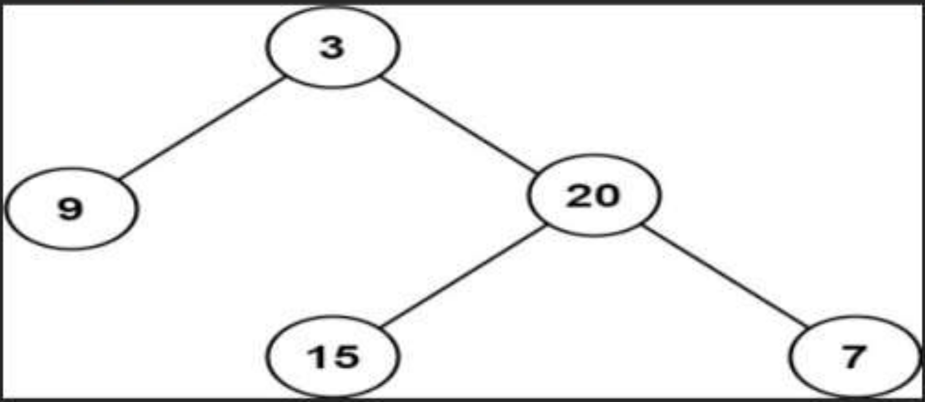
# Binary Tree

## 1.Maximum Depth of Binary Tree

Given the root of a binary tree, return *its maximum depth*.

A binary tree's **maximum depth** is the number of nodes along the longest path from the root node down to the farthest leaf node.

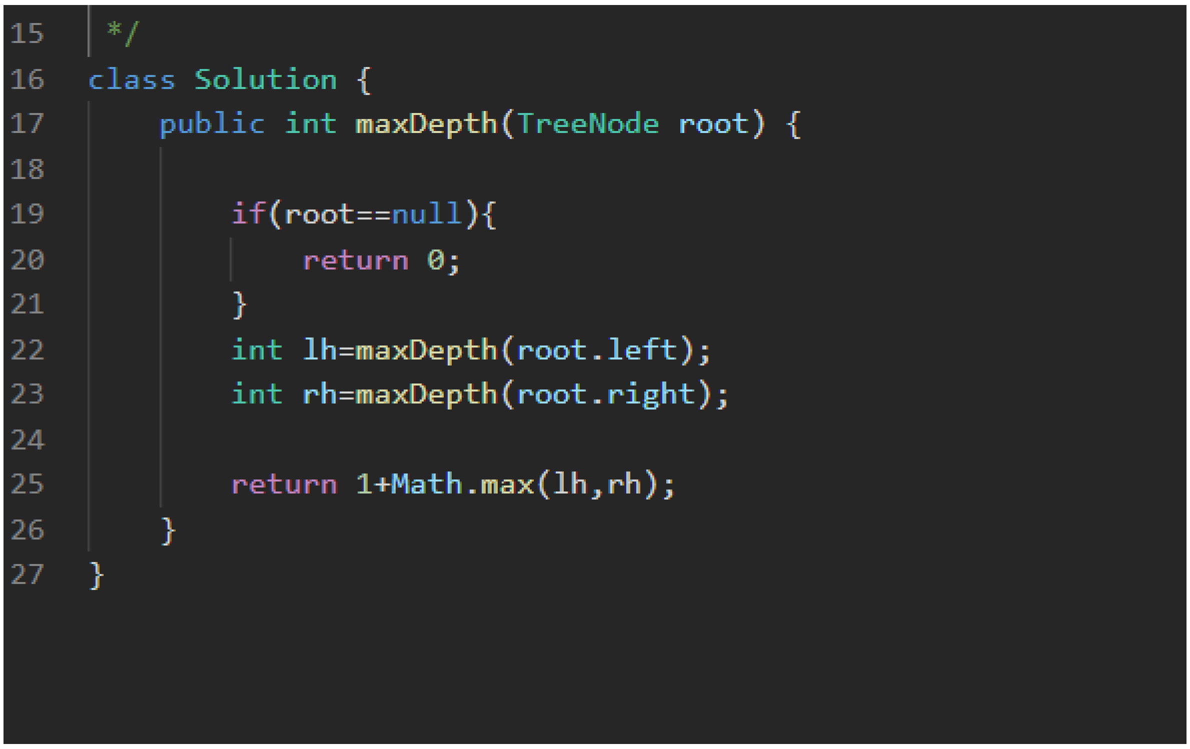
**Example 1:**



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

**Example 2:**

```
Input: root = [1,null,2]
Output: 2
```

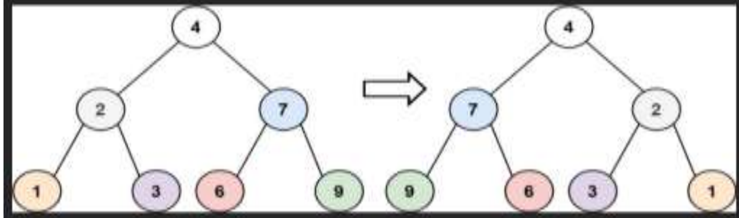


```java
 */
class Solution {
    public int maxDepth(TreeNode root) {

        if(root==null){
            return 0;
        }
        int lh=maxDepth(root.left);
        int rh=maxDepth(root.right);

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

## 2.Invert Binary Tree

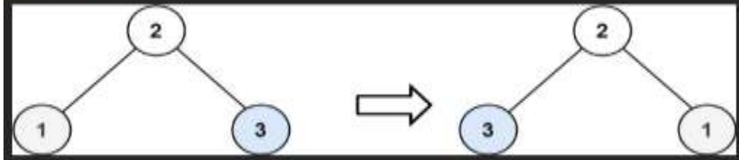
Given the `root` of a binary tree, invert the tree, and return *its root*.

**Example 1:**



```
Input: root = [4,2,7,1,3,6,9]
Output: [4,7,2,9,6,3,1]
```

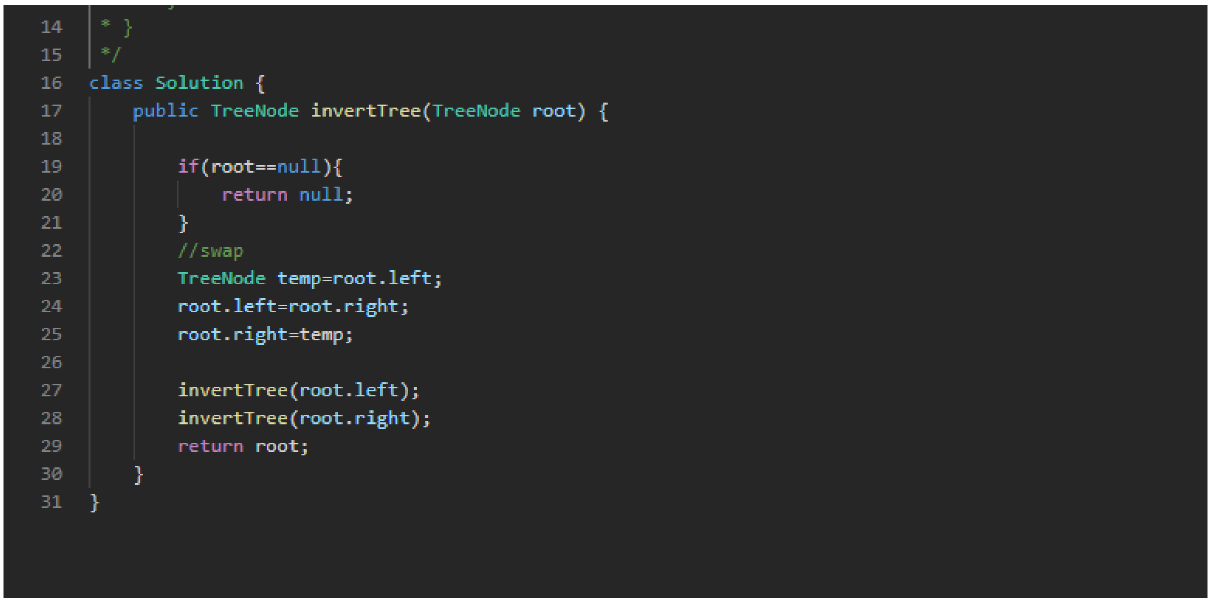
**Example 2:**



```
Input: root = [2,1,3]
Output: [2,3,1]
```

**Example 3:**

```
Input: root = []
Output: []
```

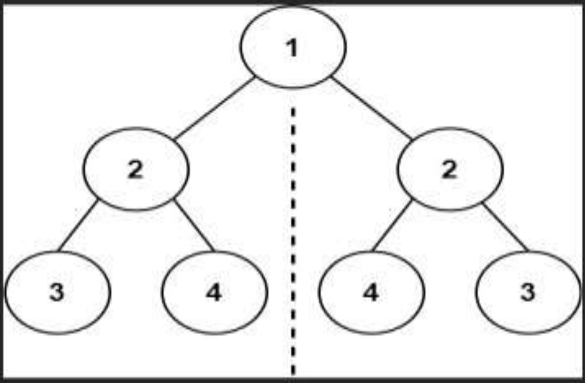


```java
 * }
 */
class Solution {
    public TreeNode invertTree(TreeNode root) {

        if(root==null){
            return null;
        }
        //swap
        TreeNode temp=root.left;
        root.left=root.right;
        root.right=temp;

        invertTree(root.left);
        invertTree(root.right);
        return root;
    }
}
```

## 3.Symmetric Tree

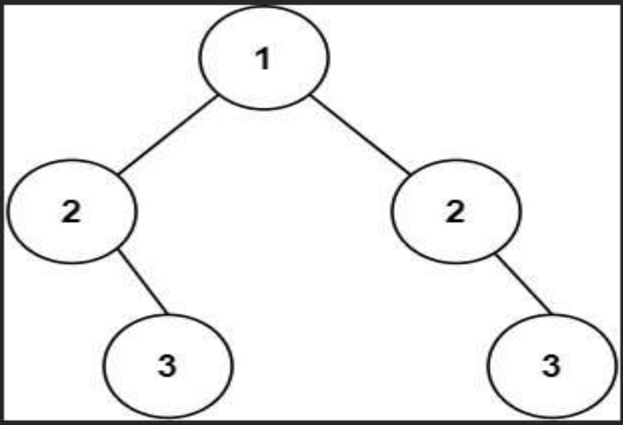
Given the `root` of a binary tree, *check whether it is a mirror of itself* (i.e., symmetric around its center).

**Example 1:**

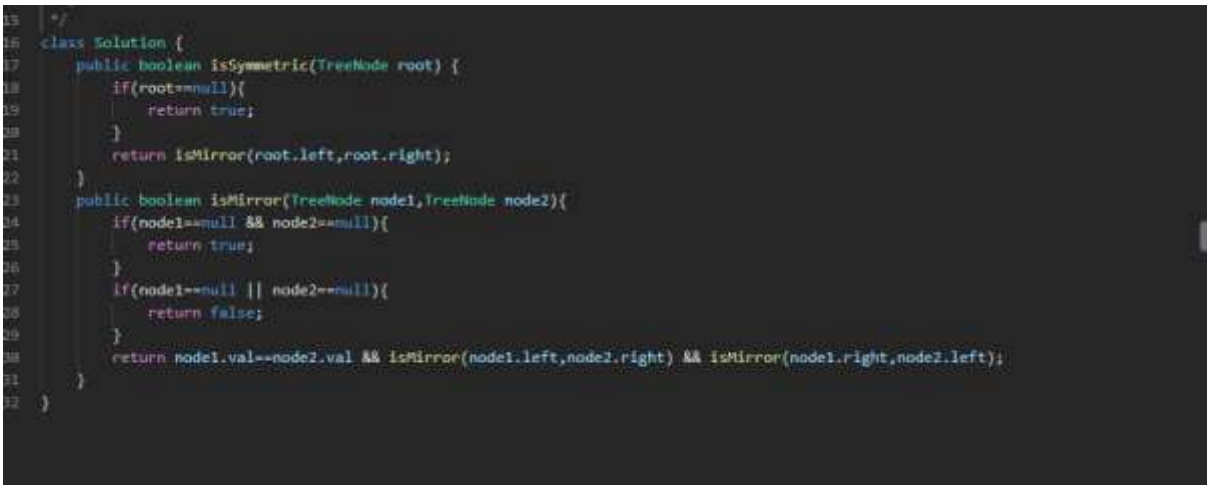


```
Input: root = [1,2,2,3,4,4,3]
Output: true
```

**Example 2:**



```
Input: root = [1,2,2,null,3,null,3]
Output: false
```



```
 */
class Solution {
    public boolean isSymmetric(TreeNode root) {
        if(root==null){
            return true;
        }
        return isMirror(root.left,root.right);
    }
    public boolean isMirror(TreeNode node1,TreeNode node2){
        if(node1==null && node2==null){
            return true;
        }
        if(node1==null || node2==null){
            return false;
        }
        return node1.val==node2.val && isMirror(node1.left,node2.right) && isMirror(node1.right,node2.left);
    }
}
```

## 4.Binary Tree Preorder Traversal

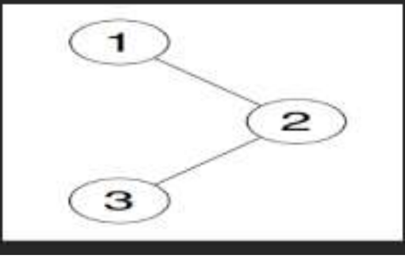
Given the `root` of a binary tree, return *the preorder traversal of its nodes' values*.

**Example 1:**

**Input:** root = [1,null,2,3]

**Output:** [1,2,3]

**Explanation:**

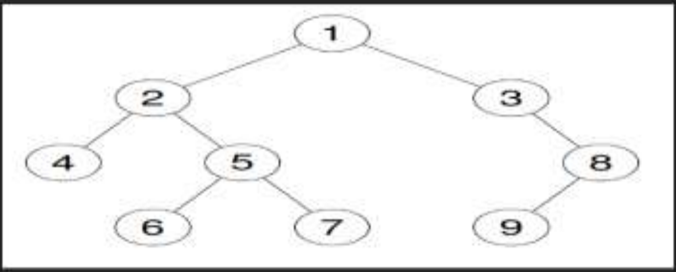


**Example 2:**

**Input:** root = [1,2,3,4,5,null,8,null,null,6,7,9]

**Output:** [1,2,4,5,6,7,3,8,9]

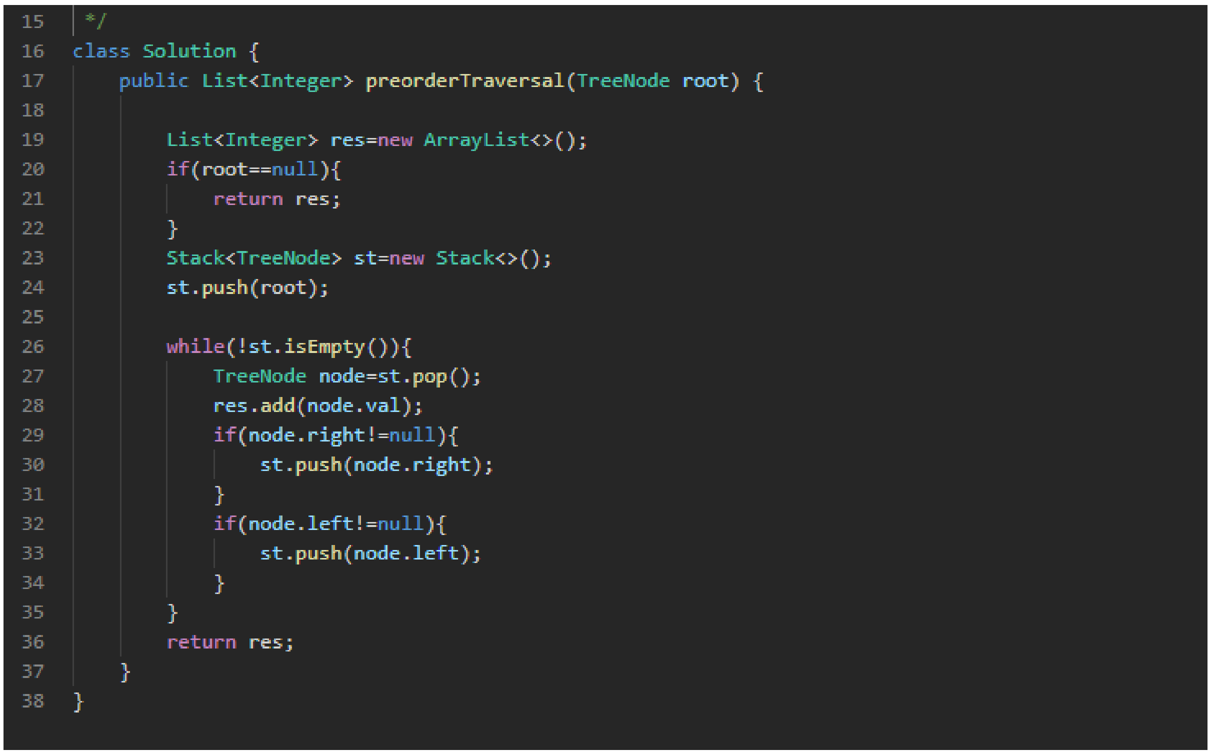
**Explanation:**



**Example 3:**

**Input:** root = []

**Output:** []



```java
 */
class Solution {
    public List<Integer> preorderTraversal(TreeNode root) {

        List<Integer> res=new ArrayList<>();
        if(root==null){
            return res;
        }
        Stack<TreeNode> st=new Stack<>();
        st.push(root);

        while(!st.isEmpty()){
            TreeNode node=st.pop();
            res.add(node.val);
            if(node.right!=null){
                st.push(node.right);
            }
            if(node.left!=null){
                st.push(node.left);
            }
        }
        return res;
    }
}
```

## 5.Binary Tree Inorder Traversal

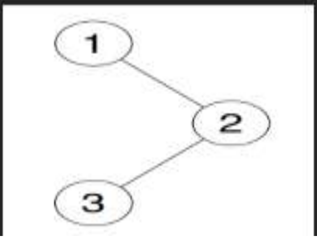
Given the `root` of a binary tree, return *the inorder traversal of its nodes' values.*

**Example 1:**

**Input:** `root = [1,null,2,3]`

**Output:** `[1,3,2]`

**Explanation:**

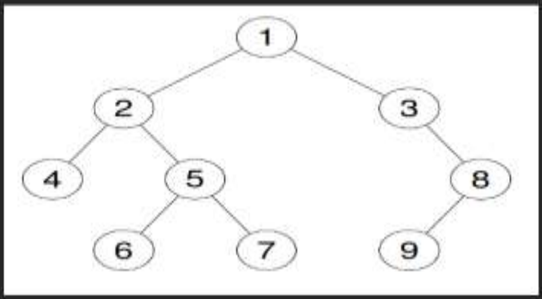


**Example 2:**

**Input:** `root = [1,2,3,4,5,null,8,null,null,6,7,9]`

**Output:** `[4,2,6,5,7,1,3,9,8]`

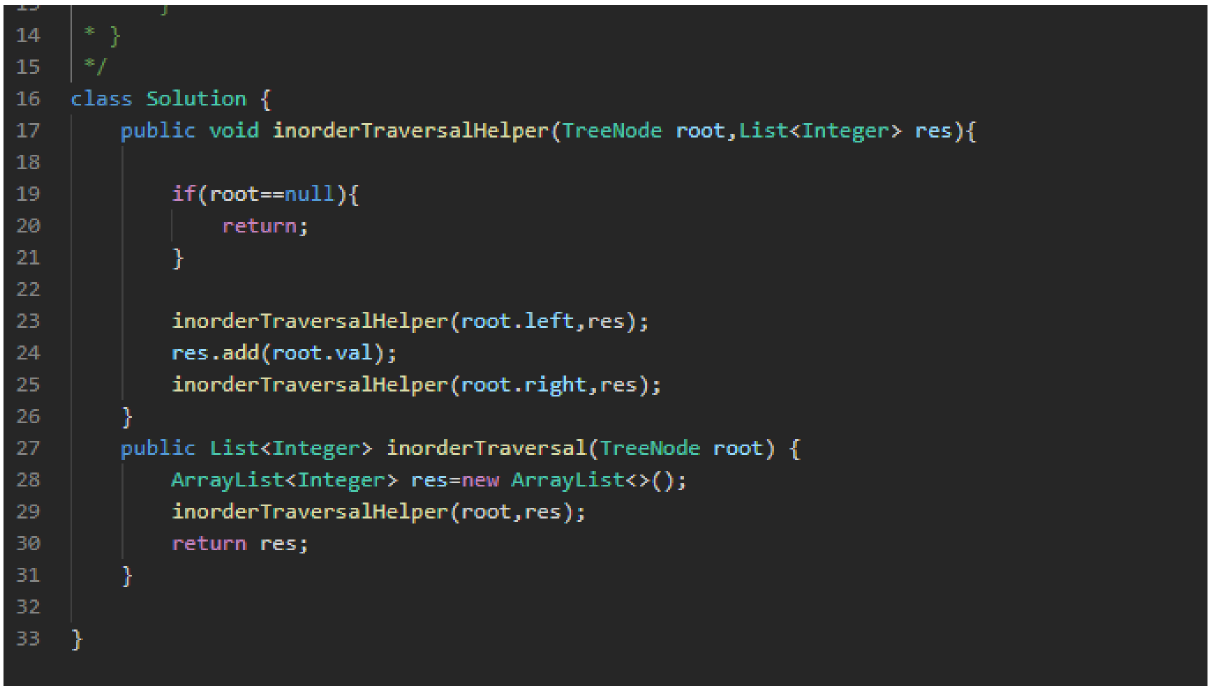
**Explanation:**



**Example 3:**

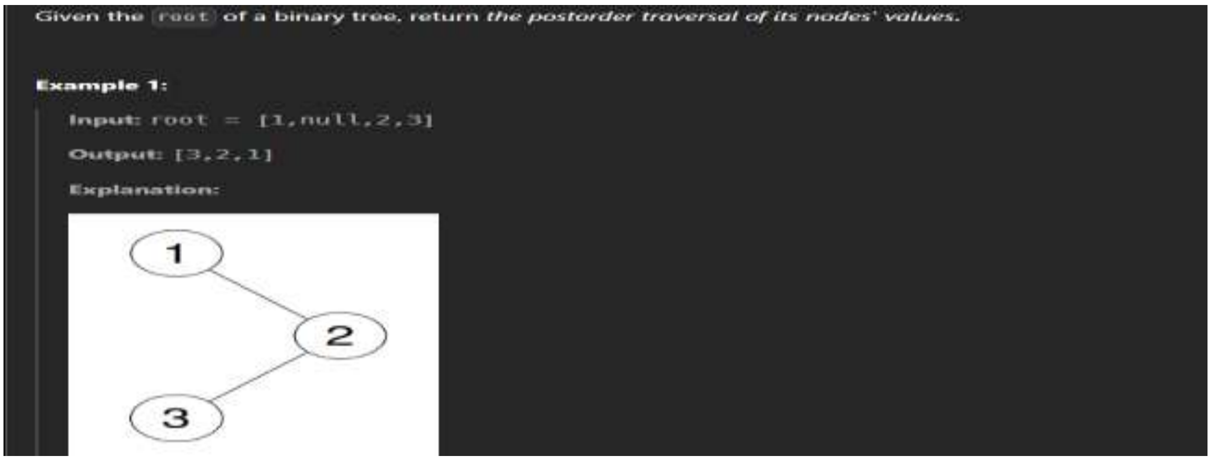
**Input:** `root = []`

**Output:** `[]`



```java
 * }
 */
class Solution {
    public void inorderTraversalHelper(TreeNode root,List<Integer> res){

        if(root==null){
            return;
        }

        inorderTraversalHelper(root.left,res);
        res.add(root.val);
        inorderTraversalHelper(root.right,res);
    }
    public List<Integer> inorderTraversal(TreeNode root) {
        ArrayList<Integer> res=new ArrayList<>();
        inorderTraversalHelper(root,res);
        return res;
    }

}
```

## 6.Binary Tree Postorder Traversal



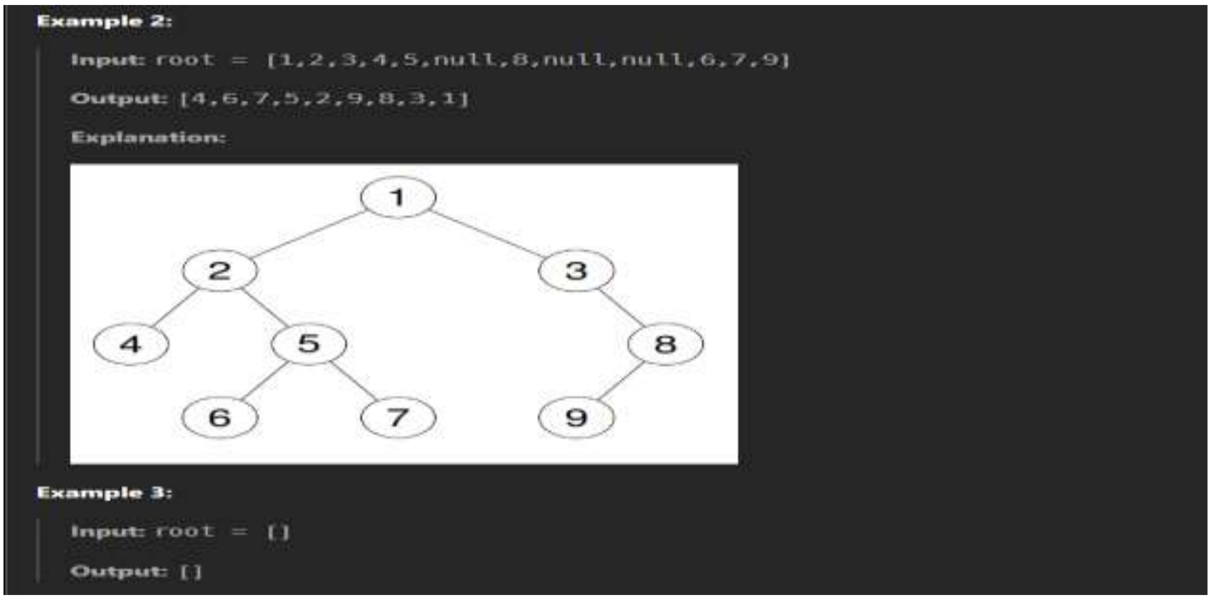
Given the `root` of a binary tree, return *the postorder traversal of its nodes' values.*

**Example 1:**

**Input:** root = [1,null,2,3]

**Output:** [3,2,1]

**Explanation:**



**Example 2:**

**Input:** root = [1,2,3,4,5,null,8,null,null,6,7,9]

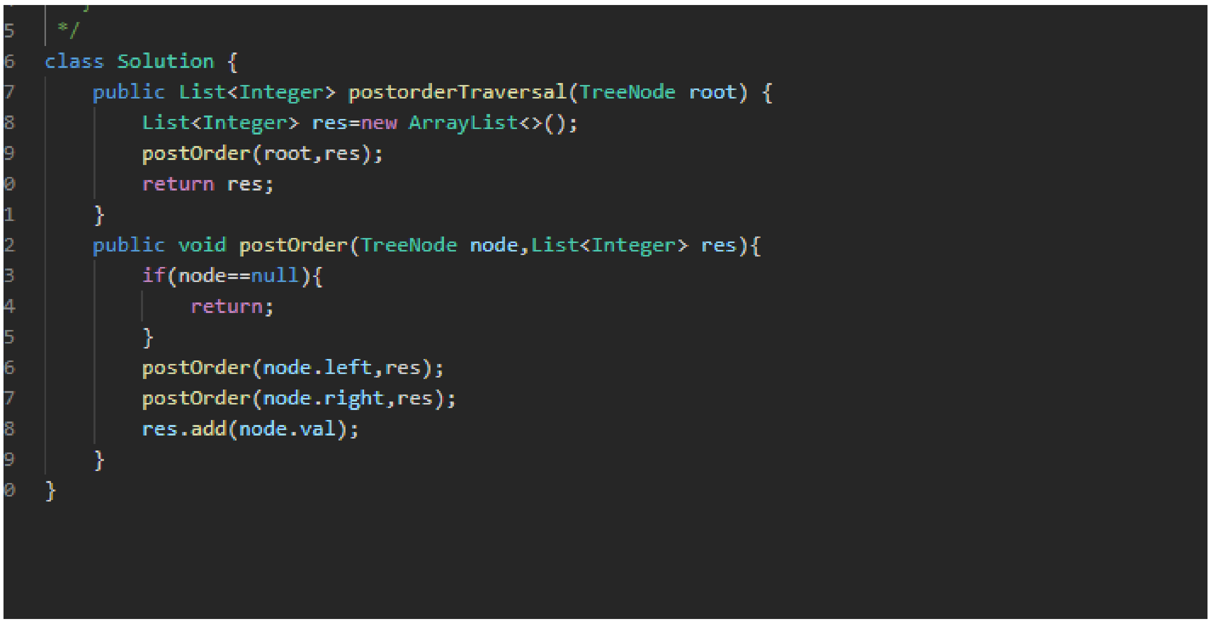
**Output:** [4,6,7,5,2,9,8,3,1]

**Explanation:**

**Example 3:**

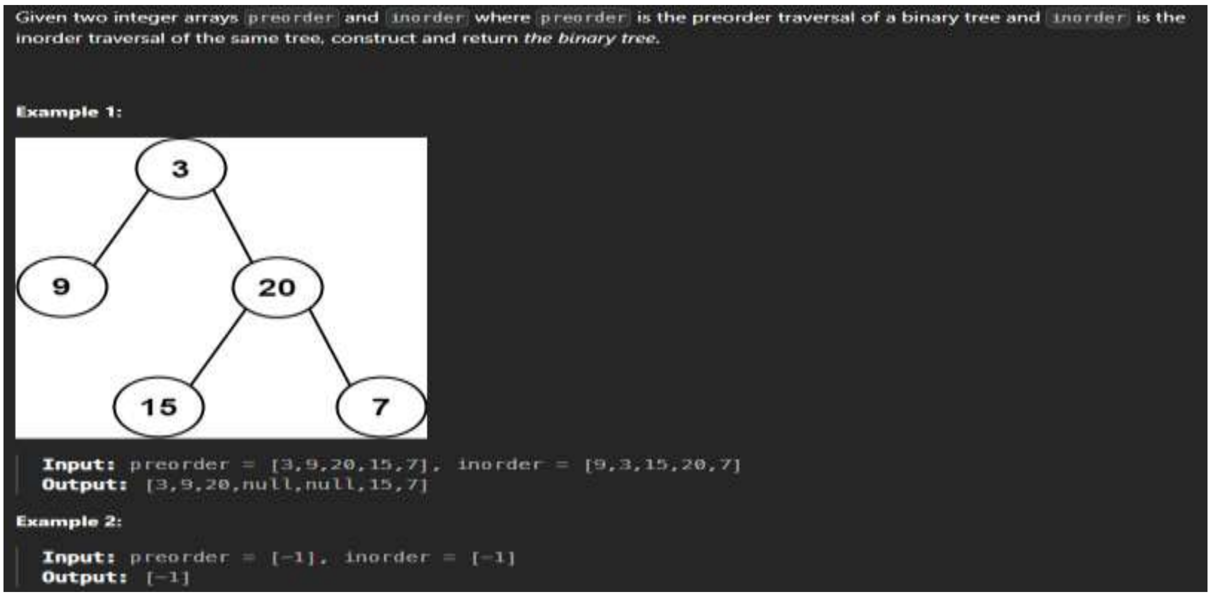
**Input:** root = []

**Output:** []



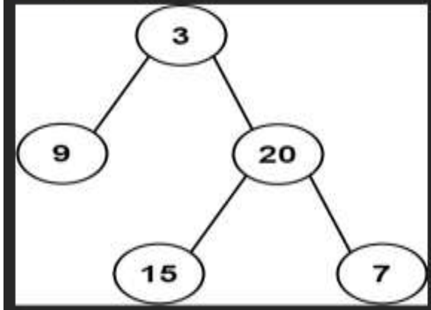
```java
 */
class Solution {
    public List<Integer> postorderTraversal(TreeNode root) {
        List<Integer> res=new ArrayList<>();
        postOrder(root,res);
        return res;
    }
    public void postOrder(TreeNode node,List<Integer> res){
        if(node==null){
            return;
        }
        postOrder(node.left,res);
        postOrder(node.right,res);
        res.add(node.val);
    }
}
```

## 7.Construct Binary Tree from preorder and Inorder Traversal



Given two integer arrays `preorder` and `inorder` where `preorder` is the preorder traversal of a binary tree and `inorder` is the inorder traversal of the same tree, construct and return *the binary tree*.

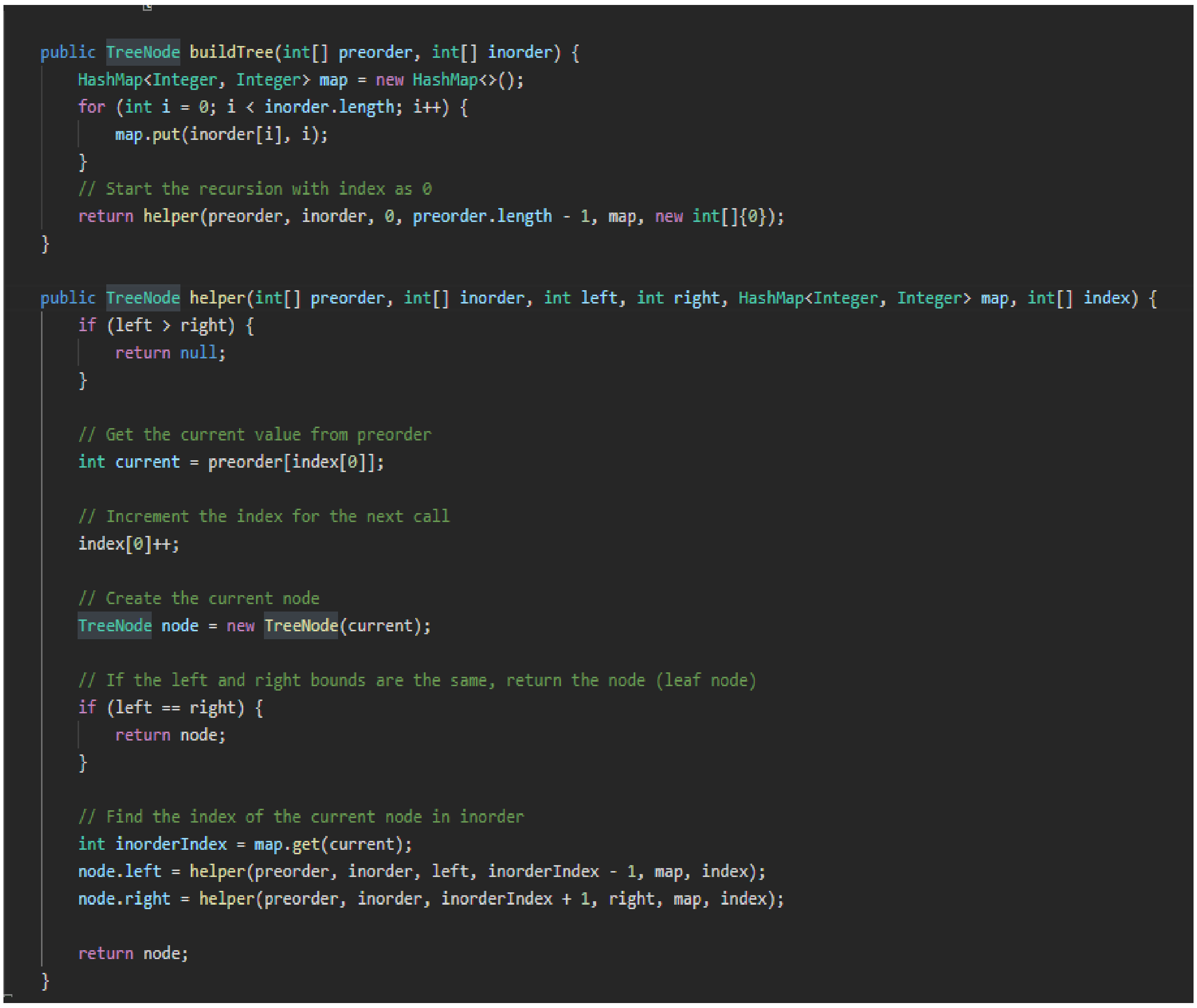
**Example 1:**



```
Input: preorder = [3,9,20,15,7], inorder = [9,3,15,20,7]
Output: [3,9,20,null,null,15,7]
```

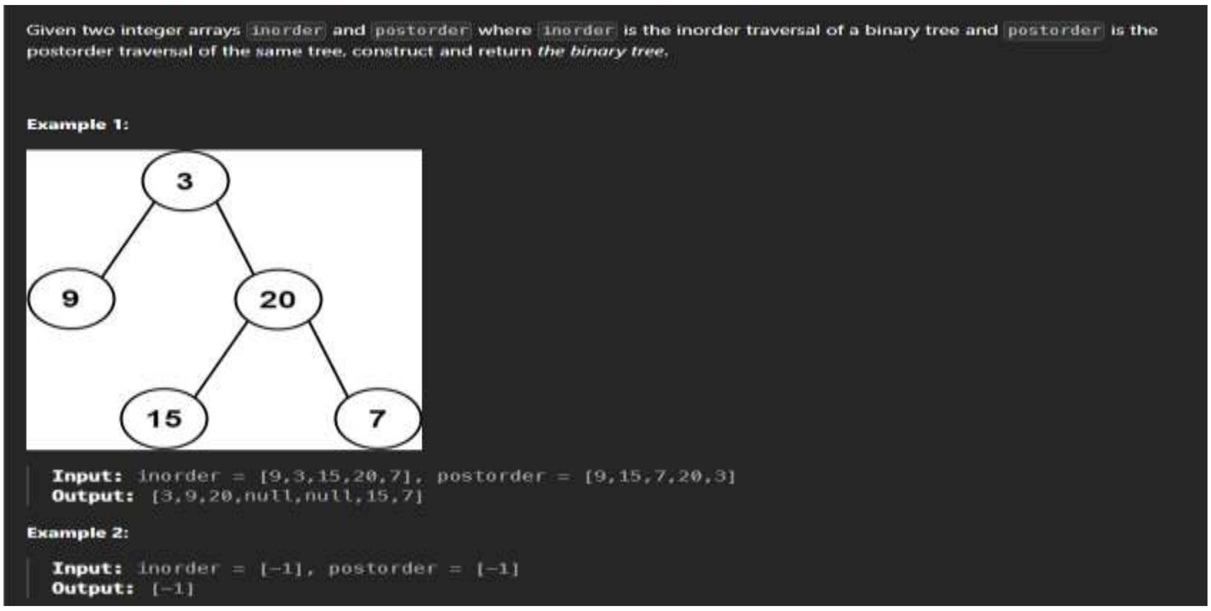
**Example 2:**

```
Input: preorder = [-1], inorder = [-1]
Output: [-1]
```



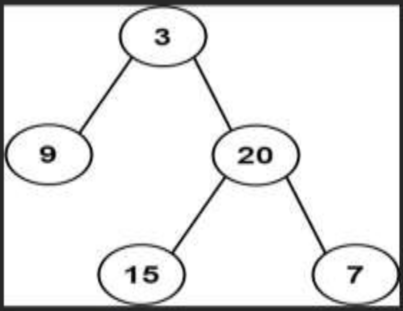
```java
public TreeNode buildTree(int[] preorder, int[] inorder) {
    HashMap<Integer, Integer> map = new HashMap<>();
    for (int i = 0; i < inorder.length; i++) {
        map.put(inorder[i], i);
    }
    // Start the recursion with index as 0
    return helper(preorder, inorder, 0, preorder.length - 1, map, new int[]{0});
}

public TreeNode helper(int[] preorder, int[] inorder, int left, int right, HashMap<Integer, Integer> map, int[] index) {
    if (left > right) {
        return null;
    }

    // Get the current value from preorder
    int current = preorder[index[0]];

    // Increment the index for the next call
    index[0]++;

    // Create the current node
    TreeNode node = new TreeNode(current);

    // If the left and right bounds are the same, return the node (leaf node)
    if (left == right) {
        return node;
    }

    // Find the index of the current node in inorder
    int inorderIndex = map.get(current);
    node.left = helper(preorder, inorder, left, inorderIndex - 1, map, index);
    node.right = helper(preorder, inorder, inorderIndex + 1, right, map, index);

    return node;
}
```

## 8. Construct Binary Tree from Inorder and postorder Traversal



Given two integer arrays `inorder` and `postorder` where `inorder` is the inorder traversal of a binary tree and `postorder` is the postorder traversal of the same tree, construct and return *the binary tree*.

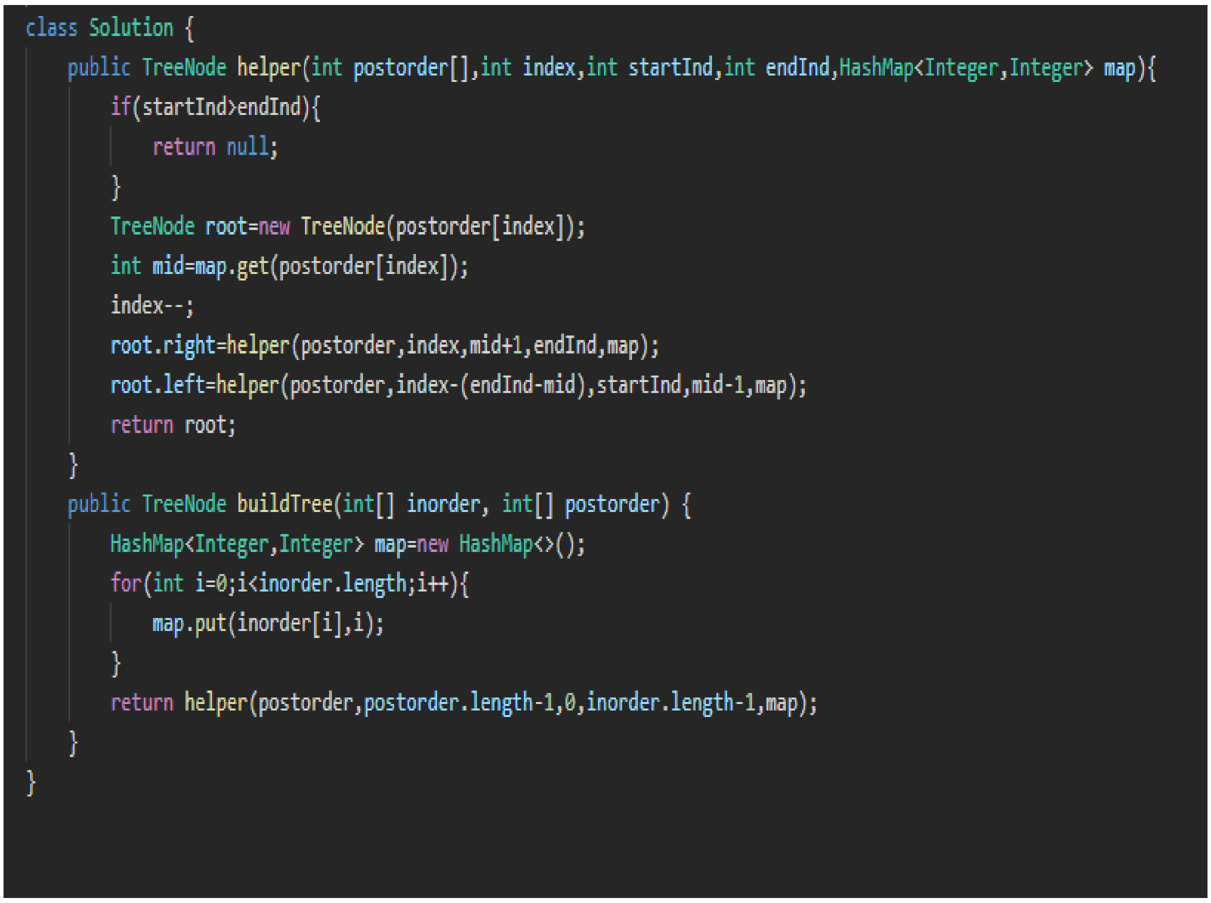
**Example 1:**



```
Input: inorder = [9,3,15,20,7], postorder = [9,15,7,20,3]
Output: [3,9,20,null,null,15,7]
```

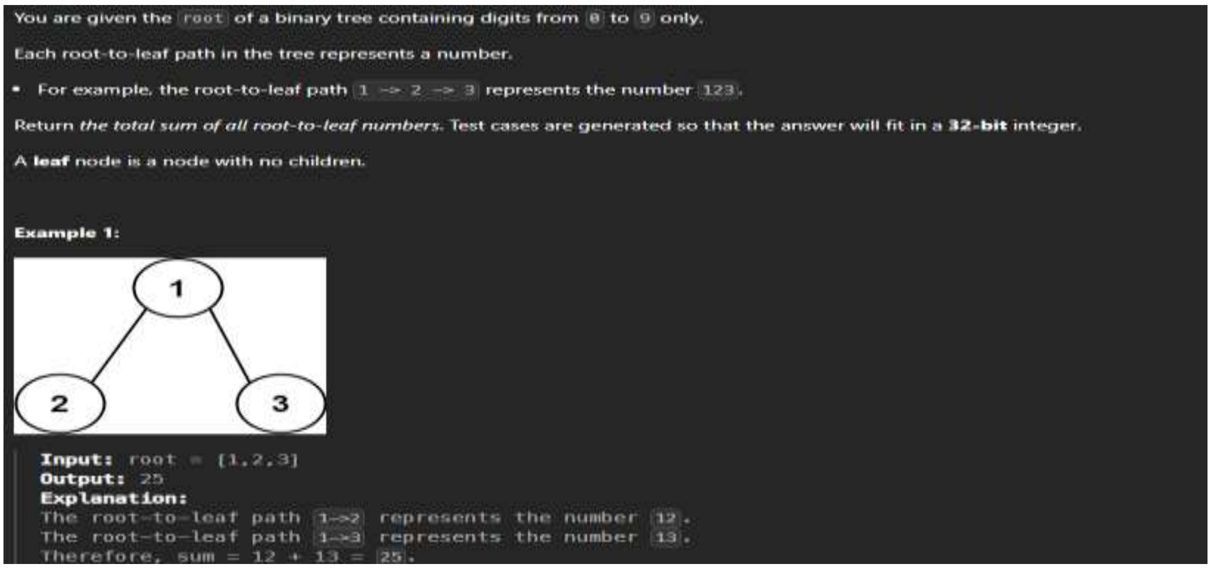
**Example 2:**

```
Input: inorder = [-1], postorder = [-1]
Output: [-1]
```



```
class Solution {
    public TreeNode helper(int postorder[],int index,int startInd,int endInd,HashMap<Integer,Integer> map){
        if(startInd>endInd){
            return null;
        }
        TreeNode root=new TreeNode(postorder[index]);
        int mid=map.get(postorder[index]);
        index--;
        root.right=helper(postorder,index,mid+1,endInd,map);
        root.left=helper(postorder,index-(endInd-mid),startInd,mid-1,map);
        return root;
    }
    public TreeNode buildTree(int[] inorder, int[] postorder) {
        HashMap<Integer,Integer> map=new HashMap<>();
        for(int i=0;i<inorder.length;i++){
            map.put(inorder[i],i);
        }
        return helper(postorder,postorder.length-1,0,inorder.length-1,map);
    }
}
```

## 9.Sum root to Leaf Number



You are given the `root` of a binary tree containing digits from `0` to `9` only.

Each root-to-leaf path in the tree represents a number.

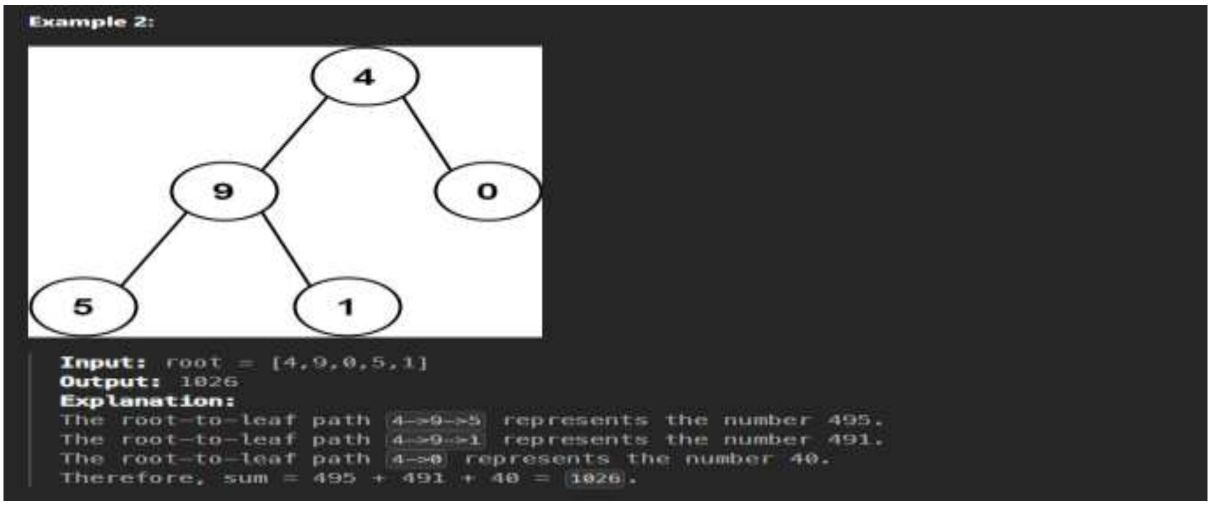
- For example, the root-to-leaf path `1 -> 2 -> 3` represents the number `123`.

Return *the total sum of all root-to-leaf numbers*. Test cases are generated so that the answer will fit in a **32-bit** integer.

A **leaf** node is a node with no children.

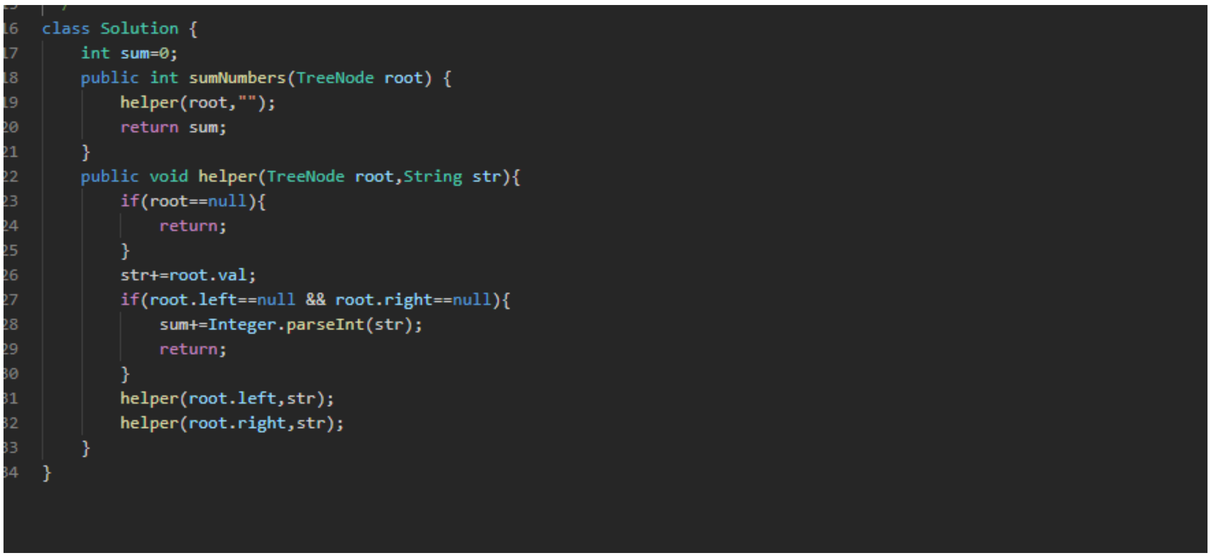
**Example 1:**

```
Input: root = [1,2,3]
Output: 25
Explanation:
The root-to-leaf path 1->2 represents the number 12.
The root-to-leaf path 1->3 represents the number 13.
Therefore, sum = 12 + 13 = 25.
```



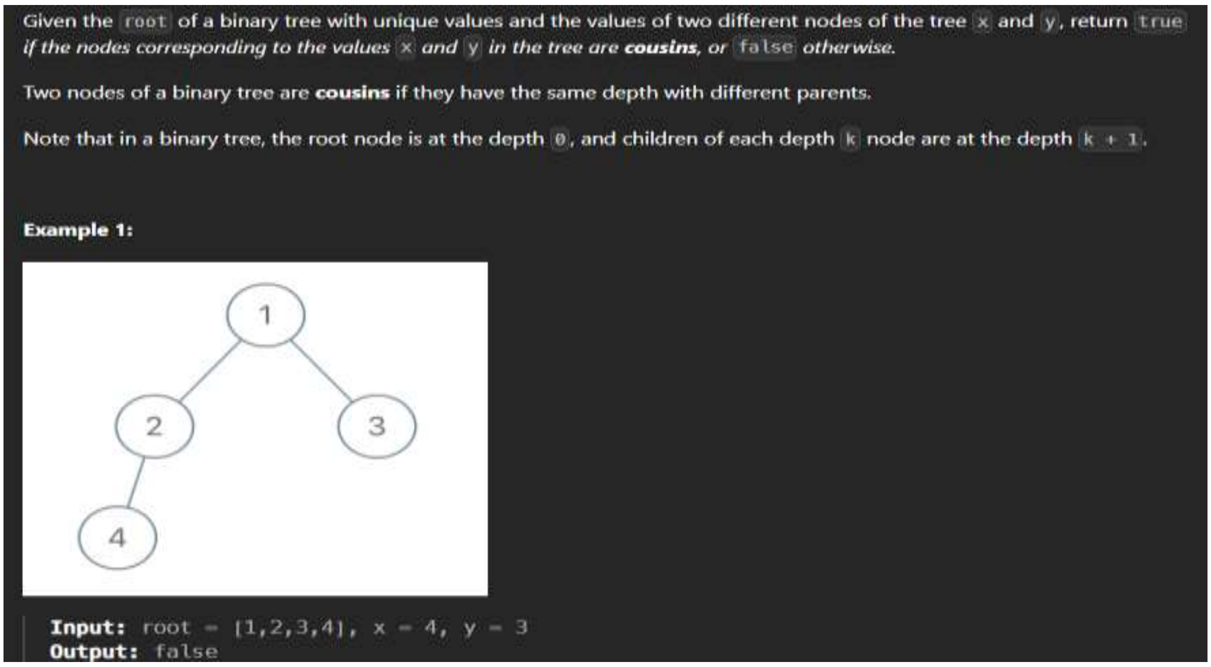
**Example 2:**

```
Input: root = [4,9,0,5,1]
Output: 1026
Explanation:
The root-to-leaf path 4->9->5 represents the number 495.
The root-to-leaf path 4->9->1 represents the number 491.
The root-to-leaf path 4->0 represents the number 40.
Therefore, sum = 495 + 491 + 40 = 1026.
```



```
class Solution {
    int sum=0;
    public int sumNumbers(TreeNode root) {
        helper(root,"");
        return sum;
    }
    public void helper(TreeNode root,String str){
        if(root==null){
            return;
        }
        str+=root.val;
        if(root.left==null && root.right==null){
            sum+=Integer.parseInt(str);
            return;
        }
        helper(root.left,str);
        helper(root.right,str);
    }
}
```

## 10.Cousins in Binary Tree

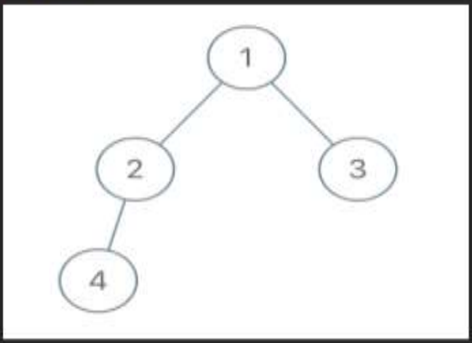


Given the `root` of a binary tree with unique values and the values of two different nodes of the tree `x` and `y`, return `true` *if the nodes corresponding to the values* `x` *and* `y` *in the tree are* ***cousins****, or* `false` *otherwise.*

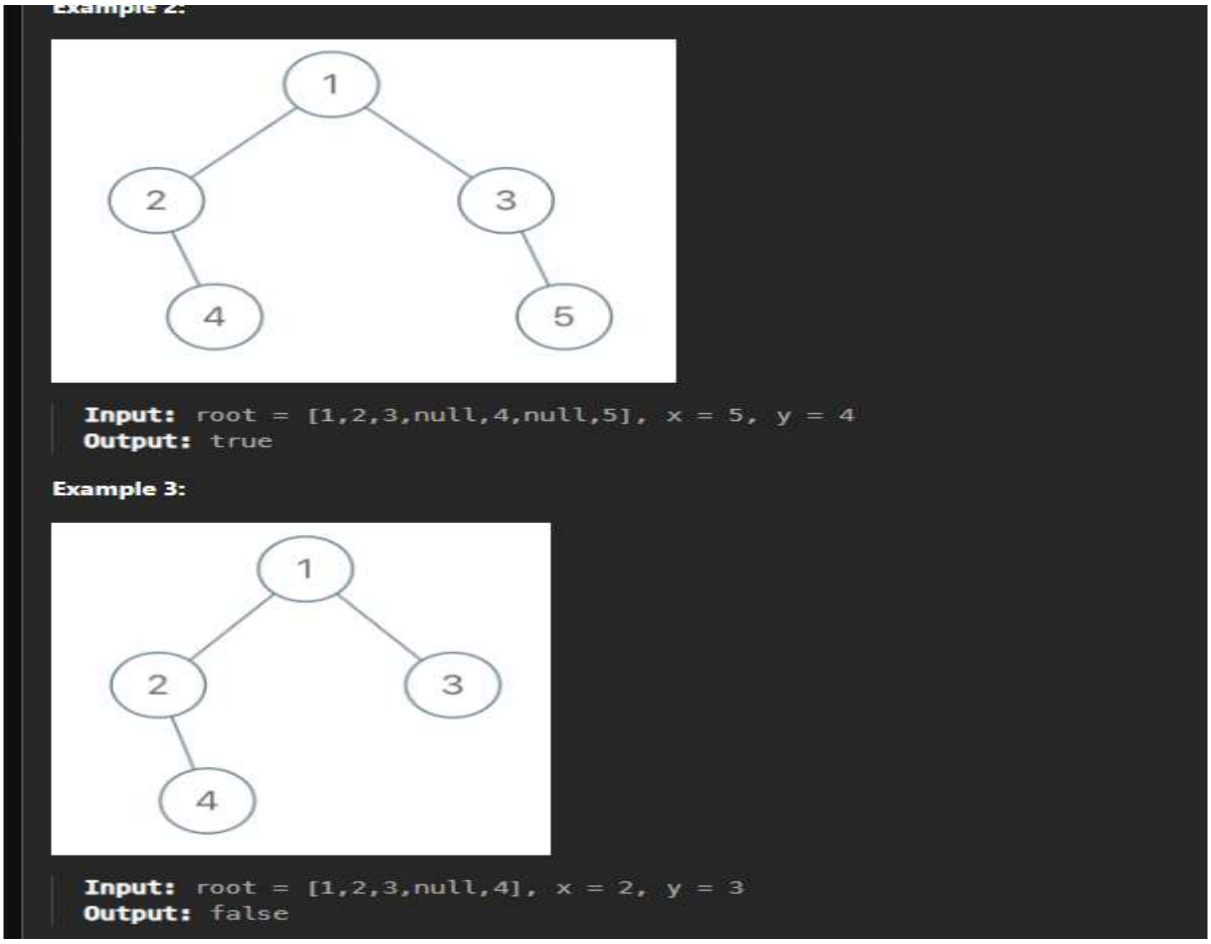
Two nodes of a binary tree are **cousins** if they have the same depth with different parents.

Note that in a binary tree, the root node is at the depth `0`, and children of each depth `k` node are at the depth `k + 1`.

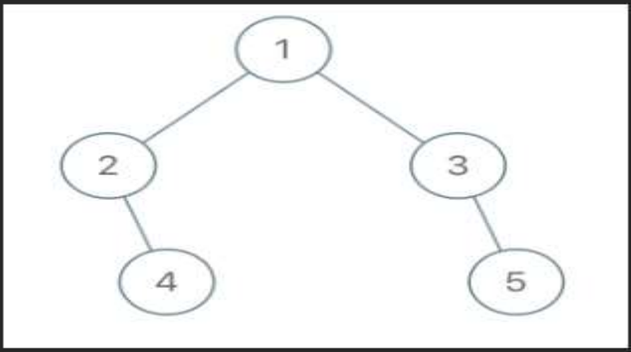
**Example 1:**



```
Input: root = [1,2,3,4], x = 4, y = 3
Output: false
```

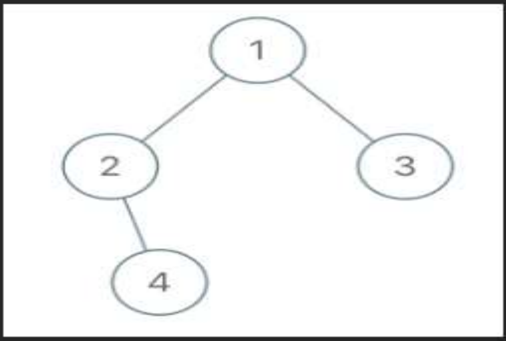


**Example 2:**

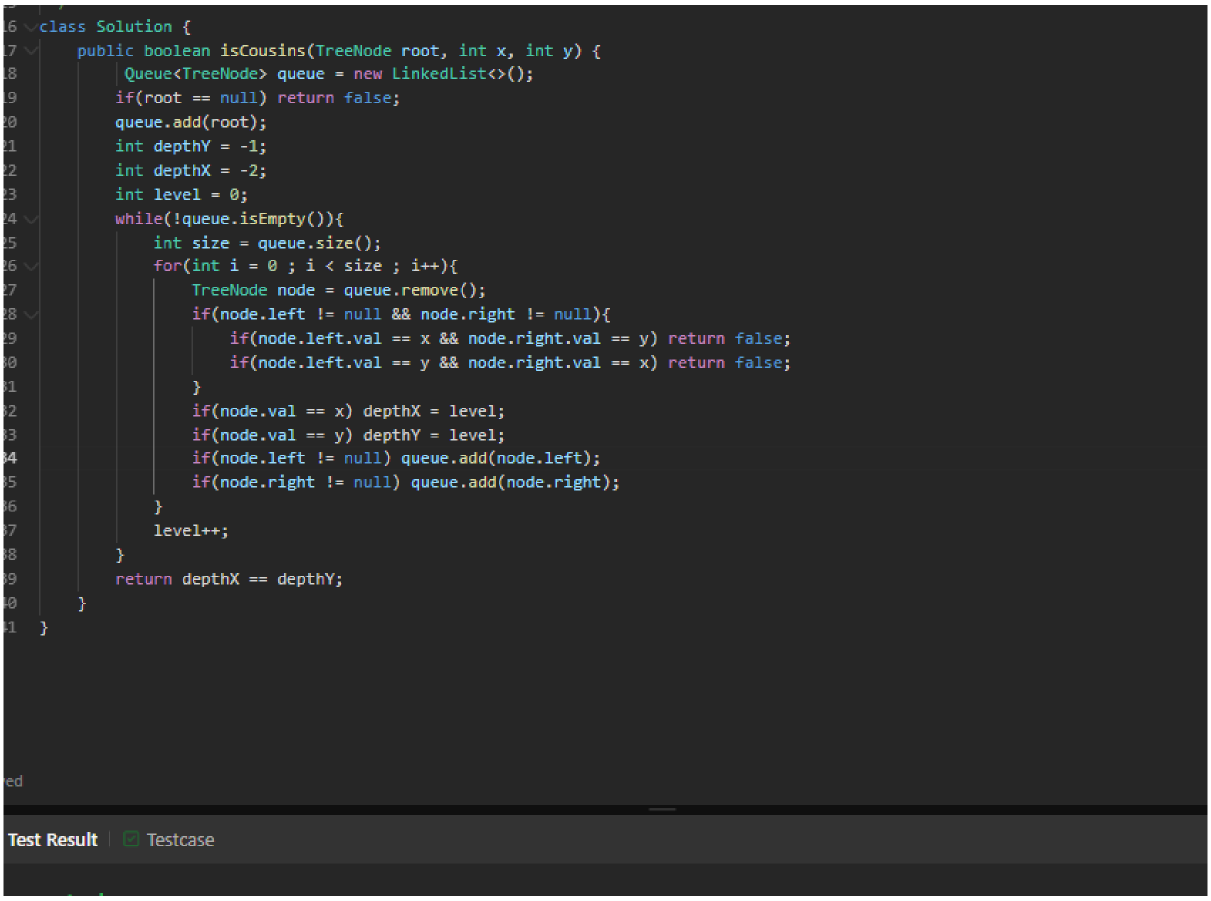


```
Input: root = [1,2,3,null,4,null,5], x = 5, y = 4
Output: true
```

**Example 3:**

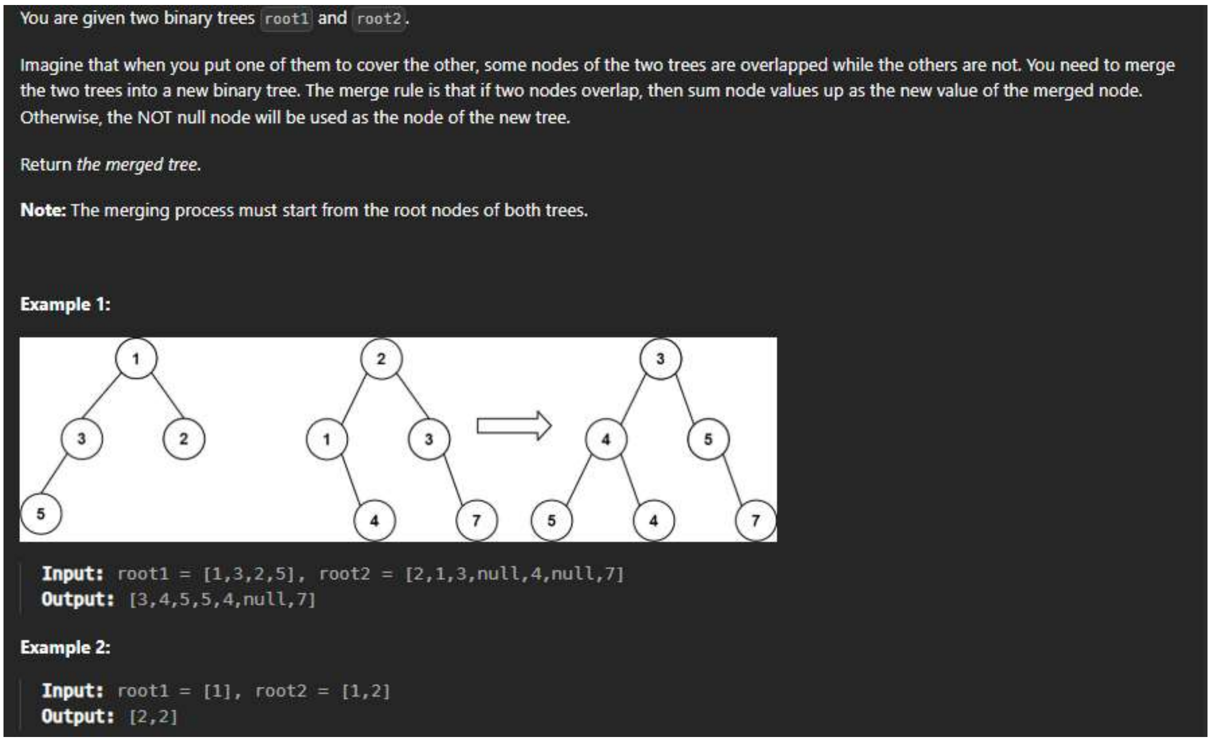


```
Input: root = [1,2,3,null,4], x = 2, y = 3
Output: false
```



```
class Solution {
    public boolean isCousins(TreeNode root, int x, int y) {
         Queue<TreeNode> queue = new LinkedList<>();
        if(root == null) return false;
        queue.add(root);
        int depthY = -1;
        int depthX = -2;
        int level = 0;
        while(!queue.isEmpty()){
            int size = queue.size();
            for(int i = 0 ; i < size ; i++){
                TreeNode node = queue.remove();
                if(node.left != null && node.right != null){
                    if(node.left.val == x && node.right.val == y) return false;
                    if(node.left.val == y && node.right.val == x) return false;
                }
                if(node.val == x) depthX = level;
                if(node.val == y) depthY = level;
                if(node.left != null) queue.add(node.left);
                if(node.right != null) queue.add(node.right);
            }
            level++;
        }
        return depthX == depthY;
    }
}
```

## 11.Merge two Binary Tree



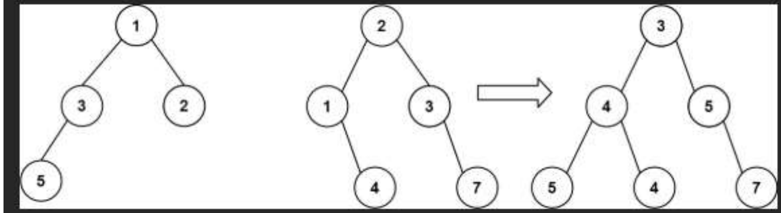
You are given two binary trees `root1` and `root2`.

Imagine that when you put one of them to cover the other, some nodes of the two trees are overlapped while the others are not. You need to merge the two trees into a new binary tree. The merge rule is that if two nodes overlap, then sum node values up as the new value of the merged node. Otherwise, the NOT null node will be used as the node of the new tree.

Return *the merged tree.*

**Note:** The merging process must start from the root nodes of both trees.

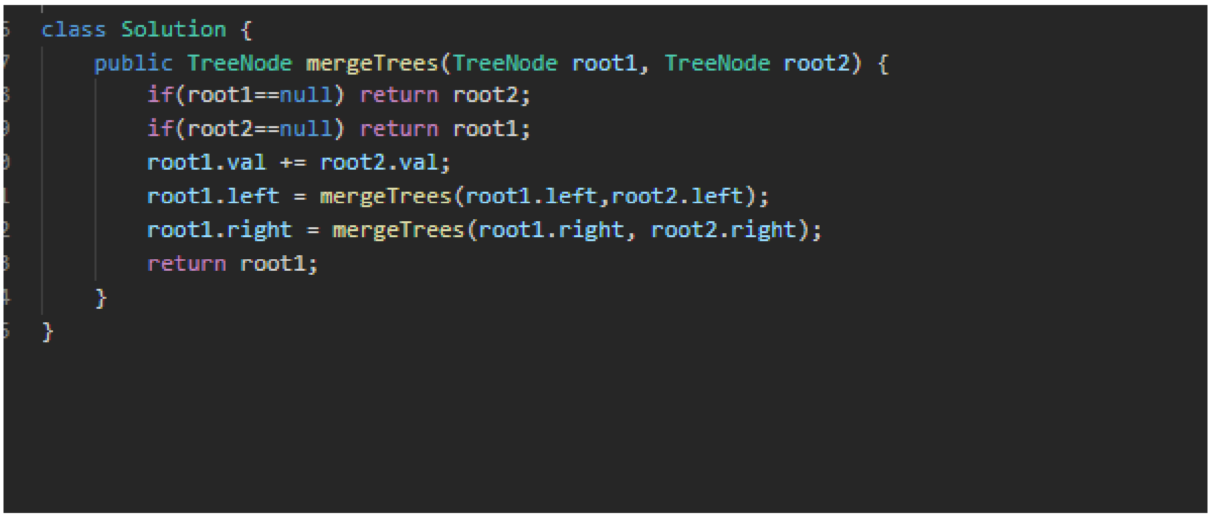
**Example 1:**



```
Input: root1 = [1,3,2,5], root2 = [2,1,3,null,4,null,7]
Output: [3,4,5,5,4,null,7]
```

**Example 2:**

```
Input: root1 = [1], root2 = [1,2]
Output: [2,2]
```



```java
class Solution {
    public TreeNode mergeTrees(TreeNode root1, TreeNode root2) {
        if(root1==null) return root2;
        if(root2==null) return root1;
        root1.val += root2.val;
        root1.left = mergeTrees(root1.left,root2.left);
        root1.right = mergeTrees(root1.right, root2.right);
        return root1;
    }
}
```

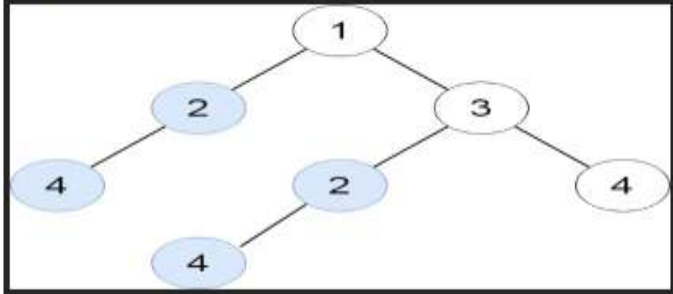
## 12.Find Duplicate Subtree

Given the `root` of a binary tree, return all **duplicate subtrees**.

For each kind of duplicate subtrees, you only need to return the root node of any **one** of them.

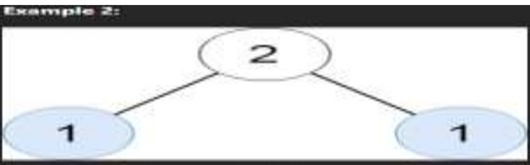
Two trees are **duplicate** if they have the **same structure** with the **same node values**.

**Example 1:**



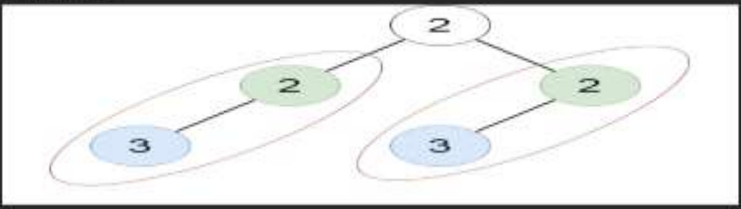
```
Input: root = [1,2,3,4,null,2,4,null,null,4]
Output: [[2,4],[4]]
```

**Example 2:**

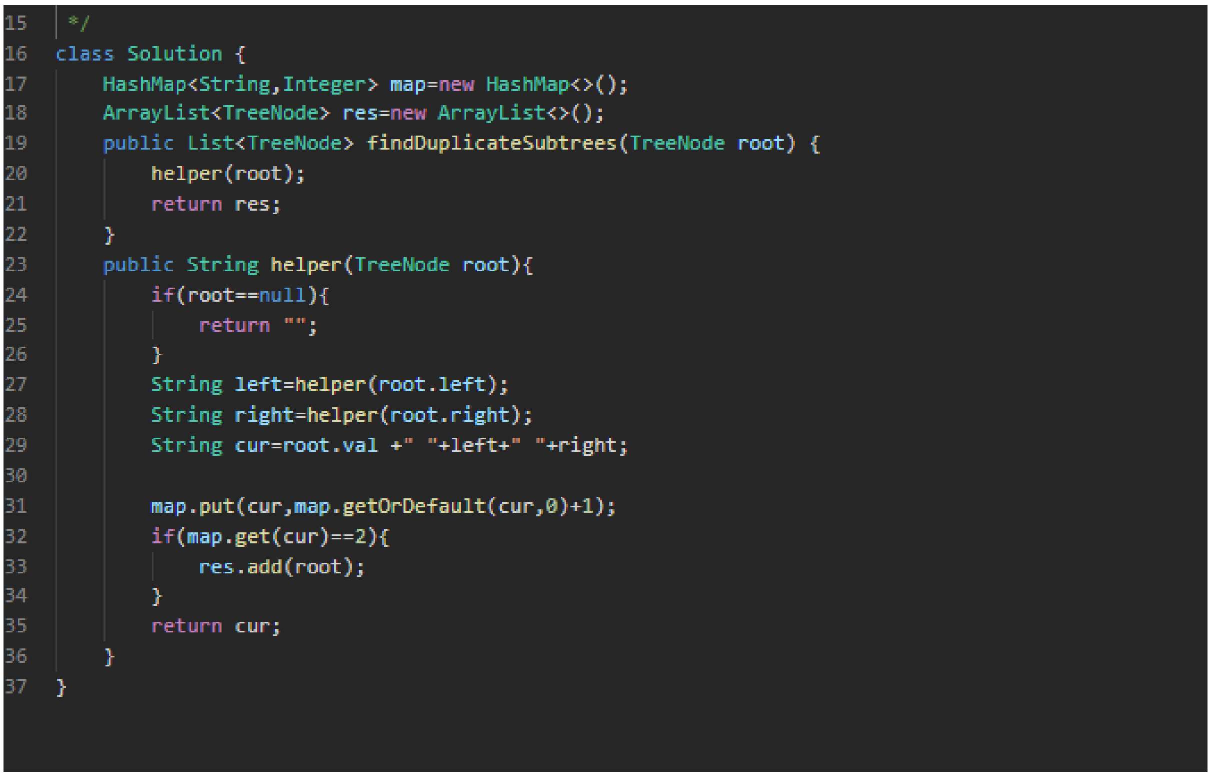


```
Input: root = [2,1,1]
Output: [[1]]
```

**Example 3:**

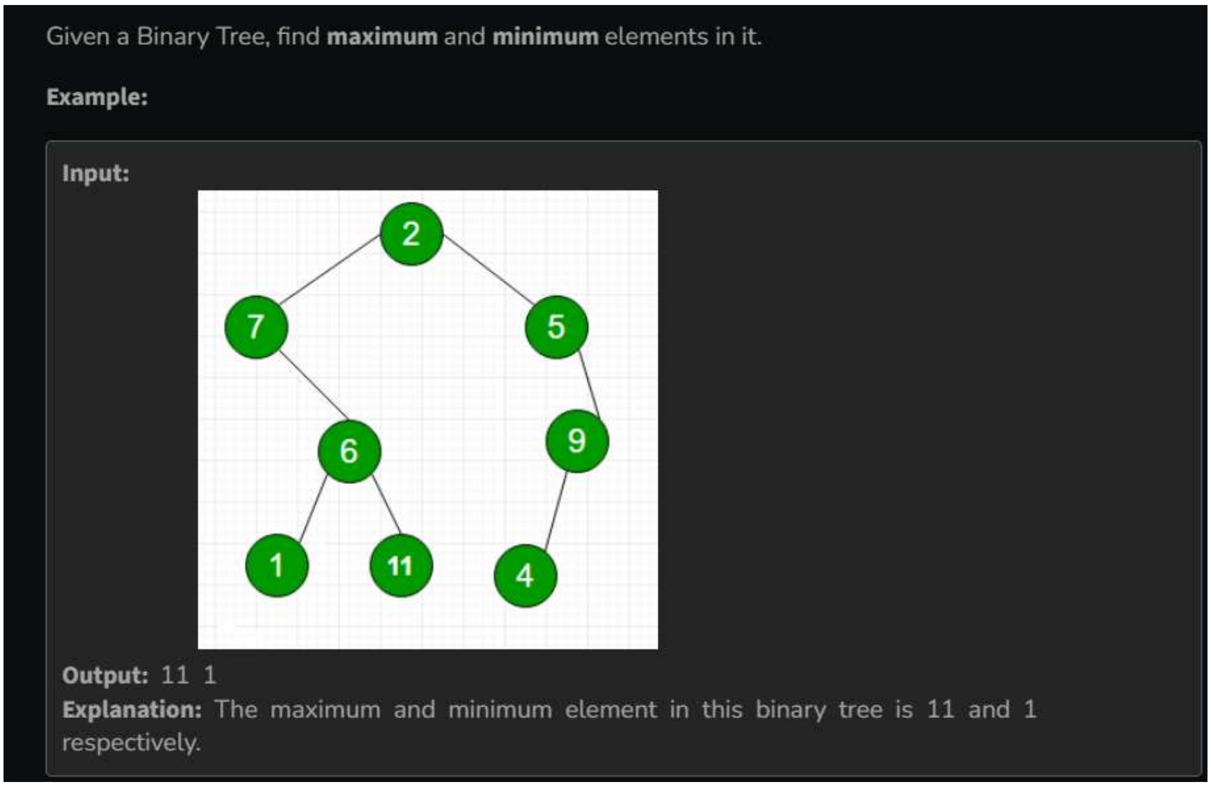


```
Input: root = [2,2,2,3,null,3,null]
Output: [[2,3],[3]]
```



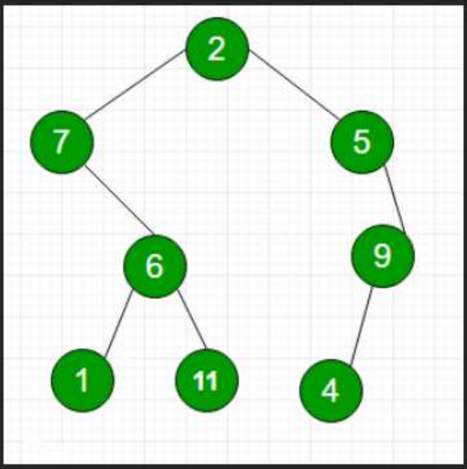
```java
 */
class Solution {
    HashMap<String,Integer> map=new HashMap<>();
    ArrayList<TreeNode> res=new ArrayList<>();
    public List<TreeNode> findDuplicateSubtrees(TreeNode root) {
        helper(root);
        return res;
    }
    public String helper(TreeNode root){
        if(root==null){
            return "";
        }
        String left=helper(root.left);
        String right=helper(root.right);
        String cur=root.val +" "+left+" "+right;

        map.put(cur,map.getOrDefault(cur,0)+1);
        if(map.get(cur)==2){
            res.add(root);
        }
        return cur;
    }
}
```

## 13.Max & Min Element in Binary Tree

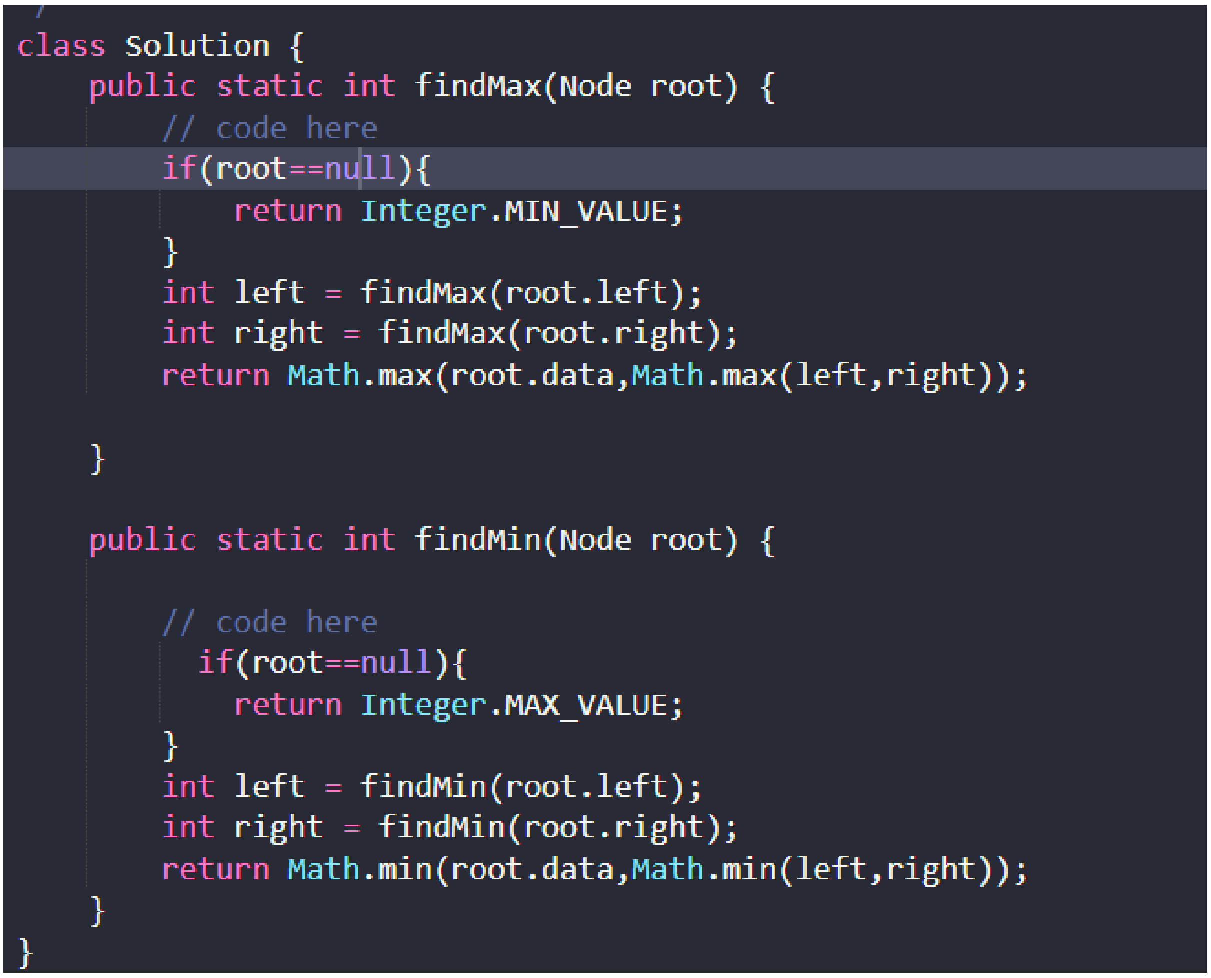

Given a Binary Tree, find **maximum** and **minimum** elements in it.

**Example:**

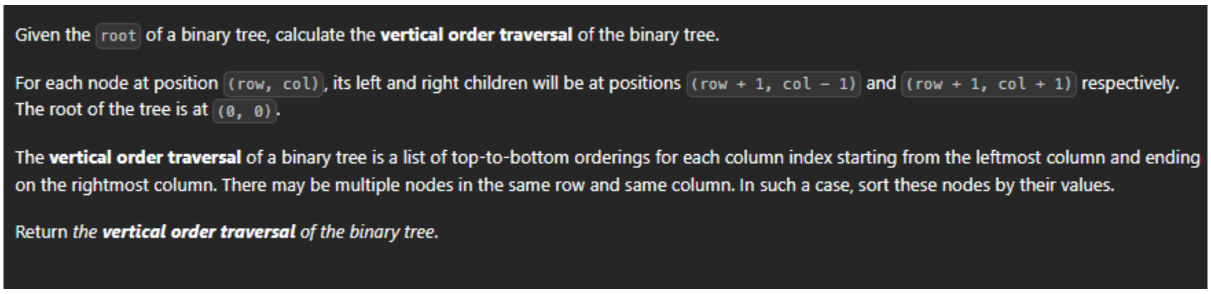
**Input:**



**Output:** 11 1
**Explanation:** The maximum and minimum element in this binary tree is 11 and 1 respectively.


```java
class Solution {
    public static int findMax(Node root) {
        // code here
        if(root==null){
            return Integer.MIN_VALUE;
        }
        int left = findMax(root.left);
        int right = findMax(root.right);
        return Math.max(root.data,Math.max(left,right));

    }

    public static int findMin(Node root) {

        // code here
          if(root==null){
            return Integer.MAX_VALUE;
        }
        int left = findMin(root.left);
        int right = findMin(root.right);
        return Math.min(root.data,Math.min(left,right));
    }
}
```

# 14.Vertical Order Traversal of Binary tree



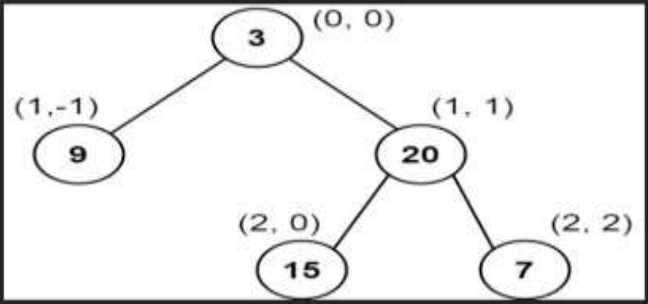
Given the `root` of a binary tree, calculate the **vertical order traversal** of the binary tree.

For each node at position `(row, col)`, its left and right children will be at positions `(row + 1, col - 1)` and `(row + 1, col + 1)` respectively. The root of the tree is at `(0, 0)`.

The **vertical order traversal** of a binary tree is a list of top-to-bottom orderings for each column index starting from the leftmost column and ending on the rightmost column. There may be multiple nodes in the same row and same column. In such a case, sort these nodes by their values.

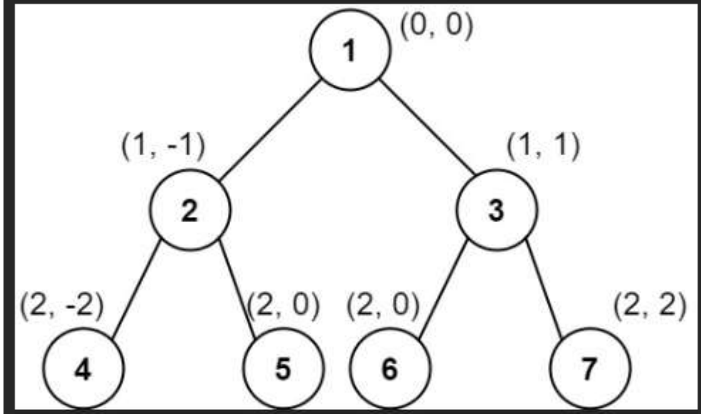
Return *the **vertical order traversal** of the binary tree.*

**Example 1:**

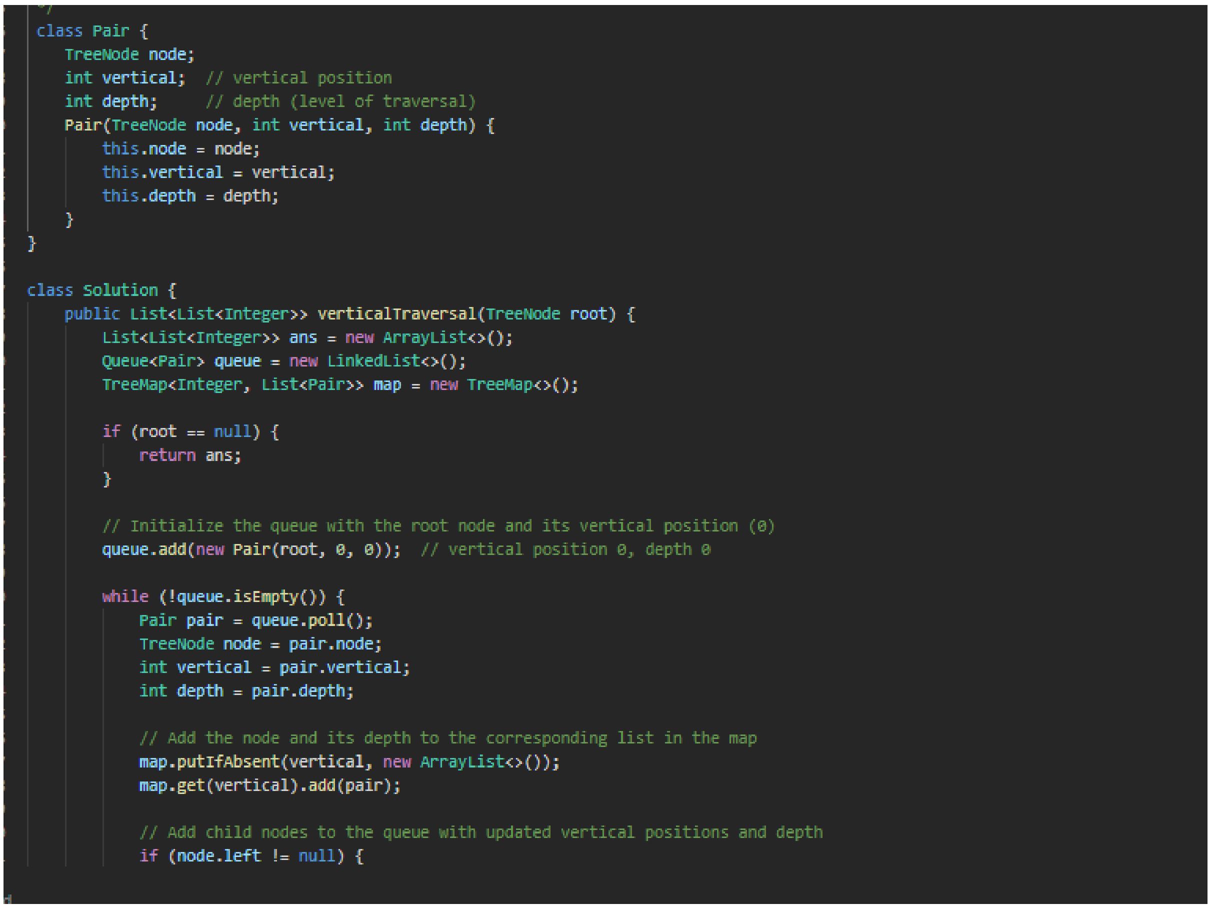


```
Input: root = [3,9,20,null,null,15,7]
Output: [[9],[3,15],[20],[7]]
Explanation:
Column -1: Only node 9 is in this column.
Column 0: Nodes 3 and 15 are in this column in that order from top to bottom.
Column 1: Only node 20 is in this column.
Column 2: Only node 7 is in this column.
```

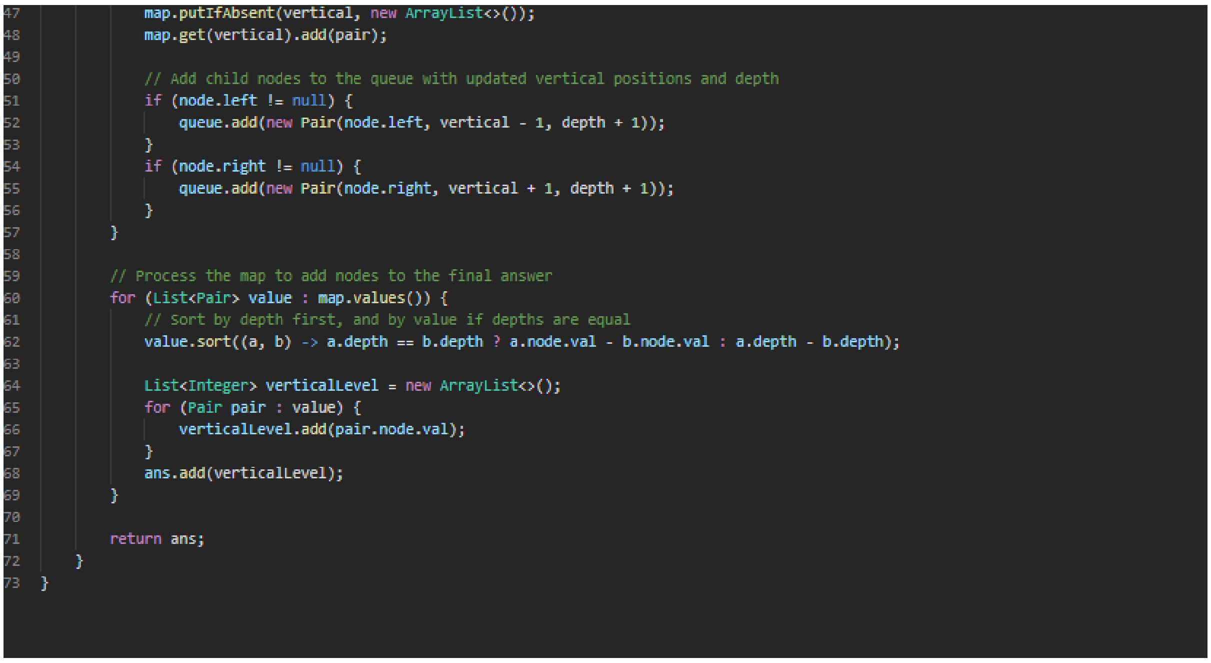
**Example 2:**



```
Input: root = [1,2,3,4,5,6,7]
Output: [[4],[2],[1,5,6],[3],[7]]
Explanation:
Column -2: Only node 4 is in this column.
Column -1: Only node 2 is in this column.
Column 0: Nodes 1, 5, and 6 are in this column.
          1 is at the top, so it comes first.
          5 and 6 are at the same position (2, 0), so we order them by their value, 5 before 6.
Column 1: Only node 3 is in this column.
Column 2: Only node 7 is in this column.
```



```java
*/
class Pair {
    TreeNode node;
    int vertical;  // vertical position
    int depth;     // depth (level of traversal)
    Pair(TreeNode node, int vertical, int depth) {
        this.node = node;
        this.vertical = vertical;
        this.depth = depth;
    }
}

class Solution {
    public List<List<Integer>> verticalTraversal(TreeNode root) {
        List<List<Integer>> ans = new ArrayList<>();
        Queue<Pair> queue = new LinkedList<>();
        TreeMap<Integer, List<Pair>> map = new TreeMap<>();

        if (root == null) {
            return ans;
        }

        // Initialize the queue with the root node and its vertical position (0)
        queue.add(new Pair(root, 0, 0));  // vertical position 0, depth 0

        while (!queue.isEmpty()) {
            Pair pair = queue.poll();
            TreeNode node = pair.node;
            int vertical = pair.vertical;
            int depth = pair.depth;

            // Add the node and its depth to the corresponding list in the map
            map.putIfAbsent(vertical, new ArrayList<>());
            map.get(vertical).add(pair);

            // Add child nodes to the queue with updated vertical positions and depth
            if (node.left != null) {
```



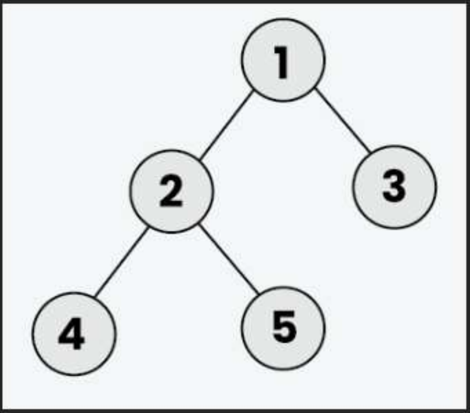
```java
            map.putIfAbsent(vertical, new ArrayList<>());
            map.get(vertical).add(pair);

            // Add child nodes to the queue with updated vertical positions and depth
            if (node.left != null) {
                queue.add(new Pair(node.left, vertical - 1, depth + 1));
            }
            if (node.right != null) {
                queue.add(new Pair(node.right, vertical + 1, depth + 1));
            }
        }

        // Process the map to add nodes to the final answer
        for (List<Pair> value : map.values()) {
            // Sort by depth first, and by value if depths are equal
            value.sort((a, b) -> a.depth == b.depth ? a.node.val - b.node.val : a.depth - b.depth);

            List<Integer> verticalLevel = new ArrayList<>();
            for (Pair pair : value) {
                verticalLevel.add(pair.node.val);
            }
            ans.add(verticalLevel);
        }

        return ans;
    }
}
```

## 15.Left View Of Binary Tree

You are given the **root** of a binary tree. Your task is to return the ***left view*** of the binary tree. The ***left view*** of a binary tree is the set of nodes visible when the tree is **viewed** from the **left side**.

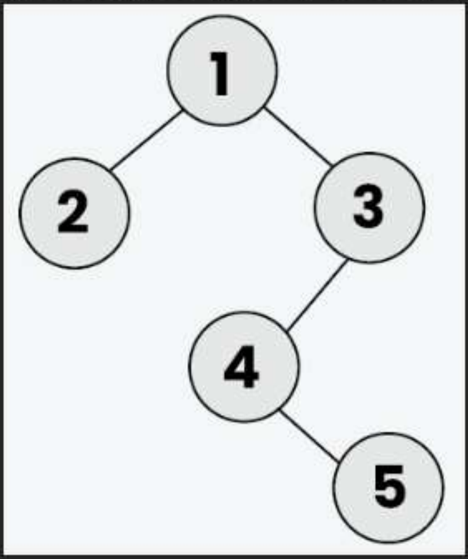
If the tree is empty, return an **empty list**.

**Input:** root[] = [1, 2, 3, 4, 5, N, N]

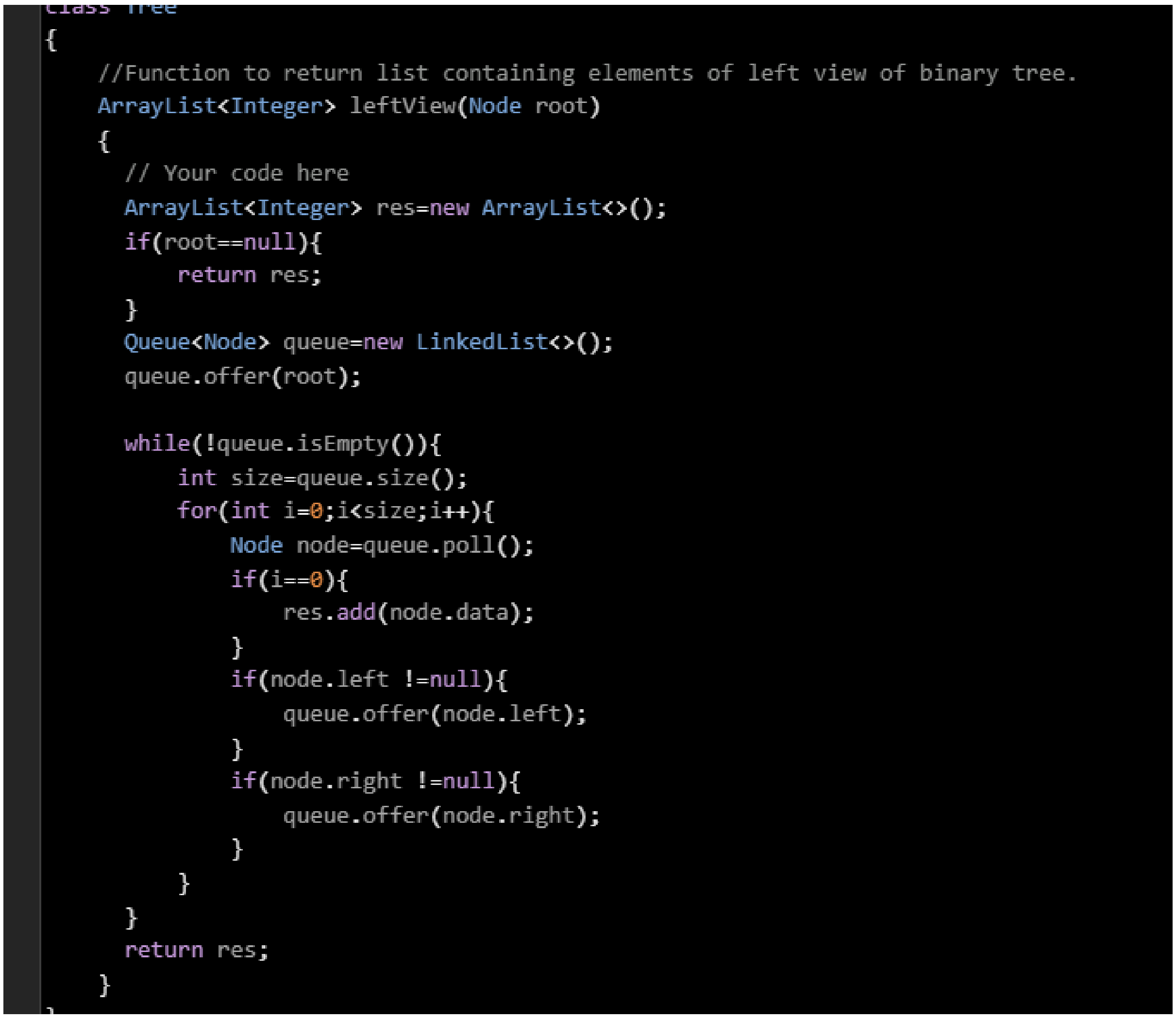


**Output:** [1, 2, 4]
**Explanation:** From the left side of the tree, only the nodes 1, 2, and 4 are visible.

**Input:** root[] = [1, 2, 3, N, N, 4, N, N, 5, N, N]

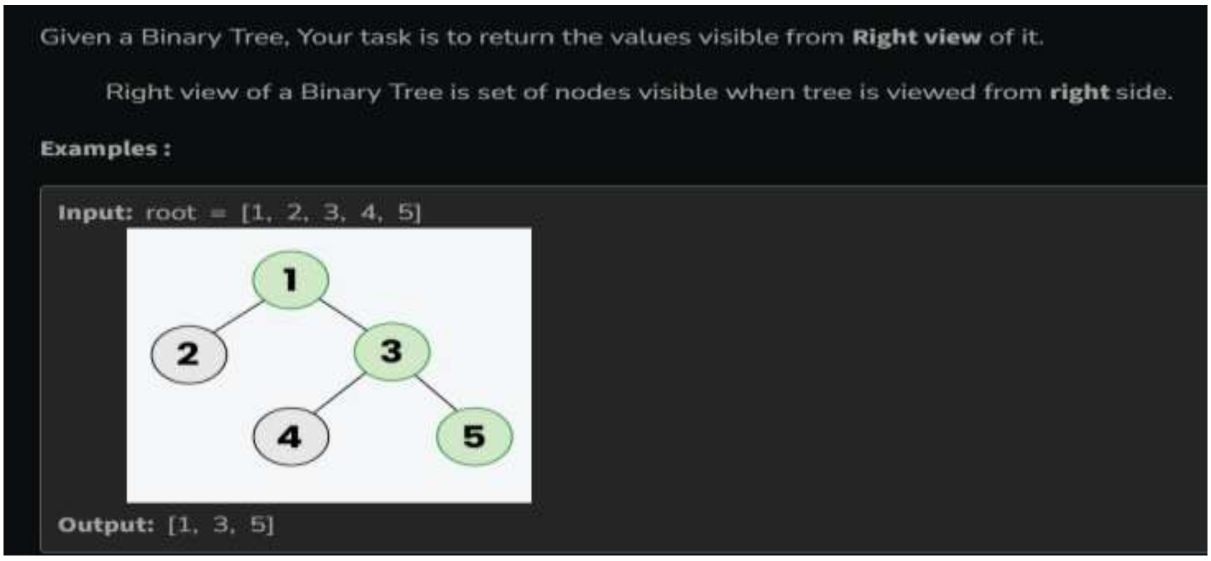


**Output:** [1, 2, 4, 5]
**Explanation:** From the left side of the tree, the nodes 1, 2, 4, and 5 are visible.



```
class Tree
{
    //Function to return list containing elements of left view of binary tree.
    ArrayList<Integer> leftView(Node root)
    {
      // Your code here
      ArrayList<Integer> res=new ArrayList<>();
      if(root==null){
          return res;
      }
      Queue<Node> queue=new LinkedList<>();
      queue.offer(root);

      while(!queue.isEmpty()){
          int size=queue.size();
          for(int i=0;i<size;i++){
              Node node=queue.poll();
              if(i==0){
                  res.add(node.data);
              }
              if(node.left !=null){
                  queue.offer(node.left);
              }
              if(node.right !=null){
                  queue.offer(node.right);
              }
          }
      }
      return res;
    }
}
```

## 16.Right view of Binary Tree

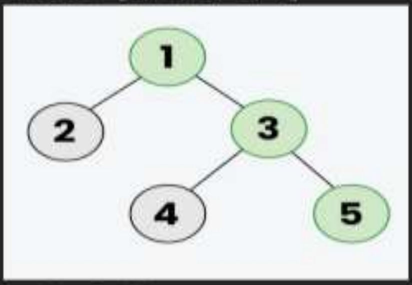


Given a Binary Tree, Your task is to return the values visible from **Right view** of it.

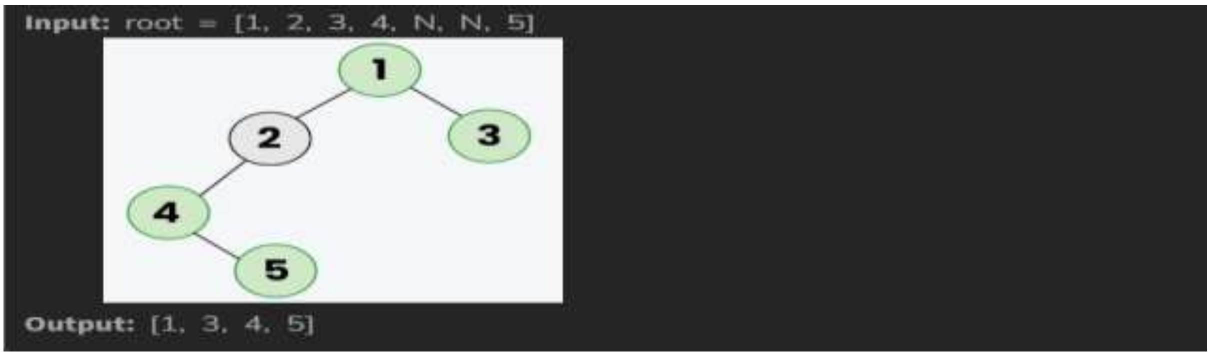
Right view of a Binary Tree is set of nodes visible when tree is viewed from **right** side.

**Examples :**

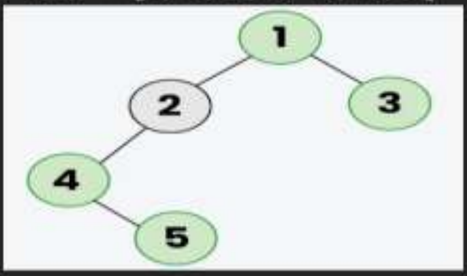
```
Input: root = [1, 2, 3, 4, 5]
```



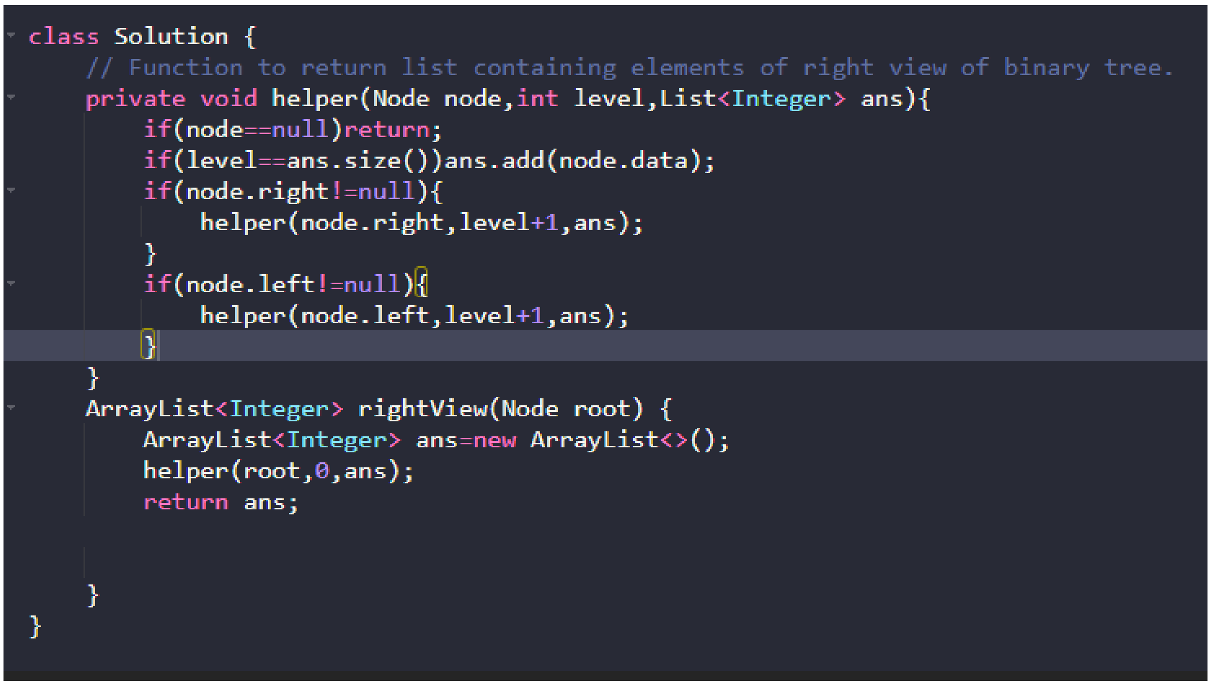
```
Output: [1, 3, 5]
```



```
Input: root = [1, 2, 3, 4, N, N, 5]
```



```
Output: [1, 3, 4, 5]
```



```
class Solution {
    // Function to return list containing elements of right view of binary tree.
    private void helper(Node node,int level,List<Integer> ans){
        if(node==null)return;
        if(level==ans.size())ans.add(node.data);
        if(node.right!=null){
            helper(node.right,level+1,ans);
        }
        if(node.left!=null){
            helper(node.left,level+1,ans);
        }
    }
    ArrayList<Integer> rightView(Node root) {
        ArrayList<Integer> ans=new ArrayList<>();
        helper(root,0,ans);
        return ans;


    }
}
```

## 17.Top View of Binary Tree

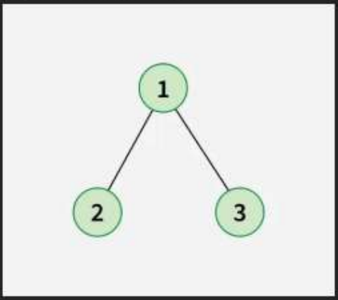
You are given a binary tree, and your task is to return its **top view**. The top view of a binary tree is the set of nodes visible when the tree is viewed from the top.

**Note:**

- Return the nodes from the leftmost node to the rightmost node.
- If two nodes are at the same position (horizontal distance) and are outside the shadow of the tree, consider the leftmost node only.

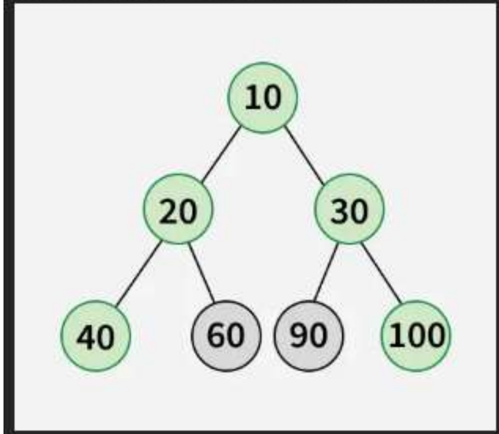
**Examples:**

**Input:** root[] = [1, 2, 3]

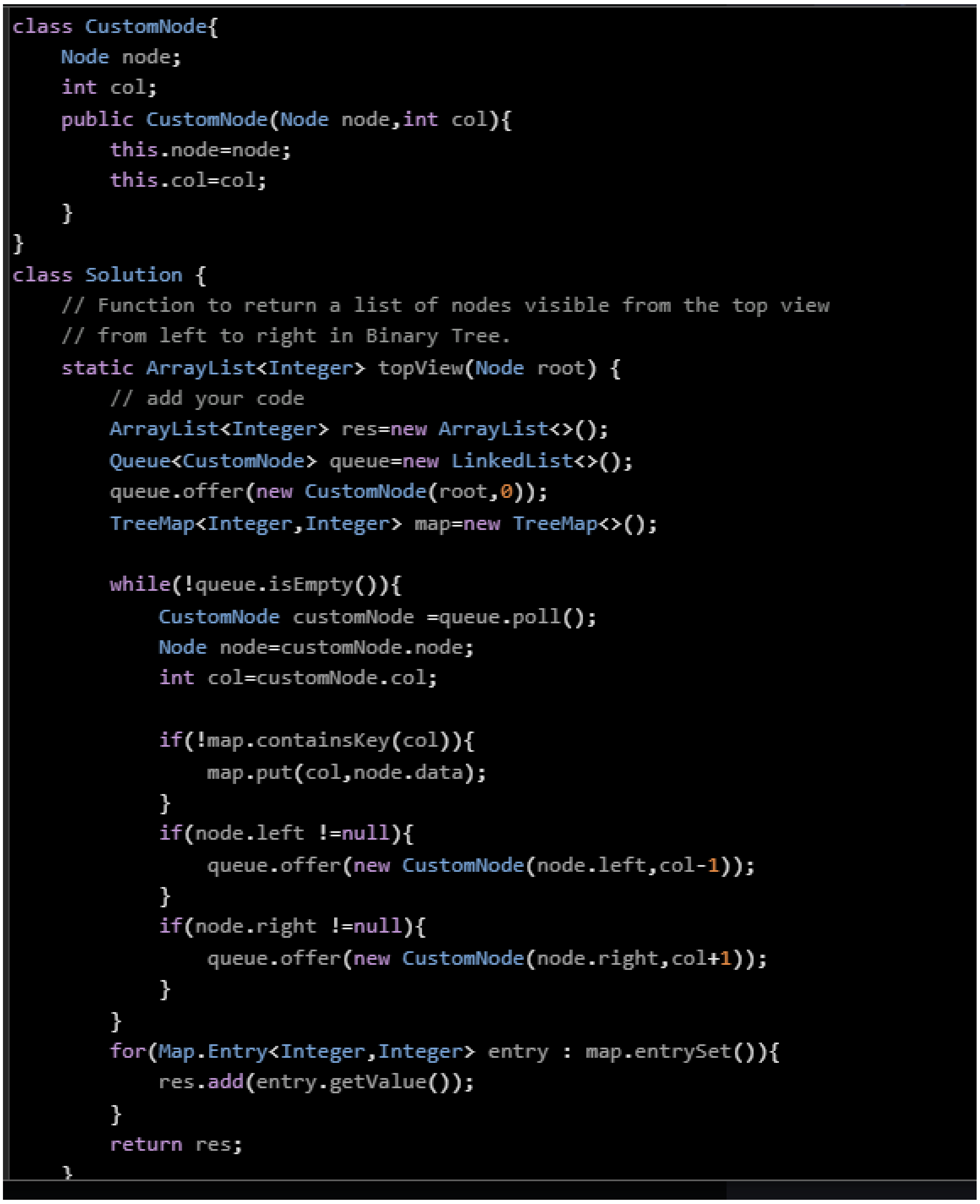


**Output:** [2, 1, 3]

**Input:** root[] = [10, 20, 30, 40, 60, 90, 100]

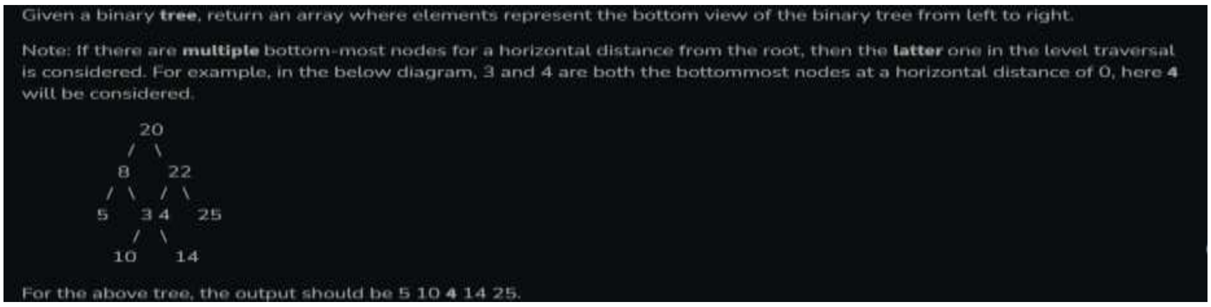


**Output:** [40, 20, 10, 30, 100]
**Explanation:** The root 10 is visible.
On the left, 40 is the leftmost node and visible, followed by 20.
On the right, 30 and 100 are visible. Thus, the top view is 40 20 10 30 100.



```java
class CustomNode{
    Node node;
    int col;
    public CustomNode(Node node,int col){
        this.node=node;
        this.col=col;
    }
}
class Solution {
    // Function to return a list of nodes visible from the top view
    // from left to right in Binary Tree.
    static ArrayList<Integer> topView(Node root) {
        // add your code
        ArrayList<Integer> res=new ArrayList<>();
        Queue<CustomNode> queue=new LinkedList<>();
        queue.offer(new CustomNode(root,0));
        TreeMap<Integer,Integer> map=new TreeMap<>();

        while(!queue.isEmpty()){
            CustomNode customNode =queue.poll();
            Node node=customNode.node;
            int col=customNode.col;

            if(!map.containsKey(col)){
                map.put(col,node.data);
            }
            if(node.left !=null){
                queue.offer(new CustomNode(node.left,col-1));
            }
            if(node.right !=null){
                queue.offer(new CustomNode(node.right,col+1));
            }
        }
        for(Map.Entry<Integer,Integer> entry : map.entrySet()){
            res.add(entry.getValue());
        }
        return res;
    }
```

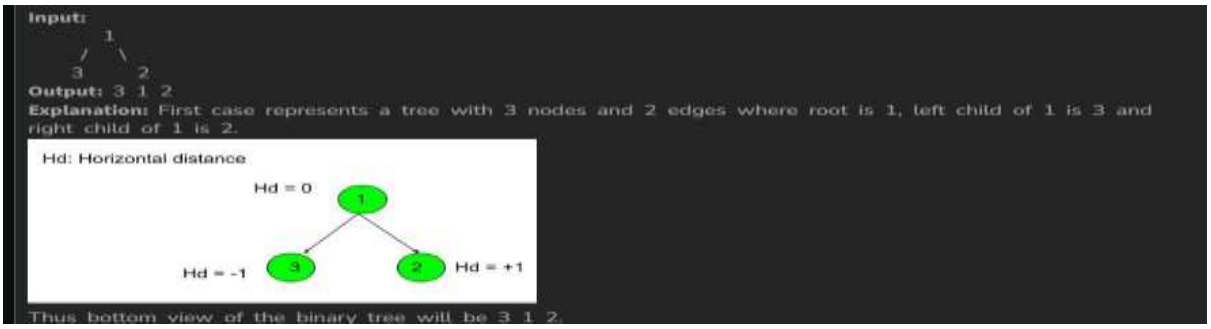
## 18.Bottom View of Binary Tree


Given a binary **tree**, return an array where elements represent the bottom view of the binary tree from left to right.

Note: If there are **multiple** bottom-most nodes for a horizontal distance from the root, then the **latter** one in the level traversal is considered. For example, in the below diagram, 3 and 4 are both the bottommost nodes at a horizontal distance of 0, here **4** will be considered.

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

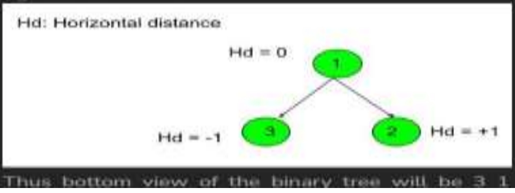
For the above tree, the output should be 5 10 **4** 14 25.


**Input:**

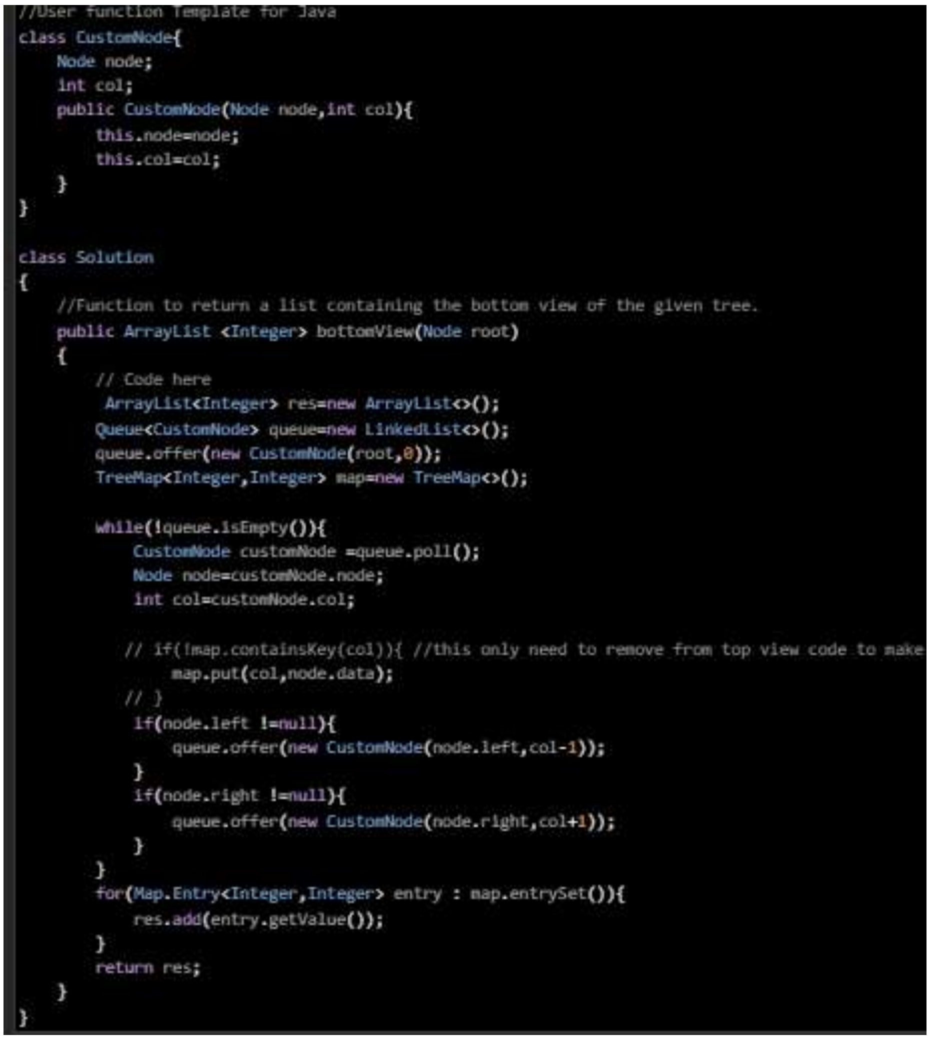
```
          1
        /   \
       3     2
```

**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.

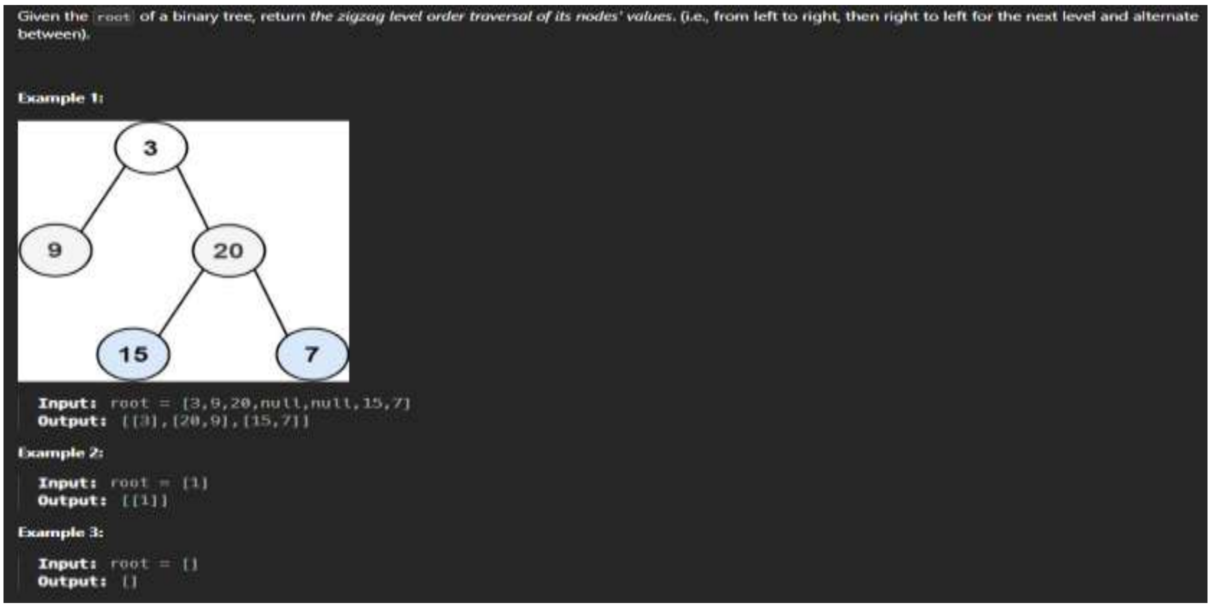


Thus bottom view of the binary tree will be 3 1 2.



```java
//User function Template for Java
class CustomNode{
    Node node;
    int col;
    public CustomNode(Node node,int col){
        this.node=node;
        this.col=col;
    }
}

class Solution
{
    //Function to return a list containing the bottom view of the given tree.
    public ArrayList <Integer> bottomView(Node root)
    {
        // Code here
         ArrayList<Integer> res=new ArrayList<>();
        Queue<CustomNode> queue=new LinkedList<>();
        queue.offer(new CustomNode(root,0));
        TreeMap<Integer,Integer> map=new TreeMap<>();

        while(!queue.isEmpty()){
            CustomNode customNode =queue.poll();
            Node node=customNode.node;
            int col=customNode.col;

           // if(!map.containsKey(col)){ //this only need to remove from top view code to make
                map.put(col,node.data);
           // }
            if(node.left !=null){
                queue.offer(new CustomNode(node.left,col-1));
            }
            if(node.right !=null){
                queue.offer(new CustomNode(node.right,col+1));
            }
        }
        for(Map.Entry<Integer,Integer> entry : map.entrySet()){
            res.add(entry.getValue());
        }
        return res;
    }
}
```

## 19.Binary Tree Zigzag Level Order Traversal



Given the `root` of a binary tree, return *the zigzag level order traversal of its nodes' values*. (i.e., from left to right, then right to left for the next level and alternate between).

**Example 1:**

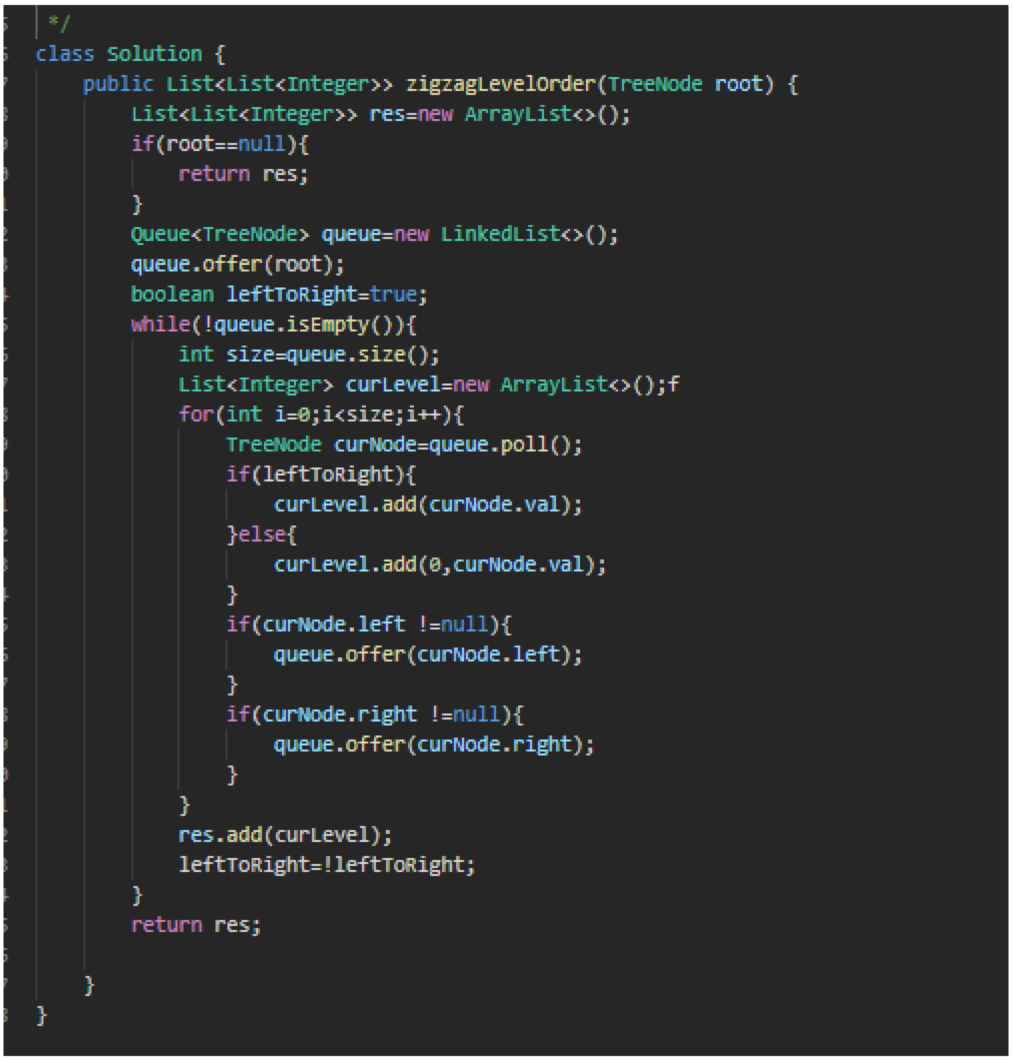
```
Input: root = [3,9,20,null,null,15,7]
Output: [[3],[20,9],[15,7]]
```

**Example 2:**

```
Input: root = [1]
Output: [[1]]
```

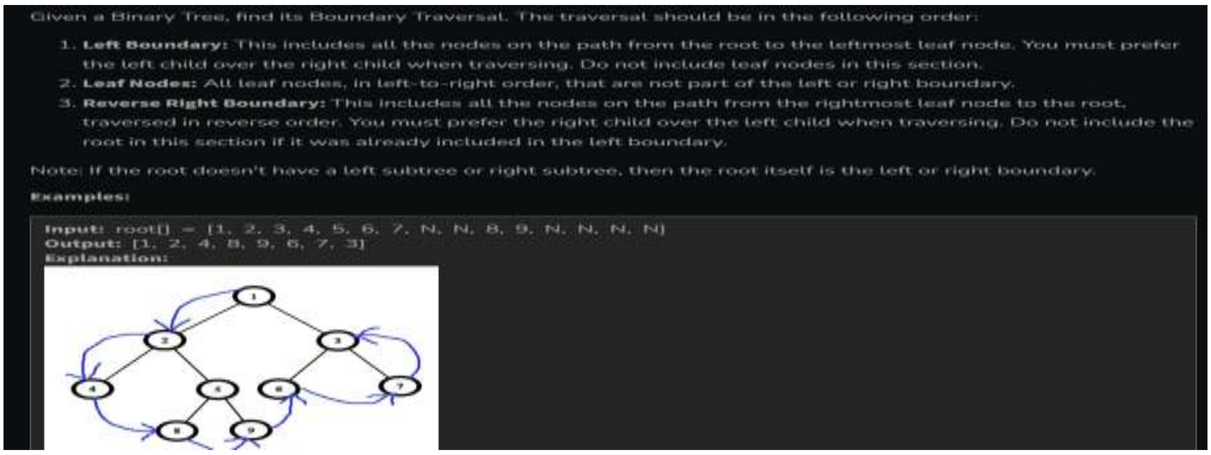
**Example 3:**

```
Input: root = []
Output: []
```



```
    */
class Solution {
    public List<List<Integer>> zigzagLevelOrder(TreeNode root) {
        List<List<Integer>> res=new ArrayList<>();
        if(root==null){
            return res;
        }
        Queue<TreeNode> queue=new LinkedList<>();
        queue.offer(root);
        boolean leftToRight=true;
        while(!queue.isEmpty()){
            int size=queue.size();
            List<Integer> curLevel=new ArrayList<>();f
            for(int i=0;i<size;i++){
                TreeNode curNode=queue.poll();
                if(leftToRight){
                    curLevel.add(curNode.val);
                }else{
                    curLevel.add(0,curNode.val);
                }
                if(curNode.left !=null){
                    queue.offer(curNode.left);
                }
                if(curNode.right !=null){
                    queue.offer(curNode.right);
                }
            }
            res.add(curLevel);
            leftToRight=!leftToRight;
        }
        return res;

    }
}
```

## 20.Tree Boundary Traversal

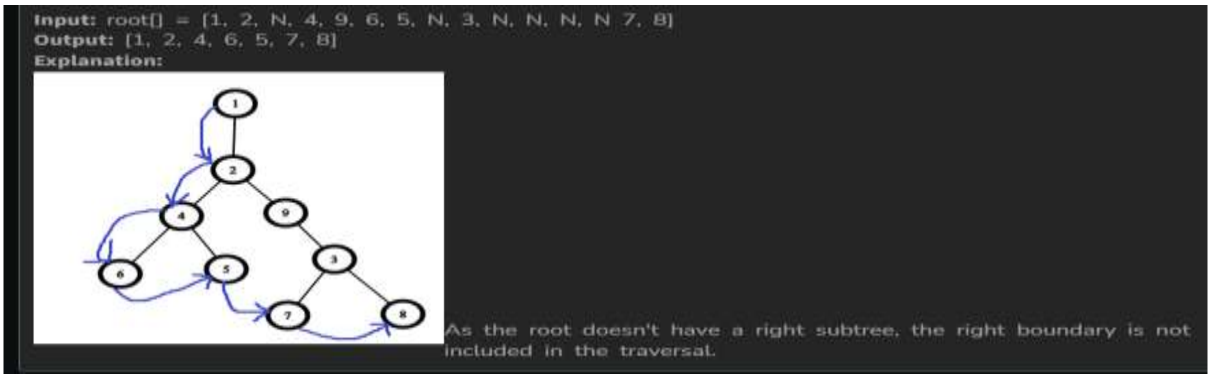

Given a Binary Tree, find its Boundary Traversal. The traversal should be in the following order:

1. **Left Boundary:** This includes all the nodes on the path from the root to the leftmost leaf node. You must prefer the left child over the right child when traversing. Do not include leaf nodes in this section.
2. **Leaf Nodes:** All leaf nodes, in left-to-right order, that are not part of the left or right boundary.
3. **Reverse Right Boundary:** This includes all the nodes on the path from the rightmost leaf node to the root, traversed in reverse order. You must prefer the right child over the left child when traversing. Do not include the root in this section if it was already included in the left boundary.

Note: If the root doesn't have a left subtree or right subtree, then the root itself is the left or right boundary.

**Examples:**

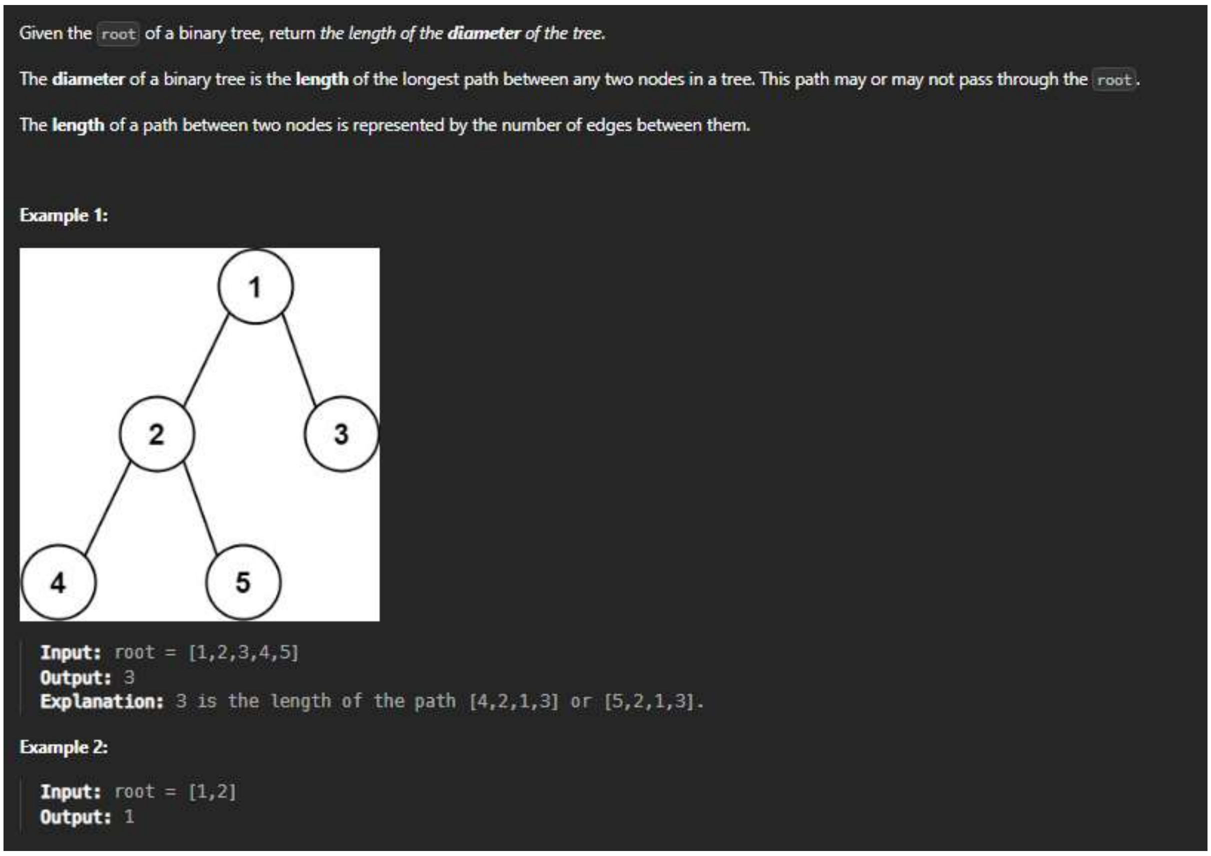
```
Input: root[] = [1, 2, 3, 4, 5, 6, 7, N, N, 8, 9, N, N, N, N]
Output: [1, 2, 4, 8, 9, 6, 7, 3]
Explanation:
```


```
Input: root[] = [1, 2, N, 4, 9, 6, 5, N, 3, N, N, N, N 7, 8]
Output: [1, 2, 4, 6, 5, 7, 8]
Explanation:
```

As the root doesn't have a right subtree, the right boundary is not included in the traversal.

```java
class Solution {

    public void helper(Node root,ArrayList<Integer> res,boolean isLeftBoundary,boolean isRightBoundary){
        if(root==null){
            return;
        }
        if(isLeftBoundary || (root.left==null && root.right==null)){
            res.add(root.data);
        }
        helper(root.left,res,isLeftBoundary,isRightBoundary && root.right==null);
        helper(root.right,res,isLeftBoundary && root.left==null ,isRightBoundary);

        if(isRightBoundary && !(isLeftBoundary || (root.left==null && root.right==null))){
            res.add(root.data);
        }
    }
    ArrayList<Integer> boundaryTraversal(Node node) {
        // code here
        ArrayList<Integer> res=new ArrayList<>();
        res.add(node.data);

        helper(node.left,res,true,false);
        helper(node.right,res,false,true);
        return res;
    }
}
```

## 21.Diameter of Binary Tree

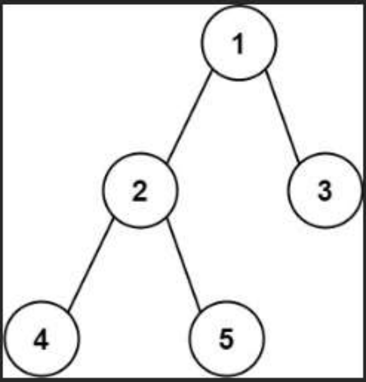


Given the `root` of a binary tree, return *the length of the **diameter** of the tree.*

The **diameter** of a binary tree is the **length** of the longest path between any two nodes in a tree. This path may or may not pass through the `root`.

The **length** of a path between two nodes is represented by the number of edges between them.

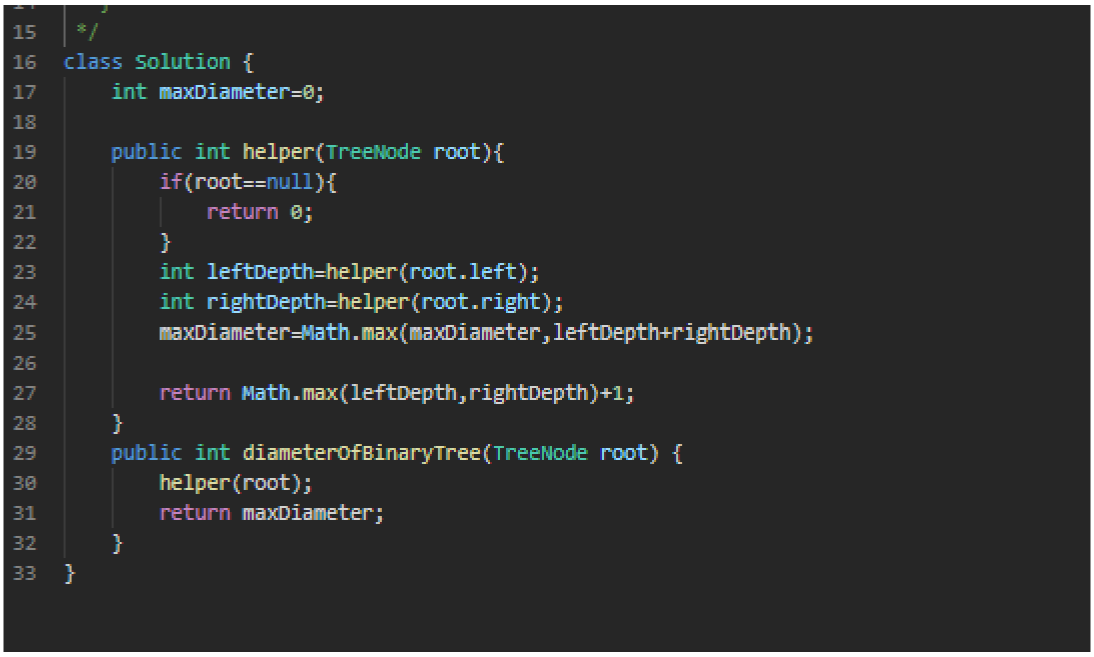
**Example 1:**



```
Input: root = [1,2,3,4,5]
Output: 3
Explanation: 3 is the length of the path [4,2,1,3] or [5,2,1,3].
```

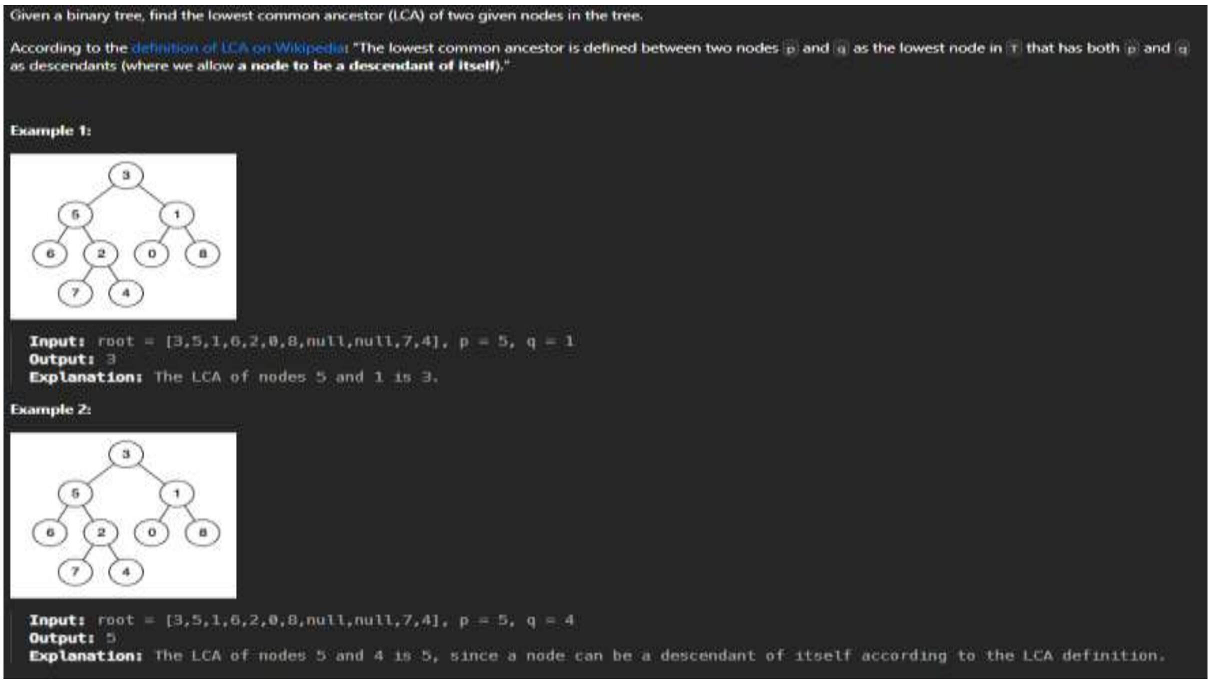
**Example 2:**

```
Input: root = [1,2]
Output: 1
```



```
 */
class Solution {
    int maxDiameter=0;

    public int helper(TreeNode root){
        if(root==null){
            return 0;
        }
        int leftDepth=helper(root.left);
        int rightDepth=helper(root.right);
        maxDiameter=Math.max(maxDiameter,leftDepth+rightDepth);

        return Math.max(leftDepth,rightDepth)+1;
    }
    public int diameterOfBinaryTree(TreeNode root) {
        helper(root);
        return maxDiameter;
    }
}
```

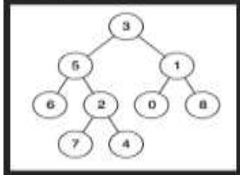
## 22.Lowest common Ansestor of Binary Tree



Given a binary tree, find the lowest common ancestor (LCA) of two given nodes in the tree.

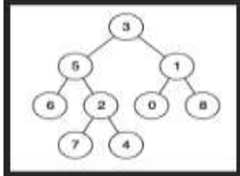
According to the definition of LCA on Wikipedia: "The lowest common ancestor is defined between two nodes `p` and `q` as the lowest node in `T` that has both `p` and `q` as descendants (where we allow **a node to be a descendant of itself**)."

**Example 1:**

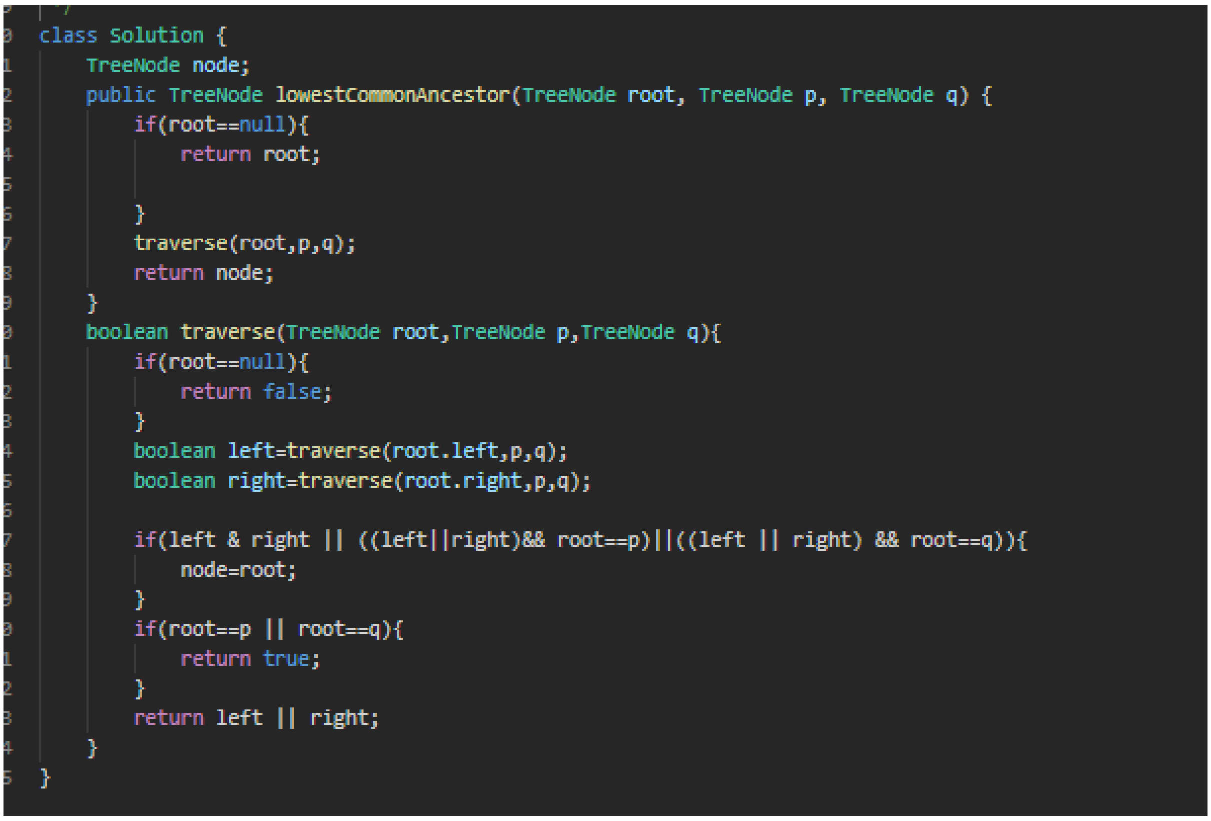


```
Input: root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 1
Output: 3
Explanation: The LCA of nodes 5 and 1 is 3.
```

**Example 2:**

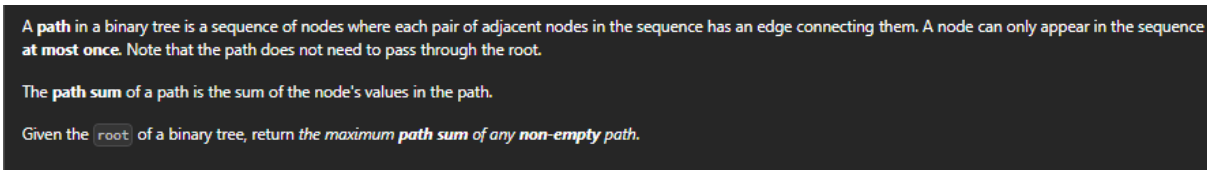


```
Input: root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 4
Output: 5
Explanation: The LCA of nodes 5 and 4 is 5, since a node can be a descendant of itself according to the LCA definition.
```



```
class Solution {
    TreeNode node;
    public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
        if(root==null){
            return root;

        }
        traverse(root,p,q);
        return node;
    }
    boolean traverse(TreeNode root,TreeNode p,TreeNode q){
        if(root==null){
            return false;
        }
        boolean left=traverse(root.left,p,q);
        boolean right=traverse(root.right,p,q);

        if(left & right || ((left||right)&& root==p)||((left || right) && root==q)){
            node=root;
        }
        if(root==p || root==q){
            return true;
        }
        return left || right;
    }
}
```

## 23.Binary Tree Maximum Path sum

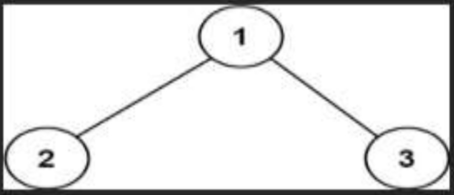


A **path** in a binary tree is a sequence of nodes where each pair of adjacent nodes in the sequence has an edge connecting them. A node can only appear in the sequence **at most once**. Note that the path does not need to pass through the root.

The **path sum** of a path is the sum of the node's values in the path.

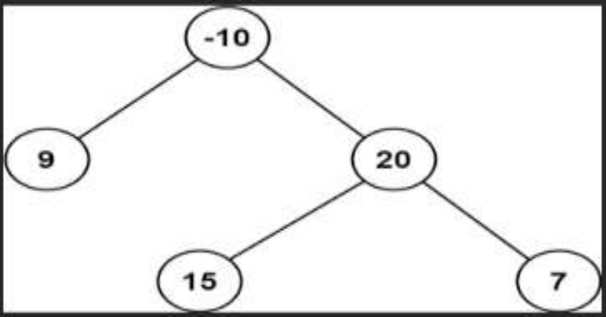
Given the `root` of a binary tree, return *the maximum* ***path sum*** *of any* ***non-empty*** *path*.

**Example 1:**

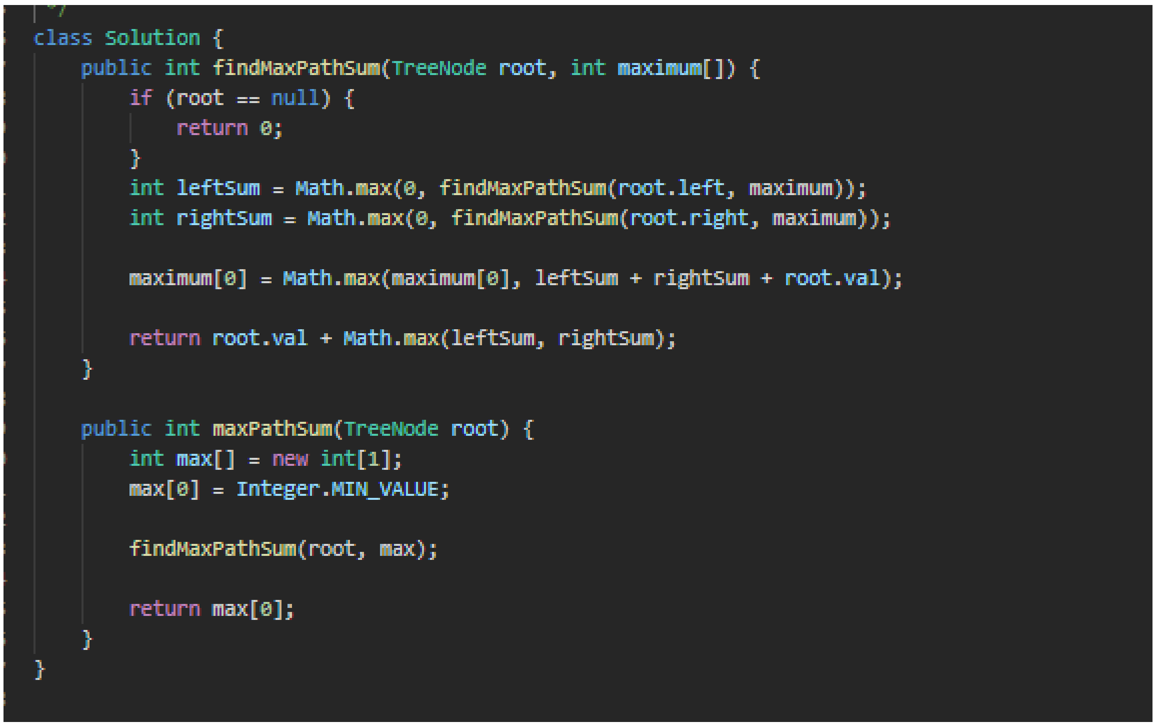


```
Input: root = [1,2,3]
Output: 6
Explanation: The optimal path is 2 -> 1 -> 3 with a path sum of 2 + 1 + 3 = 6.
```

**Example 2:**

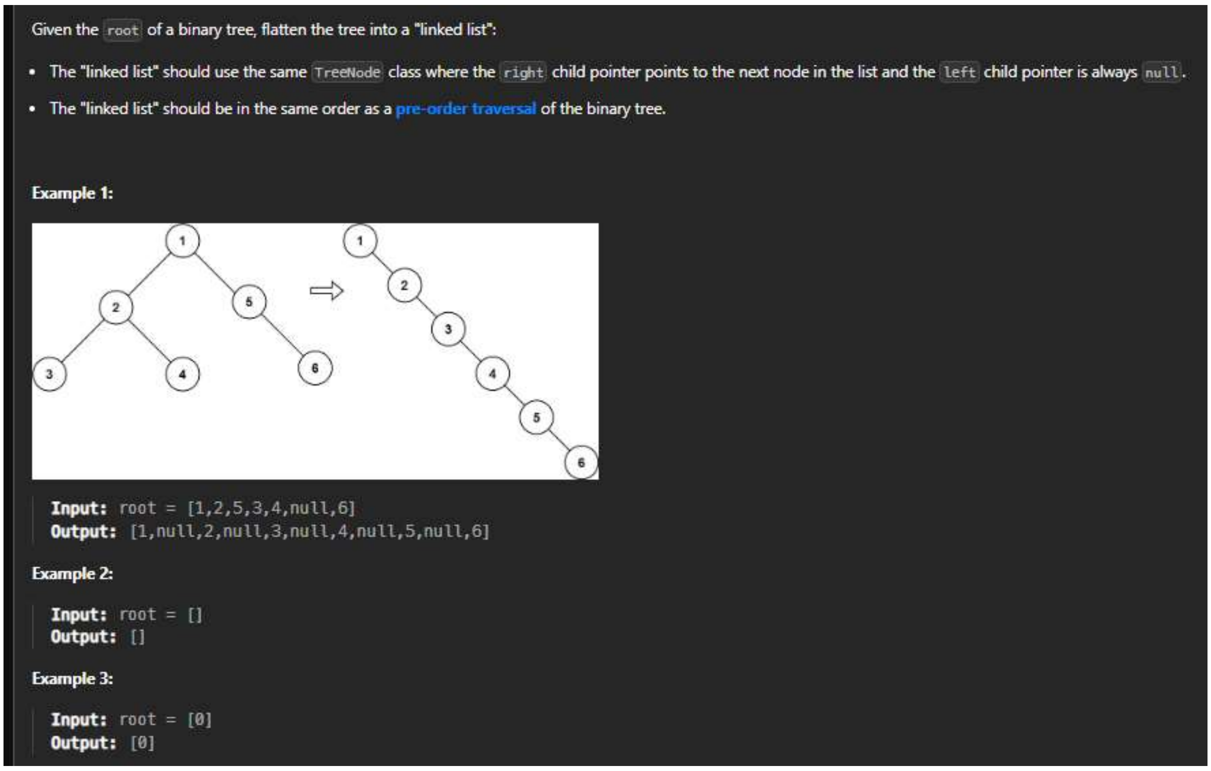


```
Input: root = [-10,9,20,null,null,15,7]
Output: 42
Explanation: The optimal path is 15 -> 20 -> 7 with a path sum of 15 + 20 + 7 = 42.
```



```
 */
class Solution {
    public int findMaxPathSum(TreeNode root, int maximum[]) {
        if (root == null) {
            return 0;
        }
        int leftSum = Math.max(0, findMaxPathSum(root.left, maximum));
        int rightSum = Math.max(0, findMaxPathSum(root.right, maximum));

        maximum[0] = Math.max(maximum[0], leftSum + rightSum + root.val);

        return root.val + Math.max(leftSum, rightSum);
    }

    public int maxPathSum(TreeNode root) {
        int max[] = new int[1];
        max[0] = Integer.MIN_VALUE;

        findMaxPathSum(root, max);

        return max[0];
    }
}
```

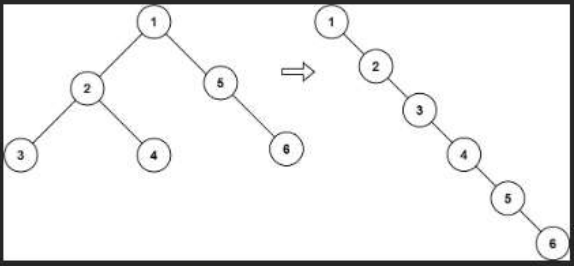
## 24.Flatten Binary Tree to Linked List



Given the `root` of a binary tree, flatten the tree into a "linked list":

- The "linked list" should use the same `TreeNode` class where the `right` child pointer points to the next node in the list and the `left` child pointer is always `null`.
- The "linked list" should be in the same order as a pre-order traversal of the binary tree.

**Example 1:**



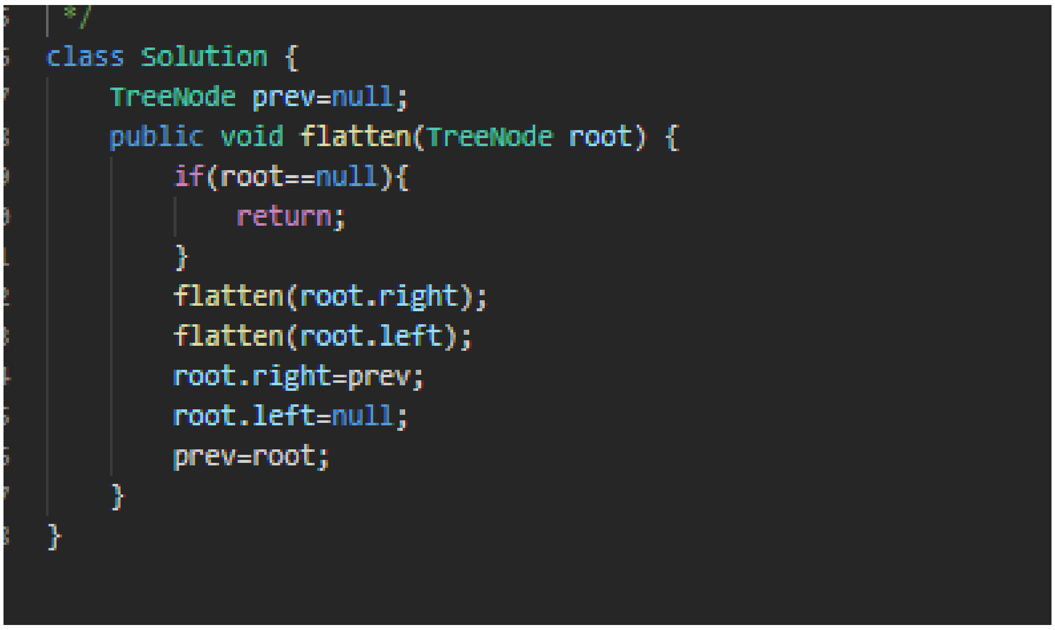
```
Input: root = [1,2,5,3,4,null,6]
Output: [1,null,2,null,3,null,4,null,5,null,6]
```

**Example 2:**

```
Input: root = []
Output: []
```

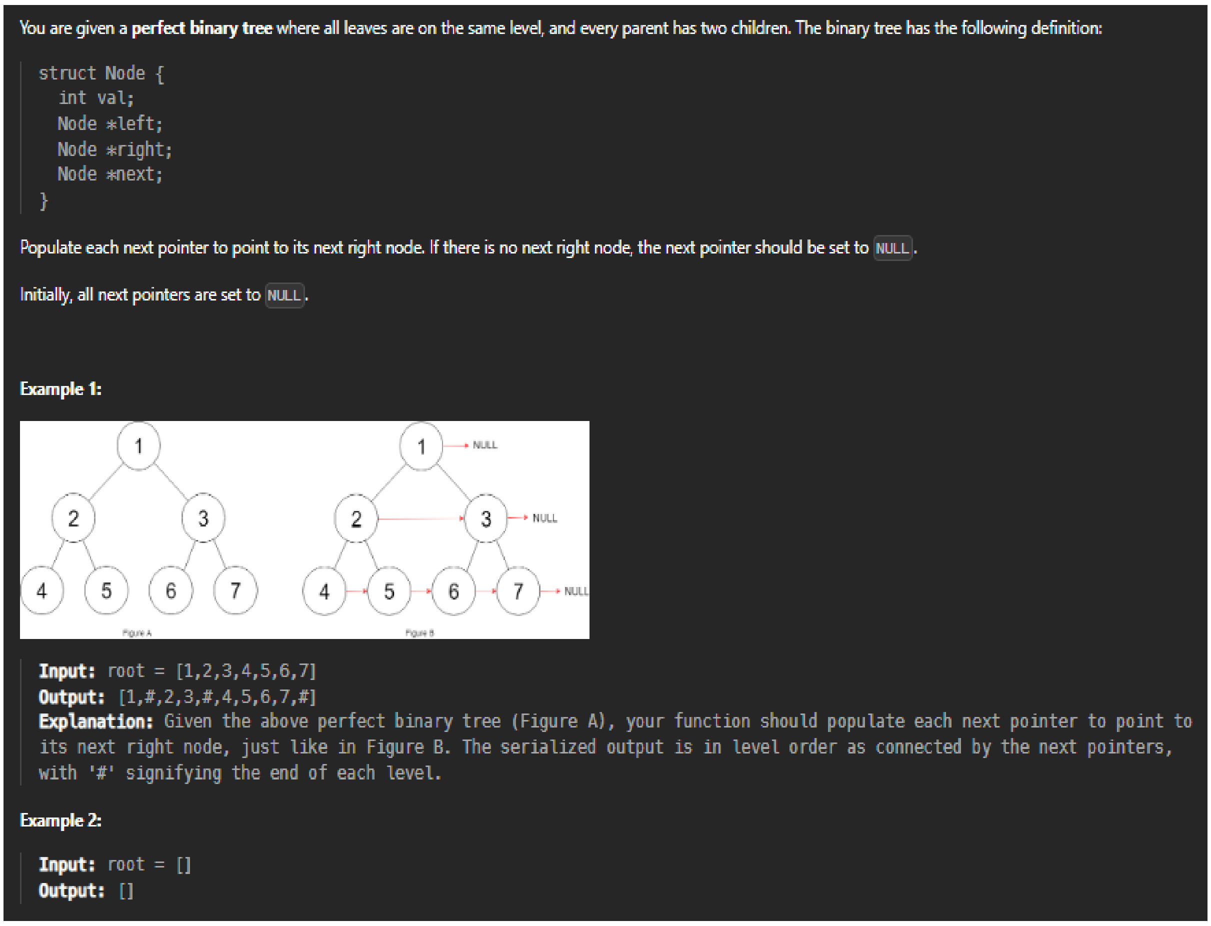
**Example 3:**

```
Input: root = [0]
Output: [0]
```



```
*/
class Solution {
    TreeNode prev=null;
    public void flatten(TreeNode root) {
        if(root==null){
            return;
        }
        flatten(root.right);
        flatten(root.left);
        root.right=prev;
        root.left=null;
        prev=root;
    }
}
```

## 25.Populating Next Right Pointer  in Each Node



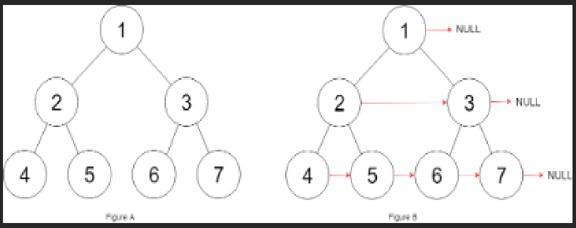
You are given a **perfect binary tree** where all leaves are on the same level, and every parent has two children. The binary tree has the following definition:

```
struct Node {
  int val;
  Node *left;
  Node *right;
  Node *next;
}
```

Populate each next pointer to point to its next right node. If there is no next right node, the next pointer should be set to `NULL`.

Initially, all next pointers are set to `NULL`.

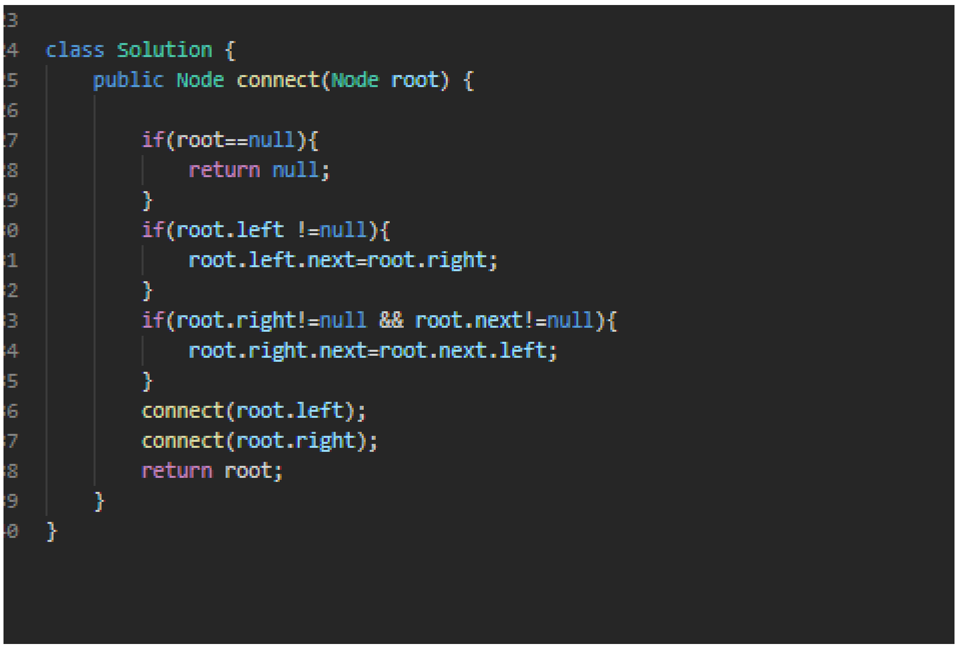
**Example 1:**



```
Input: root = [1,2,3,4,5,6,7]
Output: [1,#,2,3,#,4,5,6,7,#]
Explanation: Given the above perfect binary tree (Figure A), your function should populate each next pointer to point to
its next right node, just like in Figure B. The serialized output is in level order as connected by the next pointers,
with '#' signifying the end of each level.
```

**Example 2:**

```
Input: root = []
Output: []
```



```java

class Solution {
    public Node connect(Node root) {

        if(root==null){
            return null;
        }
        if(root.left !=null){
            root.left.next=root.right;
        }
        if(root.right!=null && root.next!=null){
            root.right.next=root.next.left;
        }
        connect(root.left);
        connect(root.right);
        return root;
    }
}
```

**https://leetcode.com/problems/serialize-and-deserialize-binary-tree/description/**