Balanced Binary Tree

Given a binary tree, determine if it is height-balanced.

For this problem, a height-balanced binary tree is defined as a binary tree in which the depth of the two subtrees of *every* node never differ by more than 1.

Solution 1

This problem is generally believed to have two solutions: the top down approach and the bottom up way.

1.The first method checks whether the tree is balanced strictly according to the definition of balanced binary tree: the difference between the heights of the two sub trees are not bigger than 1, and both the left sub tree and right sub tree are also balanced. With the helper function depth(), we could easily write the code;

```
class solution {
public:
    int depth (TreeNode *root) {
        if (root == NULL) return 0;
        return max (depth(root -> left), depth (root -> right)) + 1;
    }

    bool isBalanced (TreeNode *root) {
        if (root == NULL) return true;

        int left=depth(root->left);
        int right=depth(root->right);

        return abs(left - right) <= 1 && isBalanced(root->left) && isBalanced(roo
t->right);
    }
};
```

For the current node root, calling depth() for its left and right children actually has to access all of its children, thus the complexity is O(N). We do this for each node in the tree, so the overall complexity of isBalanced will be O(N^2). This is a top down approach.

2.The second method is based on DFS. Instead of calling depth() explicitly for each child node, we return the height of the current node in DFS recursion. When the sub tree of the current node (inclusive) is balanced, the function dfsHeight() returns a non-negative value as the height. Otherwise -1 is returned. According to the leftHeight and rightHeight of the two children, the parent node could check if the sub tree is balanced, and decides its return value.

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

In this bottom up approach, each node in the tree only need to be accessed once. Thus the time complexity is O(N), better than the first solution.

written by benlong original link here

Solution 2

Input: {1,2,2,3,3,3,3,4,4,4,4,4,4,#,#,5,5}

Output: false (based on balanced binary definition **"no 2 leaf nodes differ in distance from the root by more than 1"**)

Expected: true (base on balanced binary definition **"two subtrees of every node never differ by more than 1"**)

written by Ethan original link here

Solution 3

```
public boolean isBalanced(TreeNode root) {
    if(root==null){
        return true;
    }
    return height(root)!=-1;

}
public int height(TreeNode node){
    if(node==null){
        return 0;
    }
    int lH=height(node.left);
    if(lH==-1){
        return -1;
    }
    int rH=height(node.right);
    if(rH==-1){
        return -1;
    }
    if(lH-rH<-1 || lH-rH>1){
        return -1;
    }
    return Math.max(lH,rH)+1;
}
```

written by mingyuan original link here