

Node \*left, \*right;

Node(int data)

{

this->data = data;

this->left = this->right = nullptr;

}

};

// Iterative function to perform inorder traversal on the tree

void inorderIterative(Node\* root)

{

// create an empty stack

stack<Node\*> stack;

// start from the root node (set current node to the root node)

Node\* curr = root;

// if the current node is null and the stack is also empty, we are done

while (!stack.empty() || curr != nullptr)

{

// if the current node exists, push it into the stack (defer it)

// and move to its left child

if (curr != nullptr)

{

stack.push(curr);

curr = curr->left;

}

else {

// otherwise, if the current node is null, pop an element from the stack,

// print it, and finally set the current node to its right child

curr = stack.top();

stack.pop();

cout << curr->data << " ";

curr = curr->right;

}

}

}

int main()

{

/\* Construct the following tree

1

/ \

/ \

2 3

/ / \

/ / \

4 5 6

/ \

/ \

7 8

\*/

Node\* root = new Node(1);

root->left = new Node(2);

root->right = new Node(3);

root->left->left = new Node(4);

root->right->left = new Node(5);

root->right->right = new Node(6);

root->right->left->left = new Node(7);

root->right->left->right = new Node(8);

inorderIterative(root);

return 0;

}

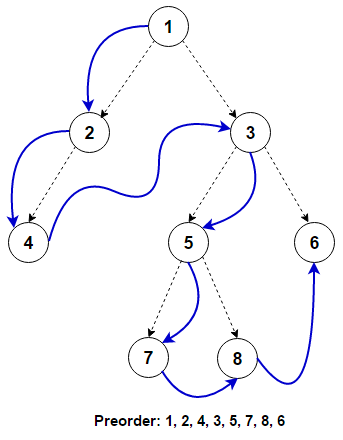
The time complexity of the above solutions is O(n), where n is the total number of nodes in the binary tree. The space complexity of the program is O(n) as the space required is proportional to the height of the tree, which can be equal to the total number of nodes in the tree in worst-case for skewed trees.

# 168. Preorder Traversal of a tree both using recursion and Iteration

For traversing a (non-empty) binary tree in a preorder fashion, we must do these three things for every node n starting from the tree’s root:

**(N)** Process n itself.  
**(L)** Recursively traverse its left subtree. When this step is finished, we are back at n again.  
**(R)** Recursively traverse its right subtree. When this step is finished, we are back at n again.

In normal preorder traversal, visit the left subtree before the right subtree. If we visit the right subtree before visiting the left subtree, it is referred to as reverse preorder traversal.



## Recursive Implementation

As we can see, only after processing any node, the left subtree is processed, followed by the right subtree. These operations can be defined recursively for each node. The recursive implementation is referred to as a [Depth–first search (DFS)](https://www.techiedelight.com/depth-first-search/), as the search tree is deepened as much as possible on each child before going to the next sibling.

Following is the C++ program that demonstrates it:

#include <iostream>

using namespace std;

// Data structure to store a binary tree node

struct Node

{

int data;

Node \*left, \*right;

Node(int data)

{

this->data = data;

this->left = this->right = nullptr;

}

};

// Recursive function to perform preorder traversal on the tree

void preorder(Node\* root)

{

// if the current node is empty

if (root == nullptr) {

return;

}

// Display the data part of the root (or current node)

cout << root->data << " ";

// Traverse the left subtree

preorder(root->left);

// Traverse the right subtree

preorder(root->right);

}

int main()

{

/\* Construct the following tree

1

/ \

/ \

2 3

/ / \

/ / \

4 5 6

/ \

/ \

7 8

\*/

Node\* root = new Node(1);

root->left = new Node(2);

root->right = new Node(3);

root->left->left = new Node(4);

root->right->left = new Node(5);

root->right->right = new Node(6);

root->right->left->left = new Node(7);

root->right->left->right = new Node(8);

preorder(root);

return 0;

}

## Iterative Implementation

To convert the above recursive procedure into an iterative one, we need an explicit stack. Following is a simple stack-based iterative algorithm to perform preorder traversal:

**iterativePreorder(node)**  
   
if (node = null)  
  return  
s —> empty stack  
s.push(node)  
while (not s.isEmpty())  
  node —> s.pop()  
  visit(node)  
  if (node.right != null)  
    s.push(node.right)  
  if (node.left != null)  
    s.push(node.left)

The algorithm can be implemented as follows in C++:

#include <iostream>

#include <stack>

using namespace std;

// Data structure to store a binary tree node

struct Node

{

int data;

Node \*left, \*right;

Node(int data)

{

this->data = data;

this->left = this->right = nullptr;

}

};

// Iterative function to perform preorder traversal on the tree

void preorderIterative(Node\* root)

{

// return if the tree is empty

if (root == nullptr)

return;

// create an empty stack and push the root node

stack<Node\*> stack;

stack.push(root);

// loop till stack is empty

while (!stack.empty())

{

// pop a node from the stack and print it

Node\* curr = stack.top();

stack.pop();

cout << curr->data << " ";

// push the right child of the popped node into the stack

if (curr->right) {

stack.push(curr->right);

}

// push the left child of the popped node into the stack

if (curr->left) {

stack.push(curr->left);

}

// the right child must be pushed first so that the left child

// is processed first (LIFO order)

}

}

int main()

{

/\* Construct the following tree

1

/ \

/ \

2 3

/ / \

/ / \

4 5 6

/ \

/ \

7 8

\*/

Node\* root = new Node(1);

root->left = new Node(2);

root->right = new Node(3);

root->left->left = new Node(4);

root->right->left = new Node(5);

root->right->right = new Node(6);

root->right->left->left = new Node(7);

root->right->left->right = new Node(8);

preorderIterative(root);

return 0;

}

The above solution can be further optimized by pushing only the right children to the stack.

#include <iostream>

#include <stack>

using namespace std;

// Data structure to store a binary tree node

struct Node

{

int data;

Node \*left, \*right;

Node(int data)

{

this->data = data;

this->left = this->right = nullptr;

}

};

// Iterative function to perform preorder traversal on the tree

void preorderIterative(Node\* root)

{

// return if the tree is empty

if (root == nullptr) {

return;

}

// create an empty stack and push the root node

stack<Node\*> stack;

stack.push(root);

// start from the root node (set current node to the root node)

Node\* curr = root;

// loop till stack is empty

while (!stack.empty())

{

// if the current node exists, print it and push its right child

// to the stack before moving to its left child

if (curr != nullptr)

{

cout << curr->data << " ";

if (curr->right) {

stack.push(curr->right);

}

curr = curr->left;

}

// if the current node is null, pop a node from the stack

// set the current node to the popped node

else {

curr = stack.top();

stack.pop();

}

}

}

int main()

{

/\* Construct the following tree

1

/ \

/ \

2 3

/ / \

/ / \

4 5 6

/ \

/ \

7 8

\*/

Node\* root = new Node(1);

root->left = new Node(2);

root->right = new Node(3);

root->left->left = new Node(4);

root->right->left = new Node(5);

root->right->right = new Node(6);

root->right->left->left = new Node(7);

root->right->left->right = new Node(8);

preorderIterative(root);

return 0;

}

The time complexity of the above solutions is O(n), where n is the total number of nodes in the binary tree. The space complexity of the program is O(n) as the space required is proportional to the tree’s height, which can be equal to the total number of nodes in the tree in the worst case for skewed trees.

**My Implementation:**

vector<int> preorderTraversal(TreeNode\* root) {

vector<int> res;

stack<TreeNode\*> st;

while(!st.empty() || root){

if(root){

res.push\_back(root->val);

st.push(root);

root = root->left;

}

else{

root = st.top()->right;

st.pop();

}

}

return res;

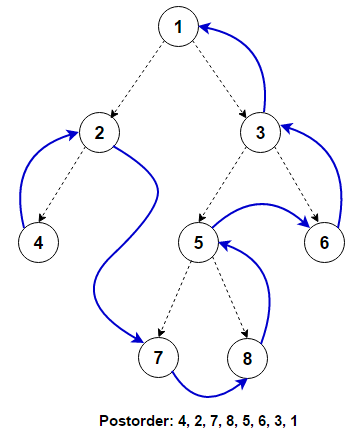
}

# 169. Postorder Traversal of a tree both using recursion and Iteration

For traversing a (non-empty) binary tree in a postorder fashion, we must do these three things for every node n starting from the tree’s root:

**(L)** Recursively traverse its left subtree. When this step is finished, we are back at n again.  
**(R)** Recursively traverse its right subtree. When this step is finished, we are back at n again.  
**(N)** Process n itself.

In normal postorder traversal, visit the left subtree before the right subtree. If we visit the right subtree before visiting the left subtree, it is referred to as reverse postorder traversal.



## Recursive Implementation

As we can see, before processing any node, the left subtree is processed first, followed by the right subtree, and the node is processed at last. These operations can be defined recursively for each node. The recursive implementation is referred to as a [Depth–first search (DFS)](https://www.techiedelight.com/depth-first-search/), as the search tree is deepened as much as possible on each child before going to the next sibling.

Following is the C++ program that demonstrates it:

#include <iostream>

using namespace std;

// Data structure to store a binary tree node

struct Node

{

int data;

Node \*left, \*right;

Node(int data)

{

this->data = data;

this->left = this->right = nullptr;

}

};

// Recursive function to perform postorder traversal on the tree

void postorder(Node\* root)

{

// if the current node is empty

if (root == nullptr) {

return;

}

// Traverse the left subtree

postorder(root->left);

// Traverse the right subtree

postorder(root->right);

// Display the data part of the root (or current node)

cout << root->data << " ";

}

int main()

{

/\* Construct the following tree

1

/ \

/ \

2 3

/ / \

/ / \

4 5 6

/ \

/ \

7 8

\*/

Node\* root = new Node(1);

root->left = new Node(2);

root->right = new Node(3);

root->left->left = new Node(4);

root->right->left = new Node(5);

root->right->right = new Node(6);

root->right->left->left = new Node(7);

root->right->left->right = new Node(8);

postorder(root);

return 0;

}

## Iterative Implementation

To convert the above recursive procedure into an iterative one, we need an explicit stack. Following is a simple stack-based iterative algorithm to perform postorder traversal:

**iterativePostorder(node)**  
   
s —> empty stack  
t —> output stack  
while (not s.isEmpty())  
  node —> s.pop()  
  t.push(node)  
   
  if (node.left <> null)  
    s.push(node.left)  
   
  if (node.right <> null)  
    s.push(node.right)  
   
while (not t.isEmpty())  
  node —> t.pop()  
  visit(node)

The algorithm can be implemented as follows in C++:

#include <iostream>

#include <stack>

using namespace std;

// Data structure to store a binary tree node

struct Node

{

int data;

Node \*left, \*right;

Node(int data)

{

this->data = data;

this->left = this->right = nullptr;

}

};

// Iterative function to perform postorder traversal on the tree

void postorderIterative(Node\* root)

{

// return if the tree is empty

if (root == nullptr) {

return;

}

// create an empty stack and push the root node

stack<Node\*> s;

s.push(root);

// create another stack to store postorder traversal

stack<int> out;

// loop till stack is empty

while (!s.empty())

{

// pop a node from the stack and push the data into the output stack

Node\* curr = s.top();

s.pop();

out.push(curr->data);

// push the left and right child of the popped node into the stack

if (curr->left) {

s.push(curr->left);

}

if (curr->right) {

s.push(curr->right);

}

}

// print postorder traversal

while (!out.empty())

{

cout << out.top() << " ";

out.pop();

}

}

int main()

{

/\* Construct the following tree

1

/ \

/ \

2 3

/ / \

/ / \

4 5 6

/ \

/ \

7 8

\*/

Node\* root = new Node(1);

root->left = new Node(2);

root->right = new Node(3);

root->left->left = new Node(4);

root->right->left = new Node(5);

root->right->right = new Node(6);

root->right->left->left = new Node(7);

root->right->left->right = new Node(8);

postorderIterative(root);

return 0;

}

The time complexity of the above solutions is O(n), where n is the total number of nodes in the binary tree. The space complexity of the program is O(n) as the space required is proportional to the height of the tree, which can be equal to the total number of nodes in the tree in worst-case for skewed trees.

# 170. Left View of a 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. The task is to complete the function **leftView()**, which accepts root of the tree as argument.

Left view of following tree is 1 2 4 8.

          1  
       /     \  
     2        3  
   /     \    /    \  
  4     5   6    7  
   \  
     8

**Example 1:**

**Input:**

  1

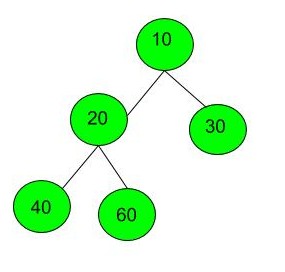
 /  \

3    2

**Output:** 1 3

**Example 2:**

**Input:**



**Output:** 10 20 40

**Your Task:**  
You just have to **complete**the function **leftView()**that prints the left view. The newline is automatically appended by the driver code.  
**Expected Time Complexity:**O(N).  
**Expected Auxiliary Space:**O(Height of the Tree).

**Constraints:**  
0 <= Number of nodes <= 100  
1 <= Data of a node <= 1000

## Solution:

Method-1 (Using Recursion)  
The left view contains all nodes that are first nodes in their levels. A simple solution is to **do**[**level order traversal**](https://www.geeksforgeeks.org/level-order-tree-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).

Below is the implementation of the above idea-

// C++ program to print left view of 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

struct Node \*newNode(int item)

{

struct Node \*temp = (struct Node \*)malloc(

sizeof(struct Node));

temp->data = item;

temp->left = temp->right = NULL;

return temp;

}

// Recursive function to print

// left view of a binary tree.

void leftViewUtil(struct Node \*root,

int level, int \*max\_level)

{

// Base Case

if (root == NULL) return;

// If this is the first Node of its level

if (\*max\_level < level)

{

cout << root->data << " ";

\*max\_level = level;

}

// Recur for left subtree first,

// then right subtree

leftViewUtil(root->left, level + 1, max\_level);

leftViewUtil(root->right, level + 1, max\_level);

}

// A wrapper over leftViewUtil()

void leftView(struct Node \*root)

{

int max\_level = 0;

leftViewUtil(root, 1, &max\_level);

}

// Driver Code

int main()

{

Node\* root = newNode(10);

root->left = newNode(2);

root->right = newNode(3);

root->left->left = newNode(7);

root->left->right = newNode(8);

root->right->right = newNode(15);

root->right->left = newNode(12);

root->right->right->left = newNode(14);

leftView(root);

return 0;

}

**Output**

10 2 7 14

**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.

**Method-2** (Using Queue):

In this method, level order traversal based solution is discussed. If we observe carefully, we will see that our main task is to print the left most node of every level. So, we will do a level order traversal on the tree and print the leftmost node at every level. Below is the implementation of above approach:

// C++ program to print left view of

// Binary Tree

#include<bits/stdc++.h>

using namespace std;

// A Binary Tree Node

struct Node

{

int data;

struct Node \*left, \*right;

};

// Utility function to create a new tree node

Node\* newNode(int data)

{

Node \*temp = new Node;

temp->data = data;

temp->left = temp->right = NULL;

return temp;

}

// function to print left view of

// binary tree

void printLeftView(Node\* root)

{

if (!root)

return;

queue<Node\*> q;

q.push(root);

while (!q.empty())

{

// number of nodes at current level

int n = q.size();

// Traverse all nodes of current level

for(int i = 1; i <= n; i++)

{

Node\* temp = q.front();

q.pop();

// Print the left most element

// at the level

if (i == 1)

cout<<temp->data<<" ";

// Add left node to queue

if (temp->left != NULL)

q.push(temp->left);

// Add right node to queue

if (temp->right != NULL)

q.push(temp->right);

}

}

}

// Driver code

int main()

{

// Let's construct the tree as

// shown in example

Node\* root = newNode(10);

root->left = newNode(2);

root->right = newNode(3);

root->left->left = newNode(7);

root->left->right = newNode(8);

root->right->right = newNode(15);

root->right->left = newNode(12);

root->right->right->left = newNode(14);

printLeftView(root);

}

**Output**

10 2 7 14

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

**My Approach:**

vector<int> leftView(Node \*root)

{

queue<Node\*> q;

Node \*curr = root;

Node \*dummy = (Node\*)malloc(sizeof(struct Node));

dummy->data = -1; dummy->left = dummy->right = NULL;

vector<int> res;

if(!root)

return res;

res.push\_back(root->data);

q.push(root);

q.push(dummy);

while(!q.empty()){

if(q.front()==dummy){

q.pop();

if(!q.empty()){

res.push\_back(q.front()->data);

q.push(dummy);

}

}

else{

if(q.front()->left)

q.push(q.front()->left);

if(q.front()->right)

q.push(q.front()->right);

q.pop();

}

}

return res;

}

Time Complexity:O(N) where N is the total number of nodes

# 171. Right View of Tree

Given a Binary Tree, find **Right view** of it. Right view of a Binary Tree is set of nodes visible when tree is viewed from **right**side.

Right view of following tree is 1 3 7 8.

          1  
       /     \  
     2        3  
   /   \      /    \  
  4     5   6    7  
    \  
     8

**Example 1:**

**Input:**

       1

   /    \

  3      2

**Output:** 1 2

**Example 2:**

**Input:**

     10

   /   \

 20     30

/   \

40  60

**Output:** 10 30 60

**Your Task:**  
Just complete the **function rightView()**that takes **node**as **parameter**and returns the right view as a list.

**Expected Time Complexity:**O(N).  
**Expected Auxiliary Space:**O(Height of the Tree).

**Constraints:**  
1 ≤ Number of nodes ≤ 105  
1 ≤ Data of a node ≤ 105

## Solution:

The problem can be solved using simple recursive traversal. We can keep track of level of a node by passing a parameter to all recursive calls. The idea is to keep track of maximum level also. And traverse the tree in a manner that right subtree is visited before left subtree. Whenever we see a node whose level is more than maximum level so far, we print the node because this is the last node in its level (Note that we traverse the right subtree before left subtree). Following is the implementation of this approach.

// C++ program to print right view of 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

struct Node \*newNode(int item)

{

struct Node \*temp = (struct Node \*)malloc(

sizeof(struct Node));

temp->data = item;

temp->left = temp->right = NULL;

return temp;

}

// Recursive function to print

// right view of a binary tree.

void rightViewUtil(struct Node \*root,

int level, int \*max\_level)

{

// Base Case

if (root == NULL) return;

// If this is the last Node of its level

if (\*max\_level < level)

{

cout << root->data << "\t";

\*max\_level = level;

}

// Recur for right subtree first,

// then left subtree

rightViewUtil(root->right, level + 1, max\_level);

rightViewUtil(root->left, level + 1, max\_level);

}

// A wrapper over rightViewUtil()

void rightView(struct Node \*root)

{

int max\_level = 0;

rightViewUtil(root, 1, &max\_level);

}

// Driver Code

int main()

{

struct Node \*root = newNode(1);

root->left = newNode(2);

root->right = newNode(3);

root->left->left = newNode(4);

root->left->right = newNode(5);

root->right->left = newNode(6);

root->right->right = newNode(7);

root->right->right->right = newNode(8);

rightView(root);

return 0;

}

**Output**

1 3 7 8

**Time Complexity:** The function does a simple traversal of the tree, so the complexity is O(n).

**Method 2:**In this method, [level order traversal based](https://www.geeksforgeeks.org/level-order-tree-traversal/) solution is discussed. If we observe carefully, we will see that our main task is to print the right most node of every level. So, we will do a level order traversal on the tree and print the last node at every level. Below is the implementation of above approach:

// C++ program to print left view of

// Binary Tree

#include <bits/stdc++.h>

using namespace std;

// A Binary Tree Node

struct Node {

int data;

struct Node \*left, \*right;

};

// Utility function to create a new tree node

Node\* newNode(int data)

{

Node\* temp = new Node;

temp->data = data;

temp->left = temp->right = NULL;

return temp;

}

// function to print Right view of

// binary tree

void printRightView(Node\* root)

{

if (root == NULL)

return;

queue<Node\*> q;

q.push(root);

while (!q.empty()) {

// get number of nodes for each level

int n = q.size();

// traverse all the nodes of the current level

while (n--) {

Node\* x = q.front();

q.pop();

// print the last node of each level

if (n == 0) {

cout << x->data << " ";

}

// if left child is not null push it into the

// queue

if (x->left)

q.push(x->left);

// if right child is not null push it into the

// queue

if (x->right)

q.push(x->right);

}

}

}

// Driver code

int main()

{

// Let's construct the tree as

// shown in 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);

printRightView(root);

}

**Output**

1 3 7 8

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

**My approach:**

vector<int> rightView(Node \*root)

{

queue<Node\*> q;

Node \*curr = root;

Node \*dummy = (Node\*)malloc(sizeof(struct Node));

vector<int> res;

if(!root)

return res;

q.push(root);

q.push(dummy);

while(!q.empty()){

if(q.front()==dummy){

q.pop();

res.push\_back(curr->data);

if(!q.empty()){

q.push(dummy);

}

}

else{

if(q.front()->left)

q.push(q.front()->left);

if(q.front()->right)

q.push(q.front()->right);

curr = q.front();

q.pop();

}

}

return res;

}

# 172. Top View of a tree

Given below is a binary tree. The task is to print the 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. For the given below tree

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

Top view will be: 4 2 1 3 7  
**Note:**Return nodes from **leftmost**node to **rightmost**node.

**Example 1:**

**Input:**

  1

 /    \

2      3

**Output:** 2 1 3

**Example 2:**

**Input:**

  10

   /      \

20        30

/   \    /    \

40   60 90    100

**Output:** 40 20 10 30 100

**Your Task:**  
Since this is a function problem. You don't have to take input. Just complete the function**topView()**that takes **root node**as parameter and returns a list of nodes visible from the top view from left to right.

**Expected Time Complexity:**O(N)  
**Expected Auxiliary Space:**O(N).

**Constraints:**  
1 ≤ N ≤ 105  
1 ≤ Node Data ≤ 105

## Solution:

The idea is to do something similar to [vertical Order Traversal](https://www.geeksforgeeks.org/print-binary-tree-vertical-order-set-2/). Like [vertical Order Traversal](https://www.geeksforgeeks.org/print-binary-tree-vertical-order-set-2/), 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;

}

**Output**

Following are nodes in top view of Binary Tree

The top view of the tree is :

2 1 3 6

**Another approach:**  
This approach does not require a queue. Here we use the two variables, one for horizontal distance of current node from the root and another for the depth of the current node from the root. We use the horizontal distance for indexing. If one node with the same horizontal 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 horizontal distance from root.

map<int, pair<int, int> > m;

// fillmap(root,horizontal\_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;

}

**Output**

Following are nodes in top view of Binary Tree

2 1 3 6

Time Complexity of the above implementation is O(nlogn) where n is the number of nodes in the given binary tree with each insertion operation in Map requiring O(log2n) complexity.

**Another approach:**

1. This approach is based on the level order traversal. We’ll keep record of current max so far left, right horizontal distances from the root.
2. And if we found less distance (or greater in magnitude) then max left so far distance then update it and also push data on this node to a stack (stack is used because in level order traversal the left nodes will appear in reverse order), or if we found greater distance then max right so far distance then update it and also push data on this node to a vector.
3. In this approach, no map is used.

// C++ Program to print Top View of a binary Tree

#include <iostream>

#include <queue>

#include <stack>

using namespace std;

// class for Tree node

class Node {

public:

Node \*left, \*right;

int data;

Node() { left = right = 0; }

Node(int data)

{

left = right = 0;

this->data = data;

}

};

/\*

1

/ \

2 3

\

4

\

5

\

6

Top view of the above binary tree is

2 1 3 6

\*/

// class for Tree

class Tree {

public:

Node\* root;

Tree() { root = 0; }

void topView()

{

// queue for holding nodes and their horizontal

// distance from the root node

queue<pair<Node\*, int> > q;

// pushing root node with distance 0

q.push(make\_pair(root, 0));

// hd is currect node's horizontal distance from

// root node l is currect left min horizontal

// distance (or max in magnitude) so far from the

// root node r is currect right max horizontal

// distance so far from the root node

int hd = 0, l = 0, r = 0;

// stack is for holding left node's data because

// they will appear in reverse order that is why

// using stack

stack<int> left;

// vector is for holding right node's data

vector<int> right;

Node\* node;

while (q.size()) {

node = q.front().first;

hd = q.front().second;

if (hd < l) {

left.push(node->data);

l = hd;

}

else if (hd > r) {

right.push\_back(node->data);

r = hd;

}

if (node->left) {

q.push(make\_pair(node->left, hd - 1));

}

if (node->right) {

q.push(make\_pair(node->right, hd + 1));

}

q.pop();

}

// printing the left node's data in reverse order

while (left.size()) {

cout << left.top() << " ";

left.pop();

}

// then printing the root node's data

cout << root->data << " ";

// finally printing the right node's data

for (auto x : right) {

cout << x << " ";

}

}

};

// Driver code

int main()

{

// Tree object

Tree t;

t.root = new Node(1);

t.root->left = new Node(2);

t.root->right = new Node(3);

t.root->left->right = new Node(4);

t.root->left->right->right = new Node(5);

t.root->left->right->right->right = new Node(6);

t.topView();

cout << endl;

return 0;

}

**Output**

2 1 3 6

As no Map is used, Time Complexity of the above implementation is O(n) where n is the number of nodes in the given binary tree.

# 173. Bottom View of a tree

Given a binary tree, print the bottom view from left to right.  
A node is included in bottom view if it can be seen when we look at the tree from bottom.

                      20  
                    /    \  
                  8       22  
                /   \        \  
              5      3       25  
                    /   \        
                  10    14

For the above tree, the bottom view is 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 bottommost nodes at horizontal distance 0, we need to print 4.

                      20  
                    /    \  
                  8       22  
                /   \     /   \  
              5      3 4     25  
                     /    \        
                 10       14

For the above tree the output should be 5 10 4 14 25.

**Example 1:**

**Input:**

1

  / \

  3 2

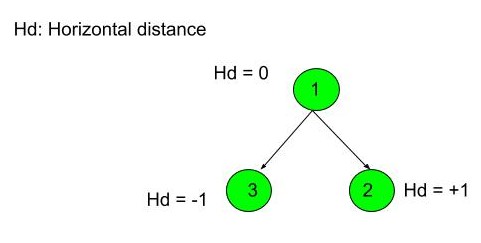
**Output:** 3 1 2

**Explanation:**

First case represents a tree with 3 nodes

and 2 edges where root is 1, left child of

1 is 3 and right child of 1 is 2.



Thus nodes of the binary tree will be

printed as such 3 1 2.

**Example 2:**

**Input:**

10

  / \

  20 30

  / \

  40 60

**Output:** 40 20 60 30

**Your Task:**  
This is a functional problem, you **don't**need to care about input, just complete the function **bottomView**() which takes the root node of the tree as input and returns an array containing the bottom view of the given tree.

**Expected Time Complexity:**O(N).  
**Expected Auxiliary Space:**O(N).

**Constraints:**  
1 <= Number of nodes <= 105  
1 <= Data of a node <= 105

## Solution:

**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.

Below is the implementation of the above:

// 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()**

**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;

}

**My approach:**

class Solution {

public:

int height(Node\* root){

if(!root)

return 0;

return max(height(root->left),height(root->right))+1;

}

vector <int> bottomView(Node \*root) {

vector<int> res;

if(!root)

return res;

Node\* dummy = (Node\*)malloc(sizeof(Node));

dummy->data = -1; dummy->left = NULL; dummy->right = NULL;

int h = height(root);

vector<Node\*> mp(2\*h+1, dummy);

queue<pair<Node\*, int>> q;

int l=0, r=0;

q.push(make\_pair(root,h));

while(!q.empty()){

Node\* nd = q.front().first;

int t = q.front().second;

q.pop();

mp[t] = nd;

if(nd->left)

q.push(make\_pair(nd->left,t-1));

if(nd->right)

q.push(make\_pair(nd->right,t+1));

}

for(auto i:mp){

if(i->data!=-1)

res.push\_back(i->data);

}

return res;

}

};

**Time Complexity:** O(n)

**Space Complexity:** O(n)

# 174. Zig-Zag traversal of a binary tree

Given a Binary Tree. Find the Zig-Zag Level Order Traversal of the Binary Tree.

**Example 1:**

**Input:**

        3

    /   \

2    1

**Output:**

3 1 2

**Example 2:**

**Input:**

           7

       /     \

      9       7

    /  \  /

   8    8   6

  /  \

  10   9

**Output:**

7 7 9 8 8 6 9 10

**Your Task:**  
You don't need to read input or print anything. Your task is to complete the function **zigZagTraversal()** which takes the root node of the Binary Tree as its input and returns a list containing the node values as they appear in the Zig-Zag Level-Order Traversal of the Tree.

**Expected Time Complexity:** O(N).  
**Expected Auxiliary Space:** O(N).

**Constraints:**  
1 <= N <= 104

## Solution:

This problem can be solved using two stacks. Assume the two stacks are current: **currentlevel and nextlevel.** We would also need a variable to keep track of the current level order(whether it is left to right or right to left). We pop from the currentlevel stack and print the nodes value. Whenever the current level order is from left to right, push the nodes left child, then its right child to the stack nextlevel. Since a stack is a LIFO(Last-In-First\_out) structure, next time when nodes are popped off nextlevel, it will be in the reverse order. On the other hand, when the current level order is from right to left, we would push the nodes right child first, then its left child. Finally, do-not forget to swap those two stacks at the end of each level(i.e., when current level is empty)   
*Below is the implementation of the above approach:*

// C++ implementation of a O(n) time method for

// Zigzag order traversal

#include <iostream>

#include <stack>

using namespace std;

// Binary Tree node

struct Node {

int data;

struct Node \*left, \*right;

};

// function to print the zigzag traversal

void zizagtraversal(struct Node\* root)

{

// if null then return

if (!root)

return;

// declare two stacks

stack<struct Node\*> currentlevel;

stack<struct Node\*> nextlevel;

// push the root

currentlevel.push(root);

// check if stack is empty

bool lefttoright = true;

while (!currentlevel.empty()) {

// pop out of stack

struct Node\* temp = currentlevel.top();

currentlevel.pop();

// if not null

if (temp) {

// print the data in it

cout << temp->data << " ";

// store data according to current

// order.

if (lefttoright) {

if (temp->left)

nextlevel.push(temp->left);

if (temp->right)

nextlevel.push(temp->right);

}

else {

if (temp->right)

nextlevel.push(temp->right);

if (temp->left)

nextlevel.push(temp->left);

}

}

if (currentlevel.empty()) {

lefttoright = !lefttoright;

swap(currentlevel, nextlevel);

}

}

}

// A utility function to create a new node

struct Node\* newNode(int data)

{

struct Node\* node = new struct Node;

node->data = data;

node->left = node->right = NULL;

return (node);

}

// driver program to test the above function

int main()

{

// create tree

struct Node\* root = newNode(1);

root->left = newNode(2);

root->right = newNode(3);

root->left->left = newNode(7);

root->left->right = newNode(6);

root->right->left = newNode(5);

root->right->right = newNode(4);

cout << "ZigZag Order traversal of binary tree is \n";

zizagtraversal(root);

return 0;

}

**Output**

ZigZag Order traversal of binary tree is

1 3 2 7 6 5 4

**Time Complexity:** O(n)   
**Space Complexity:** O(n)+(n)=O(n)

**Recursive Approach**:

The approach used here is the observable similarity to the level order traversal. Here we need to include an extra parameter to keep a track of printing each level in left-right or right-left way.

//Initial Template for C++

#include <bits/stdc++.h>

using namespace std;

struct Node {

int data;

Node \*left;

Node \*right;

Node(int val) {

data = val;

left = right = NULL;

}

};

// Function to Build Tree

Node\* buildTree(string str)

{

// Corner Case

if(str.length() == 0 || str[0] == 'N')

return NULL;

// Creating vector of strings from input

// string after splitting by space

vector<string> ip;

istringstream iss(str);

for(string str; iss >> str; )

ip.push\_back(str);

// Create the root of the tree

Node\* root = new Node(stoi(ip[0]));

// Push the root to the queue

queue<Node\*> queue;

queue.push(root);

// Starting from the second element

int i = 1;

while(!queue.empty() && i < ip.size()) {

// Get and remove the front of the queue

Node\* currNode = queue.front();

queue.pop();

// Get the current node's value from the string

string currVal = ip[i];

// If the left child is not null

if(currVal != "N") {

// Create the left child for the current node

currNode->left = new Node(stoi(currVal));

// Push it to the queue

queue.push(currNode->left);

}

// For the right child

i++;

if(i >= ip.size())

break;

currVal = ip[i];

// If the right child is not null

if(currVal != "N") {

// Create the right child for the current node

currNode->right = new Node(stoi(currVal));

// Push it to the queue

queue.push(currNode->right);

}

i++;

}

return root;

}

// Function to calculate height of tree

int treeHeight(Node \*root){

if(!root) return 0;

int lHeight = treeHeight(root->left);

int rHeight = treeHeight(root->right);

return max(lHeight, rHeight) + 1;

}

// Helper Function to store the zig zag order traversal

// of tree in a list recursively

void zigZagTraversalRecursion(Node\* root, int height, bool lor, vector<int> &ans){

// Height = 1 means the tree now has only one node

if(height <= 1){

if(root) ans.push\_back(root->data);

}

// When Height > 1

else{

if(lor){

if(root->left) zigZagTraversalRecursion(root->left, height - 1, lor, ans);

if(root->right) zigZagTraversalRecursion(root->right, height - 1, lor, ans);

}

else{

if(root->right) zigZagTraversalRecursion(root->right, height - 1, lor, ans);

if(root->left) zigZagTraversalRecursion(root->left, height - 1, lor, ans);

}

}

}

// Function to traverse tree in zig zag order

vector <int> zigZagTraversal(Node\* root)

{

vector<int> ans;

bool leftOrRight = true;

int height = treeHeight(root);

for(int i = 1; i <= height; i++){

zigZagTraversalRecursion(root, i, leftOrRight, ans);

leftOrRight = !leftOrRight;

}

return ans;

}

int main()

{

// Tree:

// 1

// / \

// / \

// / \

// 2 3

// / \ / \

// 4 5 6 7

// / \ / \ / \ / \

//8 9 10 11 12 13 14 15

string s = "1 2 3 4 5 6 7 8 9 10 11 12 13 14 15";

Node\* root = buildTree(s);

vector <int> res = zigZagTraversal(root);

cout<<"ZigZag traversal of binary tree is:"<<endl;

for (int i = 0; i < res.size (); i++) cout << res[i] << " ";

cout<<endl;

return 0;

}

Output:

*ZigZag traversal of binary tree is:*

*1 3 2 4 5 6 7 15 14 13 12 11 10 9 8*

**Another Approach:**  
In this approach, use a deque to solve the problem. Push and pop in different ways depending on if the level is odd or level is even.

Below is the implementation of the above approach:

// C++ implementation of a O(n) time method for

// Zigzag order traversal

#include <bits/stdc++.h>

#include <iostream>

using namespace std;

// Binary Tree node

class Node {

public:

int data;

Node \*left, \*right;

};

// Function to print the zigzag traversal

vector<int> zigZagTraversal(Node\* root)

{

deque<Node\*> q;

vector<int> v;

q.push\_back(root);

v.push\_back(root->data);

Node\* temp;

// set initial level as 1, because root is

// already been taken care of.

int l = 1;

while (!q.empty()) {

int n = q.size();

for (int i = 0; i < n; i++) {

// popping mechanism

if (l % 2 == 0) {

temp = q.back();

q.pop\_back();

}

else {

temp = q.front();

q.pop\_front();

}

// pushing mechanism

if (l % 2 != 0) {

if (temp->right) {

q.push\_back(temp->right);

v.push\_back(temp->right->data);

}

if (temp->left) {

q.push\_back(temp->left);

v.push\_back(temp->left->data);

}

}

else if (l % 2 == 0) {

if (temp->left) {

q.push\_front(temp->left);

v.push\_back(temp->left->data);

}

if (temp->right) {

q.push\_front(temp->right);

v.push\_back(temp->right->data);

}

}

}

l++; // level plus one

}

return v;

}

// A utility function to create a new node

struct Node\* newNode(int data)

{

struct Node\* node = new struct Node;

node->data = data;

node->left = node->right = NULL;

return (node);

}

// Driver program to test

// the above function

int main()

{

// vector to store the traversal order.

vector<int> v;

// create tree

struct Node\* root = newNode(1);

root->left = newNode(2);

root->right = newNode(3);

root->left->left = newNode(7);

root->left->right = newNode(6);

root->right->left = newNode(5);

root->right->right = newNode(4);

cout << "ZigZag Order traversal of binary tree is \n";

v = zigZagTraversal(root);

for (int i = 0; i < v.size();

i++) { // to print the order

cout << v[i] << " ";

}

return 0;

}

**Output**

ZigZag Order traversal of binary tree is

1 3 2 7 6 5 4

**My Implementaion:**

vector <int> zigZagTraversal(Node\* root)

{

vector<int> res;

if(!root)

return res;

stack<int> st;

int i=0;

queue<Node\*> q;

q.push(root);

while(!q.empty()){

int n = q.size();

while(n--){

Node\* nd = q.front();

if(nd->left)

q.push(nd->left);

if(nd->right)

q.push(nd->right);

if(i%2==0)

res.push\_back(nd->data);

else

st.push(nd->data);

q.pop();

}

while(!st.empty()){

res.push\_back(st.top());

st.pop();

}

i++;

}

return res;

}

**Time Complexity:** O(n)

**Space Complexity:** O(n)

# 175. Check if a tree is balanced or not

Given a binary tree, find if it is height balanced or not.   
A tree is height balanced if difference between heights of left and right subtrees is **not more than one** for all nodes of tree.

**A height balanced tree**  
        1  
     /     \  
   10      39  
  /  
5

**An unbalanced tree**  
        1  
     /      
   10     
  /  
5

**Example 1:**

**Input:**

      1

   /

   2

   \

    3

**Output:** 0

**Explanation:** The max difference in height

of left subtree and right subtree is 2,

which is greater than 1. Hence unbalanced

**Example 2:**

**Input:**

       10

    /   \

   20   30

  /   \

40   60

**Output:** 1

**Explanation:** The max difference in height

of left subtree and right subtree is 1.

Hence balanced.

**Your Task:**  
You don't need to take input. Just complete the function**isBalanced()**that takes root **node**as parameter and returns **true,**if the tree is balanced else returns **false**.

**Constraints:**  
1 <= Number of nodes <= 105  
0 <= Data of a node <= 106

**Expected time complexity:**O(N)  
**Expected auxiliary space:**O(h) , where h = height of tree

## Solution:

An empty tree is height-balanced. A non-empty binary tree T is balanced if:   
1) Left subtree of T is balanced   
2) Right subtree of T is balanced   
3) The difference between heights of left subtree and right subtree is not more than 1.   
The above height-balancing scheme is used in AVL trees.

To check if a tree is height-balanced, get the height of left and right subtrees. Return true if difference between heights is not more than 1 and left and right subtrees are balanced, otherwise return false.   
 /\* CPP program to check if

a tree is height-balanced or not \*/

#include <bits/stdc++.h>

using namespace std;

/\* A binary tree node has data,

pointer to left child and

a pointer to right child \*/

class node {

public:

int data;

node\* left;

node\* right;

};

/\* Returns the height of a binary tree \*/

int height(node\* node);

/\* Returns true if binary tree

with root as root is height-balanced \*/

bool isBalanced(node\* root)

{

int lh; /\* for height of left subtree \*/

int rh; /\* for height of right subtree \*/

/\* If tree is empty then return true \*/

if (root == NULL)

return 1;

/\* Get the height of left and right sub trees \*/

lh = height(root->left);

rh = height(root->right);

if (abs(lh - rh) <= 1 && isBalanced(root->left) && isBalanced(root->right))

return 1;

/\* If we reach here then

tree is not height-balanced \*/

return 0;

}

/\* UTILITY FUNCTIONS TO TEST isBalanced() FUNCTION \*/

/\* returns maximum of two integers \*/

int max(int a, int b)

{

return (a >= b) ? a : b;

}

/\* The function Compute the "height"

of a tree. Height is the number of

nodes along the longest path from

the root node down to the farthest leaf node.\*/

int height(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. \*/

node\* newNode(int data)

{

node\* Node = new node();

Node->data = data;

Node->left = NULL;

Node->right = NULL;

return (Node);

}

// Driver code

int main()

{

node\* root = newNode(1);

root->left = newNode(2);

root->right = newNode(3);

root->left->left = newNode(4);

root->left->right = newNode(5);

root->left->left->left = newNode(8);

if (isBalanced(root))

cout << "Tree is balanced";

else

cout << "Tree is not balanced";

return 0;

}

**Output:** 

Tree is not balanced

**Optimized implementation:** Above implementation can be optimized by calculating the height in the same recursion rather than calling a height() function separately. This optimization reduces time complexity to O(n).

/\* C++ program to check if a tree

is height-balanced or not \*/

#include <bits/stdc++.h>

using namespace std;

#define bool int

/\* 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;

};

/\* The function returns true if root is

balanced else false The second parameter

is to store the height of tree. Initially,

we need to pass a pointer to a location with

value as 0. We can also write a wrapper

over this function \*/

bool isBalanced(node\* root, int\* height)

{

/\* lh --> Height of left subtree

rh --> Height of right subtree \*/

int lh = 0, rh = 0;

/\* l will be true if left subtree is balanced

and r will be true if right subtree is balanced \*/

int l = 0, r = 0;

if (root == NULL) {

\*height = 0;

return 1;

}

/\* Get the heights of left and right subtrees in lh and rh

And store the returned values in l and r \*/

l = isBalanced(root->left, &lh);

r = isBalanced(root->right, &rh);

/\* Height of current node is max of heights of left and

right subtrees plus 1\*/

\*height = (lh > rh ? lh : rh) + 1;

/\* If difference between heights of left and right

subtrees is more than 2 then this node is not balanced

so return 0 \*/

if (abs(lh - rh) >= 2)

return 0;

/\* If this node is balanced and left and right subtrees

are balanced then return true \*/

else

return l && r;

}

/\* UTILITY FUNCTIONS TO TEST isBalanced() FUNCTION \*/

/\* Helper function that allocates a new node with the

given data and NULL left and right pointers. \*/

node\* newNode(int data)

{

node\* Node = new node();

Node->data = data;

Node->left = NULL;

Node->right = NULL;

return (Node);

}

// Driver code

int main()

{

int height = 0;

/\* Constructed binary tree is

1

/ \

2 3

/ \ /

4 5 6

/

7

\*/

node\* root = newNode(1);

root->left = newNode(2);

root->right = newNode(3);

root->left->left = newNode(4);

root->left->right = newNode(5);

root->right->left = newNode(6);

root->left->left->left = newNode(7);

if (isBalanced(root, &height))

cout << "Tree is balanced";

else

cout << "Tree is not balanced";

return 0;

}

**Output** 

Tree is balanced

**Time Complexity:** O(n)

# 176. Diagnol Traversal of a Binary tree

Given a Binary Tree, print the **diagonal traversal** of the 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.

**Example 1:**

**Input** :

  8

  / \

  3 10

  / \ \

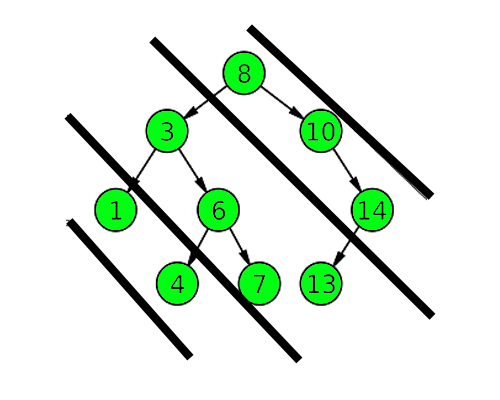
  1 6 14

  / \ /

  4 7 13

**Output** : 8 10 14 3 6 7 13 1 4

**Explanation**:

[[](http://d1hyf4ir1gqw6c.cloudfront.net/wp-content/uploads/unnamed1.png)](http://d1hyf4ir1gqw6c.cloudfront.net/wp-content/uploads/unnamed1.png" \t "_blank)

Diagonal Traversal of binary tree :

8 10 14 3 6 7 13 1 4

**Your Task:**  
You don't need to read input or print anything. The task is to complete the function**diagonal()**that takes the root nodeas input argumetsand returns the diagonal traversal of the given tree.

**Expected Time Complexity:**O(N).  
**Expected Auxiliary Space:**O(N).  
Here N is number of nodes.

**Constraints:**  
1 <= Number of nodes<= 105  
1 <= Data of a node<= 105

## Solution:

The idea is to use a map. We use different slope distances and use them as key in the map. Value in the map is a vector (or dynamic array) of nodes. We traverse the tree to store values in the map. Once map is built, we print the contents of it.

Below is the implementation of the above idea.

// C++ program for diagonal

// traversal of Binary Tree

#include <bits/stdc++.h>

using namespace std;

// Tree node

struct Node

{

int data;

Node \*left, \*right;

};

/\* root - root of the binary tree

d - distance of current line from rightmost

-topmost slope.

diagonalPrint - multimap to store Diagonal

elements (Passed by Reference) \*/

void diagonalPrintUtil(Node\* root, int d,

map<int, vector<int>> &diagonalPrint)

{

// Base case

if (!root)

return;

// Store all nodes of same

// line together as a vector

diagonalPrint[d].push\_back(root->data);

// Increase the vertical

// distance if left child

diagonalPrintUtil(root->left,

d + 1, diagonalPrint);

// Vertical distance remains

// same for right child

diagonalPrintUtil(root->right,

d, diagonalPrint);

}

// Print diagonal traversal

// of given binary tree

void diagonalPrint(Node\* root)

{

// create a map of vectors

// to store Diagonal elements

map<int, vector<int> > diagonalPrint;

diagonalPrintUtil(root, 0, diagonalPrint);

cout << "Diagonal Traversal of binary tree : \n";

for (auto it :diagonalPrint)

{

vector<int> v=it.second;

for(auto it:v)

cout<<it<<" ";

cout<<endl;

}

}

// 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

The **time complexity** of this solution is **O( N logN )** and the space complexity is **O( N )**

We can solve the same problem using queue and iterative algorithm.

#include <bits/stdc++.h>

using namespace std;

// Tree node

struct Node {

int data;

Node \*left, \*right;

};

vector<int> diagonal(Node\* root)

{

vector<int> diagonalVals;

if (!root)

return diagonalVals;

// The leftQueue will be a queue which will store all

// left pointers while traversing the tree, and will be

// utilized when at any point right pointer becomes NULL

queue<Node\*> leftQueue;

Node\* node = root;

while (node) {

// Add current node to output

diagonalVals.push\_back(node->data);

// If left child available, add it to queue

if (node->left)

leftQueue.push(node->left);

// if right child, transfer the node to right

if (node->right)

node = node->right;

else {

// If left child Queue is not empty, utilize it

// to traverse further

if (!leftQueue.empty()) {

node = leftQueue.front();

leftQueue.pop();

}

else {

// All the right childs traversed and no

// left child left

node = NULL;

}

}

}

return diagonalVals;

}

// 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);\*/

vector<int> diagonalValues = diagonal(root);

for (int i = 0; i < diagonalValues.size(); i++) {

cout << diagonalValues[i] << " ";

}

cout << endl;

return 0;

}

**Output**

[8, 10, 14, 3, 6, 7, 13, 1, 4]

The **time complexity** of this solution is **O( N logN )** and the space complexity is **O( N )**

**Approach 2 : Using Queue.**

Every node will help to generate the next diagonal. We will push the element in the queue only when its left is available. We will process the node and move to its right.

Code:

#include <bits/stdc++.h>

using namespace std;

struct Node

{

int data;

Node \*left, \*right;

};

Node\* newNode(int data)

{

Node\* node = new Node;

node->data = data;

node->left = node->right = NULL;

return node;

}

vector <vector <int>> result;

void diagonalPrint(Node\* root)

{

if(root == NULL)

return;

queue <Node\*> q;

q.push(root);

while(!q.empty())

{

int size = q.size();

vector <int> answer;

while(size--)

{

Node\* temp = q.front();

q.pop();

// traversing each component;

while(temp)

{

answer.push\_back(temp->data);

if(temp->left)

q.push(temp->left);

temp = temp->right;

}

}

result.push\_back(answer);

}

}

int main()

{

Node\* root = newNode(8);

root->left = newNode(3);

root->right = newNode(10);

root->left->left = newNode(1);

root->left->right = newNode(6);

root->right->right = newNode(14);

root->right->right->left = newNode(13);

root->left->right->left = newNode(4);

root->left->right->right = newNode(7);

diagonalPrint(root);

for(int i=0 ; i<result.size() ; i++)

{

for(int j=0 ; j<result[i].size() ; j++)

cout<<result[i][j]<<" ";

cout<<endl;

}

return 0;

}

**Output**

8 10 14

3 6 7 13

1 4

**Time Complexity: O(N)**, because we are visiting nodes once.

**Space Complexity: O(N)**, because we are using queue.

**My code:**

vector<int> diagonal(Node \*root)

{

vector<int> res;

if(!root)

return res;

queue<Node\*> q;

q.push(root);

while(!q.empty()){

Node\* nd = q.front();

q.pop();

while(nd){

res.push\_back(nd->data);

if(nd->left)

q.push(nd->left);

nd = nd->right;

}

}

return res;

}

# 178. [Construct Binary Tree from String with Bracket Representation](https://www.geeksforgeeks.org/construct-binary-tree-string-bracket-representation/)

Construct a binary tree from a string consisting of parenthesis and integers. The whole input represents a binary tree. It contains an integer followed by zero, one or two pairs of parenthesis. The integer represents the root’s value and a pair of parenthesis contains a child binary tree with the same structure. Always start to construct the left child node of the parent first if it exists.

**Examples:**

Input : "1(2)(3)"

Output : 1 2 3

Explanation :

1

/ \

2 3

Explanation: first pair of parenthesis contains

left subtree and second one contains the right

subtree. Preorder of above tree is "1 2 3".

Input : "4(2(3)(1))(6(5))"

Output : 4 2 3 1 6 5

Explanation :

4

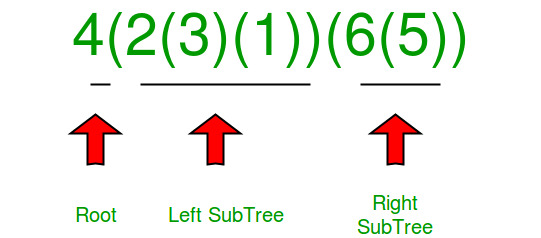
/ \

2 6

/ \ /

3 1 5

We know first character in string is root. Substring inside the first adjacent pair of parenthesis is for left subtree and substring inside second pair of parenthesis is for right subtree as in the below diagram.



## Solution:

We need to find the substring corresponding to left subtree and substring corresponding to right subtree and then recursively call on both of the substrings.

For this first find the index of starting index and end index of each substring.   
To find the index of closing parenthesis of left subtree substring, use a stack. Let the found index be stored in index variable.

/\* C++ program to construct a binary tree from

the given string \*/

#include <bits/stdc++.h>

using namespace std;

/\* A binary tree node has data, pointer to left

child and a pointer to right child \*/

struct Node {

int data;

Node \*left, \*right;

};

/\* Helper function that allocates a new node \*/

Node\* newNode(int data)

{

Node\* node = (Node\*)malloc(sizeof(Node));

node->data = data;

node->left = node->right = NULL;

return (node);

}

/\* This function is here just to test \*/

void preOrder(Node\* node)

{

if (node == NULL)

return;

printf("%d ", node->data);

preOrder(node->left);

preOrder(node->right);

}

// function to return the index of close parenthesis

int findIndex(string str, int si, int ei)

{

if (si > ei)

return -1;

// Inbuilt stack

stack<char> s;

for (int i = si; i <= ei; i++) {

// if open parenthesis, push it

if (str[i] == '(')

s.push(str[i]);

// if close parenthesis

else if (str[i] == ')') {

if (s.top() == '(') {

s.pop();

// if stack is empty, this is

// the required index

if (s.empty())

return i;

}

}

}

// if not found return -1

return -1;

}

// function to construct tree from string

Node\* treeFromString(string str, int si, int ei)

{

// Base case

if (si > ei)

return NULL;

// new root

Node\* root = newNode(str[si] - '0');

int index = -1;

// if next char is '(' find the index of

// its complement ')'

if (si + 1 <= ei && str[si + 1] == '(')

index = findIndex(str, si + 1, ei);

// if index found

if (index != -1) {

// call for left subtree

root->left = treeFromString(str, si + 2, index - 1);

// call for right subtree

root->right

= treeFromString(str, index + 2, ei - 1);

}

return root;

}

// Driver Code

int main()

{

string str = "4(2(3)(1))(6(5))";

Node\* root = treeFromString(str, 0, str.length() - 1);

preOrder(root);

}

**Output**

4 2 3 1 6 5

**Time Complexity:** O(N2)  
**Auxiliary Space:** O(N)

**Another recursive approach:**

Algorithm:

1. The very first element of the string is the root.
2. If the next two consecutive elements are “(” and “)”, this means there is no left child otherwise we will create and add the left child to the parent node recursively.
3. Once the left child is added recursively, we will look for consecutive “(” and add the right child to the parent node.
4. Encountering “)” means the end of either left or right node and we will increment the start index
5. The recursion ends when the start index is greater than equal to the end index

#include <bits/stdc++.h>

using namespace std;

// custom data type for tree building

struct Node {

int data;

struct Node\* left;

struct Node\* right;

Node(int val)

{

data = val;

left = right = NULL;

}

};

// Below function accepts string and a pointer variable as

// an argument

// and draw the tree. Returns the root of the tree

Node\* constructtree(string s, int\* start)

{

// Assuming there is/are no negative

// character/characters in the string

if (s.size() == 0 || \*start >= s.size())

return NULL;

// constructing a number from the continuous digits

int num = 0;

while (\*start < s.size() && s[\*start] != '('

&& s[\*start] != ')') {

int num\_here = (int)(s[\*start] - '0');

num = num \* 10 + num\_here;

\*start = \*start + 1;

}

// creating a node from the constructed number from

// above loop

struct Node\* root = new Node(num);

// check if start has reached the end of the string

if (\*start >= s.size())

return root;

// As soon as we see first right parenthesis from the

// current node we start to construct the tree in the

// left

if (\*start < s.size() && s[\*start] == '(') {

\*start = \*start + 1;

root->left = constructtree(s, start);

}

if (\*start < s.size() && s[\*start] == ')')

\*start = \*start + 1;

// As soon as we see second right parenthesis from the

// current node we start to construct the tree in the

// right

if (\*start < s.size() && s[\*start] == '(') {

\*start = \*start + 1;

root->right = constructtree(s, start);

}

if (\*start < s.size() && s[\*start] == ')')

\*start = \*start + 1;

return root;

}

void preorder(Node\* root)

{

if (root == NULL)

return;

cout << root->data << " ";

preorder(root->left);

preorder(root->right);

}

int main()

{

string s = "4(2(3)(1))(6(5))";

// cin>>s;

int start = 0;

Node\* root = constructtree(s, &start);

preorder(root);

return 0;

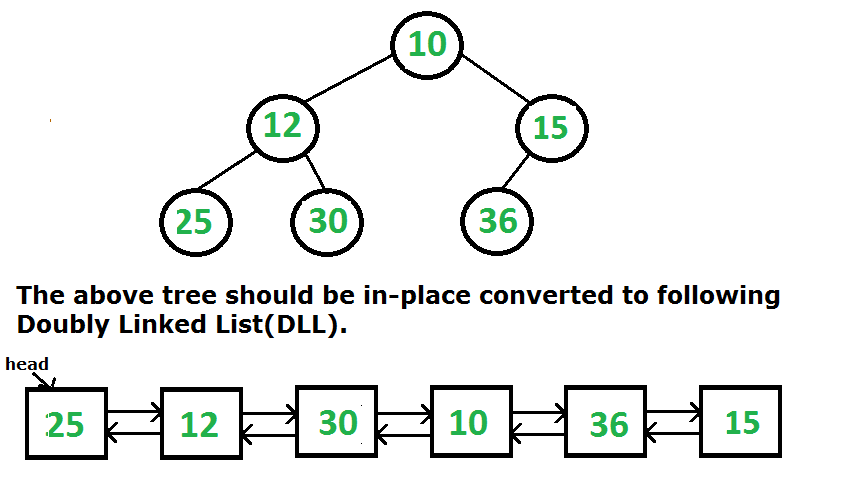
}

**Output**

4 2 3 1 6 5

# 179. Convert Binary tree into Doubly Linked List

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 (leftmost node in BT) must be the head node of the DLL.



**Example 1:**

**Input:**

      1

   /  \

  3    2

**Output:**

3 1 2

2 1 3

**Explanation:** DLL would be 3<=>1<=>2

**Example 2:**

**Input:**

       10

     /   \

20   30

  /   \

 40   60

**Output:**

40 20 60 10 30

30 10 60 20 40

**Explanation:**  DLL would be

40<=>20<=>60<=>10<=>30.

**Your Task:**  
You don't have to take input. Complete the function **bToDLL()**that takes **root**node of the tree as a parameter and returns the head of DLL . The driver code prints the DLL both ways.

**Expected Time Complexity:**O(N).  
**Expected Auxiliary Space:**O(H).  
**Note:**H is the height of the tree and this space is used implicitly for the recursion stack.

**Constraints:**  
1 ≤ Number of nodes ≤ 105  
1 ≤ Data of a node ≤ 105

## Solution:

The problem here is simpler as we don’t need to create a circular DLL, but a simple DLL. The idea behind its solution is quite simple and straight.

1. If the left subtree exists, process the left subtree
   1. Recursively convert the left subtree to DLL.
   2. Then find the inorder predecessor of the root in the left subtree (the inorder predecessor is the rightmost node in the left subtree).
   3. Make the inorder predecessor as the previous root and the root as the next in order predecessor.
2. If the right subtree exists, process the right subtree (Below 3 steps are similar to the left subtree).
   1. Recursively convert the right subtree to DLL.
   2. Then find the inorder successor of the root in the right subtree (in order the successor is the leftmost node in the right subtree).
   3. Make the inorder successor as the next root and the root as the previous inorder successor.
3. Find the leftmost node and return it (the leftmost node is always the head of a converted DLL).

Below is the source code for the 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 predecessor as previous of root

root->left = left;

}

// Convert the right subtree and link to root

if (root->right != NULL) {

// Convert the right subtree

node\* right = bintree2listUtil(root->right);

// Find inorder successor. After this loop, right

// will point to the inorder successor

for (; right->left != NULL; right = right->left)

;

// Make root as previous of successor

right->left = root;

// Make successor as next of root

root->right = right;

}

return root;

}

// The main function that first calls

// bintree2listUtil(), then follows step 3

// of the above algorithm

node\* bintree2list(node\* root)

{

// Base case

if (root == NULL)

return root;

// Convert to DLL using bintree2listUtil()

root = bintree2listUtil(root);

// bintree2listUtil() returns root node of the converted

// DLL. We need pointer to the leftmost node which is

// head of the constructed DLL, so move to the leftmost

// node

while (root->left != NULL)

root = root->left;

return (root);

}

/\* Helper function that allocates a new node with the

given data and NULL left and right pointers. \*/

node\* newNode(int data)

{

node\* new\_node = new node();

new\_node->data = data;

new\_node->left = new\_node->right = NULL;

return (new\_node);

}

/\* Function to print nodes in a given doubly linked list \*/

void printList(node\* node)

{

while (node != NULL) {

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;

}

**Output**

25 12 30 10 36 15

**Another Approach:**  
Algorithm:

1. Traverse the tree in inorder fashion.
2. While visiting each node, keep track of DLL’s head and tail pointers, insert each visited node to the end of DLL using tail pointer.
3. Return head of the list.

Below is the implementation of the above approach:

// 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.\*/

void bintree2listUtil(node\* root, node\*\* head, node\*\* tail)

{

if (root == NULL)

return NULL;

node\* left = root->left;

node\* right = root->right;

bintree2listUtil(root->left, head, tail);

if (\*head == NULL) {

\*head = root;

}

root->left = \*tail;

if (\*tail != NULL) {

(\*tail)->right = root;

}

\*tail = root;

bintree2listUtil(root->right, head, tail);

}

// The main function that first calls

// bintree2listUtil()

node\* bintree2list(node\* root)

{

// Base case

if (root == NULL)

return root;

node\* head = NULL;

node\* tail = NULL;

bintree2listUtil(root, &head, &tail);

return 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 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;

}

**Output**

25 12 30 10 36 15

**Another approach**

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 in order traversal of the tree. In order to traversal, we keep track of the previously visited node and change the left pointer to the previous node. See *fixPrevPtr()*implementation below.   
**2) *Fix Right Pointers*:**The above is intuitive and simple. How to change the right pointers to point to the 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 the previous node in DLL, we can linearly traverse the complete DLL using these pointers. The traversal would be from last to the 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()*implementation below.

// 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;

}

**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 a given Binary Tree. The solution simply does two traversals of all Binary Tree nodes.

**Another Approach:**

In this post, a third solution is discussed which seems to be the simplest of all. The idea is to do in order 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 to the *prev*and previous of this node as *prev*.

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

**Another approach:**

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;

};

// 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;

}

// 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 BToDLL(Node\* root, Node\*& head)

{

// Base cases

if (root == NULL)

return;

// Recursively convert right subtree

BToDLL(root->right, head);

// insert root into DLL

root->right = head;

// Change left pointer of previous head

if (head != NULL)

head->left = root;

// Change head of Doubly linked list

head = root;

// Recursively convert left subtree

BToDLL(root->left, head);

}

// 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.

# 180. Convert Binary tree into Sum tree

Given a Binary Tree of size N , where each node can have positive or negative values. Convert this to a tree where each node contains the sum of the left and right sub trees of the original tree. The values of leaf nodes are changed to 0.

**Example 1:**

**Input:**

10

/ \

-2 6

/ \ / \

8 -4 7 5

**Output:**

20

/ \

4 12

/ \ / \

0 0 0 0

**Explanation:**

(4-2+12+6)

/ \

(8-4) (7+5)

/ \ / \

0 0 0 0

**Your Task:**  
You dont need to read input or print anything. Complete the function**toSumTree()** which takes root node as input parameter and modifies the given tree in-place.

**Note:** If you click on Compile and Test the output will be the in-order traversal of the modified tree.

**Expected Time Complexity:**O(N)  
**Expected Auxiliary Space:**O(height of tree)

**Constraints:**  
1 ≤ N ≤ 104

## 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

// old 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;

}

**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.

**My Implementation:**

class Solution {

public:

int fun(Node\* &root){

if(!root)

return 0;

int t = root->data;

int l = fun(root->left);

int r = fun(root->right);

root->data = l + r;

if(root->left)

root->data += root->left->data;

if(root->right)

root->data += root->right->data;

return t;

}

// 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

void toSumTree(Node \*node)

{

if(!node)

return;

fun(node);

}

};

# 181. [Construct Binary tree from Inorder and preorder traversal](https://practice.geeksforgeeks.org/problems/construct-tree-1/1)

**Construct Tree from Inorder & Preorder**

Given 2 Arrays of Inorder and preorder traversal. Construct a tree and print the Postorder traversal.

**Example 1:**

**Input:**

N = 4

inorder[] = {1 6 8 7}

preorder[] = {1 6 7 8}

**Output:** 8 7 6 1

**Example 2:**

**Input:**

N = 6

inorder[] = {3 1 4 0 5 2}

preorder[] = {0 1 3 4 2 5}

**Output:** 3 4 1 5 2 0

**Explanation:** The tree will look like

    0

    /     \

   1       2

 /   \    /

3    4   5

**Your Task:**  
Your task is to complete the function **buildTree()**which takes 3 arguments(inorder traversal array, preorder traversal array, and size of tree n) and returns the root node to the tree constructed. You are not required to print anything and a new line is added automatically (The post order of the returned tree is printed by the driver's code.)

**Expected Time Complexity**: O(N\*N).  
**Expected Auxiliary Space:** O(N).

**Constraints:**  
1<=Number of Nodes<=1000

## Solution:

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

A

/ \

/ \

D B E F C

We recursively follow the above steps and get the following tree.

A

/ \

/ \

B C

/ \ /

/ \ /

D E F

Algorithm: buildTree()   
1) Pick an element from Preorder. Increment a Preorder Index Variable (preIndex in below code) to pick the next element in the next recursive call.   
2) Create a new tree node tNode with the data as the 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 a left subtree of tNode.   
5) Call buildTree for elements after inIndex and make the built tree as a right subtree of tNode.   
6) return tNode.

/\* 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 function is here just to test buildTree() \*/

void printInorder(node\* node)

{

if (node == NULL)

return;

/\* first recur on left child \*/

printInorder(node->left);

/\* then print the data of node \*/

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 Inorder traversal \*/

cout << "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^2)**. The worst case occurs when the 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 function is here just to test buildTree() \*/

void printInorder(struct Node\* node)

{

if (node == NULL)

return;

printInorder(node->left);

printf("%c ", node->data);

printInorder(node->right);

}

/\* Driver program to test above functions \*/

int main()

{

char in[] = { 'D', 'B', 'E', 'A', 'F', 'C' };

char pre[] = { 'A', 'B', 'D', 'E', 'C', 'F' };

int len = sizeof(in) / sizeof(in[0]);

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

/\* Let us test the built tree by printing

Inorder 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)

**Tree from Postorder and Inorder**

Given **inorder**and **postorder** traversals of a Binary Tree in the arrays **in[]** and **post[]** respectively. The task is to construct the binary tree from these traversals.

**Example 1:**

**Input:**

N = 8

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

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

**Output:** 1 2 4 8 5 3 6 7

**Explanation:** For the given postorder and

inorder traversal of tree the  resultant

binary tree will be

          1

       /     \

     2        3

   /  \ / \

  4    5   6    7

   \

     8

**Example 2:**

**Input:**

N = 5

in[] = 9 5 2 3 4

post[] = 5 9 3 4 2

**Output:** 2 9 5 4 3

**Explanation:**

the  resultant binary tree will be

           2

        /    \

       9      4

 \ /

    5 3

**Your Task:**  
You do not need to read input or print anything. Complete the function **buildTree()**which takes the inorder, postorder traversals and the number of nodes in the tree as input parameters and returns the root node of the newly constructed Binary Tree.  
**The generated output contains the preorder traversal of the constructed tree.**

**Expected Time Complexity:**O(N2)  
**Expected Auxiliary Space:**O(N)

**Constraints:**  
1 <= N <= 103  
1 <= in[i], post[i] <= 103

## Solution:

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 the root always appears at the end of postorder traversal.  
**2)** We search “1” in in[] to find the left and right subtrees of the root. Everything on the left of “1” in in[] is in the left subtree and everything on right is in the 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 the above idea. One important observation is, we recursively call for the right subtree before the left subtree as we decrease the index of the 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);

/\* 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 function is here just to test \*/

void preOrder(Node\* node)

{

if (node == NULL)

return;

printf("%d ", node->data);

preOrder(node->left);

preOrder(node->right);

}

// Driver code

int main()

{

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

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

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

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

cout << "Preorder of the constructed tree : \n";

preOrder(root);

return 0;

}

**Output**

Preorder of the constructed tree :

1 2 4 8 5 3 6 7

**Time Complexity:** O(n2)

**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 If given that element in the tree are not repeated.

/\* C++ program to construct tree using inorder and

postorder traversals \*/

#include <bits/stdc++.h>

using namespace std;

/\* A binary tree node has data, pointer to left

child and a pointer to right child \*/

struct Node {

int data;

Node \*left, \*right;

};

// Utility function to create a new node

Node\* newNode(int data);

/\* Recursive function to construct binary of size n

from Inorder traversal in[] and Postorder traversal

post[]. Initial values of inStrt and inEnd should

be 0 and n -1. The function doesn't do any error

checking for cases where inorder and postorder

do not form a tree \*/

Node\* buildUtil(int in[], int post[], int inStrt,

int inEnd, int\* pIndex, unordered\_map<int, int>& mp)

{

// Base case

if (inStrt > inEnd)

return NULL;

/\* Pick current node from Postorder traversal

using postIndex and decrement postIndex \*/

int curr = post[\*pIndex];

Node\* node = newNode(curr);

(\*pIndex)--;

/\* If this node has no children then return \*/

if (inStrt == inEnd)

return node;

/\* Else find the index of this node in Inorder

traversal \*/

int iIndex = mp[curr];

/\* Using index in Inorder traversal, construct

left and right subtress \*/

node->right = buildUtil(in, post, iIndex + 1,

inEnd, pIndex, mp);

node->left = buildUtil(in, post, inStrt,

iIndex - 1, pIndex, mp);

return node;

}

// This function mainly creates an unordered\_map, then

// calls buildTreeUtil()

struct Node\* buildTree(int in[], int post[], int len)

{

// Store indexes of all items so that we

// we can quickly find later

unordered\_map<int, int> mp;

for (int i = 0; i < len; i++)

mp[in[i]] = i;

int index = len - 1; // Index in postorder

return buildUtil(in, post, 0, len - 1,

&index, mp);

}

/\* Helper function that allocates a new node \*/

Node\* newNode(int data)

{

Node\* node = (Node\*)malloc(sizeof(Node));

node->data = data;

node->left = node->right = NULL;

return (node);

}

/\* This function is here just to test \*/

void preOrder(Node\* node)

{

if (node == NULL)

return;

printf("%d ", node->data);

preOrder(node->left);

preOrder(node->right);

}

// Driver code

int main()

{

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

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

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

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

cout << "Preorder of the constructed tree : \n";

preOrder(root);

return 0;

}

**Output**

Preorder of the constructed tree :

1 2 4 8 5 3 6 7

**Time Complexity:** O(n)

# 182. Find minimum swaps required to convert a Binary tree into BST

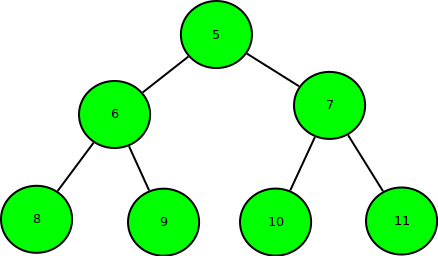
Given the array representation of Complete Binary Tree i.e, if index i is the parent, index 2\*i + 1 is the left child and index 2\*i + 2 is the right child. The task is to find the minimum number of swap required to convert it into Binary Search Tree.

Examples:

Input : arr[] = { 5, 6, 7, 8, 9, 10, 11 }

Output : 3

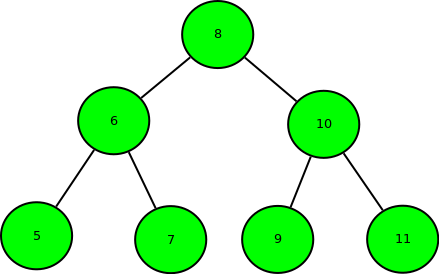
Binary tree of the given array:



Swap 1: Swap node 8 with node 5.

Swap 2: Swap node 9 with node 10.

Swap 3: Swap node 10 with node 7.

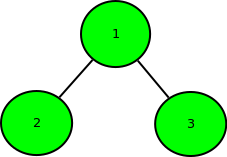


So, minimum 3 swaps are required.

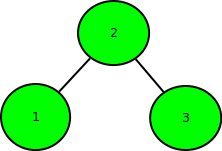
Input : arr[] = { 1, 2, 3 }

Output : 1

Binary tree of the given array:



After swapping node 1 with node 2.



So, only 1 swap required.

## Solution:

The idea is to use the fact that inorder traversal of Binary Search Tree is in increasing order of their value.   
So, find the inorder traversal of the Binary Tree and store it in the array and try to sort the array. The minimum number of swap required to get the array sorted will be the answer. Please refer below post to find minimum number of swaps required to get the array sorted.  
[Minimum number of swaps required to sort an array](https://www.geeksforgeeks.org/minimum-number-swaps-required-sort-array/)  
**Time Complexity:**O(n log n).

// C++ program for Minimum swap required

// to convert binary tree to binary search tree

#include<bits/stdc++.h>

using namespace std;

// Inorder Traversal of Binary Tree

void inorder(int a[], std::vector<int> &v,

int n, int index)

{

// if index is greater or equal to vector size

if(index >= n)

return;

inorder(a, v, n, 2 \* index + 1);

// push elements in vector

v.push\_back(a[index]);

inorder(a, v, n, 2 \* index + 2);

}

// Function to find minimum swaps to sort an array

int minSwaps(std::vector<int> &v)

{

std::vector<pair<int,int> > t(v.size());

int ans = 0;

for(int i = 0; i < v.size(); i++)

t[i].first = v[i], t[i].second = i;

sort(t.begin(), t.end());

for(int i = 0; i < t.size(); i++)

{

// second element is equal to i

if(i == t[i].second)

continue;

else

{

// swaping of elements

swap(t[i].first, t[t[i].second].first);

swap(t[i].second, t[t[i].second].second);

}

// Second is not equal to i

if(i != t[i].second)

--i;

ans++;

}

return ans;

}

// Driver code

int main()

{

int a[] = { 5, 6, 7, 8, 9, 10, 11 };

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

std::vector<int> v;

inorder(a, v, n, 0);

cout << minSwaps(v) << endl;

}

# 183. Check if Binary tree is Sum tree or not

Given a Binary Tree. Return **1** if, for every node **X** in the tree other than the leaves, its value is equal to the sum of its left subtree's value and its right subtree's value. Else return **0**.

An empty tree is also a Sum Tree as the sum of an empty tree can be considered to be 0. A leaf node is also considered a Sum Tree.

**Example 1:**

**Input:**

3

/ \

1 2

**Output:** 1

**Explanation:**

The sum of left subtree and right subtree is

1 + 2 = 3, which is the value of the root node.

Therefore,the given binary tree is a **sum tree**.

**Example 2:**

**Input:**

10

/ \

20 30

/ \

10 10

**Output:** 0

**Explanation:**

The given tree is not a sum tree.

For the root node, sum of elements

in left subtree is 40 and sum of elements

in right subtree is 30. Root element = 10

which is not equal to 30+40.

**Your Task:**  
You don't need to read input or print anything. Complete the function **isSumTree()**which takes **root**node as input parameter and returns true if the tree is a SumTree else it returns false.

**Expected Time Complexity:**O(N)  
**Expected Auxiliary Space:** O(Height of the Tree)

**Constraints:**  
1 ≤ number of nodes ≤ 104

## Solution:

**Method 1 (Simple)**  
Get the sum of nodes in the left subtree and right subtree. Check if the sum calculated is equal to the root’s data. Also, recursively check if the left and right subtrees are SumTrees.

// C++ program to check if Binary tree

// is sum tree or not

#include <iostream>

using namespace std;

// A binary tree node has data,

// left child and right child

struct node

{

int data;

struct node\* left;

struct node\* right;

};

// A utility function to get the sum

// of values in tree with root as root

int sum(struct node \*root)

{

if (root == NULL)

return 0;

return sum(root->left) + root->data +

sum(root->right);

}

// Returns 1 if sum property holds for

// the given node and both of its children

int isSumTree(struct node\* node)

{

int ls, rs;

// If node is NULL or it's a leaf

// node then return true

if (node == NULL ||

(node->left == NULL &&

node->right == NULL))

return 1;

// Get sum of nodes in left and

// right subtrees

ls = sum(node->left);

rs = sum(node->right);

// If the node and both of its

// children satisfy the property

// return 1 else 0

if ((node->data == ls + rs) &&

isSumTree(node->left) &&

isSumTree(node->right))

return 1;

return 0;

}

// Helper function that allocates a new node

// with the given data and NULL left and right

// pointers.

struct node\* newNode(int data)

{

struct node\* node = (struct node\*)malloc(

sizeof(struct node));

node->data = data;

node->left = NULL;

node->right = NULL;

return(node);

}

// Driver code

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))

cout << "The given tree is a SumTree ";

else

cout << "The given tree is not a SumTree ";

getchar();

return 0;

}

**Output**

The given tree is a SumTree

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

**Method 2 (Tricky)**  
Method 1 uses sum() to get the sum of nodes in left and right subtrees. Method 2 uses the following rules to get the sum directly.   
1) If the node is a leaf node then the sum of the subtree rooted with this node is equal to the value of this node.   
2) If the node is not a leaf node then the sum of the subtree rooted with this node is twice the value of this node (Assuming that the tree rooted with this node is SumTree).

// C++ program to check if Binary tree

// is sum tree or not

#include<bits/stdc++.h>

using namespace std;

/\* A binary tree node has data,

left child and right child \*/

struct node

{

int data;

node\* left;

node\* right;

};

/\* Utility function to check if

the given node is leaf or not \*/

int isLeaf(node \*node)

{

if(node == NULL)

return 0;

if(node->left == NULL && node->right == NULL)

return 1;

return 0;

}

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

tree \*/

int isSumTree(node\* node)

{

int ls; // for sum of nodes in left subtree

int rs; // for sum of nodes in right subtree

/\* If node is NULL or it's a leaf node then

return true \*/

if(node == NULL || isLeaf(node))

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.

\*/

node\* newNode(int data)

{

node\* node1 = new node();

node1->data = data;

node1->left = NULL;

node1->right = NULL;

return(node1);

}

/\* Driver code \*/

int main()

{

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))

cout << "The given tree is a SumTree ";

else

cout << "The given tree is not a SumTree ";

return 0;

}

**Output:**

The given tree is a SumTree

**Time Complexity:** O(n)

**My Implementation:**

class Solution

{

public:

bool fun(Node\* root, int &sum){

if(!root){

sum = 0;

return true;

}

int lval, rval;

bool l, r;

l = fun(root->left, lval);

r = fun(root->right, rval);

bool res = false;

if((!root->left && !root->right) || (l && r && root->data == lval+rval))

res = true;

sum = lval + rval + root->data;

return res;

}

bool isSumTree(Node\* root)

{

int sum = 0;

return fun(root, sum);

}

};

**Time Complexity:** O(n)

# 184. Check if all leaf nodes are at same level or not

Given a Binary Tree, check if all leaves are at same level or not.

**Example 1:**

**Input:**

1

/ \

2 3

**Output:** 1

**Explanation:**

Leaves 2 and 3 are at same level.

**Example 2:**

**Input:**

10

/ \

20 30

/ \

10 15

**Output:** 0

**Explanation:**

Leaves 10, 15 and 30 are not at same level.

**Your Task:**  
You dont need to read input or print anything. Complete the function **check()** which takes root node as input parameter and returns true/false depending on whether all the leaf nodes are at the same level or not.

**Expected Time Complexity:**O(N)  
**Expected Auxiliary Space:** O(height of tree)

**Constraints:**  
1 ≤ N ≤ 10^3

## Solution:

**Method 1 (Recursive)**

The idea is to first find the level of the leftmost leaf and store it in a variable leafLevel. Then compare level of all other leaves with leafLevel, if same, return true, else return false. We traverse the given Binary Tree in a Preorder fashion. An argument leaflevel is passed to all calls. The value of leafLevel is initialized as 0 to indicate that the first leaf is not yet seen yet. The value is updated when we find first leaf. Level of subsequent leaves (in preorder) is compared with leafLevel.

// C++ program to check if all leaves

// are at same level

#include <bits/stdc++.h>

using namespace std;

// A binary tree node

struct Node

{

int data;

struct Node \*left, \*right;

};

// A utility function to allocate

// a new tree node

struct Node\* newNode(int data)

{

struct Node\* node = (struct Node\*) malloc(sizeof(struct Node));

node->data = data;

node->left = node->right = NULL;

return node;

}

/\* Recursive function which checks whether

all leaves are at same level \*/

bool checkUtil(struct Node \*root,

int level, int \*leafLevel)

{

// Base case

if (root == NULL) return true;

// If a leaf node is encountered

if (root->left == NULL &&

root->right == NULL)

{

// When a leaf node is found

// first time

if (\*leafLevel == 0)

{

\*leafLevel = level; // Set first found leaf's level

return true;

}

// If this is not first leaf node, compare

// its level with first leaf's level

return (level == \*leafLevel);

}

// If this node is not leaf, recursively

// check left and right subtrees

return checkUtil(root->left, level + 1, leafLevel) &&

checkUtil(root->right, level + 1, leafLevel);

}

/\* The main function to check

if all leafs are at same level.

It mainly uses checkUtil() \*/

bool check(struct Node \*root)

{

int level = 0, leafLevel = 0;

return checkUtil(root, level, &leafLevel);

}

// Driver Code

int main()

{

// Let us create tree shown in third example

struct Node \*root = newNode(12);

root->left = newNode(5);

root->left->left = newNode(3);

root->left->right = newNode(9);

root->left->left->left = newNode(1);

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

if (check(root))

cout << "Leaves are at same level\n";

else

cout << "Leaves are not at same level\n";

getchar();

return 0;

}

**Output:**

Leaves are at same level

**Time Complexity:** The function does a simple traversal of the tree, so the complexity is O(n).

**Method 2 (Iterative)**

It can also be solved by an iterative approach.  
The idea is to iteratively traverse the tree, and when you encounter the first leaf node, store its level in result variable, now whenever you encounter any leaf node, compare its level with previously stored result, they are the same then proceed for the rest of tree, else return false.

// C++ program to check if all leaf nodes are at

// same level of binary tree

#include <bits/stdc++.h>

using namespace std;

// tree node

struct Node {

int data;

Node \*left, \*right;

};

// returns a new tree Node

Node\* newNode(int data)

{

Node\* temp = new Node();

temp->data = data;

temp->left = temp->right = NULL;

return temp;

}

// return true if all leaf nodes are

// at same level, else false

int checkLevelLeafNode(Node\* root)

{

if (!root)

return 1;

// create a queue for level order traversal

queue<Node\*> q;

q.push(root);

int result = INT\_MAX;

int level = 0;

// traverse until the queue is empty

while (!q.empty()) {

int size = q.size();

level += 1;

// traverse for complete level

while(size > 0){

Node\* temp = q.front();

q.pop();

// check for left child

if (temp->left) {

q.push(temp->left);

// if its leaf node

if(!temp->left->right && !temp->left->left){

// if it's first leaf node, then update result

if (result == INT\_MAX)

result = level;

// if it's not first leaf node, then compare

// the level with level of previous leaf node

else if (result != level)

return 0;

}

}

// check for right child

if (temp->right){

q.push(temp->right);

// if it's leaf node

if (!temp->right->left && !temp->right->right)

// if it's first leaf node till now,

// then update the result

if (result == INT\_MAX)

result = level;

// if it is not the first leaf node,

// then compare the level with level

// of previous leaf node

else if(result != level)

return 0;

}

size -= 1;

}

}

return 1;

}

// driver program

int main()

{

// construct a tree

Node\* root = newNode(1);

root->left = newNode(2);

root->right = newNode(3);

root->left->right = newNode(4);

root->right->left = newNode(5);

root->right->right = newNode(6);

int result = checkLevelLeafNode(root);

if (result)

cout << "All leaf nodes are at same level\n";

else

cout << "Leaf nodes not at same level\n";

return 0;

}

**Output:**

All leaf nodes are at same level

**Time Complexity :** O(n)

# 185. Check if a Binary Tree contains duplicate subtrees of size 2 or more [ IMP ]

Given a binary tree, find out whether it contains a duplicate sub-tree of size two or more, or not.

**Example 1 :**

**Input :**

1

/ \

2 3

/ \ \

4 5 2

/ \

4 5

**Output :** 1

**Explanation :**

2

/ \

4 5

is the duplicate sub-tree.

**Example 2 :**

**Input :**

1

/ \

2 3

**Output:** 0

**Explanation:** There is no duplicate sub-tree

in the given binary tree.

**Your Task:**  
You don't need to read input or print anything. Your task is to complete the function **dupSub()** which takes root of the tree as the only arguement and returns 1 if the binary tree contains a duplicate sub-tree of size two or more, else 0.  
**Note:** Two same leaf nodes are not considered as subtree as size of a leaf node is one.

**Constraints:**  
1 ≤ length of string ≤ 100

## Solution:

**[ Method 1]**   
A simple solution is that, we pick every node of tree and try to find is any sub-tree of given tree is present in tree which is identical with that sub-tree. Here we can use below post to find if a subtree is present anywhere else in tree.   
[Check if a binary tree is subtree of another binary tree](https://www.geeksforgeeks.org/check-if-a-binary-tree-is-subtree-of-another-binary-tree/)

**[Method 2 ]( Efficient solution )**   
An Efficient solution based on[tree serialization](https://www.geeksforgeeks.org/serialize-deserialize-binary-tree/) and [hashing](https://www.geeksforgeeks.org/hashing-set-1-introduction/). The idea is to serialize subtrees as strings and store the strings in hash table. Once we find a serialized tree (which is not a leaf) already existing in hash-table, we return true.

Below The implementation of above idea.

// C++ program to find if there is a duplicate

// sub-tree of size 2 or more.

#include<bits/stdc++.h>

using namespace std;

// Separator node

const char MARKER = '$';

// Structure for a binary tree node

struct Node

{

char key;

Node \*left, \*right;

};

// A utility function to create a new node

Node\* newNode(char key)

{

Node\* node = new Node;

node->key = key;

node->left = node->right = NULL;

return node;

}

unordered\_set<string> subtrees;

// This function returns empty string if tree

// contains a duplicate subtree of size 2 or more.

string dupSubUtil(Node \*root)

{

string s = "";

// If current node is NULL, return marker

if (root == NULL)

return s + MARKER;

// If left subtree has a duplicate subtree.

string lStr = dupSubUtil(root->left);

if (lStr.compare(s) == 0)

return s;

// Do same for right subtree

string rStr = dupSubUtil(root->right);

if (rStr.compare(s) == 0)

return s;

// Serialize current subtree

s = s + root->key + lStr + rStr;

// If current subtree already exists in hash

// table. [Note that size of a serialized tree

// with single node is 3 as it has two marker

// nodes.

if (s.length() > 3 &&

subtrees.find(s) != subtrees.end())

return "";

subtrees.insert(s);

return s;

}

// Driver program to test above functions

int main()

{

Node \*root = newNode('A');

root->left = newNode('B');

root->right = newNode('C');

root->left->left = newNode('D');

root->left->right = newNode('E');

root->right->right = newNode('B');

root->right->right->right = newNode('E');

root->right->right->left= newNode('D');

string str = dupSubUtil(root);

(str.compare("") == 0) ? cout << " Yes ": cout << " No " ;

return 0;

}

**Output:**

Yes

**My Implementation:**

class Solution {

public:

int res = 0;

unordered\_set<string> st;

string fun(Node\* root){

string str = "";

if(!root)

return str;

str = "(" + fun(root->left) + root->data + fun(root->right) + ")";

if( (root->left || root->right) && (st.find(str)!=st.end()) )

res = 1;

else

st.insert(str);

return str;

}

/\*This function returns true if the tree contains

a duplicate subtree of size 2 or more else returns false\*/

int dupSub(Node \*root) {

fun(root);

return res;

}

};

# 186. Check if 2 trees are mirror or not

# Symmetric Tree

Given a Binary Tree. Check whether it is Symmetric or not, i.e. whether the binary tree is a **Mirror image of itself** or not.

**Example 1:**

**Input:**

5

/ \

1 1

/ \

2 2

**Output:** True

**Explanation:** Tree is mirror image of

itself i.e. tree is symmetric

**Example 2:**

**Input:**

5

/ \

10 10

/ \ \

20 20 30

**Output:** False

**Your Task:**  
You don't need to read input or print anything. Your task is to complete the function **isSymmetric()** which takes the root of the Binary Tree as its input and returns True if the given Binary Tree is a same as the Mirror image of itself. Else, it returns False.

**Expected Time Complexity:** O(N).  
**Expected Auxiliary Space:** O(Height of the Tree).

**Constraints:**  
0<=Number of nodes<=100

## Solution:

class Solution{

public:

bool pairSymmetric(Node\* root1, Node\* root2){

if(!root1 && !root2)

return true;

if(!root1 || !root2)

return false;

if(root1->data==root2->data && pairSymmetric(root1->left, root2->right) && pairSymmetric(root1->right, root2->left))

return true;

return false;

}

// return true/false denoting whether the tree is Symmetric or not

bool isSymmetric(struct Node\* root)

{

if(!root)

return true;

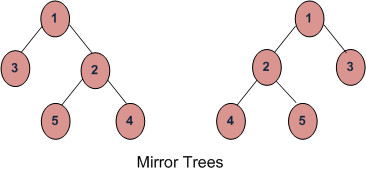
return pairSymmetric(root->left, root->right);

}

};

# Check if two trees are Mirror

Given two Binary Trees, write a function that returns true if two trees are mirror of each other, else false. For example, the function should return true for following input trees.



## Solution:

For two trees ‘a’ and ‘b’ to be mirror images, the following three conditions must be true:

1. Their root node’s key must be same
2. Left subtree of root of ‘a’ and right subtree root of ‘b’ are mirror.
3. Right subtree of ‘a’ and left subtree of ‘b’ are mirror.

Below is implementation of above idea.

// C++ program to check if two trees are mirror

// of each other

#include<bits/stdc++.h>

using namespace std;

/\* A binary tree node has data, pointer to

left child and a pointer to right child \*/

struct Node

{

int data;

Node\* left, \*right;

};

/\* Given two trees, return true if they are

mirror of each other \*/

/\*As function has to return bool value instead integer value\*/

bool areMirror(Node\* a, Node\* b)

{

/\* Base case : Both empty \*/

if (a==NULL && b==NULL)

return true;

// If only one is empty

if (a==NULL || b == NULL)

return false;

/\* Both non-empty, compare them recursively

Note that in recursive calls, we pass left

of one tree and right of other tree \*/

return a->data == b->data &&

areMirror(a->left, b->right) &&

areMirror(a->right, b->left);

}

/\* Helper function that allocates a new node \*/

Node\* newNode(int data)

{

Node\* node = new Node;

node->data = data;

node->left = node->right = NULL;

return(node);

}

/\* Driver program to test areMirror() \*/

int main()

{

Node \*a = newNode(1);

Node \*b = newNode(1);

a->left = newNode(2);

a->right = newNode(3);

a->left->left = newNode(4);

a->left->right = newNode(5);

b->left = newNode(3);

b->right = newNode(2);

b->right->left = newNode(5);

b->right->right = newNode(4);

areMirror(a, b)? cout << "Yes" : cout << "No";

return 0;

}

**Output :**

Yes

**Time Complexity :** O(n)

# Check mirror in n-ary tree

Given two n-ary trees, the task is to check if they are the mirror of each other or not. Print “Yes” if they are the mirror of each other else “No”.

**Examples:**

Input : Node = 3, Edges = 2

Edge 1 of first N-ary: 1 2

Edge 2 of first N-ary: 1 3

Edge 1 of second N-ary: 1 3

Edge 2 of second N-ary: 1 2

Output : Yes

https://media.geeksforgeeks.org/wp-content/uploads/mirrorimage-1.png

Input : Node = 3, Edges = 2

Edge 1 of first N-ary: 1 2

Edge 2 of first N-ary: 1 3

Edge 1 of second N-ary: 1 2

Edge 2 of second N-ary: 1 3

Output : No

## Solution:

**Approach  1: (Using Hashing)**

The idea is to use an **unordered map of stacks** to check if given N-ary tree are mirror of each other or not.   
Let the first n-ary tree be t1 and the second n-ary tree is t2. For each node in t1, push its connected node in their corresponding stack in the map. Now, for each node in t2, their connected node match with the top of the stack, then pop elements from the stack.

Otherwise, if the node does not match with the top of the stack then it means two trees are not mirror of each other.   
Now, for each corresponding node do the following:

1. Iterate over map of stack

Push all connected nodes of each node of first tree in map of stack.

2. Again iterate over map for each node of second tree

**For example :**

Let us take one node X of second tree

For this node X , check in map which stack is used

a = Top of that stack for node X present in second tree;

b = Connected node of X in second tree

if (a != b)

return false;

pop node X from stack.

// C++ program to check if two n-ary trees are

// mirror.

#include <bits/stdc++.h>

using namespace std;

// Function to check given two trees are mirror

// of each other or not

int checkMirrorTree(int M, int N, int u1[ ],

int v1[ ] , int u2[], int v2[])

{

// Map to store nodes of the tree

unordered\_map<int , stack<int>>mp;

// Traverse first tree nodes

for (int i = 0 ; i < N ; i++ )

{

mp[u1[i]].push(v1[i]);

}

// Traverse second tree nodes

for (int i = 0 ; i < N ; i++)

{

if(mp[u2[i]].top() != v2[i])

return 0;

mp[u2[i]].pop();

}

return 1;

}

// Driver code

int main()

{

int M = 7, N = 6;

//Tree 1

int u1[] = { 1, 1, 1, 10, 10, 10 };

int v1[] = { 10, 7, 3, 4, 5, 6 };

//Tree 2

int u2[] = { 1, 1, 1, 10, 10, 10 };

int v2[] = { 3, 7, 10, 6, 5, 4 };

if(checkMirrorTree(M, N, u1, v1, u2, v2))

cout<<"Yes";

else

cout<<"No";

return 0;

}

**Output**

Yes

**Approach 2: (Using LinkedList):**

The main approach is to use one list of stack and one list of queue to store to value of nodes given in the form of two arrays.

1. Initialize both the lists with empty stack and empty queues respectively.

2. Now, iterate over the lists

Push all connected nodes of each node of first tree in list of stack and second tree list of queue.

3. Now iterate over the array and pop item from both stack and queue and check if they are same, if not same then retur

// Java program to check two n-ary trees are mirror.

import java.io.\*;

import java.util.\*;

class GFG {

// Function to check given two trees are mirror

// of each other or not

static int checkMirrorTree(int n, int e, int[] A, int[] B) {

//Lists to store nodes of the tree

List<Stack<Integer>> s = new ArrayList<>();

List<Queue<Integer>> q = new ArrayList<>();

// initializing both list with empty stack and queue

for (int i = 0; i <= n; i++) {

s.add(new Stack<>());

Queue<Integer> queue = new LinkedList<>();

q.add(queue);

}

// add all nodes of tree 1 to list of stack and tree 2 to list of queue

for (int i = 0; i < 2 \* e; i += 2) {

s.get(A[i]).push(A[i + 1]);

q.get(B[i]).add(B[i + 1]);

}

// now take out the stack and queues

// for each of the nodes and compare them

// one by one

for (int i = 1; i <= n; i++) {

while (!s.get(i).isEmpty() && !q.get(i).isEmpty()) {

int a = s.get(i).pop();

int b = q.get(i).poll();

if (a != b) {

return 0;

}

}

}

return 1;

}

public static void main (String[] args) {

int n = 3;

int e = 2;

int A[] = { 1, 2, 1, 3 };

int B[] = { 1, 3, 1, 2 };

if (checkMirrorTree(n, e, A, B) == 1) {

System.out.println("Yes");

} else {

System.out.println("No");

}

}

}

**Output**

Yes

**My Implementation:**

Given two **n**-ary trees. Check if they are mirror images of each other or not. You are also given **e** denoting the number of edges in both trees, and two arrays, **A[]**and**B[]**. Each array has 2\*e space separated values u,v denoting an edge from u to v for the both trees.

**Example 1:**

**Input:**

**n =** 3, **e =** 2

**A[] =** {1, 2, 1, 3}

**B[] =** {1, 3, 1, 2}

**Output:**

1

**Explanation:**

1 1

/ \ / \

2 3 3 2

As we can clearly see, the second tree

is mirror image of the first.

**Example 2:**

**Input:**

**n =** 3, **e =** 2

**A[] =** {1, 2, 1, 3}

**B[] =** {1, 2, 1, 3}

**Output:**

0

**Explanation:**

1 1

/ \ / \

2 3 2 3

As we can clearly see, the second tree

isn't mirror image of the first.

**Your Task:**  
You don't need to read input or print anything. Your task is to complete the function **checkMirrorTree()** which takes 2 Integers n, and e;  and two arrays A[] and B[] of size 2\*e as input and returns 1 if the trees are mirror images of each other and 0 if not.

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

**Constraints:**  
1 <= n,e <= 105

Below Implementation is based on level order traversal:

class Solution {

public:

int checkMirrorTree(int n, int e, int A[], int B[]) {

queue<int> q;

stack<pair<int, int>> st;

int i=0,j=0,res=1;

q.push(A[0]);

while(!q.empty() && i<e){

int n = q.size();

while(n--){

int a = q.front();

q.pop();

while(i<e && A[2\*i]==a){

st.push(make\_pair(A[2\*i],A[2\*i+1]));

q.push(A[2\*i+1]);

i++;

}

}

while(!st.empty()){

pair<int, int> p = st.top();

st.pop();

if(p.first!=B[2\*j] || p.second!=B[2\*j+1]){

//cout<<p.first<<" "<<p.second<<" "<<j<<endl;

res = 0;

break;

}

j++;

}

if(!res)

break;

}

return res;

}

};

# 187. Sum of Nodes on the Longest path from root to leaf node

Given a binary tree of size **N.** Your task is to complete the function **sumOfLongRootToLeafPath(),** that find the sum of all nodes on the longest path from root to leaf node.  
If two or more paths compete for the longest path, then the path having maximum sum of nodes is being considered.

**Example 1:**

**Input:**

4

/ \

2 5

/ \ / \

7 1 2 3

/

6

**Output:** 13

**Explanation:**

**4**

/ \

**2** 5

/ \ / \

7 **1** 2 3

/

**6**

The highlighted nodes **(4, 2, 1, 6)** above are

part of the longest root to leaf path having

sum = (4 + 2 + 1 + 6) = 13

**Example 2:**

**Input:**

  1

  / \

  2 3

  / \ / \

  4 5 6 7

**Output:** 11

**Your Task:**  
You don't need to read input or print anything. Your task is to complete the function **sumOfLongRootToLeafPath()**which takes root node of the tree as input parameter and returns an integer denoting the sum of the longest root to leaf path of the tree. If the tree is empty, return 0.

**Expected Time Complexity:** O(N)  
**Expected Auxiliary Space:** O(N)

**Constraints:**  
1 <= Number of nodes <= 104  
1 <= Data of a node <= 104

## Solution:

**Approach:** Recursively find the length and sum of nodes of each root to leaf path and accordingly update the maximum sum.  
**Algorithm:** 

**sumOfLongRootToLeafPath(root, sum, len, maxLen, maxSum)**

if root == NULL

if maxLen < len

maxLen = len

maxSum = sum

else if maxLen == len && maxSum is less than sum

maxSum = sum

return

sumOfLongRootToLeafPath(root-left, sum + root-data,

len + 1, maxLen, maxSum)

sumOfLongRootToLeafPath(root-right, sum + root-data,

len + 1, maxLen, maxSum)

**sumOfLongRootToLeafPathUtil(root)**

if (root == NULL)

return 0

Declare maxSum = Minimum Integer

Declare maxLen = 0

sumOfLongRootToLeafPath(root, 0, 0, maxLen, maxSum)

return maxSum

// C++ implementation to find the sum of nodes

// on the longest path from root to leaf node

#include <bits/stdc++.h>

using namespace std;

// Node of a binary tree

struct Node {

int data;

Node\* left, \*right;

};

// function to get a new node

Node\* getNode(int data)

{

// allocate memory for the node

Node\* newNode = (Node\*)malloc(sizeof(Node));

// put in the data

newNode->data = data;

newNode->left = newNode->right = NULL;

return newNode;

}

// function to find the sum of nodes on the

// longest path from root to leaf node

void sumOfLongRootToLeafPath(Node\* root, int sum,

int len, int& maxLen, int& maxSum)

{

// if true, then we have traversed a

// root to leaf path

if (!root) {

// update maximum length and maximum sum

// according to the given conditions

if (maxLen < len) {

maxLen = len;

maxSum = sum;

} else if (maxLen == len && maxSum < sum)

maxSum = sum;

return;

}

// recur for left subtree

sumOfLongRootToLeafPath(root->left, sum + root->data,

len + 1, maxLen, maxSum);

// recur for right subtree

sumOfLongRootToLeafPath(root->right, sum + root->data,

len + 1, maxLen, maxSum);

}

// utility function to find the sum of nodes on

// the longest path from root to leaf node

int sumOfLongRootToLeafPathUtil(Node\* root)

{

// if tree is NULL, then sum is 0

if (!root)

return 0;

int maxSum = INT\_MIN, maxLen = 0;

// finding the maximum sum 'maxSum' for the

// maximum length root to leaf path

sumOfLongRootToLeafPath(root, 0, 0, maxLen, maxSum);

// required maximum sum

return maxSum;

}

// Driver program to test above

int main()

{

// binary tree formation

Node\* root = getNode(4); /\* 4 \*/

root->left = getNode(2); /\* / \ \*/

root->right = getNode(5); /\* 2 5 \*/

root->left->left = getNode(7); /\* / \ / \ \*/

root->left->right = getNode(1); /\* 7 1 2 3 \*/

root->right->left = getNode(2); /\* / \*/

root->right->right = getNode(3); /\* 6 \*/

root->left->right->left = getNode(6);

cout << "Sum = "

<< sumOfLongRootToLeafPathUtil(root);

return 0;

}

**Output**

Sum = 13

**Time Complexity:** O(n)

**Another Approach:**Using level order traversal

1. Create a structure containing the current Node, level and sum in the path.
2. Push the root element with level 0 and sum as the root’s data.
3. Pop the front element and update the maximum level sum and maximum level if needed.
4. Push the left and right nodes if exists.
5. Do the same for all the nodes in tree.

#include <bits/stdc++.h>

using namespace std;

//Building a tree node having left and right pointers set to null initially

struct Node

{

Node\* left;

Node\* right;

int data;

//constructor to set the data of the newly created tree node

Node(int element){

data = element;

this->left = nullptr;

this->right = nullptr;

}

};

int longestPathLeaf(Node\* root){

/\* structure to store current Node,it's level and sum in the path\*/

struct Element{

Node\* data;

int level;

int sum;

};

/\*

maxSumLevel stores maximum sum so far in the path

maxLevel stores maximum level so far

\*/

int maxSumLevel = root->data,maxLevel = 0;

/\* queue to implement level order traversal \*/

list<Element> que;

Element e;

/\* Each element variable stores the current Node, it's level, sum in the path \*/

e.data = root;

e.level = 0;

e.sum = root->data;

/\* push the root element\*/

que.push\_back(e);

/\* do level order traversal on the tree\*/

while(!que.empty()){

Element front = que.front();

Node\* curr = front.data;

que.pop\_front();

/\* if the level of current front element is greater than the maxLevel so far then update maxSum\*/

if(front.level > maxLevel){

maxSumLevel = front.sum;

maxLevel = front.level;

}

/\* if another path competes then update if the sum is greater than the previous path of same height\*/

else if(front.level == maxLevel && front.sum > maxSumLevel)

maxSumLevel = front.sum;

/\* push the left element if exists\*/

if(curr->left){

e.data = curr->left;

e.sum = e.data->data;

e.sum += front.sum;

e.level = front.level+1;

que.push\_back(e);

}

/\*push the right element if exists\*/

if(curr->right){

e.data = curr->right;

e.sum = e.data->data;

e.sum += front.sum;

e.level = front.level+1;

que.push\_back(e);

}

}

/\*return the answer\*/

return maxSumLevel;

}

//Helper function

int main() {

Node\* root = new Node(4);

root->left = new Node(2);

root->right = new Node(5);

root->left->left = new Node(7);

root->left->right = new Node(1);

root->right->left = new Node(2);

root->right->right = new Node(3);

root->left->right->left = new Node(6);

cout << longestPathLeaf(root) << "\n";

return 0;

}

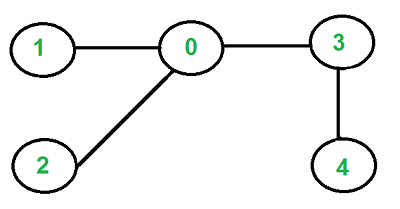
**Output**

13

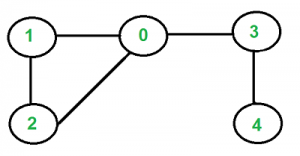
***Time Complexity:****O(N)*  
***Auxiliary Space:****O(N)*

# 188. Check if given graph is tree or not. [ IMP ]

Write a function that returns true if a given undirected graph is tree and false otherwise. For example, the following graph is a tree.



But the following graph is not a tree. 



## Solution:

An undirected graph is tree if it has following properties.   
1) There is no cycle.   
2) The graph is connected.  
For an undirected graph we can either use [BFS](https://www.geeksforgeeks.org/breadth-first-traversal-for-a-graph/)or [DFS](https://www.geeksforgeeks.org/depth-first-traversal-for-a-graph/)to detect above two properties.  
**How to detect cycle in an undirected graph?**   
We can either use BFS or DFS. For every visited vertex ‘v’, if there is an adjacent ‘u’ such that u is already visited and u is not parent of v, then there is a cycle in graph. If we don’t find such an adjacent for any vertex, we say that there is no cycle (See[Detect cycle in an undirected graph](https://www.geeksforgeeks.org/detect-cycle-undirected-graph/) for more details).  
**How to check for connectivity?**   
Since the graph is undirected, we can start BFS or DFS from any vertex and check if all vertices are reachable or not. If all vertices are reachable, then graph is connected, otherwise not.

// A C++ Program to check whether a graph is tree or not

#include<iostream>

#include <list>

#include <limits.h>

using namespace std;

// Class for an undirected graph

class Graph

{

int V; // No. of vertices

list<int> \*adj; // Pointer to an array for adjacency lists

bool isCyclicUtil(int v, bool visited[], int parent);

public:

Graph(int V); // Constructor

void addEdge(int v, int w); // to add an edge to graph

bool isTree(); // returns true if graph is tree

};

Graph::Graph(int V)

{

this->V = V;

adj = new list<int>[V];

}

void Graph::addEdge(int v, int w)

{

adj[v].push\_back(w); // Add w to v’s list.

adj[w].push\_back(v); // Add v to w’s list.

}

// A recursive function that uses visited[] and parent to

// detect cycle in subgraph reachable from vertex v.

bool Graph::isCyclicUtil(int v, bool visited[], int parent)

{

// Mark the current node as visited

visited[v] = true;

// Recur for all the vertices adjacent to this vertex

list<int>::iterator i;

for (i = adj[v].begin(); i != adj[v].end(); ++i)

{

// If an adjacent is not visited, then recur for

// that adjacent

if (!visited[\*i])

{

if (isCyclicUtil(\*i, visited, v))

return true;

}

// If an adjacent is visited and not parent of current

// vertex, then there is a cycle.

else if (\*i != parent)

return true;

}

return false;

}

// Returns true if the graph is a tree, else false.

bool Graph::isTree()

{

// Mark all the vertices as not visited and not part of

// recursion stack

bool \*visited = new bool[V];

for (int i = 0; i < V; i++)

visited[i] = false;

// The call to isCyclicUtil serves multiple purposes.

// It returns true if graph reachable from vertex 0

// is cyclcic. It also marks all vertices reachable

// from 0.

if (isCyclicUtil(0, visited, -1))

return false;

// If we find a vertex which is not reachable from 0

// (not marked by isCyclicUtil(), then we return false

for (int u = 0; u < V; u++)

if (!visited[u])

return false;

return true;

}

// Driver program to test above functions

int main()

{

Graph g1(5);

g1.addEdge(1, 0);

g1.addEdge(0, 2);

g1.addEdge(0, 3);

g1.addEdge(3, 4);

g1.isTree()? cout << "Graph is Tree\n":

cout << "Graph is not Tree\n";

Graph g2(5);

g2.addEdge(1, 0);

g2.addEdge(0, 2);

g2.addEdge(2, 1);

g2.addEdge(0, 3);

g2.addEdge(3, 4);

g2.isTree()? cout << "Graph is Tree\n":

cout << "Graph is not Tree\n";

return 0;

}

**Output:**

Graph is Tree

Graph is not Tree

# 189. [Find Largest subtree sum in a tree](https://www.geeksforgeeks.org/find-largest-subtree-sum-tree/)

Given a binary tree, task is to find subtree with maximum sum in tree.  
**Examples:** 

Input : 1

/ \

2 3

/ \ / \

4 5 6 7

Output : 28

As all the tree elements are positive,

the largest subtree sum is equal to

sum of all tree elements.

Input : 1

/ \

-2 3

/ \ / \

4 5 -6 2

Output : 7

Subtree with largest sum is : -2

/ \

4 5

Also, entire tree sum is also 7.

## Solution:

**Approach :**Do post order traversal of the binary tree. At every node, find left subtree value and right subtree value recursively. The value of subtree rooted at current node is equal to sum of current node value, left node subtree sum and right node subtree sum. Compare current subtree sum with overall maximum subtree sum so far.

**Implementation :**

// C++ program to find largest subtree

// sum in a given binary tree.

#include <bits/stdc++.h>

using namespace std;

// Structure of a tree node.

struct Node {

int key;

Node \*left, \*right;

};

// Function to create new tree node.

Node\* newNode(int key)

{

Node\* temp = new Node;

temp->key = key;

temp->left = temp->right = NULL;

return temp;

}

// Helper function to find largest

// subtree sum recursively.

int findLargestSubtreeSumUtil(Node\* root, int& ans)

{

// If current node is null then

// return 0 to parent node.

if (root == NULL)

return 0;

// Subtree sum rooted at current node.

int currSum = root->key +

findLargestSubtreeSumUtil(root->left, ans)

+ findLargestSubtreeSumUtil(root->right, ans);

// Update answer if current subtree

// sum is greater than answer so far.

ans = max(ans, currSum);

// Return current subtree sum to

// its parent node.

return currSum;

}

// Function to find largest subtree sum.

int findLargestSubtreeSum(Node\* root)

{

// If tree does not exist,

// then answer is 0.

if (root == NULL)

return 0;

// Variable to store maximum subtree sum.

int ans = INT\_MIN;

// Call to recursive function to

// find maximum subtree sum.

findLargestSubtreeSumUtil(root, ans);

return ans;

}

// Driver function

int main()

{

/\*

1

/ \

/ \

-2 3

/ \ / \

/ \ / \

4 5 -6 2

\*/

Node\* root = newNode(1);

root->left = newNode(-2);

root->right = newNode(3);

root->left->left = newNode(4);

root->left->right = newNode(5);

root->right->left = newNode(-6);

root->right->right = newNode(2);

cout << findLargestSubtreeSum(root);

return 0;

}

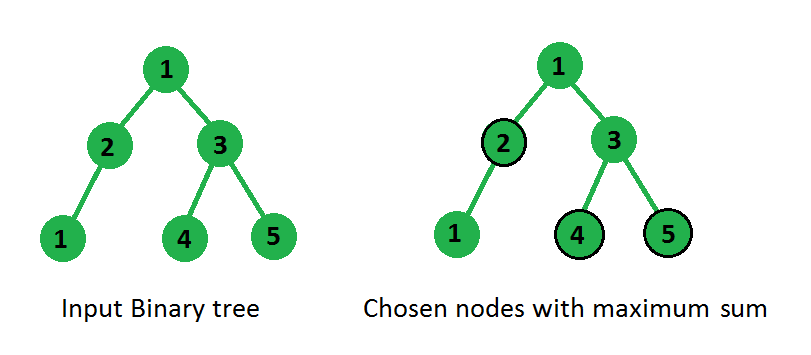
**Output:**

7

**Time Complexity:**O(n), where n is number of nodes.   
**Auxiliary Space:**O(n), function call stack size.

# 190. Maximum Sum of nodes in Binary tree such that no two are adjacent

Given a binary tree with a value associated with each node, we need to choose a subset of these nodes such that sum of chosen nodes is maximum under a constraint that no two chosen node in subset should be directly connected that is, if we have taken a node in our sum then we can’t take its any children or parents in consideration and vice versa.



**Example 1:**

**Input:**

11

  / \

  1 2

**Output:** 11

**Explanation:** The maximum sum is sum of

node 11.

**Example 2:**

**Input:**

1

**/ \**

2 3

  / / \

  4 5 6

**Output:** 16

**Explanation:** The maximum sum is sum of

nodes 1 4 5 6, i.e 16. These nodes are

non adjacent.

**Your Task:**  
You don't need to read input or print anything. You just have to complete function **getMaxSum()**which accepts root node of the tree as parameter and returns the maximum sum as described.

**Expected Time Complexity:**O(Number of nodes in the tree).  
**Expected Auxiliary Space:**O(Height of the Tree).

**Constraints:**  
1 ≤ Number of nodes in the tree ≤ 10000

1 ≤ Value of each node ≤ 100000

## Solution:

**Method 1**  
We can solve this problem by considering the fact that both node and its children can’t be in sum at the same time, so when we take a node into our sum we will call recursively for its grandchildren or if we don’t take this node then we will call for all its children nodes and finally we will choose maximum from both of the results.   
It can be seen easily that the above approach can lead to solving the same subproblem many times, for example in the above diagram node 1 calls node 4 and 5 when its value is chosen and node 3 also calls them when its value is not chosen so these nodes are processed more than once. We can stop solving these nodes more than once by memoizing the result at all nodes.   
In the below code, a map is used for memorizing the result which stores the result of the complete subtree rooted at a node in the map so that if it is called again, the value is not calculated again instead stored value from the map is returned directly.

Please see the below code for a better understanding.

// C++ program to find maximum sum from a subset of

// nodes of binary tree

#include <bits/stdc++.h>

using namespace std;

/\* A binary tree node structure \*/

struct node

{

int data;

struct node \*left, \*right;

};

/\* Utility function to create a new Binary Tree node \*/

struct node\* newNode(int data)

{

struct node \*temp = new struct node;

temp->data = data;

temp->left = temp->right = NULL;

return temp;

}

// Declaration of methods

int sumOfGrandChildren(node\* node);

int getMaxSum(node\* node);

int getMaxSumUtil(node\* node, map<struct node\*, int>& mp);

// method returns maximum sum possible from subtrees rooted

// at grandChildrens of node 'node'

int sumOfGrandChildren(node\* node, map<struct node\*, int>& mp)

{

int sum = 0;

// call for children of left child only if it is not NULL

if (node->left)

sum += getMaxSumUtil(node->left->left, mp) +

getMaxSumUtil(node->left->right, mp);

// call for children of right child only if it is not NULL

if (node->right)

sum += getMaxSumUtil(node->right->left, mp) +

getMaxSumUtil(node->right->right, mp);

return sum;

}

// Utility method to return maximum sum rooted at node 'node'

int getMaxSumUtil(node\* node, map<struct node\*, int>& mp)

{

if (node == NULL)

return 0;

// If node is already processed then return calculated

// value from map

if (mp.find(node) != mp.end())

return mp[node];

// take current node value and call for all grand children

int incl = node->data + sumOfGrandChildren(node, mp);

// don't take current node value and call for all children

int excl = getMaxSumUtil(node->left, mp) +

getMaxSumUtil(node->right, mp);

// choose maximum from both above calls and store that in map

mp[node] = max(incl, excl);

return mp[node];

}

// Returns maximum sum from subset of nodes

// of binary tree under given constraints

int getMaxSum(node\* node)

{

if (node == NULL)

return 0;

map<struct node\*, int> mp;

return getMaxSumUtil(node, mp);

}

// Driver code to test above methods

int main()

{

node\* root = newNode(1);

root->left = newNode(2);

root->right = newNode(3);

root->right->left = newNode(4);

root->right->right = newNode(5);

root->left->left = newNode(1);

cout << getMaxSum(root) << endl;

return 0;

}

**Output**

11

This article is contributed by Utkarsh Trivedi.

**Method 2 (Using pair)**   
Return a pair for each node in the binary tree such that the first of the pair indicates maximum sum when the data of a node is included and the second indicates maximum sum when the data of a particular node is not included.

// C++ program to find maximum sum in Binary Tree

// such that no two nodes are adjacent.

#include<iostream>

using namespace std;

class Node

{

public:

int data;

Node\* left, \*right;

Node(int data)

{

this->data = data;

left = NULL;

right = NULL;

}

};

pair<int, int> maxSumHelper(Node \*root)

{

if (root==NULL)

{

pair<int, int> sum(0, 0);

return sum;

}

pair<int, int> sum1 = maxSumHelper(root->left);

pair<int, int> sum2 = maxSumHelper(root->right);

pair<int, int> sum;

// This node is included (Left and right children

// are not included)

sum.first = sum1.second + sum2.second + root->data;

// This node is excluded (Either left or right

// child is included)

sum.second = max(sum1.first, sum1.second) +

max(sum2.first, sum2.second);

return sum;

}

int maxSum(Node \*root)

{

pair<int, int> res = maxSumHelper(root);

return max(res.first, res.second);

}

// Driver code

int main()

{

Node \*root= new Node(10);

root->left= new Node(1);

root->left->left= new Node(2);

root->left->left->left= new Node(1);

root->left->right= new Node(3);

root->left->right->left= new Node(4);

root->left->right->right= new Node(5);

cout << maxSum(root);

return 0;

}

**Output**

21

**Time complexity**: O(n)

**Method 3(**Using dynamic programming**)**

Store the maximum sum by including a node or excluding the node in a dp array or unordered map. Recursively calls for grandchildren of nodes if the node is included or calls for neighbours if the node is excluded.

// C++ program to find maximum sum in Binary Tree

// such that no two nodes are adjacent.

#include<iostream>

#include<bits/stdc++.h>

using namespace std;

class Node

{

public:

int data;

Node\* left, \*right;

Node(int data)

{

this->data = data;

left = NULL;

right = NULL;

}

};

//declare map /dp array as global

unordered\_map<Node \*, int> umap;

int maxSum(Node \* root)

{

//base case

if(!root)return 0;

//if the max sum from the node is already in map,return the value

if(umap[root])return umap[root];

//if the current node(root) is included in result

//then find maximum sum

int inc =root->data;

//if left of node exsists, addd their grandchildren

if(root->left)

{

inc+=maxSum(root->left->left) + maxSum(root->left->right);

}

//if right of node exsist,add their grandchildren

if(root->right)

{

inc+=maxSum(root->right->left)+maxSum(root->right->right);

}

//if the current node(root) is excluded, find the maximum sum

int ex=maxSum(root->left)+maxSum(root->right);

//store the maximum of including & excluding the node in map

umap[root]=max(inc,ex);

}

// Driver code

int main()

{

Node \*root= new Node(10);

root->left= new Node(1);

root->left->left= new Node(2);

root->left->left->left= new Node(1);

root->left->right= new Node(3);

root->left->right->left= new Node(4);

root->left->right->right= new Node(5);

cout << maxSum(root);

return 0;

}

**Output**

21

**Time complexity:**O(n)  
**Auxiliary** **Space:**O(n)

# 191. Print all "K" Sum paths in a Binary tree

A binary tree and a number k are given. Print every path in the tree with sum of the nodes in the path as k.   
A path can start from any node and end at any node and must be downward only, i.e. they need not be root node and leaf node; and negative numbers can also be there in the tree.  
**Examples:** 

Input : k = 5

Root of below binary tree:

1

/ \

3 -1

/ \ / \

2 1 4 5

/ / \ \

1 1 2 6

Output :

3 2

3 1 1

1 3 1

4 1

1 -1 4 1

-1 4 2

5

1 -1 5

## Solution:

Kindly note that this problem is significantly different from [finding k-sum path from root to leaves](https://www.geeksforgeeks.org/print-paths-root-specified-sum-binary-tree/). Here each node can be treated as root, hence the path can start and end at any node.  
The basic idea to solve the problem is to do a preorder traversal of the given tree. We also need a container (vector) to keep track of the path that led to that node. At each node we check if there are any path that sums to k, if any we print the path and proceed recursively to print each path.  
Below is the implementation of the same.

// C++ program to print all paths with sum k.

#include <bits/stdc++.h>

using namespace std;

// utility function to print contents of

// a vector from index i to it's end

void printVector(const vector<int>& v, int i)

{

for (int j = i; j < v.size(); j++)

cout << v[j] << " ";

cout << endl;

}

// binary tree node

struct Node {

int data;

Node \*left, \*right;

Node(int x)

{

data = x;

left = right = NULL;

}

};

// This function prints all paths that have sum k

void printKPathUtil(Node\* root, vector<int>& path, int k)

{

// empty node

if (!root)

return;

// add current node to the path

path.push\_back(root->data);

// check if there's any k sum path

// in the left sub-tree.

printKPathUtil(root->left, path, k);

// check if there's any k sum path

// in the right sub-tree.

printKPathUtil(root->right, path, k);

// check if there's any k sum path that

// terminates at this node

// Traverse the entire path as

// there can be negative elements too

int f = 0;

for (int j = path.size() - 1; j >= 0; j--) {

f += path[j];

// If path sum is k, print the path

if (f == k)

printVector(path, j);

}

// Remove the current element from the path

path.pop\_back();

}

// A wrapper over printKPathUtil()

void printKPath(Node\* root, int k)

{

vector<int> path;

printKPathUtil(root, path, k);

}

// Driver code

int main()

{

Node\* root = new Node(1);

root->left = new Node(3);

root->left->left = new Node(2);

root->left->right = new Node(1);

root->left->right->left = new Node(1);

root->right = new Node(-1);

root->right->left = new Node(4);

root->right->left->left = new Node(1);

root->right->left->right = new Node(2);

root->right->right = new Node(5);

root->right->right->right = new Node(2);

int k = 5;

printKPath(root, k);

return 0;

}

**Output:**   
 3 2

3 1 1

1 3 1

4 1

1 -1 4 1

-1 4 2

5

1 -1 5

**Time Complexity: O(n\*h\*h)**, as maximum size of path vector can be h

**Space Complexity: O(h)**

# 192. Find LCA in a Binary tree

Given a Binary Tree with all **unique** values and two nodes value**n1** and **n2**. The task is to find the**lowestcommon ancestor** of the given two nodes. We may assume that either both n1 and n2 are present in the tree or none of them is present.

**Example 1:**

**Input:**n1 = 2 , n2 = 3

1

/\

2 3

**Output:**

1

**Explanation:**LCA of 2 and 3 is 1.

**Example 2:**

**Input:**n1 = 3 , n2 = 4

5

/

2

/\

3 4

**Output:**

2

**Explanation:** LCA of 3 and 4 is 2.

**Your Task:**  
You don't have to read input or print anything. Your task is tocomplete the function **lca()**that takes nodes, **n1, and n2** as parameters and returns **LCA**node as output. Otherwise return a node with value **-1** if both **n1**and **n2**is not present in the tree.

**Expected Time Complexity:**O(N).  
**Expected Auxiliary Space:**O(Height of Tree).

**Constraints:**  
1 ≤ Number of nodes ≤ 105  
1 ≤ Data of a node ≤ 105

## Solution:

We have discussed an efficient solution to find [LCA in Binary Search Tree](https://www.geeksforgeeks.org/lowest-common-ancestor-in-a-binary-search-tree/). In Binary Search Tree, using BST properties, we can find LCA in O(h) time where h is the height of the tree. Such an implementation is not possible in Binary Tree as keys Binary Tree nodes don’t follow any order. The following are different approaches to find LCA in Binary Tree.

**Method 1 (By Storing root to n1 and root to n2 paths):**   
Following is a simple O(n) algorithm to find LCA of n1 and n2.   
**1)**Find a path from the root to n1 and store it in a vector or array.   
**2)**Find a path from the root to n2 and store it in another vector or array.   
**3)** Traverse both paths till the values in arrays are the same. Return the common element just before the mismatch.

Following is the implementation of the 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:***The 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)**   
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 a single traversal of Binary Tree and without extra storage for path arrays.   
The idea is to traverse the tree starting from the root. If any of the given keys (n1 and n2) matches with the root, then the root is LCA (assuming that both keys are present). If the root doesn’t match with any of the keys, we recur for the left and right subtree. The node which has one key present in its left subtree and the other key present in the right subtree is the LCA. If both keys lie in the left subtree, then the left subtree has LCA also, otherwise, LCA lies in the 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

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

Note that the above method assumes that keys are present in Binary Tree. If one key is present and the 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 bypassing two boolean variables v1 and v2. v1 is set as true when n1 is present in the tree and v2 is set as true if n2 is present in the 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

# 193. Find distance between 2 nodes in a Binary tree

Given a binary tree and two node values your task is to find the minimum distance between them.

**Example 1:**

**Input:**

1

  / \

  2 3

a = 2, b = 3

**Output:** 2

**Explanation:** The tree formed is:

      1

     /   \

   2     3

We need the distance between 2 and 3.

Being at node 2, we need to take two

steps ahead in order to reach node 3.

The path followed will be:

2 -> 1 -> 3. Hence, the result is 2.

**Your Task:**  
You don't need to read input or print anything. Your task is to complete the function **findDist()**which takes the **root**node of the Tree and the two node values **a** and **b** as input parameters and returns the minimum distance between the nodes represented by the two given node values.

**Expected Time Complexity:**O(N).  
**Expected Auxiliary Space:**O(Height of the Tree).

**Constraints:**  
1 <= Number of nodes <= 104  
1 <= Data of a node <= 105

## Solution:

The distance between two nodes can be obtained in terms of [lowest common ancestor](https://www.geeksforgeeks.org/lowest-common-ancestor-binary-tree-set-1/). Following is the formula.

**Dist(n1, n2) = Dist(root, n1) + Dist(root, n2) - 2\*Dist(root, lca)**

'n1' and 'n2' are the two given keys

'root' is root of given Binary Tree.

'lca' is lowest common ancestor of n1 and n2

Dist(n1, n2) is the distance between n1 and n2.

Following is the implementation of the above approach. The implementation is adopted from the last code provided in [Lowest Common Ancestor Post](https://www.geeksforgeeks.org/lowest-common-ancestor-binary-tree-set-1/).

/\* C++ program to find distance between n1 and n2 using

one traversal \*/

#include <iostream>

using namespace std;

// A Binary Tree Node

struct Node

{

struct Node \*left, \*right;

int key;

};

// Utility function to create a new tree Node

Node\* newNode(int key)

{

Node \*temp = new Node;

temp->key = key;

temp->left = temp->right = NULL;

return temp;

}

// Returns level of key k if it is present in tree,

// otherwise returns -1

int findLevel(Node \*root, int k, int level)

{

// Base Case

if (root == NULL)

return -1;

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

// or right subtree, return true;

if (root->key == k)

return level;

int l = findLevel(root->left, k, level+1);

return (l != -1)? l : findLevel(root->right, k, level+1);

}

// This function returns pointer to LCA of two given

// values n1 and n2. It also sets d1, d2 and dist if

// one key is not ancestor of other

// d1 --> To store distance of n1 from root

// d2 --> To store distance of n2 from root

// lvl --> Level (or distance from root) of current node

// dist --> To store distance between n1 and n2

Node \*findDistUtil(Node\* root, int n1, int n2, int &d1,

int &d2, int &dist, int lvl)

{

// Base case

if (root == NULL) return NULL;

// If either n1 or n2 matches with root's key, report

// the presence by returning root (Note that if a key is

// ancestor of other, then the ancestor key becomes LCA

if (root->key == n1)

{

d1 = lvl;

return root;

}

if (root->key == n2)

{

d2 = lvl;

return root;

}

// Look for n1 and n2 in left and right subtrees

Node \*left\_lca = findDistUtil(root->left, n1, n2,

d1, d2, dist, lvl+1);

Node \*right\_lca = findDistUtil(root->right, n1, n2,

d1, d2, dist, lvl+1);

// If both of the above calls return Non-NULL, then

// one key is present in once subtree and other is

// present in other. So this node is the LCA

if (left\_lca && right\_lca)

{

dist = d1 + d2 - 2\*lvl;

return root;

}

// Otherwise check if left subtree or right subtree

// is LCA

return (left\_lca != NULL)? left\_lca: right\_lca;

}

// The main function that returns distance between n1

// and n2. This function returns -1 if either n1 or n2

// is not present in Binary Tree.

int findDistance(Node \*root, int n1, int n2)

{

// Initialize d1 (distance of n1 from root), d2

// (distance of n2 from root) and dist(distance

// between n1 and n2)

int d1 = -1, d2 = -1, dist;

Node \*lca = findDistUtil(root, n1, n2, d1, d2,

dist, 1);

// If both n1 and n2 were present in Binary

// Tree, return dist

if (d1 != -1 && d2 != -1)

return dist;

// If n1 is ancestor of n2, consider n1 as root

// and find level of n2 in subtree rooted with n1

if (d1 != -1)

{

dist = findLevel(lca, n2, 0);

return dist;

}

// If n2 is ancestor of n1, consider n2 as root

// and find level of n1 in subtree rooted with n2

if (d2 != -1)

{

dist = findLevel(lca, n1, 0);

return dist;

}

return -1;

}

// Driver program to test above functions

int main()

{

// Let us create binary tree given in the

// above example

Node \* root = newNode(1);

root->left = newNode(2);

root->right = newNode(3);

root->left->left = newNode(4);

root->left->right = newNode(5);

root->right->left = newNode(6);

root->right->right = newNode(7);

root->right->left->right = newNode(8);

cout << "Dist(4, 5) = " << findDistance(root, 4, 5);

cout << "nDist(4, 6) = " << findDistance(root, 4, 6);

cout << "nDist(3, 4) = " << findDistance(root, 3, 4);

cout << "nDist(2, 4) = " << findDistance(root, 2, 4);

cout << "nDist(8, 5) = " << findDistance(root, 8, 5);

return 0;

}

**Output:**

Dist(4, 5) = 2

Dist(4, 6) = 4

Dist(3, 4) = 3

Dist(2, 4) = 1

Dist(8, 5) = 5

***Time Complexity:*** Time complexity of the above solution is O(n) as the method does a single tree traversal.

**Better Solution :**  
We first find the LCA of two nodes. Then we find the distance from LCA to two nodes.

/\* C++ Program to find distance between n1 and n2

using one traversal \*/

#include <iostream>

using namespace std;

// A Binary Tree Node

struct Node {

struct Node \*left, \*right;

int key;

};

// Utility function to create a new tree Node

Node\* newNode(int key)

{

Node\* temp = new Node;

temp->key = key;

temp->left = temp->right = NULL;

return temp;

}

Node\* LCA(Node\* root, int n1, int n2)

{

// Your code here

if (root == NULL)

return root;

if (root->key == n1 || root->key == n2)

return root;

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

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

if (left != NULL && right != NULL)

return root;

if (left == NULL && right == NULL)

return NULL;

if (left != NULL)

return LCA(root->left, n1, n2);

return LCA(root->right, n1, n2);

}

// Returns level of key k if it is present in

// tree, otherwise returns -1

int findLevel(Node\* root, int k, int level)

{

if (root == NULL)

return -1;

if (root->key == k)

return level;

int left = findLevel(root->left, k, level + 1);

if (left == -1)

return findLevel(root->right, k, level + 1);

return left;

}

int findDistance(Node\* root, int a, int b)

{

// Your code here

Node\* lca = LCA(root, a, b);

int d1 = findLevel(lca, a, 0);

int d2 = findLevel(lca, b, 0);

return d1 + d2;

}

// Driver program to test above functions

int main()

{

// Let us create binary tree given in

// the above example

Node\* root = newNode(1);

root->left = newNode(2);

root->right = newNode(3);

root->left->left = newNode(4);

root->left->right = newNode(5);

root->right->left = newNode(6);

root->right->right = newNode(7);

root->right->left->right = newNode(8);

cout << "Dist(4, 5) = " << findDistance(root, 4, 5);

cout << "\nDist(4, 6) = " << findDistance(root, 4, 6);

cout << "\nDist(3, 4) = " << findDistance(root, 3, 4);

cout << "\nDist(2, 4) = " << findDistance(root, 2, 4);

cout << "\nDist(8, 5) = " << findDistance(root, 8, 5);

return 0;

}

**Output**

Dist(4, 5) = 2

Dist(4, 6) = 4

Dist(3, 4) = 3

Dist(2, 4) = 1

Dist(8, 5) = 5

Thanks to NILMADHAB MONDAL for suggesting this solution.

**Another Better Solution (one pass):**

We know that distance between two node(let suppose n1 and n2) = distance between LCA and n1 + distance between LCA and n2.

A general solution using above formula that may come to your mind is  :

*int findDistance(Node\* root, int n1, int n2) {*

*if (!root) return 0;*

*if (root->data == n1 || root->data == n2)*

*return 1;*

*int left = findDistance(root->left, n1, n2);*

*int right = findDistance(root->right, n1, n2);*

*if (left  && right)*

*return left + right;*

*else if (left || right)*

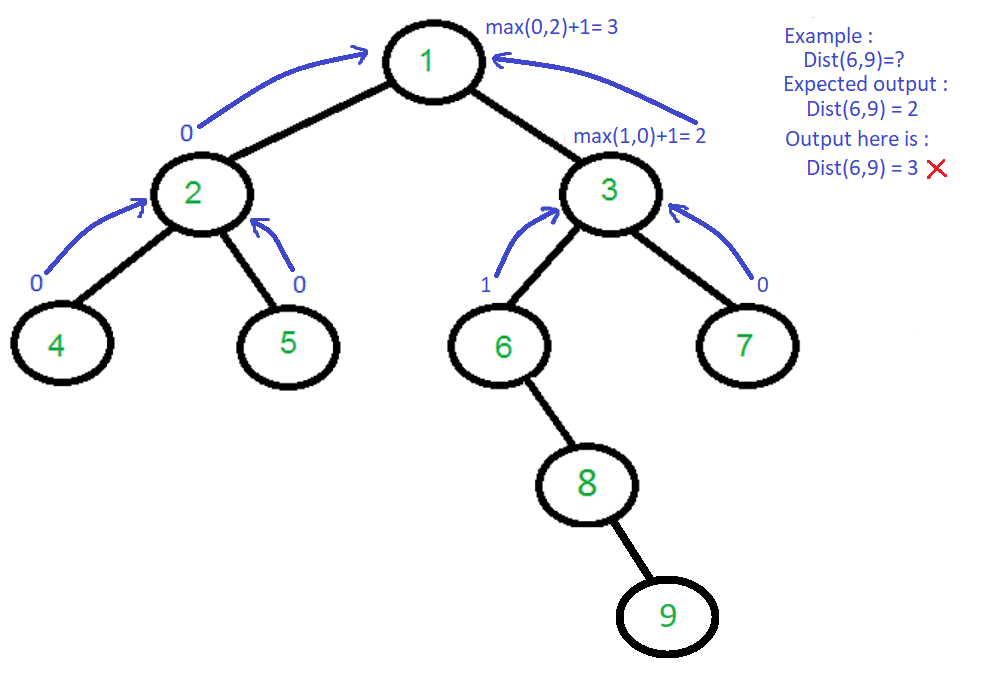
*return max(left, right) + 1;*

*return 0;*

*}*

But this solution has a flaw (a missing edge case)  when  n2 is Descendant of n1 or n1 is Descendant of n2.

Below is dry run of above code with edge case example :



In the above binary tree expected output is 2 but the function will give output as 3. This situation is overcome in the solution code given below :

**Note :** both n1 and n2 should be present in Binary Tree.

/\* C++ Program to find distance between n1 and n2

using one traversal \*/

#include <iostream>

using namespace std;

// A Binary Tree Node

struct Node {

struct Node \*left, \*right;

int key;

};

// Utility function to create a new tree Node

Node\* newNode(int key)

{

Node\* temp = new Node;

temp->key = key;

temp->left = temp->right = NULL;

return temp;

}

//Global variable to store distance

//between n1 and n2.

int ans;

//Function that finds distance between two node.

int \_findDistance(Node\* root, int n1, int n2)

{

if (!root) return 0;

int left = \_findDistance(root->left, n1, n2);

int right = \_findDistance(root->right, n1, n2);

//if any node(n1 or n2) is found

if (root->key == n1 || root->key == n2)

{

//check if their is any descendant(n1 or n2)

//if decendant exist then distance between descendant

//and current root will be our answer.

if (left || right)

{

ans = max(left, right);

return 0;

}

else

return 1;

}

//if current root is LCA of n1 and n2.

else if (left && right)

{

ans = left + right;

return 0;

}

//if their is a descendent(n1 or n2).

else if (left || right)

//increment its distance

return max(left, right) + 1;

//if neither n1 nor n2 exist as descendant.

return 0;

}

// The main function that returns distance between n1

// and n2.

int findDistance(Node\* root, int n1, int n2)

{

ans = 0;

\_findDistance(root, n1, n2);

return ans;

}

// Driver program to test above functions

int main()

{

// Let us create binary tree given in

// the above example

Node\* root = newNode(1);

root->left = newNode(2);

root->right = newNode(3);

root->left->left = newNode(4);

root->left->right = newNode(5);

root->right->left = newNode(6);

root->right->right = newNode(7);

root->right->left->right = newNode(8);

cout << "Dist(4, 5) = " << findDistance(root, 4, 5);

cout << "\nDist(4, 6) = " << findDistance(root, 4, 6);

cout << "\nDist(3, 4) = " << findDistance(root, 3, 4);

cout << "\nDist(2, 4) = " << findDistance(root, 2, 4);

cout << "\nDist(8, 5) = " << findDistance(root, 8, 5);

return 0;

}

**Output**

Dist(4, 5) = 2

Dist(4, 6) = 4

Dist(3, 4) = 3

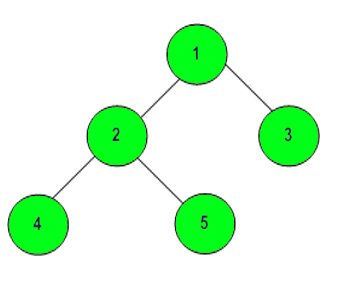
Dist(2, 4) = 1

Dist(8, 5) = 5

# 194. Kth Ancestor of node in a Binary tree

Given a binary tree of size  **N**, a **node** and a positive integer **k**., Your task is to complete the function **kthAncestor()**, the function should return the **kth** ancestor of the given node in the binary tree. If there does not exist any such ancestor then return -1.

**Example 1:**



**Input:**

K = 2

Node = 4

**Output:** 1

**Explanation:**

Since, K is 2 and node is 4, so we

first need to locate the node and

look k times its ancestors.

Here in this Case node 4 has 1 as his

2nd Ancestor aka the Root of the tree.

**Example 2:**

**Input:**

k=1

node=3

1

/ \

2 3

**Output:**

1

**Explanation:**

K=1 and node=3 ,Kth ancestor of node 3 is 1.

**Expected Time Complexity:** O(N)  
**Expected Auxiliary Space:** O(N)

**Constraints:**  
1<=N<=104  
1<= K <= 100

## Solution:

The idea to do this is to first traverse the binary tree and store the ancestor of each node in an array of size n. For example, suppose the array is anecestor[n]. Then at index i, ancestor[i] will store the ancestor of ith node. So, the 2nd ancestor of ith node will be ancestor[ancestor[i]] and so on. We will use this idea to calculate the kth ancestor of the given node. We can use [level order traversal](https://www.geeksforgeeks.org/level-order-tree-traversal/) to populate this array of ancestors.

Below is the implementation of above idea.

/\* C++ program to calculate Kth ancestor of given node \*/

#include <iostream>

#include <queue>

using namespace std;

// A Binary Tree Node

struct Node

{

int data;

struct Node \*left, \*right;

};

// function to generate array of ancestors

void generateArray(Node \*root, int ancestors[])

{

// There will be no ancestor of root node

ancestors[root->data] = -1;

// level order traversal to

// generate 1st ancestor

queue<Node\*> q;

q.push(root);

while(!q.empty())

{

Node\* temp = q.front();

q.pop();

if (temp->left)

{

ancestors[temp->left->data] = temp->data;

q.push(temp->left);

}

if (temp->right)

{

ancestors[temp->right->data] = temp->data;

q.push(temp->right);

}

}

}

// function to calculate Kth ancestor

int kthAncestor(Node \*root, int n, int k, int node)

{

// create array to store 1st ancestors

int ancestors[n+1] = {0};

// generate first ancestor array

generateArray(root,ancestors);

// variable to track record of number of

// ancestors visited

int count = 0;

while (node!=-1)

{

node = ancestors[node];

count++;

if(count==k)

break;

}

// print Kth ancestor

return node;

}

// Utility function to create a new tree node

Node\* newNode(int data)

{

Node \*temp = new Node;

temp->data = data;

temp->left = temp->right = NULL;

return temp;

}

// Driver program to test above functions

int main()

{

// Let us create binary tree shown in above diagram

Node \*root = newNode(1);

root->left = newNode(2);

root->right = newNode(3);

root->left->left = newNode(4);

root->left->right = newNode(5);

int k = 2;

int node = 5;

// print kth ancestor of given node

cout<<kthAncestor(root,5,k,node);

return 0;

}

**Output:**

1

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

**Method 2:** In this method first we will get an element whose ancestor has to be searched and from that node, we will decrement count one by one till we reach that ancestor node.   
for example –

consider the tree given below:-

         (1)

        /    \

      (4)   (2)

     /    \      \

   (3)  (7)    (6)

              \

              (8)

Then check if k=0 if yes then return that element as an ancestor else climb a level up and reduce k (k = k-1).  
**Initially k = 2**   
First we search for 8 then,   
at 8 => check if(k == 0) but k = 2 so k = k-1 => k = 2-1 = 1 and climb a level up i.e. at 7   
at 7 => check if(k == 0) but k = 1 so k = k-1 => k = 1-1 = 0 and climb a level up i.e. at 4   
at 4 => check if(k == 0) yes k = 0 return this node as ancestor.

// C++ program for finding

// kth ancestor of a particular node

#include<bits/stdc++.h>

using namespace std;

// Structure for a node

struct node{

int data;

struct node \*left, \*right;

node(int x)

{

data = x;

left = right = NULL;

}

};

// Program to find kth ancestor

bool ancestor(struct node\* root, int item, int &k)

{

if(root == NULL)

return false;

// Element whose ancestor is to be searched

if(root->data == item)

{

//reduce count by 1

k = k-1;

return true;

}

else

{

// Checking in left side

bool flag = ancestor(root->left,item,k);

if(flag)

{

if(k == 0)

{

// If count = 0 i.e. element is found

cout<<"["<<root->data<<"] ";

return false;

}

// if count !=0 i.e. this is not the

// ancestor we are searching for

// so decrement count

k = k-1;

return true;

}

// Similarly Checking in right side

bool flag2 = ancestor(root->right,item,k);

if(flag2)

{

if(k == 0)

{

cout<<"["<<root->data<<"] ";

return false;

}

k = k-1;

return true;

}

}

}

// Driver Code

int main()

{

struct node\* root = new node(1);

root->left = new node(4);

root->left->right = new node(7);

root->left->left = new node(3);

root->left->right->left = new node(8);

root->right = new node(2);

root->right->right = new node(6);

int item,k;

item = 3;

k = 1;

int loc = k;

bool flag = ancestor(root,item,k);

if(flag)

cout<<"Ancestor doesn't exist\n";

else

cout<<"is the "<<loc<<"th ancestor of ["<<

item<<"]"<<endl;

return 0;

}

**Output**

[4] is the 1th ancestor of [3]

**Approach 2:**

We have discussed a BFS based solution for this problem in our [previous](https://www.geeksforgeeks.org/kth-ancestor-node-binary-tree/) article. If you observe that solution carefully, you will see that the basic approach was to first find the node and then backtrack to the kth parent. The same thing can be done using recursive [DFS](https://www.geeksforgeeks.org/depth-first-traversal-for-a-graph/) without using an extra array.   
The idea of using DFS is to first find the given node in the tree and then backtrack k times to reach to the kth ancestor, once we have reached the kth parent, we will simply print the node and return NULL.   
Below is the implementation of above idea:

/\* C++ program to calculate Kth ancestor of given node \*/

#include <bits/stdc++.h>

using namespace std;

// A Binary Tree Node

struct Node

{

int data;

struct Node \*left, \*right;

};

// temporary node to keep track of Node returned

// from previous recursive call during backtrack

Node\* temp = NULL;

// recursive function to calculate Kth ancestor

Node\* kthAncestorDFS(Node \*root, int node , int &k)

{

// Base case

if (!root)

return NULL;

if (root->data == node||

(temp = kthAncestorDFS(root->left,node,k)) ||

(temp = kthAncestorDFS(root->right,node,k)))

{

if (k > 0)

k--;

else if (k == 0)

{

// print the kth ancestor

cout<<"Kth ancestor is: "<<root->data;

// return NULL to stop further backtracking

return NULL;

}

// return current node to previous call

return root;

}

}

// Utility function to create a new tree node

Node\* newNode(int data)

{

Node \*temp = new Node;

temp->data = data;

temp->left = temp->right = NULL;

return temp;

}

// Driver program to test above functions

int main()

{

// Let us create binary tree shown in above diagram

Node \*root = newNode(1);

root->left = newNode(2);

root->right = newNode(3);

root->left->left = newNode(4);

root->left->right = newNode(5);

int k = 2;

int node = 5;

// print kth ancestor of given node

Node\* parent = kthAncestorDFS(root,node,k);

// check if parent is not NULL, it means

// there is no Kth ancestor of the node

if (parent)

cout << "-1";

return 0;

}

**Output:**

Kth ancestor is: 1

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

**Approach 3:** First we find the path of given key data from the root and we will store it into a [vector](https://www.geeksforgeeks.org/vector-in-cpp-stl/) then we simply return the kth index of the vector from the last.   
Below is the implementation of the above approach:

// C++ implementation of the approach

#include <bits/stdc++.h>

using namespace std;

// Structure of Tree

struct node {

node \*left, \*right;

int data;

};

// To create a new node

node\* newNode(int data)

{

node\* temp = new node;

temp->left = temp->right = NULL;

temp->data = data;

return temp;

}

// Function to find the path from

// root to the target node

bool RootToNode(node\* root, int key, vector<int>& v)

{

if (root == NULL)

return false;

// Add current node to the path

v.push\_back(root->data);

// If current node is the target node

if (root->data == key)

return true;

// If the target node exists in

// the left or the right sub-tree

if (RootToNode(root->left, key, v)

|| RootToNode(root->right, key, v))

return true;

// Remove the last inserted node as

// it is not a part of the path

// from root to target

v.pop\_back();

return false;

}

// Driver code

int main()

{

struct node\* root = newNode(1);

root->left = newNode(2);

root->right = newNode(3);

root->left->left = newNode(4);

root->left->right = newNode(5);

root->right->left = newNode(6);

root->right->right = newNode(7);

// Given node

int target = 4;

// Vector to store the path from

// root to the given node

vector<int> v;

// Find the path from root to the target node

RootToNode(root, target, v);

int k = 2;

// Print the Kth ancestor

if (k > v.size() - 1 || k <= 0)

cout << -1;

else

cout << v[v.size() - 1 - k];

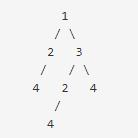
}

**Output:**

1

# 195. Find all Duplicate subtrees in a Binary tree [ IMP ]

Given a binary tree of size **N**, your task is to that find all duplicate subtrees from the given binary tree.  
  
**Example:**

**Input :** 

**Output :** 2 4

  4

**Explanation:** Above Trees are two

duplicate subtrees.i.e http://contribute.geeksforgeeks.org/wp-content/uploads/tree2-1.png and http://contribute.geeksforgeeks.org/wp-content/uploads/tree3.png

Therefore,you need to return above trees

root in the form of a list.

**Your Task:**  
You don't need to take input. Just complete the function**printAllDups()**that takes the root **node**as a parameter and returns an array of Node\*, which contains all the duplicate subtree.  
**Note:** Here the Output of every Node printed in the Pre-Order tree traversal format.

**Constraints:**  
1<=T<=100  
1<=N<=100

## Solution:

The idea is to use [hashing](https://www.geeksforgeeks.org/hashing-data-structure/). We store [inorder traversals](https://www.geeksforgeeks.org/tree-traversals-inorder-preorder-and-postorder/) of subtrees in a hash. Since simple inorder traversal cannot uniquely identify a tree, we use symbols like ‘(‘ and ‘)’ to represent NULL nodes.   
We pass an [Unordered Map in C++](https://www.geeksforgeeks.org/unordered_map-in-stl-and-its-applications/) as an argument to the helper function which recursively calculates inorder string and increases its count in map. If any string gets repeated, then it will imply duplication of the subtree rooted at that node so push that node in the Final result and return the vector of these nodes.

// C++ program to find averages of all levels

// in a binary tree.

#include <bits/stdc++.h>

using namespace std;

/\* A binary tree node has data, pointer to

left child and a pointer to right child \*/

struct Node {

int data;

struct Node\* left, \*right;

};

string inorder(Node\* node, unordered\_map<string, int>& m)

{

if (!node)

return "";

string str = "(";

str += inorder(node->left, m);

str += to\_string(node->data);

str += inorder(node->right, m);

str += ")";

// Subtree already present (Note that we use

// unordered\_map instead of unordered\_set

// because we want to print multiple duplicates

// only once, consider example of 4 in above

// subtree, it should be printed only once.

if (m[str] == 1)

cout << node->data << " ";

m[str]++;

return str;

}

// Wrapper over inorder()

void printAllDups(Node\* root)

{

unordered\_map<string, int> m;

inorder(root, m);

}

/\* Helper function that allocates a

new node with the given data and

NULL left and right pointers. \*/

Node\* newNode(int data)

{

Node\* temp = new Node;

temp->data = data;

temp->left = temp->right = NULL;

return temp;

}

// Driver code

int main()

{

Node\* root = NULL;

root = newNode(1);

root->left = newNode(2);

root->right = newNode(3);

root->left->left = newNode(4);

root->right->left = newNode(2);

root->right->left->left = newNode(4);

root->right->right = newNode(4);

printAllDups(root);

return 0;

}

**Output:**

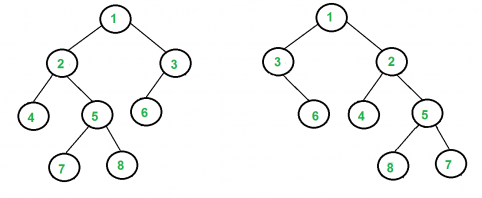
4 2

**Time Complexity:**O(N^2)  Since string copying takes O(n) extra time.

**Auxiliary Space:**O(N^2) Since we are hashing a string for each node and length of this string can be of the order N.

# 196. Tree Isomorphism Problem

Given two Binary Trees. Check whether they are Isomorphic or not.

**Note:**  
Two trees are called isomorphic if one can be obtained from another by a series of flips, i.e. by swapping left and right children of several nodes. Any number of nodes at any level can have their children swapped. Two empty trees are isomorphic.  
For example, the following two trees are isomorphic with the following sub-trees flipped: 2 and 3, NULL and 6, 7 and 8.  
[](https://media.geeksforgeeks.org/wp-content/cdn-uploads/ISomorphicTrees-e1368593305854.png)

**Example 1:**

**Input:**

**T1** 1 **T2:** 1

  / \ / \

  2 3 3 2

  / /

  44

**Output:** No

**Example 2:**

**Input:**

**T1** 1 **T2:** 1

  / \ / \

  2 3 3 2

  / \

  44

**Output:** Yes

**Your Task:**  
You don't need to read input or print anything. Your task is to complete the function**isomorphic()**that takesthe root nodes of both the Binary Trees as its input and returns True if the two trees are isomorphic. Else, it returns False. (The driver code will print Yes if the returned values are true, otherwise false.)

**Expected Time Complexity:**O(min(M, N)) where M and N are the sizes of the two trees.  
**Expected Auxiliary Space:**O(min(H1, H2)) where H1 and H2 are the heights of the two trees.

**Constraints:**  
1<=Number of nodes<=105

## Solution:

We simultaneously traverse both trees. Let the current internal nodes of two trees being traversed be **n1**and **n2** respectively. There are following two conditions for subtrees rooted with n1 and n2 to be isomorphic.   
**1)** Data of n1 and n2 is same.   
**2)**One of the following two is true for children of n1 and n2   
……**a)** Left child of n1 is isomorphic to left child of n2 and right child of n1 is isomorphic to right child of n2.   
……**b)** Left child of n1 is isomorphic to right child of n2 and right child of n1 is isomorphic to left child of n2.  
 // A C++ program to check if two given trees are isomorphic

#include <iostream>

using namespace std;

/\* A binary tree node has data, pointer to left and right children \*/

struct node

{

int data;

struct node\* left;

struct node\* right;

};

/\* Given a binary tree, print its nodes in reverse level order \*/

bool isIsomorphic(node\* n1, node \*n2)

{

// Both roots are NULL, trees isomorphic by definition

if (n1 == NULL && n2 == NULL)

return true;

// Exactly one of the n1 and n2 is NULL, trees not isomorphic

if (n1 == NULL || n2 == NULL)

return false;

if (n1->data != n2->data)

return false;

// There are two possible cases for n1 and n2 to be isomorphic

// Case 1: The subtrees rooted at these nodes have NOT been "Flipped".

// Both of these subtrees have to be isomorphic, hence the &&

// Case 2: The subtrees rooted at these nodes have been "Flipped"

return

(isIsomorphic(n1->left,n2->left) && isIsomorphic(n1->right,n2->right))||

(isIsomorphic(n1->left,n2->right) && isIsomorphic(n1->right,n2->left));

}

/\* Helper function that allocates a new node with the

given data and NULL left and right pointers. \*/

node\* newNode(int data)

{

node\* temp = new node;

temp->data = data;

temp->left = NULL;

temp->right = NULL;

return (temp);

}

/\* Driver program to test above functions\*/

int main()

{

// Let us create trees shown in above diagram

struct node \*n1 = newNode(1);

n1->left = newNode(2);

n1->right = newNode(3);

n1->left->left = newNode(4);

n1->left->right = newNode(5);

n1->right->left = newNode(6);

n1->left->right->left = newNode(7);

n1->left->right->right = newNode(8);

struct node \*n2 = newNode(1);

n2->left = newNode(3);

n2->right = newNode(2);

n2->right->left = newNode(4);

n2->right->right = newNode(5);

n2->left->right = newNode(6);

n2->right->right->left = newNode(8);

n2->right->right->right = newNode(7);

if (isIsomorphic(n1, n2) == true)

cout << "Yes";

else

cout << "No";

return 0;

}

**Output:**

Yes

**Time Complexity:** The above solution does a traversal of both trees. So time complexity is O(min(m,n)\*2) or O(min(m,n)) where m and n are number of nodes in given trees.