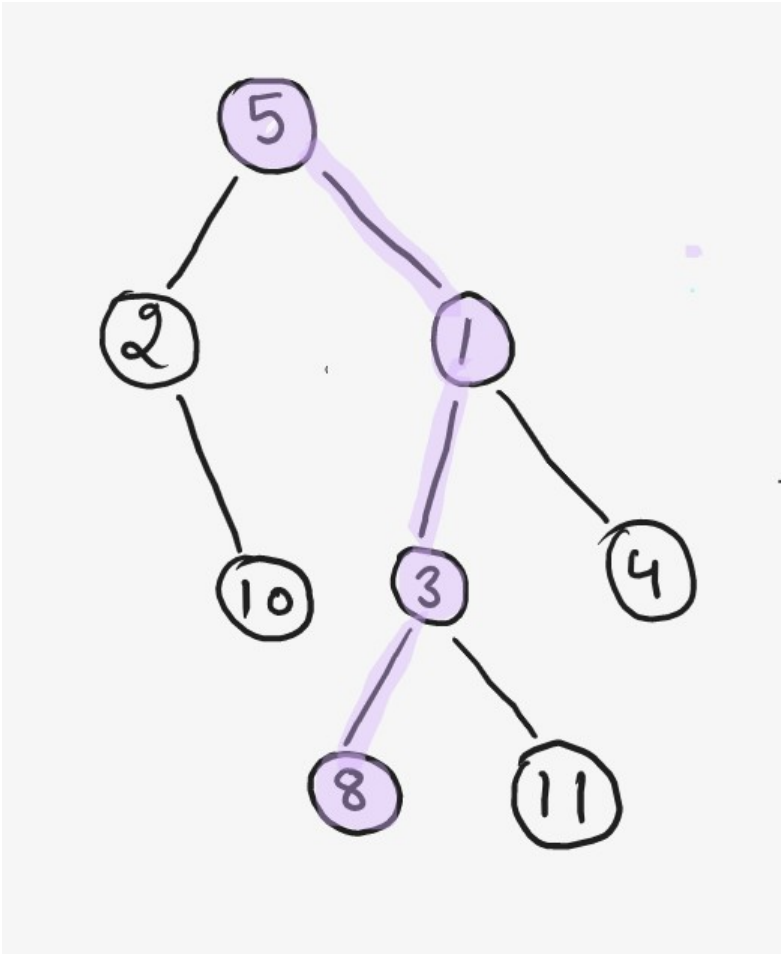
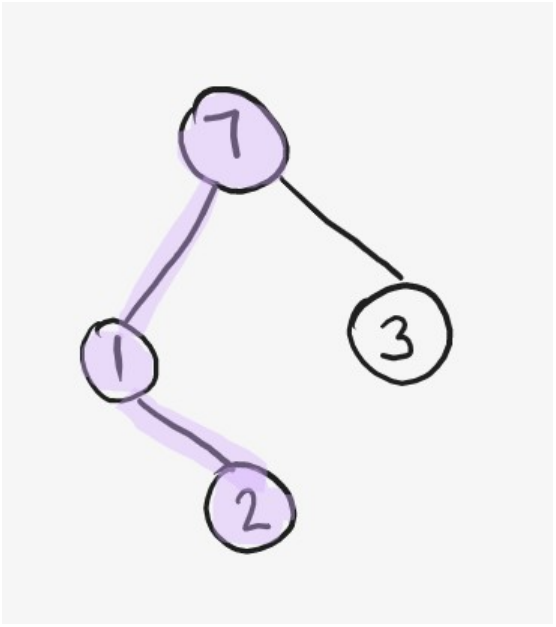
**Maximum depth of a Binary Tree**
**Problem Statement:** Find the **Maximum Depth** of Binary Tree. Maximum Depth is the **count of nodes of the longest path** from the root node to the leaf node.
**Examples**:
**Input Format**: Given the root of Binary Tree

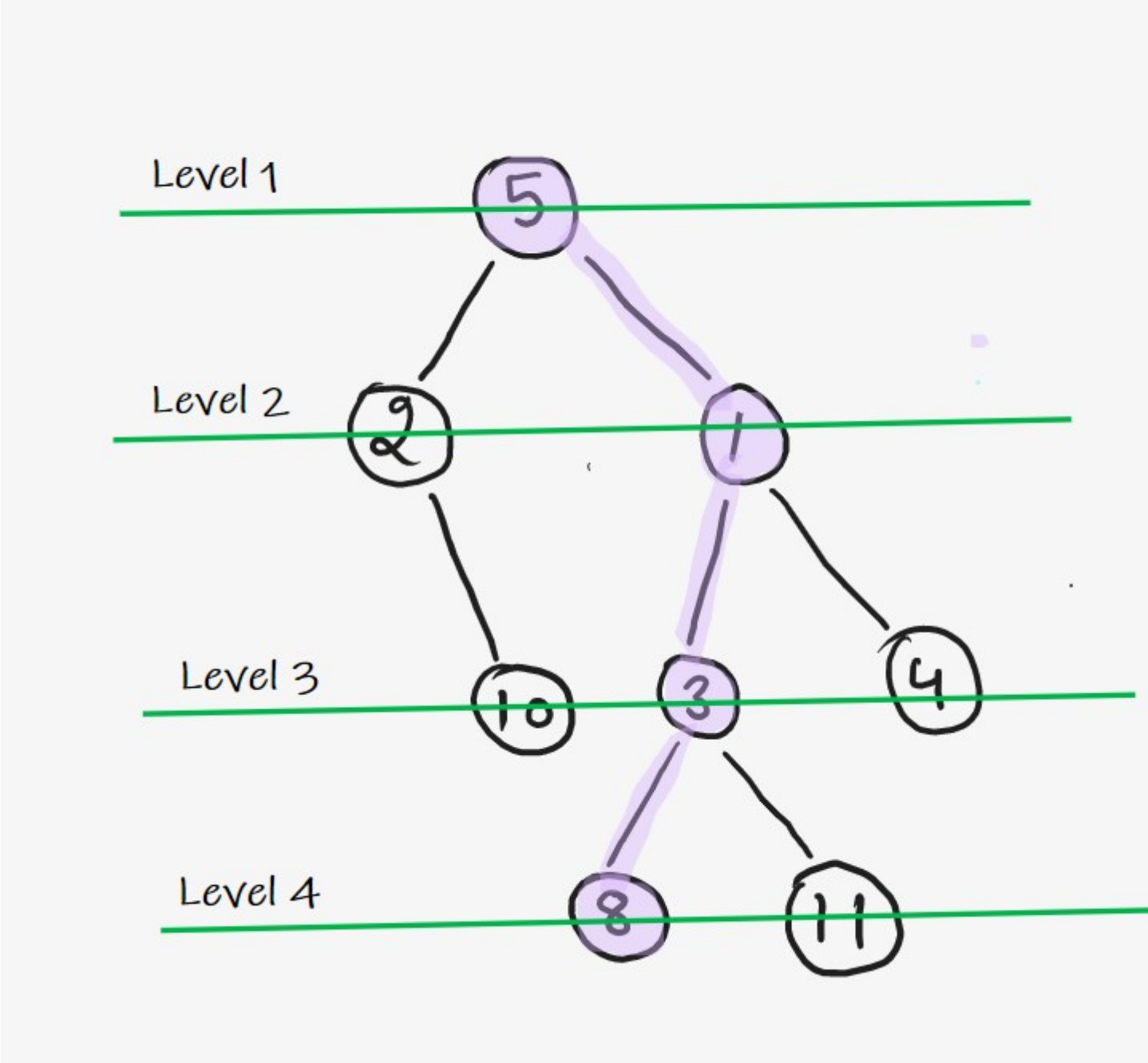



**Result**: 4
**Explanation**: Maximum Depth in this tree is 4 if we follow path 5 – 1 – 3 – 8 or 5 – 1 – 3 – 11
**Input Format:** Given the root of Binary Tree




**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**
***Disclaimer***: *Don't jump directly to the solution, try it out yourself first.*
**Solution 1:**
**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.

**Code**:

- Java Code

```
public class tUf {
    private static int levelOrder( TreeNode root ){
        if( root == null ){
            return 0;
        }

        LinkedList<TreeNode> queue = new LinkedList<>();
        queue.addLast(root);

        int level = 0;

        while( queue.size() > 0 ){
            int size = queue.size();

            while( size-- > 0 ){
                TreeNode remNode = queue.removeFirst();
                if( remNode.left != null ){
                    queue.addLast( remNode.left );
                }
                if( remNode.right != null ){
                        queue.addLast( remNode.right );
                }
            }

            level++;
        }

        return level;
    }
}
```

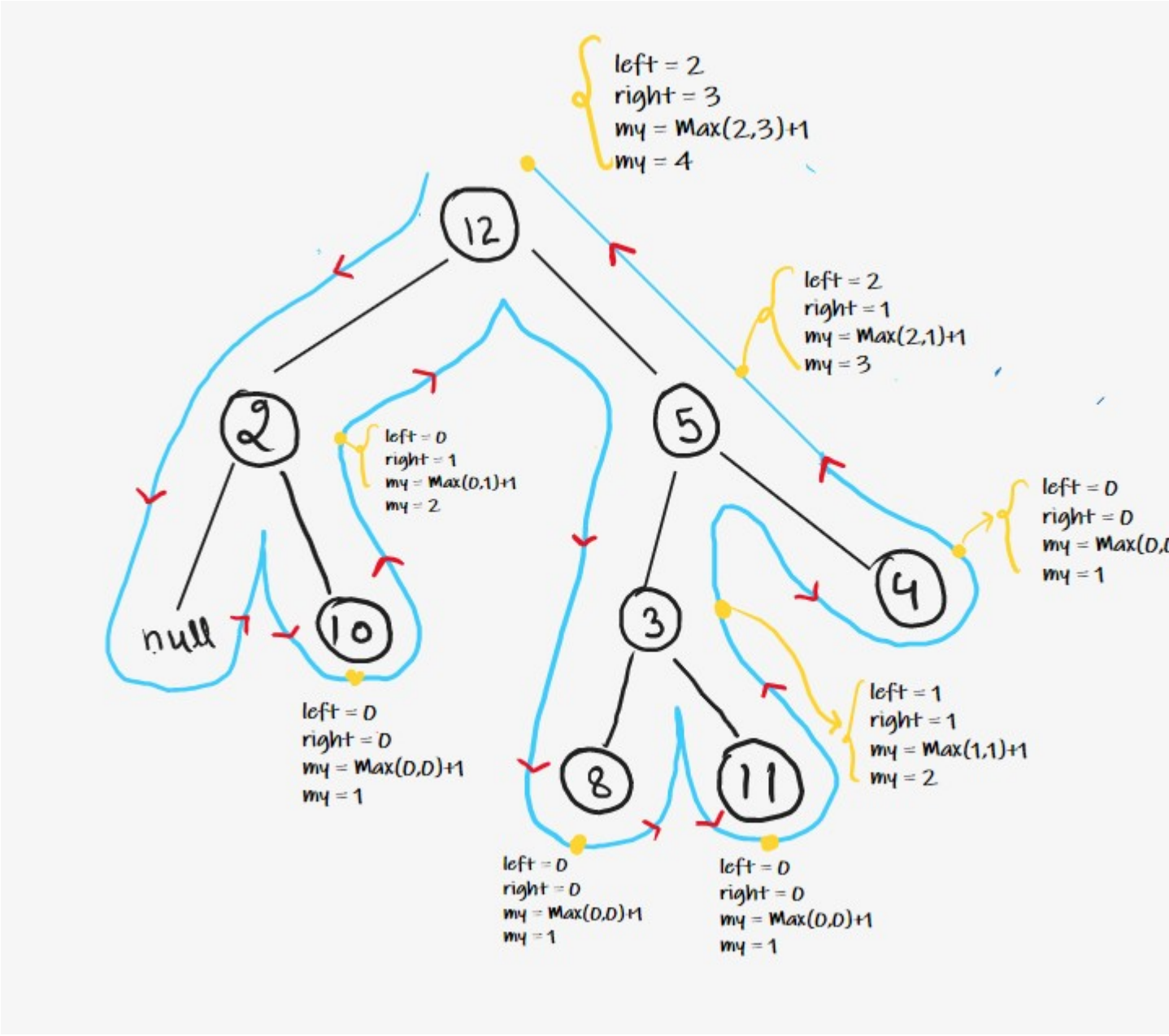
**Time Complexity:** O(N)
**Space Complexity:** O(N) ( Queue data structure is used )
**Solution 2:**
**Intuition: Recursively ( Post Order Traversal )**
If we have to do it recursively, then what we can think of is, If I have Maximum Depth of Left subtree and Maximum Depth of Right subtree then what 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**:

- C++ Code

- Java 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.