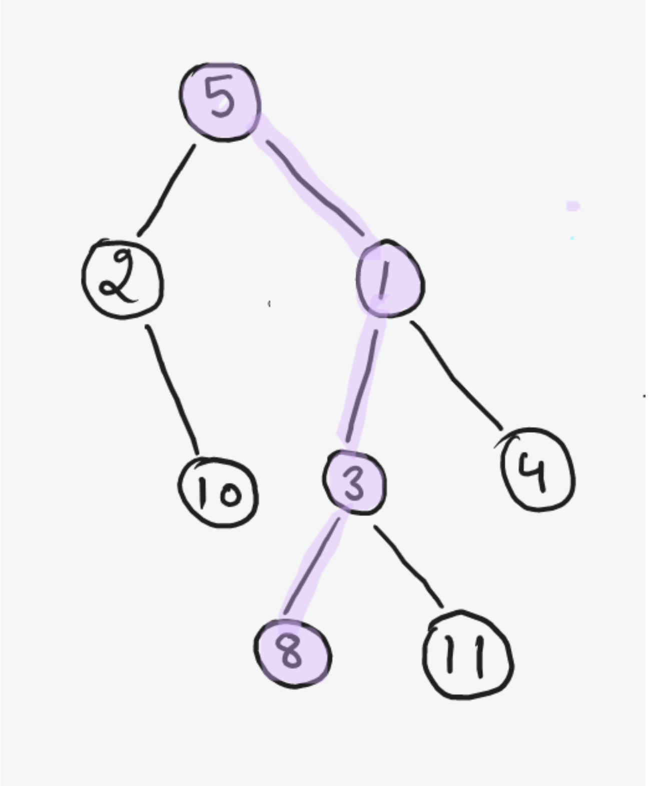
Maximum depth of a Binary Tree

Problem Statement: Find the Maximum Depth of Binary Tree. Maximum Depth is the count of nodes of the longest path from the root node to the leaf node.

Examples:

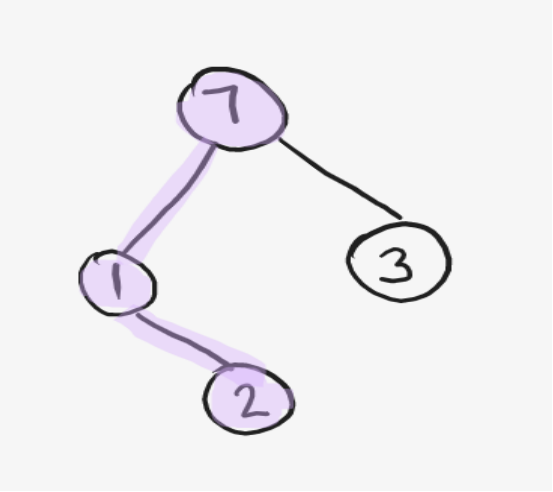
Input Format: Given the root of Binary Tree



Result: 4

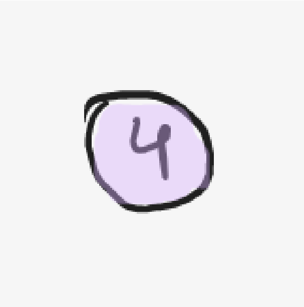
Explanation: Maximum Depth in this tree is 4 if we follow path 5 – 1 – 3 – 8 or 5 – 1 – 3 – 11

Input Format: Given the root of Binary Tree



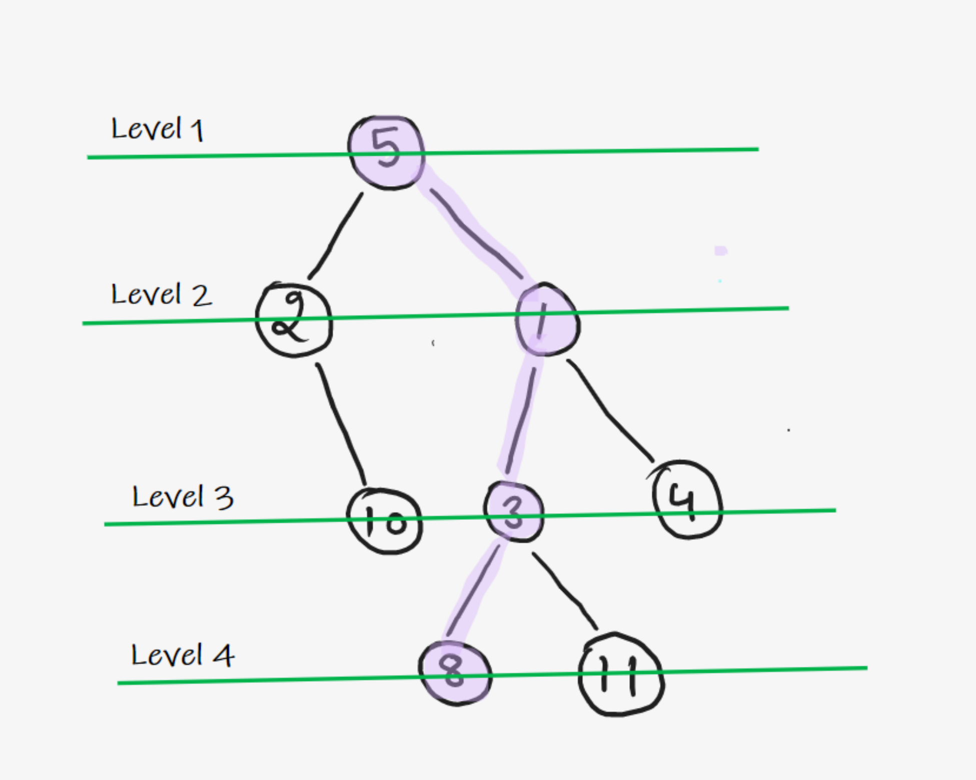
Result: 3
Explanation: Maximum Depth in this tree is 3 , if we follow path 7 – 1 – 2. If we follow 7 – 3 path then depth is 2 (not optimal)

Input Format: Given the root of Binary Tree



Result: 1
Explanation: Maximum Depth in this tree is 1 as there is only one node which is the root node.
Note: We are counting depth in terms of Node, if the question was given in terms of edges then the answer will be 0 in the above case.

Solution
Intuition + Approach: Using LEVEL ORDER TRAVERSAL
If we observe carefully, the depth of the Binary Tree is the number of levels in the binary tree. So, if we simply do a level order traversal on the binary tree and keep a count of the number of levels, it will be our answer.



In this example, if we start traversing the tree level-wise, then we can reach at max Level 4, so our answer is 4. Because the maximum depth we can achieve is indicated by the last level at which we can travel.

```
vector<vector<int>> levelorderTraversal(TreeNode *root)
{
   vector<vector<int>> ans;
   if(root == NULL) return ans;
   queue<TreeNode*> q;
   q.push(root);
   while(!q.empty()){
    vector<int> level;
    int size = q.size();
    for (int i = 0; i < size; i++)
    {
        TreeNode* node = q.front();
        q.pop();
        if(node->left != NULL)q.push(node->left);
        if(node->right != NULL)q.push(node->right);
        level.push_back(node->data);
    }
    ans.push_back(level);
   }
   return ans;
}
```

Time Complexity: O(N)
Space Complexity: O(N) (Queue data structure is used)

Solution 2:
Intuition: Recursively (Post Order Traversal)
If we have to do it recursively, then what we can think of is, If I have Maximum Depth of Left subtree and Maximum Depth of Right subtree then what will be the height or depth of the tree?
Exactly,

```
1 + max(depth of left subtree, depth of right subtree)
```

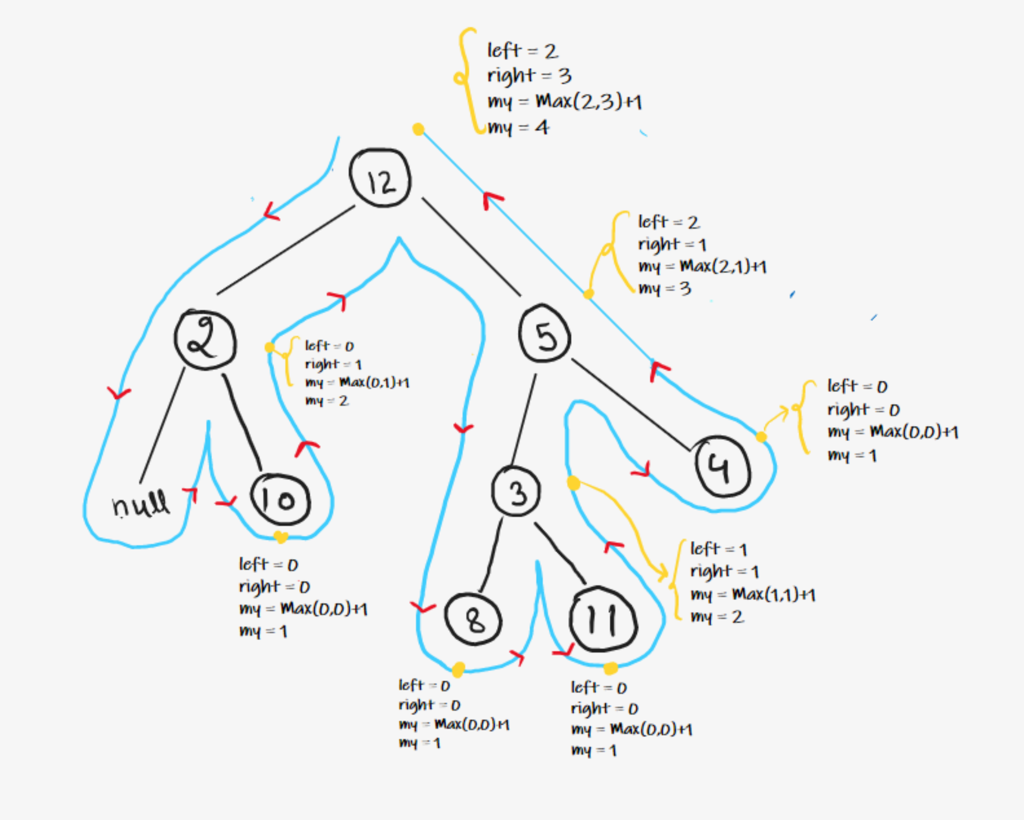
So, to calculate the Maximum Depth, we can simply take the maximum of the depths of the left and right subtree and add 1 to it.
Why take Maximum?? Because we need maximum depth so if we know left & right children's maximum depth then we'll definitely get to the maximum depth of the entire tree.
Approach :

- We start to travel recursively and do our work in Post Order.
- Reason behind using Post Order comes from our intuition , that if we know the result of left and right child then we can calculate the result using that.

- **This is exactly an indication of PostOrder, because in PostOrder we already calculated results for left and right children than we do it for current node.**
- **So for every node post order, we do Max(left result , right result) + 1 and return it to the previous call.**
- **Base Case is when root == null so we need to return 0;**

Dry Run :

In Post Order, we start to travel on the example given in the below diagram



- **Reach on Node 10 , Left child = null so 0 , Right child = null so 0 & add 1 for node 10 so max depth till node 10 is max(0,0) + 1 = 1.**
- **Reach on Node 2 , Left child = null so 0 , Right child = will give 1 & add 1 for node 2 so max depth till node 2 is max(0,1) + 1 = 2.**
- **Reach on Node 8 , Left child = null so 0 , Right child = null so 0 & add 1 for node 8 so max depth till node 8 is max(0,0) + 1 = 1.**
- **Reach on Node 11 , Left child = null so 0 , Right child = null so 0 & add 1 for node 11 so max depth till node 11 is max(0,0) + 1 = 1.**
- **Reach on Node 3 , Left child will give 1 , Right child = will give 1 & add 1 for node 3 so max depth till node 3 is max(1,1) + 1 = 2.**
- **Reach on Node 4 , Left child = null so 0 , Right child = null so 0 & add 1 for node 4 so max depth till node 4 is max(0,0) + 1 = 1.**
- **Reach on Node 5 , Left child will give 2 , Right child = will give 1 & add 1 for node 5 so max depth till node 5 is max(2,1) + 1 = 3.**

- **Reach on Node 12 , Left child will give 2 , Right child = will give 3 & add 1 for node 12 so max depth till node 12 is max(2,3) + 1 = 4.**
- **Hence 4 is our final ans.**

Code:

```
class Solution {
public:
    int maxDepth(TreeNode* root) {
        if(root == NULL) return 0;

        int lh = maxDepth(root->left);
        int rh = maxDepth(root->right);

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

Time Complexity: O(N)

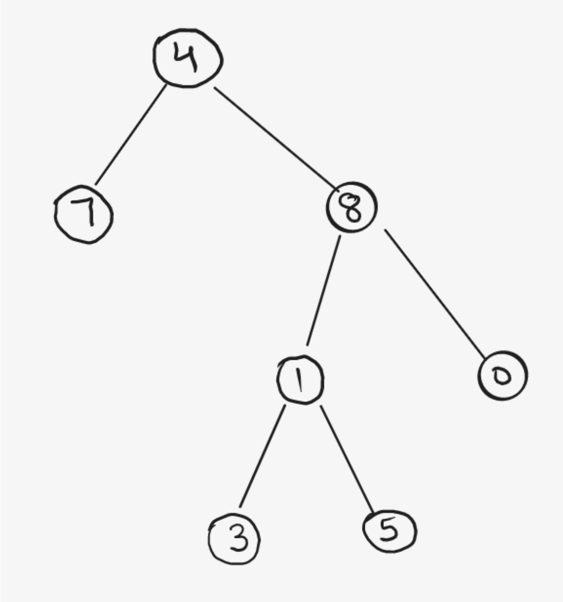
Space Complexity: O(1) Extra Space + O(H) Recursion Stack space, where "H" is the height of the binary tree.

Check if the Binary Tree is Balanced Binary Tree

Problem Statement: Check whether the given Binary Tree is a **Balanced Binary Tree** or not. A binary tree is balanced if, for all nodes in the tree, the difference between left and right subtree height is not more than 1.

Examples:

Input Format: Given the root of Binary Tree

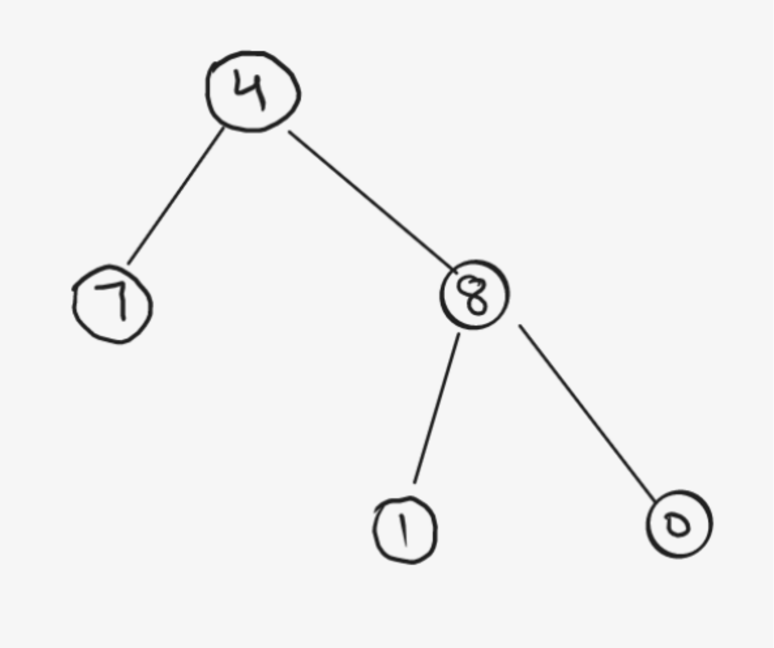


Binary Tree

Result: False

Explanation: At Node 4, Left Height = 1 & Right Height = 3, Absolute Difference is 2 which is greater than 1, Hence, not a balanced tree.

Input Format: Given the root of Binary Tree



Result: True

Explanation: All Nodes in the tree have an Absolute Difference of Left Height & Right Height not more than 1.

Solution

Solution 1: Naive approach

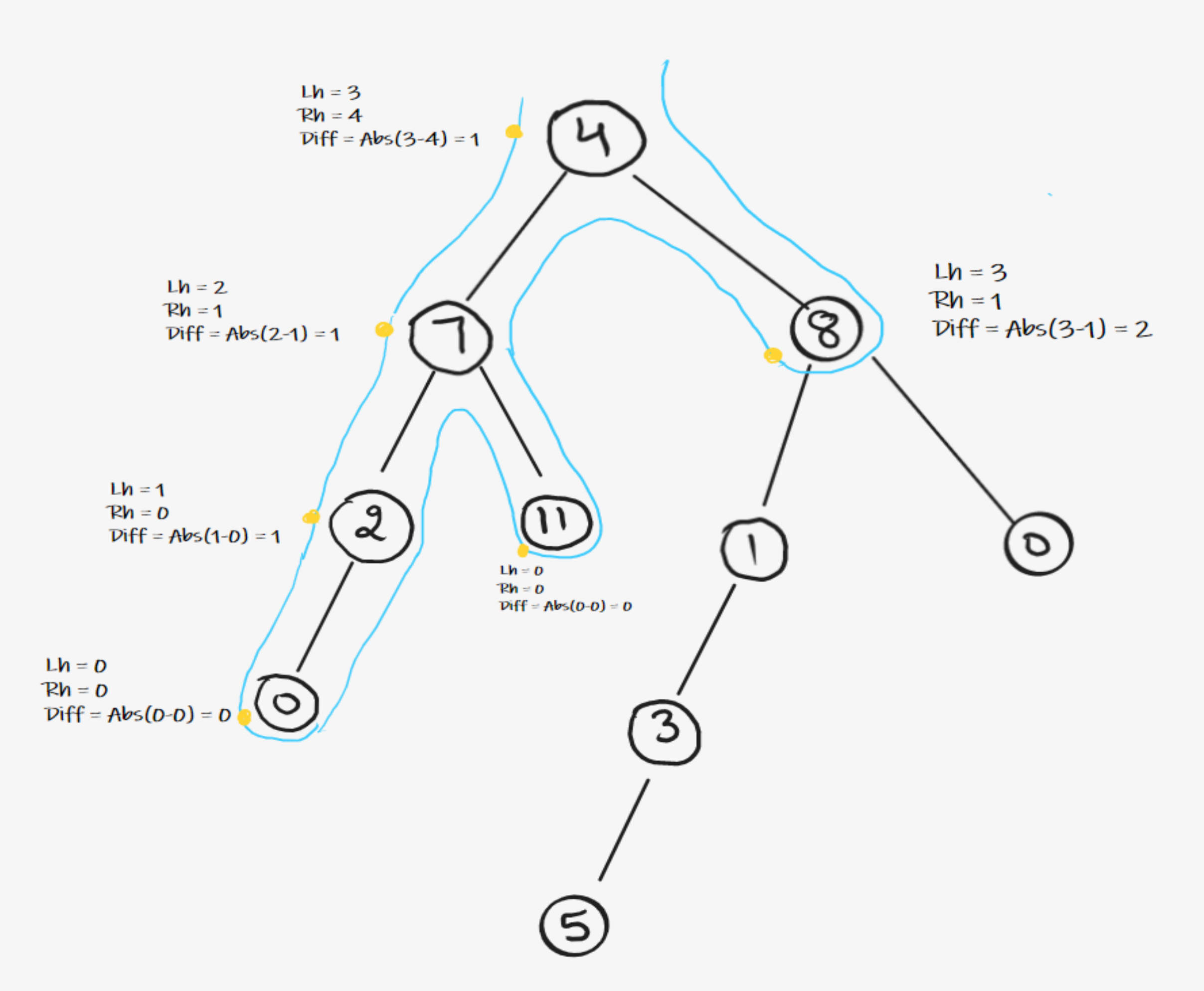
Intuition + Approach:

For a Balanced Binary Tree, Check left subtree height and right subtree height for every node present in the tree. Hence, traverse the tree recursively and calculate the height of left and right subtree from every node, and whenever the condition of Balanced tree violates, simply return false.

Condition for Balanced Binary Tree

For all Nodes , Absolute(Left Subtree Height – Right Subtree Height) <= 1

Dry-run :



Start traversing the tree, the example given in above diagram:

- Reach on **Node 4**, call Height Function , Left height = 3 , Right height = 4 so Absolute Difference between two is Abs(3 – 4) = 1.
- Reach on **Node 7**, call Height Function , Left height = 2 , Right height = 1 so Absolute Difference between two is Abs(2 – 1) = 1.
- Reach on **Node 2**, call Height Function , Left height = 1 , Right height = 0 so Absolute Difference between two is Abs(1 – 0) = 1.
- Reach on **Node 0**, call Height Function , Left height = 0 , Right height = 0 so Absolute Difference between two is Abs(0 – 0) = 0.
- Now, on **PostOrder of Node 0,** the left subtree (null) gives true & right subtree (null) gives true , as both are true , return true.
- Now, on **PostOrder of Node 2,** the left subtree (0) gives true & right subtree (null) gives true , as both are true , return true.

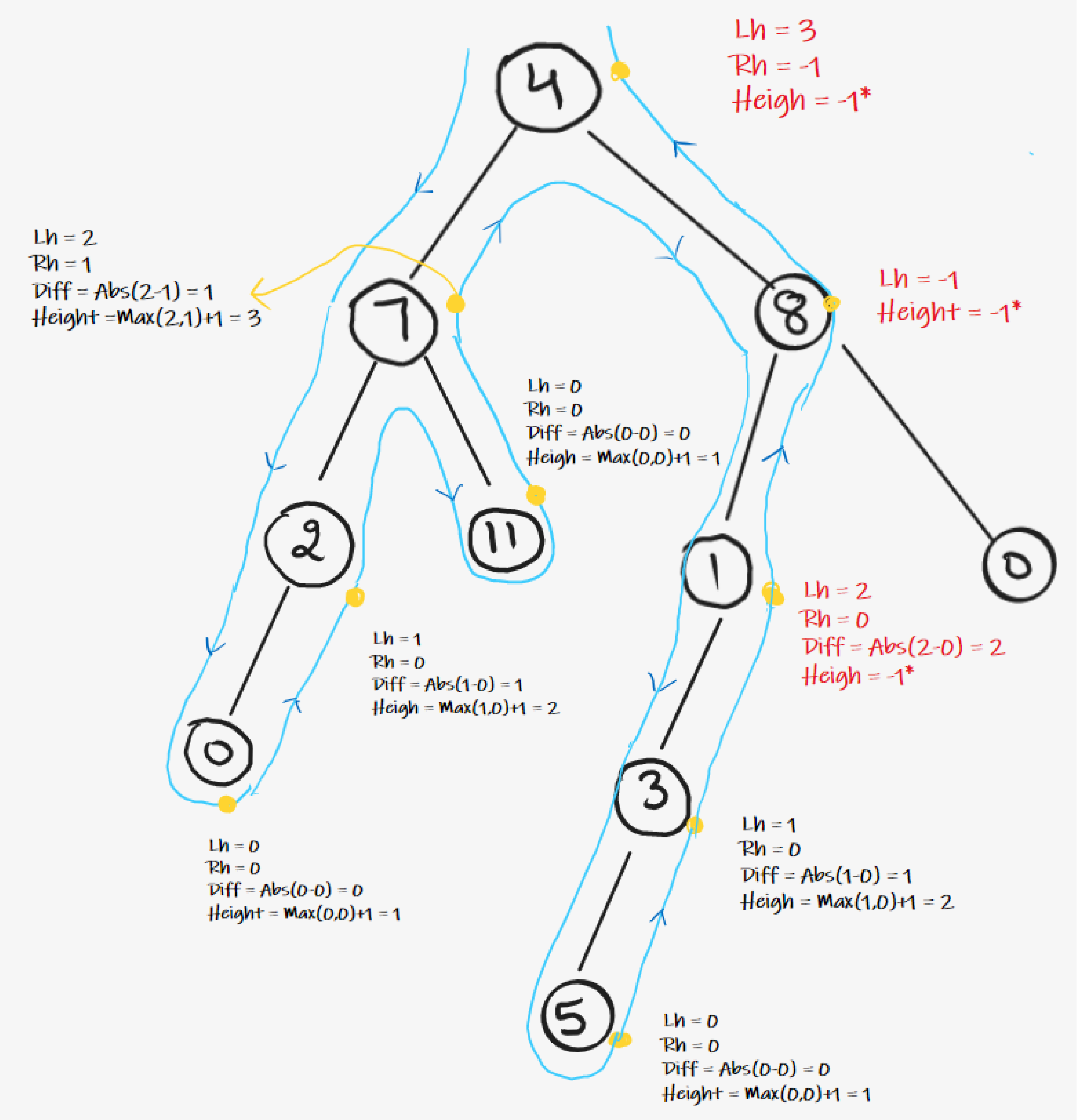
- Reach on **Node 11**, call Height Function , Left height = 0 , Right height = 0 so Absolute Difference between two is Abs(0 – 0) = 0.
- Now , on **PostOrder of Node 11,** the left subtree (null) gives true & right subtree (null) gives true , as both are true , return true.
- Now ,on **PostOrder of Node 7,** the left subtree (2) gives true & right subtree (11) gives true , as both are true , return true.
- Reach on **Node 8**, call Height Function , Left height = 3 , Right height = 1 so Absolute Difference between two is Abs(3 – 1) = 2. Here Condition violates , simply return false, no need to call further
- Now , on **PostOrder of Node 4,** the left subtree (7) gives true & right subtree (8) gives false , so any one of subtree gives false , return false.

Time Complexity: O(N*N) (For every node, Height Function is called which takes O(N) Time. Hence for every node it becomes N*N)
Space Complexity: O(1) (Extra Space) **+ O(H)** (Recursive Stack Space where **"H"** is the height of tree)

Solution 2: Post Order Traversal
Intuition: Can we optimize the above brute force solution? Which operation do you think can be skipped to optimize the time complexity?
Ain't we traversing the subtrees again and again in the above example?
Yes, so can we skip the repeated traversals?
What if we can make use of post-order traversal?
So, the idea is to use post-order traversal. Since, in postorder traversal, we first traverse the left and right subtrees and then visit the parent node, similarly instead of calculating the height of the left subtree and right subtree every time at the root node, use post-order traversal, and keep calculating the heights of the left and right subtrees and perform the validation.
Approach :

- Start traversing the tree recursively and do work in Post Order.
- For each call, caculate the height of the root node, and return it to previous calls.
- Simultaneously, in the Post Order of every node , Check for condition of balance as information of left and right subtree height is available.
- If it is balanced , simply return height of current node and if not then return -1.
- Whenever the subtree result is -1 , simply keep on returning -1.

Dry Run :
In Post Order, Start traversing the tree on the example given in below diagram



- Reach on **Node 0**, Left child = null so 0 Height , Right child = null so 0 Height , Difference is 0-0 = 0 , (0 <= 1) so return height , i.e. Max(0,0) + 1 = 1.
- Reach on **Node 2** , Left subtree height = 1 , Right subtree height = 0, Difference is 1-0 = 1 , (1 <= 1) so return height , i.e. Max(1,0) + 1 = 2.
- Reach on **Node 11** , Left child = null so 0 Height , Right child = null so 0 Height , Difference is 0-0 = 0 , (0 <= 1) so return height , i.e. Max(0,0) + 1 = 1.
- Reach on **Node 7** , Left subtree height = 2 , Right subtree height = 1, Difference is 2-1 = 1 , (1 <= 1) so return height , i.e. Max(2,1) + 1 = 3.

- Reach on **Node 5** , Left child = null so 0 Height , Right child = null so 0 Height , Difference is 0-0 = 0 , (0 <= 1) so return height , i.e. Max(0,0) + 1 = 1.
- Reach on **Node 3** , Left subtree height = 1 , Right subtree height = 0, Difference is 1-0 = 1 , (1 <= 1) so return height , i.e. Max(1,0) + 1 = 2.
- Reach on **Node 1** , Left subtree height = 2 , Right subtree height = 0, Difference is 2-0 = 2 , (2 > 1) i.e. Tree is **not Balanced** , so return -1.
- Reach on **Node 8** , Left subtree height = **-1** , indicates that tree is not balanced, simply return -1;
- Reach on **Node 4** , Left subtree height = 3 , Right subtree height = **-1**, therefore indicates that tree is not balanced , simply return -1;
- In the Main function , If the final Height of tree is -1 return false as tree is not balanced , else return true.

Code:

```
   int maxDepth(TreeNode *root){
    if(root == NULL) return 0;

    int leftTree = maxDepth(root->left);
    int rightTree = maxDepth(root->right);

    if(leftTree == -1 || rightTree == -1) return -1;
    if(abs(leftTree-rightTree)>1) return -1;
    return max(leftTree, rightTree)+1;

}
```

Time Complexity: O(N)

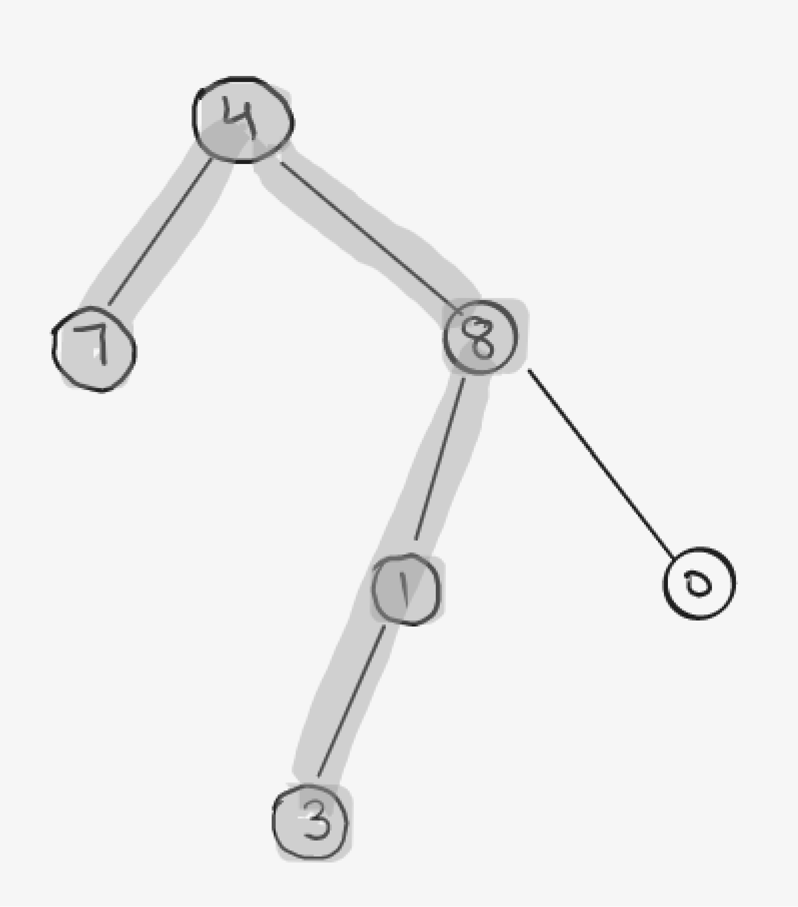
Space Complexity: O(1) Extra Space + O(H) Recursion Stack space (Where “H” is the height of binary tree)

Calculate the Diameter of a Binary Tree

Problem Statement: Find the Diameter of a Binary Tree. **Diameter** is the length of the longest path between any 2 nodes in the tree and this path may or may not pass from the root.

Example 1:

Input Format: Given the root of Binary Tree

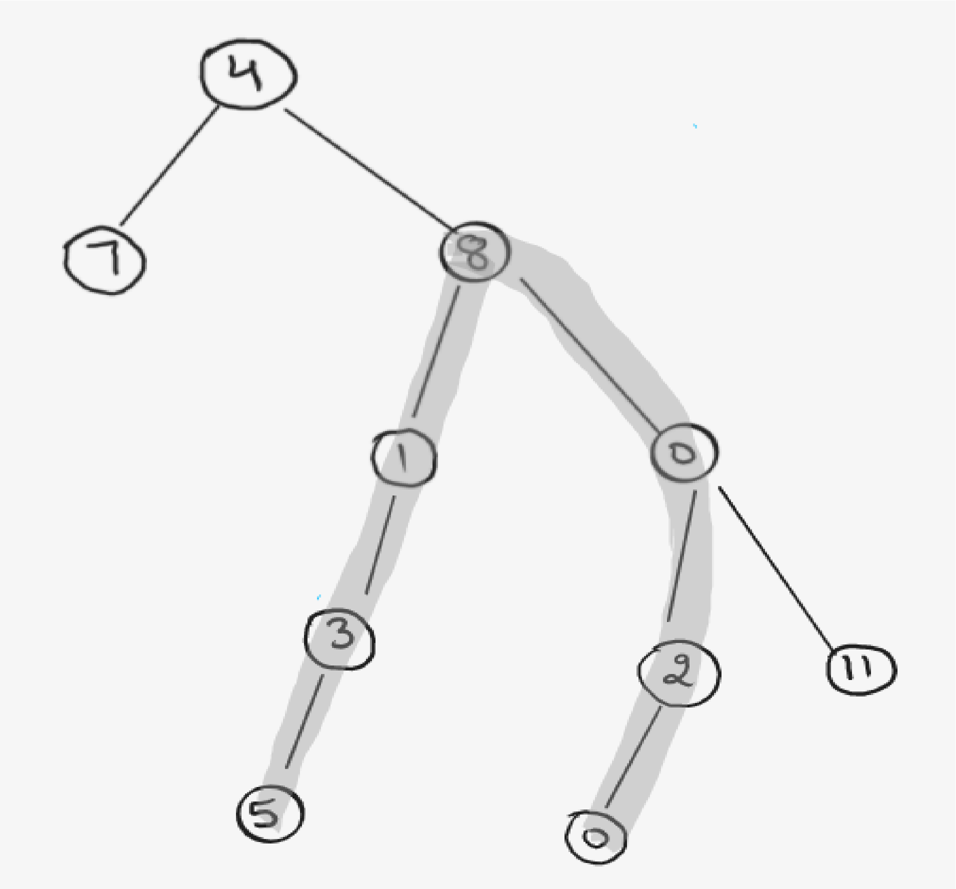


Diameter of a Binary Tree

Result: 4

Explanation: Longest Path available is 7 – 4 – 8 – 1 – 3 of length 4

Example 2:
Input Format: Given the root of Binary Tree



Result: 6
Explanation: Longest Path available is 5 – 3 – 1 – 8 – 0 – 2 – 0 of length 6. (Path is not Passing from root).

Solutions for Diameter of a Binary Tree

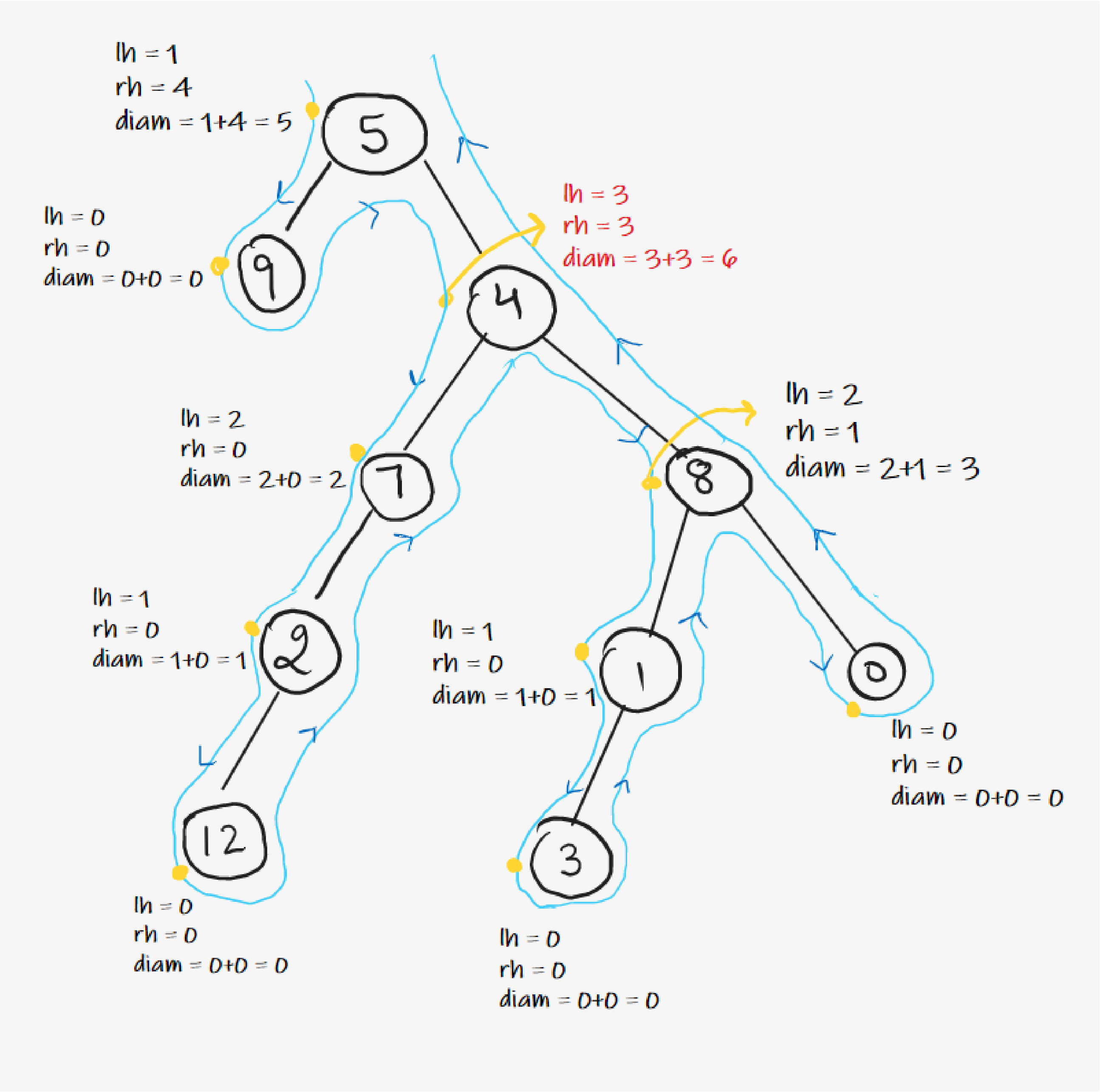
Solution 1: Naive approach
Intuition :
The idea is to consider every node as a *Curving Point* in diameter. For our understanding, we can define the curving point as the node on the diameter path which has the maximum height. In the above examples, the Curving Point is node **4** in Example 1 and node **8** in Example 2.
Now, if we observe carefully, we can see that diameter of the tree can be defined as left subtree height + right subtree height from the Curving Point.
So, the idea to find the Curving Point is, consider every node in the tree as a curving point and calculate the diameter for every curving point and return the maximum of all diameters.

```
Diameter at given Curving Point = Left Height + Right Height
```

Approach :

- Traverse the tree recursively.
- At every node, calculate height of left and right subtrees.
- Calculate the diameter for every node using the above formula.
- Calculate the maximum of all diameters. This can be done simply using a variable passed by reference in the recursive calls or a global static variable.

dry-run :



Start traversing the tree.

- Initialize Maximum Diameter variable with Integer.MIN_VALUE.
- Reach on **Node 5** , call Height Function , Left height = 1 , Right height = 4 so Diameter is (1 + 4) = 5. Hence , Maximum Diameter = Max(Integer.MIN_VALUE , 5) = 5.
- Reach on **Node 9** , call Height Function , Left height = 0 , Right height = 0 so Diameter is (0 + 0) = 0. Hence , Maximum Diameter = Max(5 , 0) = 5.
- Reach on **Node 4** , call Height Function , Left height = 3 , Right height = 3 so Diameter is (3 + 3) = 6. Hence , Maximum Diameter = Max(5 , 6) = 6.
- Reach on **Node 7** , call Height Function , Left height = 2 , Right height = 0 so Diameter is (2 + 0) = 2. Hence , Maximum Diameter = Max(6,2) = 6.
- Reach on **Node 2** , call Height Function , Left height = 1 , Right height = 0 so Diameter is (1 + 0) = 1. Hence , Maximum Diameter = Max(6 , 1) = 6.
- Reach on **Node 12** , call Height Function , Left height = 0 , Right height = 0 so Diameter is (0 + 0) = 0. Hence , Maximum Diameter = Max(6 , 0) = 6.
- Reach on **Node 8** , call Height Function , Left height = 2 , Right height = 1 so Diameter is (2 + 1) = 3. Hence , Maximum Diameter = Max(6 , 3) = 6.
- Reach on **Node 1** , call Height Function , Left height = 1 , Right height = 0 so Diameter is (1 + 0) = 1. Hence , Maximum Diameter = Max(6 , 1) = 6.
- Reach on **Node 3** , call Height Function , Left height = 0 , Right height = 0 so Diameter is (0 + 0) = 0. Hence , Maximum Diameter = Max(6 , 0) = 6.
- Reach on **Node 0** , call Height Function , Left height = 0 , Right height = 0 so Diameter is (0 + 0) = 0. Hence , Maximum Diameter = Max(6 , 0) = 6.
-

Time Complexity: O(N*N) (For every node, Height Function is called which takes O(N) time hence for every node it becomes N*N)
Space Complexity: O(1) (Extra Space) **+ O(H)** (Recursive Stack Space where **"H"** is the height of tree)

Solution 2: Post Order Traversal
Intuition :
Is it possible to optimize the above solution further? Which operation do you think is very repetitive in nature in the above solution?
Height of the subtrees.
Can we use postorder traversal to calculate everything in a single traversal of the tree?
Yes, as in post-order traversal, we have to completely traverse the left and right subtree before visiting the root node.

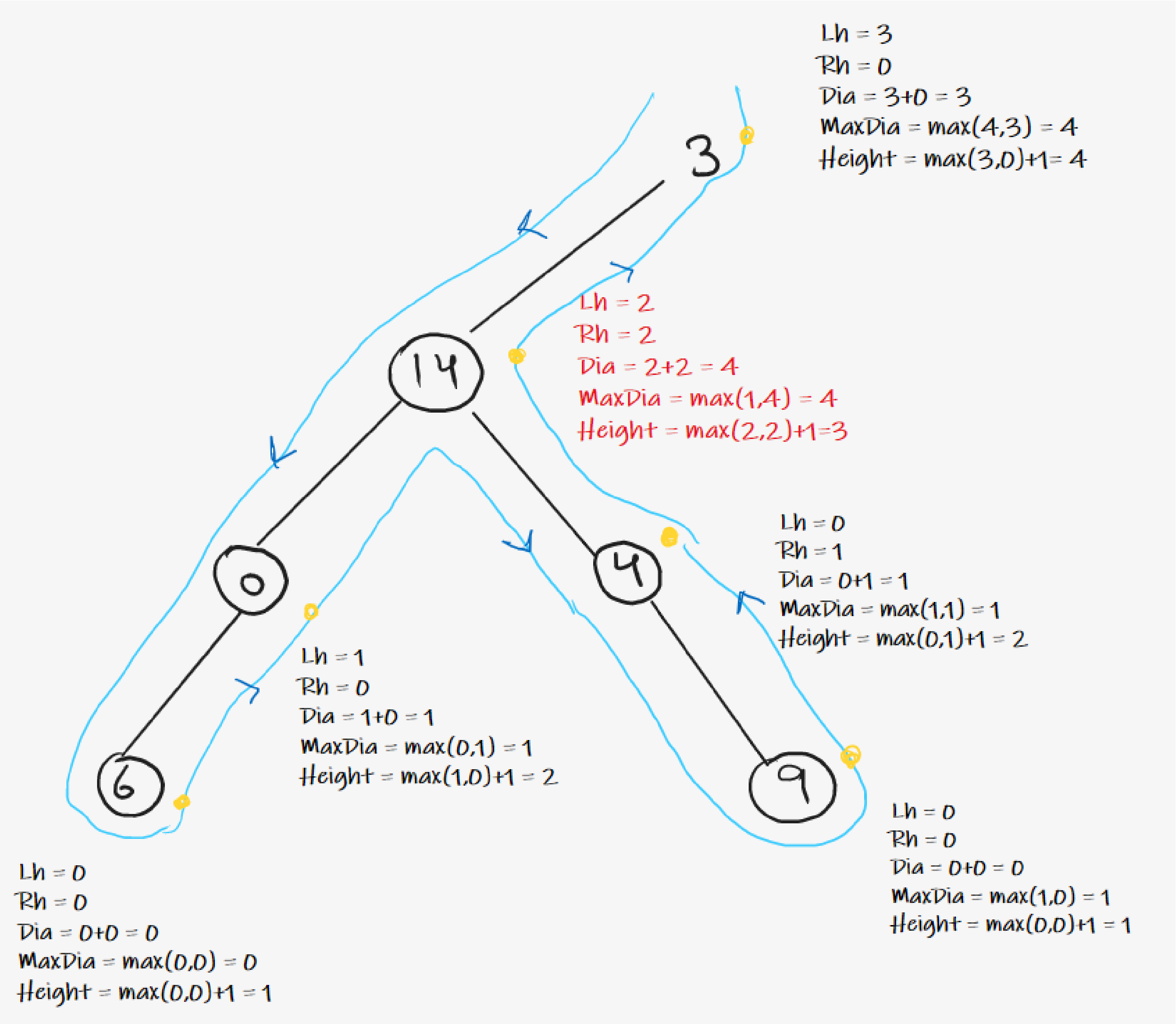
So, the idea is to use post-order traversal and keep calculating the height of the left and right subtrees. Once we have the heights at the current node, we can easily calculate both the diameter and height of the current node.

Approach :

- Start traversing the tree recursively and do work in Post Order.
- In the Post Order of every node , calculate diameter and height of the current node.
- If current diameter is maximum then update the variable used to store the maximum diameter.
- Return height of current node to the previous recursive call.

Dry Run :

In Post Order, Start traversing the tree:



- Reach on **Node 6 ,** Left height = 0 as left == null , Right height = 0 as right == null so Diameter is (0 + 0) = 0. Hence , Maximum Diameter = Max(0 , 0) = 0 and return height = max(0,0)+1 = 1.
- Reach on **Node 0**, Left height = 1 , Right height = 0 as right == null so Diameter is (1 + 0) = 1. Hence , Maximum Diameter = Max(0 , 1) = 1 and return height = max(1,0)+1 = 2.
- Reach on **Node 9 ,** Left height = 0 as left == null , Right height = 0 as right == null so Diameter is (0 + 0) = 0. Hence , Maximum Diameter = Max(1 , 0) = 1 and return height = max(0,0)+1 = 1.
- Reach on **Node 4 ,** Left height = 0 as left == null , Right height = 1 , so Diameter is (0 + 1) = 1. Hence , Maximum Diameter = Max(1 , 1) = 1 and return height = max(0,1)+1 = 2.
- Reach on **Node 14 ,** Left height = 2 , Right height = 2 , so Diameter is (2 + 2) = 4. Hence , Maximum Diameter = Max(1 , 4) = 4 and return height = max(2,2)+1 = 3.
- Reach on **Node 3 ,** Left height = 3 , Right height = 0 as right == null , so Diameter is (3 + 0) = 3. Hence , Maximum Diameter = Max(4 , 3) = 4 and return height = max(3,0)+1 = 4.
- Hence , the maximum diameter is 4 .

Code:

```
int traverse(TreeNode * root, int &maxi){
    if(root == NULL) return 0;

    int leftTree = traverse(root->left, maxi);
    int rightTree = traverse(root->right, maxi);
    maxi = max(maxi, leftTree+rightTree);
    return max(leftTree, rightTree)+1;
}
int maxDepth(TreeNode *root){

    int maxi = 0;
    traverse(root, maxi);
    return maxi;

}
```

Time Complexity: O(N)

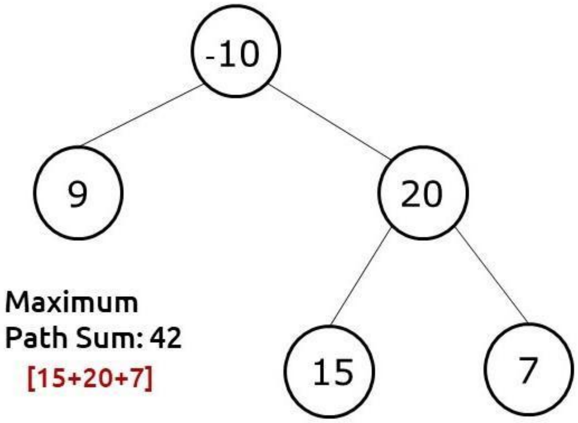
Space Complexity: O(1) Extra Space + O(H) Recursion Stack space (Where **"H"** is the height of binary tree)

Maximum Sum Path in Binary Tree

Problem Statement: Write a program to find the maximum sum path in a binary tree. A path in a binary tree is a sequence of nodes where every adjacent pair of nodes are connected by an edge. A node can only appear in the sequence at most once. A path need not pass from the root. We need to find the path with the maximum sum in the binary tree.

Example:

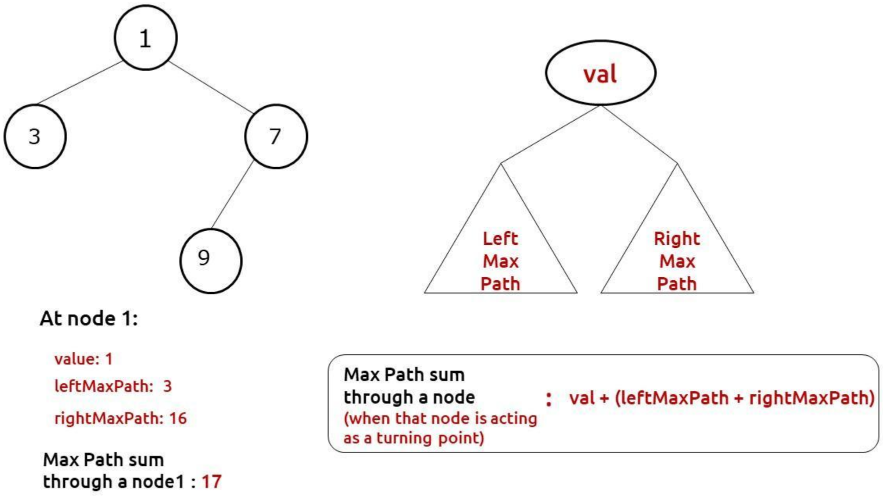
`Input:`



`Output:` `The Max Path Sum for the Tree is 42`

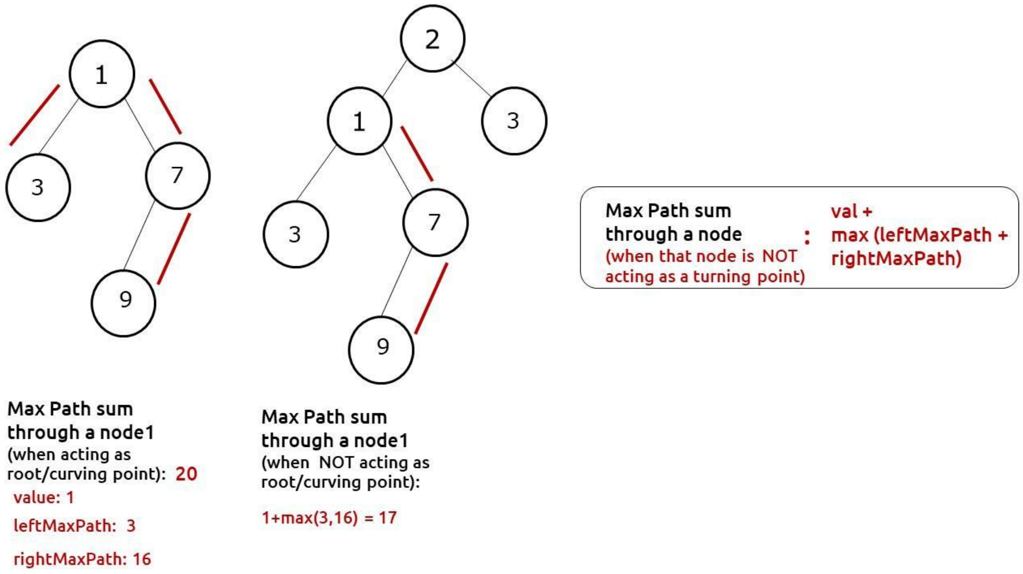
Solution:

Approach: A brute force approach would be to generate all paths and compare them. Generating all paths will be a time-costly activity therefore we need to look for something else. We first need to define what is the maximum path sum through a given node (when that node is acting as the root node/curving point). At a given node with a value, if we find the max leftSumPath in the left-subtree and the max rightSumPath in the right subtree, then the maxPathSum through that node is value+(leftSumPath+rightSumPath).



Now we can apply this formula at every node by doing a simple tree traversal and storing the maximum value (our answer) in a reference variable.

For our recursion to work, it is very important to understand what value we return from our function. In our recursive function, we find and compare the maxPathSum from a given node when it is the root/turning point of the path. But what we return is the maxPathSum of that same node when it is **NOT** the root/turning point of the path. To find the latter maxPath, we no longer have the liberty to consider both leftMaxPath and rightMaxPath, we will simply take the maximum of the two and it to the value of the node.



To summarize:

- Initialize a maxi variable to store our final answer.
- Do a simple tree traversal. At each node, find recursively its leftMaxPath and its rightMaxPath.
- Calculate the maxPath through the node as val + (leftMaxPath + rightMaxPath) and update maxi accordingly.
- Return the maxPath when node is not the curving point as val + max(leftMaxPath, rightMaxPath).

Code:

```
int traverse(TreeNode* root , int &maxi) {
        if(root==NULL) return 0;

        int leftSum = traverse(root->left, maxi);
        int rightSum = traverse(root->right, maxi);

        maxi = max(maxi, leftSum+rightSum+root->data);

        return max(leftSum, rightSum)+root->data <0 ? 0 : max(leftSum,
rightSum)+root->data ;
    }
int maxPathSum(TreeNode* root) {
        int maxi = INT_MIN;
        traverse(root, maxi);
        return maxi;
    }
```

The Max Path Sum for this tree is 42

Time Complexity: O(N).

Reason: We are doing a tree traversal.

Space Complexity: O(N)

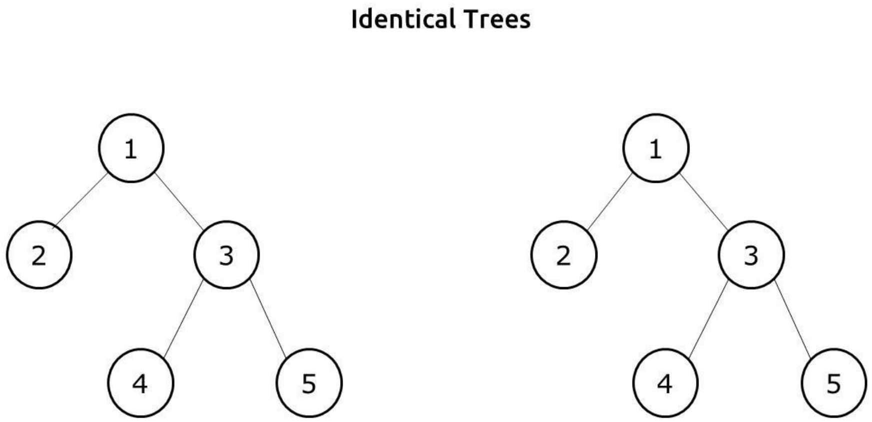
Reason: Space is needed for the recursion stack. In the worst case (skewed tree), space complexity can be O(N).

Check if two trees are identical

Problem Statement: Given two Binary Tree. Write a program to check if two trees are identical or not.

Example 1:

Input:



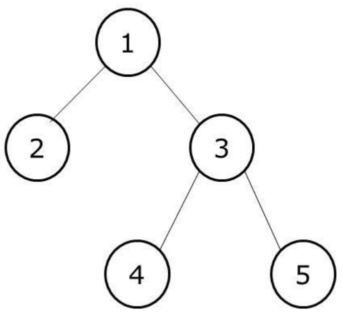
Output: Two Trees are identical

Exaplaination: Two trees T1 and T2 are said to be identical if these three conditions are met for every pair of nodes :
Value of a node in T1 is equal to the value of the corresponding node in T2.
Left subtree of this node is identical to the left subtree of the corresponding node.
Right subtree of this node is identical to the right subtree of the corresponding node.

Example 2:

`Input:`

Non Identical Trees

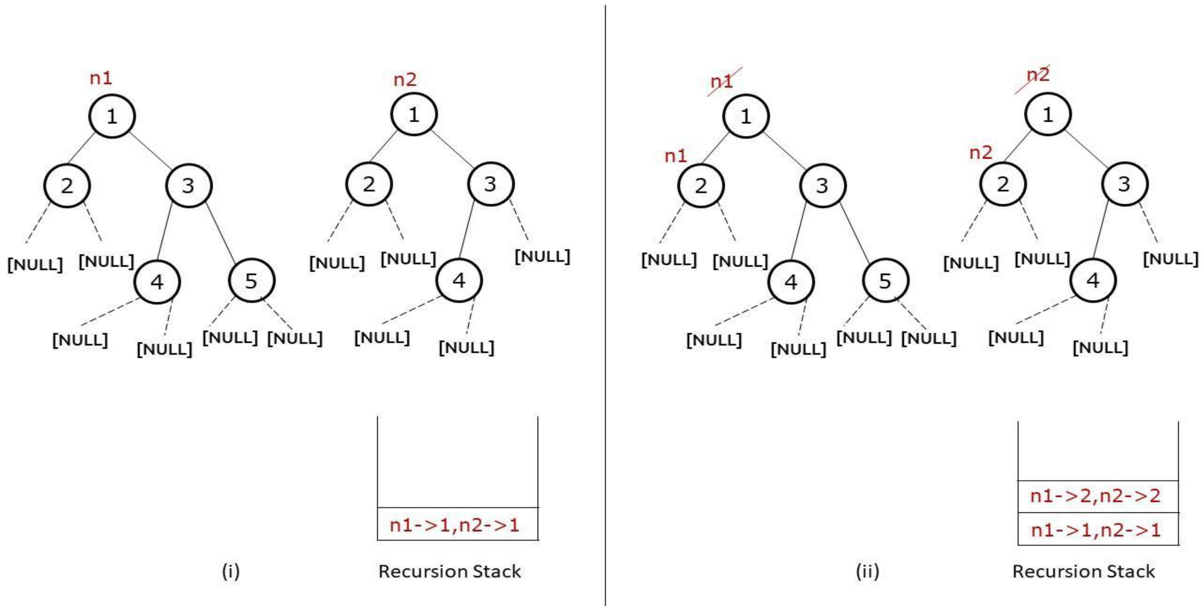


`Output:` `Two Trees are not identical`

Solution:

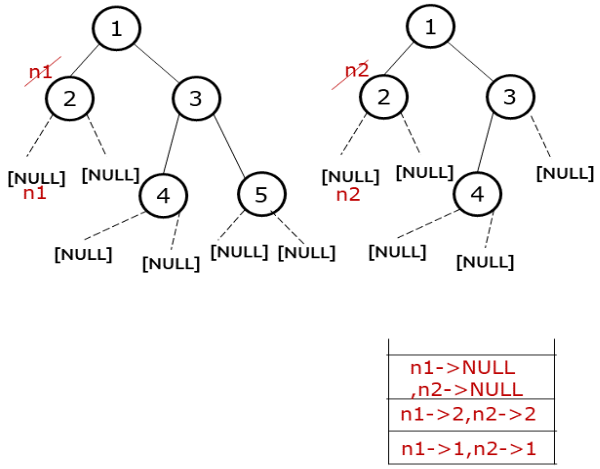
Approach: In order to check whether two trees are identical or not, we need to traverse the trees. While traversing we first check the value of the nodes, if they are unequal we can simply return false, as trees are non-identical. If they are the same, then we need to **recursively** check their left child as well as the right child. When we get all the three values as true(node values, left child, right child) we can conclude that these are identical trees and can return true. Any other combination will return false.

Dry Run:

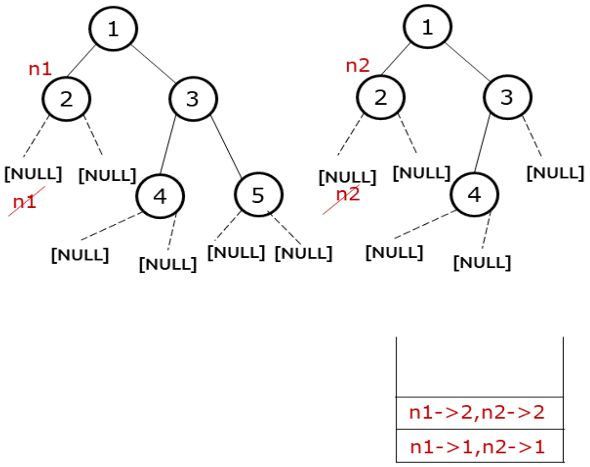


(i) Initially we have variables n1 and n2 pointing to the roots of both trees. This function is inside our recursion stack as well.

(ii) First we check the values at the nodes. As the values of nodes are equal, we proceed further and recursively call the left child of both nodes. This second function is pushed to our recursion stack.



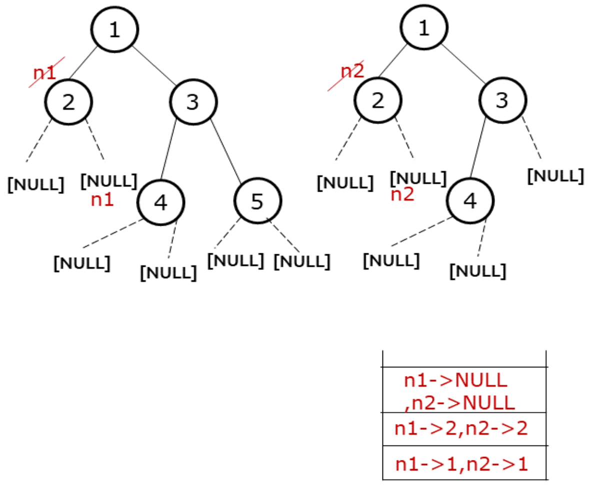
(iii) Recursion Stack



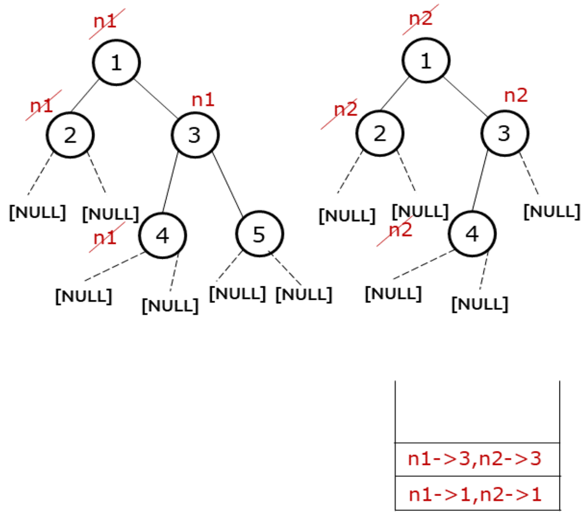
(iv) Recursion Stack

(iii) Again we check the nodes. As they are equal, we again check for the left subtrees and add the function to our recursion stack.

(**iv)** Both n1 and n2 point to NULL, so the base condition will be true and we will return true from this function.



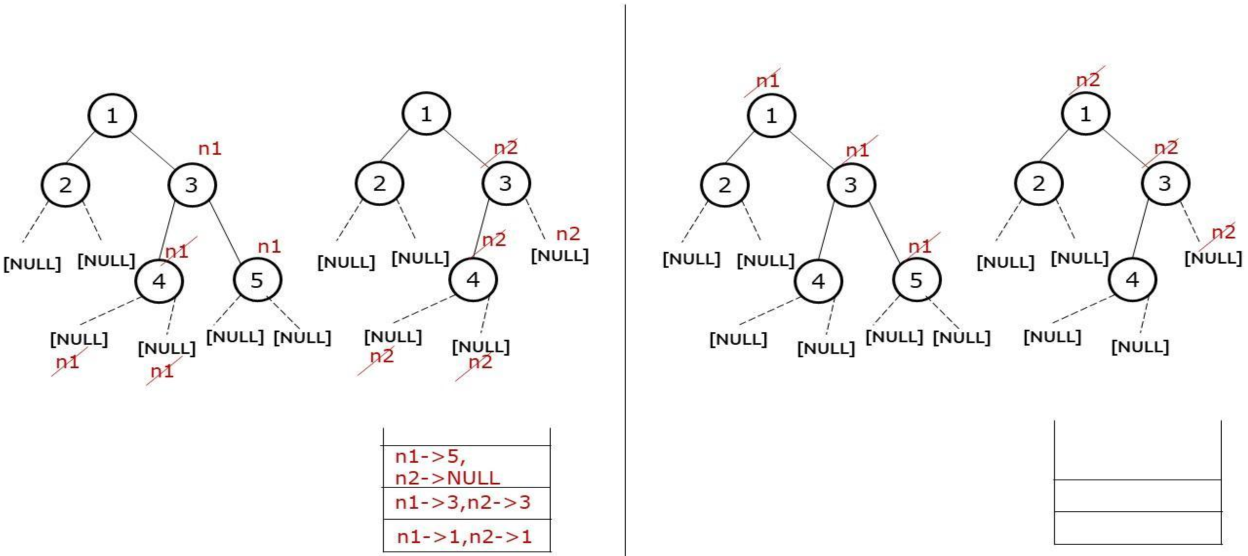
(v) Recursion Stack



(vi) Recursion Stack

(v) At nodes pointing to 2, two out of three conditions return true, so we will proceed further and check the third condition. We will recursively call the right child of the nodes. Again that function will be added to our recursive stack.

(vi) Similar to step 4, n1 and n2 point to NULL, therefore we will again return true and the function will be removed from our recursion stack. Now at nodes pointing to 2, all three conditions return true (node values, left child and right child), therefore we can return true from this function and remove it from the recursion stack. Then at node 1, two conditions are true, therefore we call the right child of n1 and n2 to check the third condition.



(vii) Recursion Stack

(viii) Recursion Stack

(vii) At nodes pointing to 3, we first check the data values which are equal, then we recursively call the function to check for their left child. It turns out to be identical and we get the return value true, then we recursively call the function to check for their right child.

(viii) Now n1 points to a node with value 5 whereas n2 points to NULL, this can’t be the case with an identical tree, our second base condition hits and we return false from this function. In the parent function where n1 and n2 point to 3, **only two out of three conditions return true,** therefore we will return the value false. In its parent function where n1 and n2 points to 1, again **only two out of three conditions return true,** therefore we will return false. Hence we will return a false value.

As our first function returns a false to our main function, we can conclude that these two trees are **Non-identical.**

Code:

```
#include <bits/stdc++.h>
using namespace std;
bool isIdentical(node * node1, node * node2) {
  if (node1 == NULL && node2 == NULL)
    return true;
  else if (node1 == NULL || node2 == NULL)
    return false;

  return ((node1 -> data == node2 -> data) && isIdentical(node1 ->
left, node2 -> left) && isIdentical(node1 -> right, node2 -> right));
}
```

Output:

Two trees are non-identical

Time Complexity: O(N).

Reason: We are doing a tree traversal.

Space Complexity: O(N)

Reason: Space is needed for the recursion stack. In the worst case (skewed tree), space complexity can be O(N).

Zig Zag Traversal Of Binary Tree

Problem Statement: Given the root of a binary tree, return the zigzag level order traversal of Binary Tree. (i.e., from left to right, then right to left for the next level and alternate between).

Examples:

Example 1:

Input: root = [3,9,20,null,null,15,7]

Output: [[3],[20,9],[15,7]]

Explaination: From the root, we follow this terminology, left to right -> right to left -> left to right and so on so forth.

Example 2:

Input: root = [[0]]

Output : [[0]]

Explanation: We just have a single node which acts as the root, so going from left to right, we get just one node that is the root node itself.

Intuition: Considering the fact that we need to print the nodes, level by level, our first guess would definitely be that it must be related to level order traversal. If we closely examine, for even levels we need to go from left to right while for odd levels we need to go from right to left.

Approach: The above idea, could be implemented with a queue. We initially keep an empty queue and push the root node. We also need to keep the left to right bool variable that keeps check of the current level we are in. As we traverse nodes in the queue, we need to push them in a temporary array. If left to right is false we need to reverse the array and push it in our data structure or else, simply push it in our data structure. In the end, when we have finished traversing the current level, we need to toggle our left to the right variable.

Code:

```
vector<vector<int>> levelorderTraversal(TreeNode *root)
{
   vector<vector<int>> ans;
   if(root == NULL) return ans;
   queue<TreeNode*> q;
   q.push(root);
   int count = 0;
   while(!q.empty()){
    vector<int> level;
    int size = q.size();
    for (int i = 0; i < size; i++)
    {
        TreeNode* node = q.front();
        q.pop();

        if(node->left != NULL)q.push(node->left);
        if(node->right != NULL)q.push(node->right);

        level.push_back(node->data);
    }
```

```
      if(count%2==1){
          reverse(level.begin(), level.end());
      }
      ans.push_back(level);
      count++;
    }
    return ans;
}
```

Output:

```
Zig Zag Traversal of Binary Tree
3
20 9
15 7
```

Time Complexity: O(N)

Space Complexity: O(N)