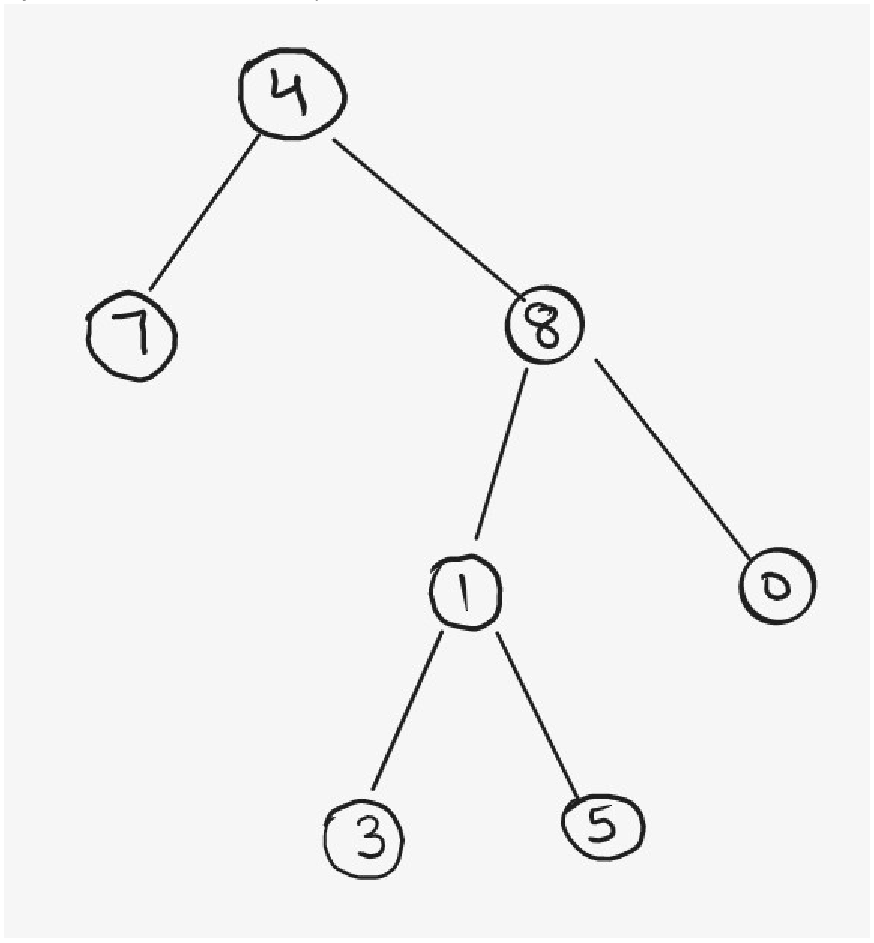
Check if the Binary Tree is Balanced Binary Tree

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:

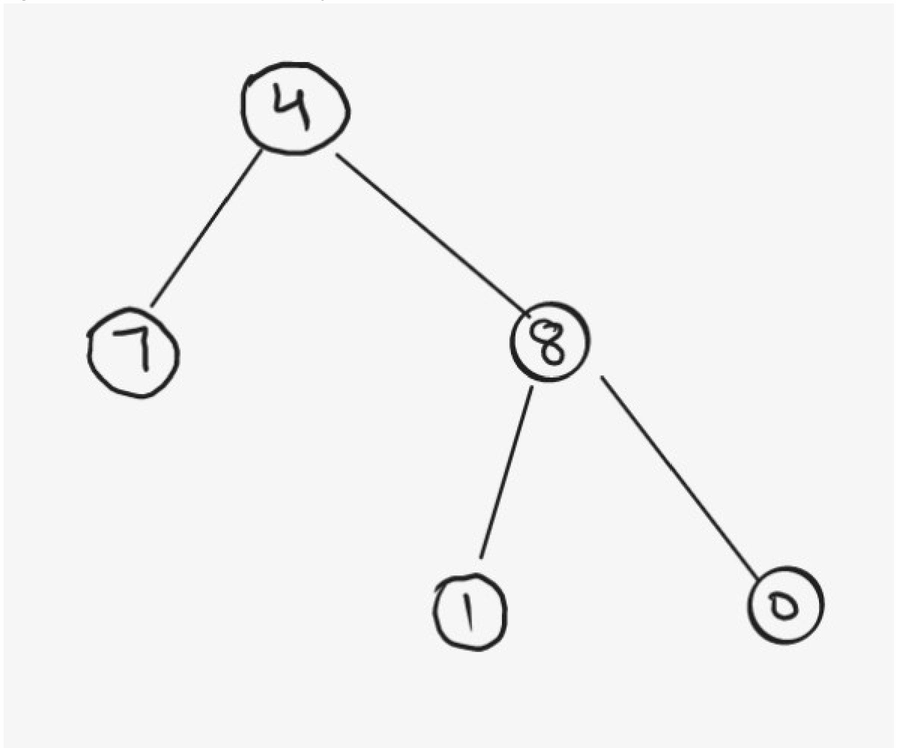
Input Format: Given the root of Binary Tree



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.

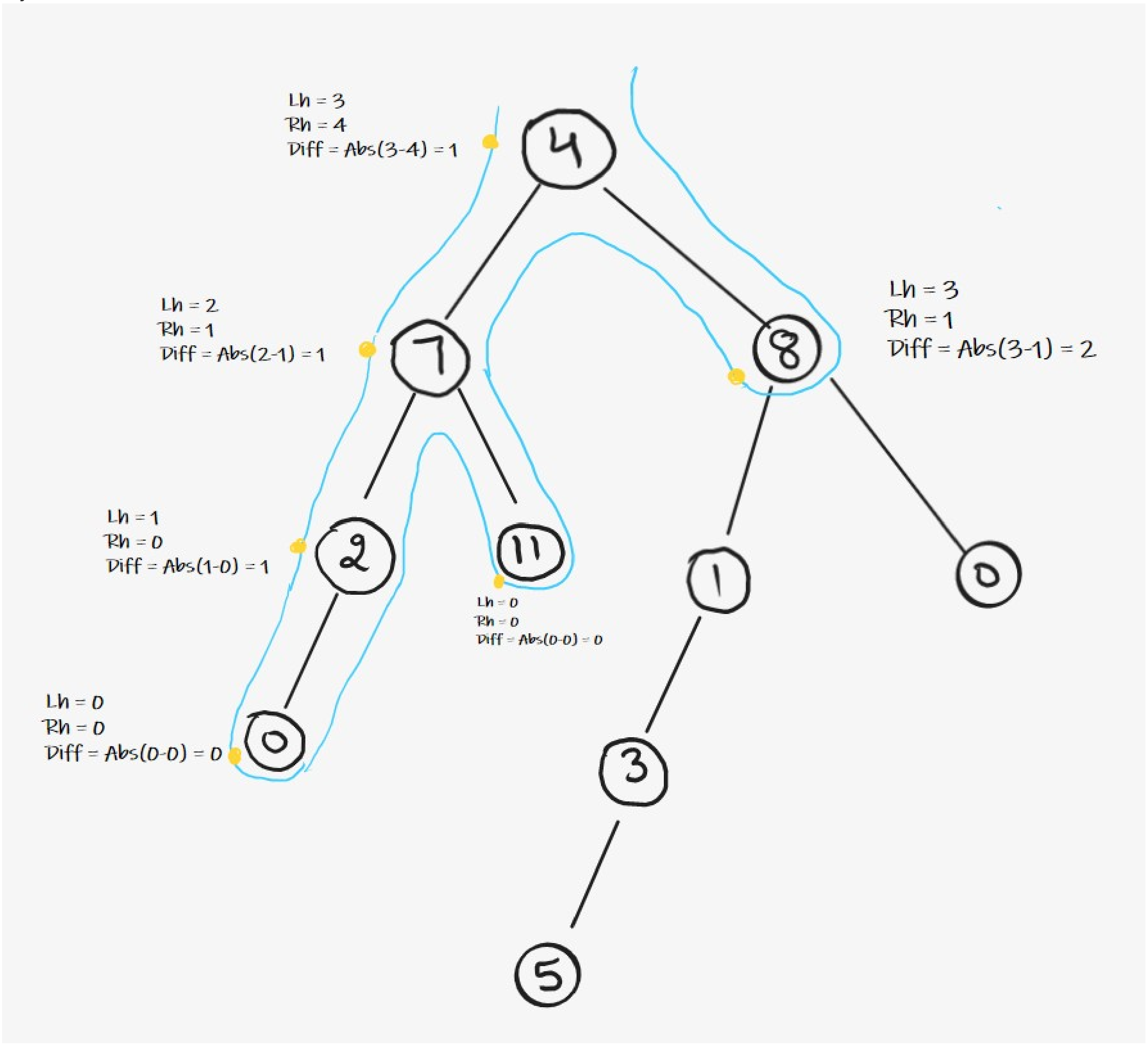
Input Format: Given the root of Binary Tree



Result: True

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

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

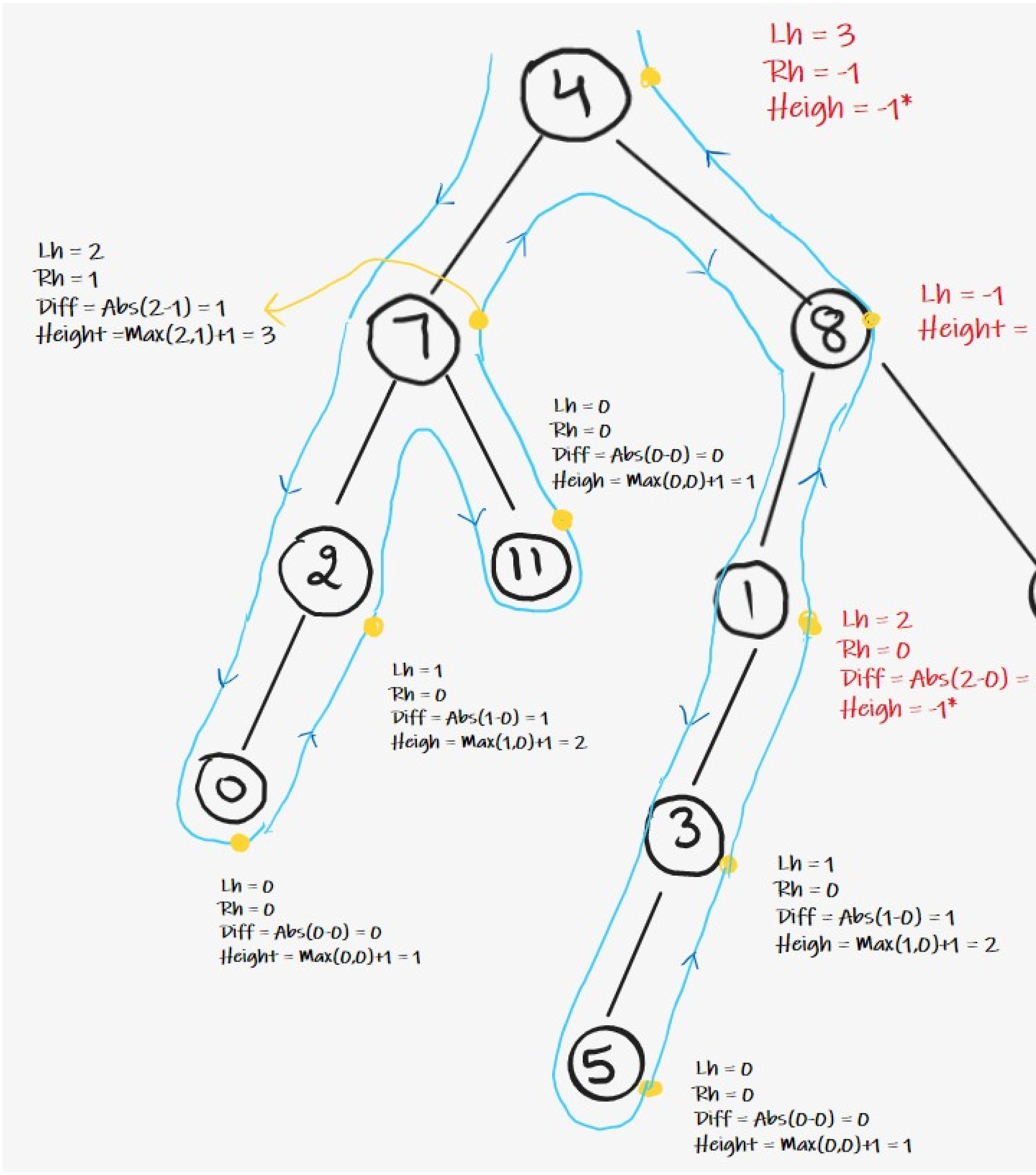
Ain’t we traversing the subtrees again and again in the above example?
Yes, so can we skip the repeated traversals?
What if we can make use of post-order traversal?
So, the idea is to use post-order traversal. Since, in postorder traversal, we first traverse the left and right subtrees and then visit 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:

- C++ Code
- Java Code

```
class Solution {

public:
    bool isBalanced(TreeNode* root) {
        return dfsHeight (root) != -1;
    }

    int dfsHeight (TreeNode *root) {

        if (root == NULL) return 0;

        int leftHeight = dfsHeight (root -> left);

        if (leftHeight == -1)
            return -1;

        int rightHeight = dfsHeight (root -> right);

        if (rightHeight == -1)
            return -1;

        if (abs(leftHeight - rightHeight) > 1)
            return -1;

        return max (leftHeight, rightHeight) + 1;
    }
};
```

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