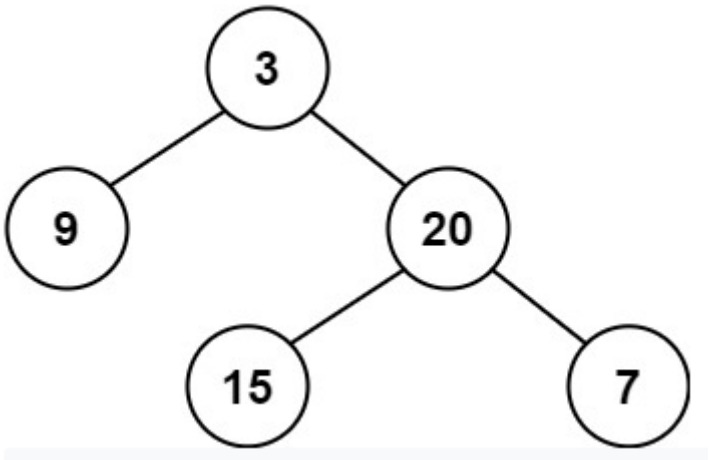
https://leetcode.com/problems/balanced-binary-tree/

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

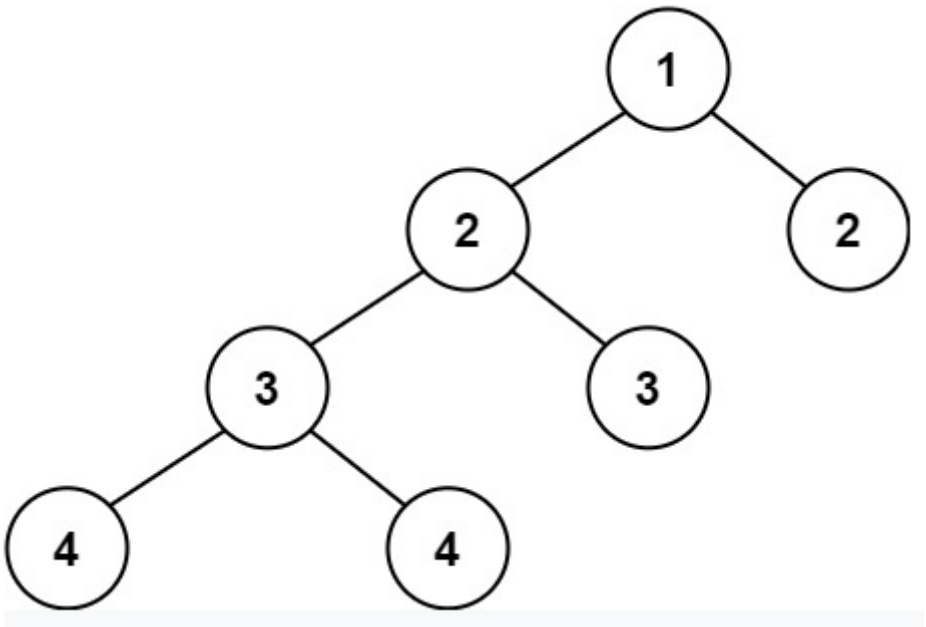
Example 1:



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

Output: true

Example 2:



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

Output: false

Example 3:

Input: root = []

Output: true

Constraints:

- The number of nodes in the tree is in the range [0, 5000].
- -10^4 <= Node.val <= 10^4

What is Balanced Binary Tree ?

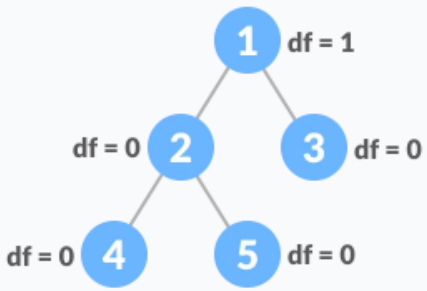
Refer to

https://www.programiz.com/dsa/balanced-binary-tree

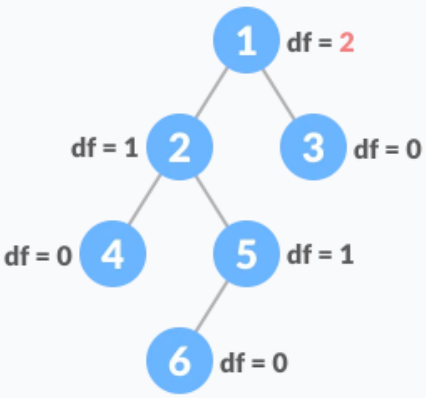
A balanced binary tree, also referred to as a height-balanced binary tree, is defined as a binary tree in which the height of the left and right subtree of **any node** differ by not more than 1.

To learn more about the height of a tree/node, visit Tree Data Structure. Following are the conditions for a height-balanced binary tree:

1. difference between the left and the right subtree for any node is not more than one
2. the left subtree is balanced
3. the right subtree is balanced



Balanced Binary Tree with depth at each level



df = |height of left child - height of right child|

Unbalanced Binary Tree with depth at each level

Attempt 1: 2022-11-08

Solution 1: Divide and Conquer (30min, O(n^2) as multiple passes required, not postorder traversal, only left + right child traversal as DFS)

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode() {}
 *     TreeNode(int val) { this.val = val; }
 *     TreeNode(int val, TreeNode left, TreeNode right) {
 *         this.val = val;
 *         this.left = left;
 *         this.right = right;
 *     }
 * }
 */
class Solution {
    public boolean isBalanced(TreeNode root) {
        if(root == null) {
            return true;
        }
        int left_height = getHeight(root.left);
        int right_height = getHeight(root.right);
        int root_diff = Math.abs(left_height - right_height);
        // Divide
        boolean left = isBalanced(root.left);
        boolean right = isBalanced(root.right);
        // Conquer
        return root_diff <= 1 && left && right;
    }

    private int getHeight(TreeNode root) {
        // Base caes: root being null means tree doesn't exist
        if(root == null) {
            return 0;
```

```
        }
        // Get the depth of the left and right subtree using recursion
        int leftHeight = getHeight(root.left);
        int rightHeight = getHeight(root.right);
        // Choose the larger one and add the root to it
        return Math.max(leftHeight, rightHeight) + 1;
    }
}

Time Complexity: O(n^2), where n is number of nodes in the Binary Tree
Space Complexity: O(n)
```

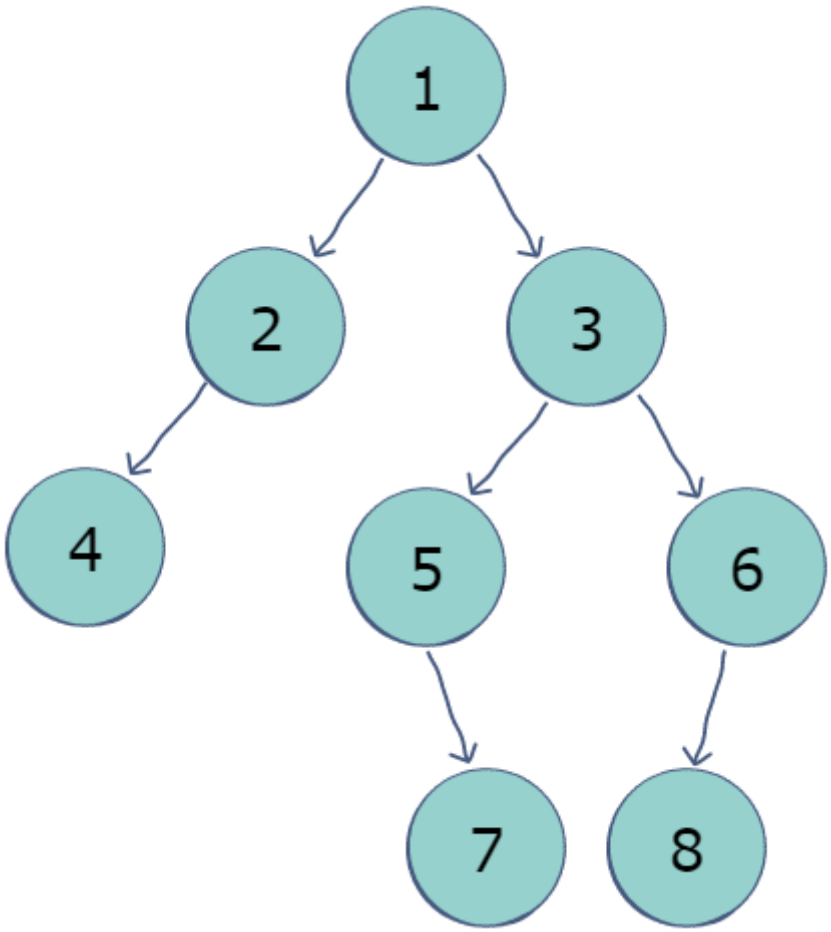
How to get height (maximum depth) of a binary tree ?

Refer to

https://www.educative.io/answers/finding-the-maximum-depth-of-a-binary-tree

The **maximum depth of a binary tree** is the number of nodes from the root down to the furthest leaf node. In other words, it is the height of a binary tree.

Consider the binary tree illustrated below:



The maximum depth, or height, of this tree is 4; node 7 and node 8 are both four nodes away from the *root*.

Algorithm

The algorithm uses recursion to calculate the maximum height:

1. Recursively calculate the height of the tree to the left of the root.
2. Recursively calculate the height of the tree to the right of the root.
3. Pick the larger height from the two answers and add one to it (to account for the root node).

```
class Node
{
  int value;
  Node left, right;

  Node(int val)
  {
    value = val;
    left = right = null;
  }
}
```

```java
class BinaryTree
{
  Node root;
  int maxDepth(Node root)
  {
    // Root being null means tree doesn't exist.
    if (root == null)
      return 0;
    // Get the depth of the left and right subtree
    // using recursion.
    int leftDepth = maxDepth(root.left);
    int rightDepth = maxDepth(root.right);
    // Choose the larger one and add the root to it.
    if (leftDepth > rightDepth)
      return (leftDepth + 1);
    else
      return (rightDepth + 1);
  }

  // Driver code
  public static void main(String[] args)
  {
    BinaryTree tree = new BinaryTree();
    tree.root = new Node(1);
    tree.root.left = new Node(2);
    tree.root.right = new Node(3);
    tree.root.left.left = new Node(4);
    tree.root.right.left = new Node(5);
    tree.root.right.right = new Node(6);
    tree.root.right.right.left = new Node(8);
    tree.root.right.left.right = new Node(7);
    System.out.println("Max depth: " + tree.maxDepth(tree.root));
  }
}
```

Solution 2: Divide and Conquer (30min, simple O(n) version, height of tree starts from 0, so -1 is free to use as a flag, not postorder traversal, only left + right child traversal as DFS)

```java
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode() {}
 *     TreeNode(int val) { this.val = val; }
 *     TreeNode(int val, TreeNode left, TreeNode right) {
 *         this.val = val;
 *         this.left = left;
 *         this.right = right;
 *     }
 * }
 */
class Solution {
    public boolean isBalanced(TreeNode root) {
        return helper(root) != -1;
    }

    private int helper(TreeNode root) {
        if(root == null) {
            return 0;
        }
        int left_height = helper(root.left);
        int right_height = helper(root.right);
        if(left_height == -1 || right_height == -1 || Math.abs(left_height - right_height) > 1) {
            return -1;
        }
        return 1 + Math.max(left_height, right_height);
    }
}

Time Complexity: O(n), where n is number of nodes in the Binary Tree
Space Complexity: O(n)
```

Refer to

https://leetcode.com/problems/balanced-binary-tree/discuss/35691/The-bottom-up-O(N)-solution-would-be-better

This problem is generally believed to have two solutions: the multiple passes approach and the one pass 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(root->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 multiple passes 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 one pass approach, each node in the tree only need to be accessed once. Thus the time complexity is O(N), better than the first solution.

simple O(n) version: (height of tree starts from 0, so -1 is free to use as a flag)

https://leetcode.com/problems/balanced-binary-tree/discuss/35691/The-bottom-up-O(N)-solution-would-be-better/198436

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

    private int helper(TreeNode root){
        if(root == null){
            return 0;
        }
        int left = helper(root.left);
        int right = helper(root.right);
```

```
        if(left == -1 || right == -1 || Math.abs(left - right) > 1){
            return -1;
        }
        return Math.max(left, right) + 1;
    }
```

How the Solution 2 use -1 return as flag comes up ?

https://leetcode.com/problems/balanced-binary-tree/discuss/254230/Thinking-process-of-bottom-up-solution

From the recursive perspective, we know that we need to know 2 things:

1. If left/right subtrees is balanced

2. The height of left/right subtreeThen I think how can I get both of them? The only way to do it is: **return both of them in the recursion function** i.e. {height, isBalanced}, rather than just return the height like the top-down solution. But the problem is that balanced is boolean data type and height is an int data type. We cannot declare an array with different data type in Java(However Python can do it :P). So I use intas the replacement of balanced boolean data type: -1 as false, 1 as true. Thus the result can be stored in the array int[] cur = new int[2].

```
class Solution {
    public boolean isBalanced(TreeNode root) {
            // corner case
        if(root == null) return true;

        int[] res = getHeight(root);
        return res[1] == 1;
    }

    // return [height, balanced]
    public int[] getHeight(TreeNode root){
        // base case
        if(root == null) return new int[]{0, 1};

        int[] cur = new int[2];

        int[] left = getHeight(root.left);
        if(left[1] == -1){
```

```
            cur[1] = -1; // unbalanced, do not care about height anymore
            return cur;
        }

        int[] right = getHeight(root.right);
        if(right[1] == -1){
            cur[1] = -1; // unbalanced, do not care about height anymore
            return cur;
        }

        if(Math.abs(left[0] - right[0]) > 1){
            cur[1] = -1; // unbalanced, do not care about height anymore
            return cur;
        }

        // set [height, balanced]
        cur[0] = Math.max(left[0], right[0]) + 1; // set height
        cur[1] = 1; // set balanced
        return cur;
    }

}
```

Optimized {height, isBalanced}

But notice that the height of a tree is **always** >= 0, and we do not care about the height when the subtree is already confirmed imbalanced. **So we can use -1 to represents imbalanced, then we can merge the int array of size 2 to just a int value to save some space.** That's the magic!!! (But to be honest, who cares such little space. O(2) == O(1), the value is that if you are familiar with this, you can directly use -1 which is easier to write)

Final code:

```
class Solution {
    public boolean isBalanced(TreeNode root) {
            // corner case
        if(root == null) return true;

        return getHeight(root) != -1;
    }
```

```java

    // return the height of tree rooted at `root`if balanced, otherwise -1
    public int getHeight(TreeNode root){
        // base case
        if(root == null) return 0;

        int left = getHeight(root.left);
        if(left == -1) return -1;
        int right = getHeight(root.right);
        if(right == -1) return -1;

        if(Math.abs(left - right) > 1) return -1;
        return Math.max(left, right) + 1;
    }
}
```

Refer to

L104.Maximum Depth of Binary Tree (Ref.L222)

L222.Count Complete Tree Nodes (Ref.L104,L1448,L333)