Description

110. 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.

```
Example 1:
Given the following tree [3,9,20,null,null,15,7]:

    3
   / \
  9  20
    /  \
   15   7
Return true.

Example 2:
Given the following tree [1,2,2,3,3,null,null,4,4]:

       1
      / \
     2   2
    / \
   3   3
  / \
 4   4
Return false.
```

Idea

From the definition of the height-balanced binary tree, we know the depth of the left and right child never differ by more than 1. And both left and right child has to be height-balenced too. So

for any node, we need to know the depth of node.left and depth of node.right, and we need to know a boolean which indicates whether node.left is balanced, or node.right is balanced. So we can construct a new class contains a integer of node's depth, and a boolean of node's isBalanced. For any node, if any of its children is not balanced, we return false directly. If both of its children is balanced, we see if thier height differs by more than 1, if it does, return false, otherwise it is a balanced tree, we return true.

Java

```java
class Solution {

    class ReturnType {
        int depth;
        boolean isBalanced;
        public ReturnType(int depth, boolean isBalanced) {
            this.depth = depth;
            this.isBalanced = isBalanced;
        }
    }

    public boolean isBalanced(TreeNode root) {
        return helper(root).isBalanced;
    }

    private ReturnType helper(TreeNode root) {
        ReturnType res = new ReturnType(0, true);
        if (root == null) {
            return res;
        }

        ReturnType left = helper(root.left);
        ReturnType right = helper(root.right);
        if (!left.isBalanced || !right.isBalanced
            || Math.abs(left.depth - right.depth) > 1) {
            res.isBalanced = false;
            return res;
        }

        res.depth = Math.max(left.depth, right.depth) + 1;
        return res;
    }
}
```

C++

```
class Solution {
public:
    bool isBalanced(TreeNode* root) {
        return helper(root).isBalanced;
    }

private:
    struct ReturnType {
        int depth;
        bool isBalanced;
        ReturnType(int depth, bool isBalanced):
            depth(depth), isBalanced(isBalanced){}
    };

    ReturnType helper(TreeNode* root) {
        ReturnType ret(0, true);
        if (root == NULL) {
            return ret;
        }

        ReturnType left = helper(root->left);
        ReturnType right = helper(root->right);
        if (!left.isBalanced || !right.isBalanced
            || abs(left.depth - right.depth) > 1) {
            ret.isBalanced = false;
            return ret;
        }

        ret.depth = max(left.depth, right.depth) + 1;
        return ret;
    }
};
```

We can also combine the depth and boolean into a single integer, where -1 means false, and >= 0 means the actual depth. But the code's readability is not good, so I do not prefer to use this.

Java

```
class Solution {
    public boolean isBalanced(TreeNode root) {
        return helper(root) >= 0;
    }

    private int helper(TreeNode root) {
        if (root == null) {
            return 0;
```

```
        }

        int left = helper(root.left);
        int right = helper(root.right);
        if (left < 0 || right < 0 || Math.abs(left - right) > 1) {
            return -1;
        }

        return Math.max(left, right) + 1;
    }
}
```

C++

```
class Solution {
public:
    bool isBalanced(TreeNode* root) {
        return helper(root) >= 0;
    }

private:
    int helper(TreeNode* root) {
        if (root == NULL) {
            return 0;
        }

        int left = helper(root->left);
        int right = helper(root->right);
        if (left < 0 || right < 0 || abs(left - right) > 1) {
            return -1;
        }

        return max(left, right) + 1;
    }
};
```

Summary

- Divid and conquer.
- Use ReturnType class if we need more than one information from bottom up.