110. Balanced Binary Tree

Easy · Tree · Depth-First Search · Binary Tree

Problem Description

The problem presents a scenario where one is given a binary tree and is tasked with determining if the tree is *height-balanced*. In the context of this problem, a height-balanced binary tree is defined as a binary tree where the heights of the two child subtrees of any node differ by no more than one. Additionally, the subtrees of any node must also be height-balanced.

Intuition

The solution to this problem involves a recursive approach, where we traverse the tree to check the height of both left and right subtrees for each node and verify the balance condition (that the height difference is no more than one).

The intuition behind the solution is to perform a post-order traversal of the tree. A post-order traversal means we check the subtrees before dealing with the current node, which naturally allows us to check whether the subtrees are balanced and also to calculate their heights.

If at any point we find that the subtree is not balanced (i.e., the height difference between the left and right subtree is greater than 1), we can immediately conclude that the tree is not height-balanced.

The function `height` calculates the height of a tree rooted at a given node. If the left or right subtree is unbalanced (represented by a height of -1), or if the current node's subtrees' heights differ by more than 1, it returns -1, indicating that the tree is not balanced starting from this node. If the subtrees are balanced, the function returns the actual height, which is 1 plus the maximum of the heights of the left and right subtrees.

The `isBalanced` function calls the `height` function with the root node and checks if the returned value is non-negative. A non-negative return value indicates that the tree is balanced, whereas a value of -1 indicates that it is not.

By using this approach, we only need to traverse each node once, giving us an efficient algorithm with a time complexity of O(n) where n is the number of nodes in the tree.

Solution Approach

The solution implements a recursive function to determine the height of a given subtree and whether the subtree is balanced.

Within this approach, a few key concepts and algorithms are used:

- **Post-order Traversal:** The function `height` uses post-order traversal of the tree. At each node, it first checks the height of its left and right subtrees before performing any action on the current node. Only after we have the heights of the subtrees do we check if they satisfy the balance criterion.
- **Recursion:** Recursion is a fundamental part of the solution. The `height` function calls itself to find the height of the left subtree and the right subtree.
- **Termination Checks:** The function includes condition checks to terminate early if an imbalance is found. When the left or right subtree has an imbalance (height of `-1`), or their height difference is more than `1`, the function returns `-1` immediately, avoiding further unnecessary checks or recursive calls.
- **Height Calculation:** To calculate a subtree's height, the function considers the larger height of its left or right subtree and adds `1` to account for the current node.

Let's break down the code step-by-step:

- We define the `height` function that takes a `TreeNode` as an argument. This function will return an integer representing the height of the tree rooted at the given node or `-1` if the subtree is unbalanced:

```python
def height(root):
    if root is None:
        return 0
    l, r = height(root.left), height(root.right)
    if l == -1 or r == -1 or abs(l - r) > 1:
        return -1
    return 1 + max(l, r)
```

- If the current node `root` is `None`, which means we've reached the end of a branch, it returns `0` as the height.
- It uses the same `height` function to get the height of the left and right subtrees.
- The function checks if either `l` or `r` is `-1`, indicating that the respective subtree is unbalanced. It also checks if the absolute difference in heights `abs(l - r)` is greater than `1`. If any of these conditions are true, the current subtree cannot be height-balanced and the function returns `-1`.
- If all is well, the function computes the height of the current tree by adding `1` to the maximum of the heights of the left and the right subtrees.
- The main function `isBalanced` calls the `height` function on the `root` and checks if the result is greater than or equal to `0`. A result of `-1` would mean the tree is not height-balanced:

```python
class Solution:
    def isBalanced(self, root: Optional[TreeNode]) -> bool:
        return height(root) >= 0
```

This recursive solution is elegant and efficient because it computes both the balance status and height simultaneously, avoiding repeated calculations that would occur if we treated each concern separately.

Example Walkthrough

Let's use a small example to illustrate the solution approach. Consider the following binary tree:

```
    3
   / \
  9  20
    /  \
   15   7
```

In this example, we have a binary tree where the root node has a value of 3, the left child is a leaf node with a value of 9, and the right child has a value of 20. The right child has its own children (15 on the left and 7 on the right), both leaf nodes.

Now, let's walk through the solution algorithm step-by-step:

1. The `isBalanced` function will begin by calling the `height` function on the root node (with value 3).
2. The `height` function checks if the root node is `None`. Since it's not, it proceeds to find the height of the left and right subtrees.
3. The `height` function is recursively called on the left child (value 9) which is a leaf node. As it does not have any children, the function returns `1` because the height of a leaf node is `1`.
4. Similarly, the `height` function is recursively called on the right child (value 20).
5. For node 20, the function calls itself on node 15 (left child) and node 7 (right child), both of which are leaf nodes and so return `1`.
6. Now, with `l` (the height of the left subtree of node 20) and `r` (the height of the right subtree of node 20) both equal to `1`, the function proceeds to check if the subtree rooted at node 20 is balanced. Since `abs(l - r) = 0` which does not exceed `1`, it is balanced and the function returns `1` + max(`l`, `r`) = `2`.
7. Back at the root node (value 3), we now have `l` (the height of the left subtree) as `1` and `r` (the height of the right subtree) as `2`. The function checks if the subtree rooted at node 3 is balanced. `abs(1 - 2)` equals `1`, which is within the allowed difference, so it is balanced, and the function returns `1` + max(`1`, `2`) = `3`.
8. Since there was no step where the `height` function returned `-1`, indicating an imbalance, the overall result of the `height` function when called on the root is the height of the tree `3`, which is a non-negative number.
9. The `isBalanced` function finally checks the returned value from the `height` function. Since the value is not `-1`, `isBalanced` returns `True`, indicating that the binary tree is indeed height-balanced.

Thus, using the given solution approach, the binary tree in our example has been determined to be height-balanced efficiently in `O(n)` time complexity, where `n` is the number of nodes in the tree.

Python Solution

```python
# Definition for a binary tree node.
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Solution:
    def isBalanced(self, root: Optional[TreeNode]) -> bool:
        # Helper function to calculate the height of a binary tree rooted at `node`.
        def calculate_height(node):
            # Base case: if the node is None, the height is 0.
            if node is None:
                return 0

            # Recursively find the height of the left and right subtrees.
            left_height = calculate_height(node.left)
            right_height = calculate_height(node.right)

            # If either subtree is unbalanced (indicated by a height of -1),
            # or the difference in heights is greater than 1, the tree is unbalanced.
            if left_height == -1 or right_height == -1 or abs(left_height - right_height) > 1:
                return -1

            # If the tree is balanced, return the height of the tree.
            # Height of a node is 1 plus the maximum of the heights of its subtrees.
            return 1 + max(left_height, right_height)

        # The tree is balanced if calculate_height does not return -1.
        return calculate_height(root) >= 0

```

Java Solution

```java
/**
 * Definition for a binary tree node.
 */
class TreeNode {
    int val;        // Variable for the value of the node.
    TreeNode left;  // Pointer to the left child.
    TreeNode right; // Pointer to the right child.

    // Default constructor.
    TreeNode() {}

    // Constructor with the node's value.
    TreeNode(int val) { this.val = val; }

    // Constructor with the node's value, left, and right children.
    TreeNode(int val, TreeNode left, TreeNode right) {
        this.val = val;
        this.left = left;
        this.right = right;
    }
}

class Solution {
    /**
     * Determines if a binary tree is balanced.
     * In a balanced tree, the height of the two subtrees of any node never differ by more than one.
     *
     * @param root The root of the binary tree.
     * @return true if the tree is balanced, false otherwise.
     */
    public boolean isBalanced(TreeNode root) {
        // A non-negative height indicates that the tree is balanced,
        // while -1 represents an imbalance.
        return calculateHeight(root) >= 0;
    }

    /**
     * Recursively calculates the height of a binary tree.
     * Returns -1 if the subtree is unbalanced.
     *
     * @param node The node to calculate height of.
     * @return The height of the tree if balanced, otherwise -1.
     */
    private int calculateHeight(TreeNode node) {
        // Tree with no nodes has height 0.
        if (node == null) {
            return 0;
        }

        // Recursively find the height of the left and right subtrees.
        int leftHeight = calculateHeight(node.left);
        int rightHeight = calculateHeight(node.right);

        // Check if left or right subtree is unbalanced or if they differ in height by more than 1.
        if (leftHeight == -1 || rightHeight == -1 || Math.abs(leftHeight - rightHeight) > 1) {
            return -1; // Tree is not balanced.
        }

        // Current node height is max of left and right subtree heights plus 1 (for the current node).
        return 1 + Math.max(leftHeight, rightHeight);
    }
}

```

C++ Solution

```cpp
/**
 * Definition for a binary tree node.
 */
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode() : val(0), left(nullptr), right(nullptr) {}
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
    TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
};

class Solution {
public:
    /**
     * Checks if a binary tree is height-balanced.
     * A binary tree is balanced if the depth of the two subtrees of every node
     * never differ by more than 1.
     *
     * @param root A pointer to the root node of the binary tree.
     * @return A boolean indicating whether the tree is balanced.
     */
    bool isBalanced(TreeNode* root) {
        // Lambda function to recursively find the height of the tree.
        // It returns -1 if the tree is unbalanced at any point.
        std::function<int(TreeNode*)> findHeight = [&](TreeNode* node) -> int {
            if (!node) {
                // Tree is empty or this is a leaf node, hence the height is 0.
                return 0;
            }
            int leftHeight = findHeight(node->left);
            int rightHeight = findHeight(node->right);

            if (leftHeight == -1 || rightHeight == -1 || abs(leftHeight - rightHeight) > 1) {
                // If the left or right subtree is unbalanced, or if the current subtree
                // is unbalanced (difference in height > 1), return -1 as an indicator.
                return -1;
            }
            // Otherwise, calculate the height of this subtree,
            // which is 1 + the maximum height of its left or right subtree.
            return 1 + std::max(leftHeight, rightHeight);
        };

        // If the height function returns -1, the tree is unbalanced.
        // Otherwise, it is balanced.
        return findHeight(root) >= 0;
    }
};

```

Typescript Solution

```typescript
// Function to determine if a binary tree is height-balanced
// A binary tree in which the left and right subtrees of every node differ in height by no more than 1.
function isBalanced(root: TreeNode | null): boolean {

  // Helper function to compute the depth of the tree and check for balance
  // Returns the height of a node or -1 if the subtree is not balanced
  function depthFirstSearch(node: TreeNode | null): number {
    // An empty node has a depth of 0
    if (node == null) {
      return 0;
    }

    // Recursively obtain the depth of the left subtree
    const leftDepth = depthFirstSearch(node.left);
    // Recursively obtain the depth of the right subtree
    const rightDepth = depthFirstSearch(node.right);

    // If the left or right subtree is not balanced, or the difference
    // between their depths is more than 1, the current subtree is not balanced
    if (leftDepth === -1 || rightDepth === -1 || Math.abs(leftDepth - rightDepth) > 1) {
      return -1;
    }

    // Depth of current node is 1 plus the maximum depth of its left or right subtrees
    return 1 + Math.max(leftDepth, rightDepth);
  };

  // The tree is balanced if the depth first search does not return -1
  return depthFirstSearch(root) > -1;
}

// Definition for a binary tree node (given in the problem statement)
interface TreeNode {
  val: number;
  left: TreeNode | null;
  right: TreeNode | null;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the given recursive function is `O(N)`, where `N` is the number of nodes in the Tree. Each node in the tree is visited once, and the height is calculated. The function returns `-1` immediately if a subtree is found to be imbalanced, which eliminates the need for further computation on that subtree. The check for balance (`abs(l - r) > 1`) is `O(1)` for each node.

Space Complexity

The space complexity is `O(H)`, where `H` is the height of the tree. This space is used by the call stack during the recursive calls. In the worst case scenario (a degenerate tree), the space complexity can become `O(N)`, where each level of the tree has only one node, so the call stack grows to the height of the tree, which is `N`. In the best case scenario (a completely balanced tree), the space complexity would be `O(log(N))`, as the height of the tree would be `log(N)`, representing the maximum number of recursive calls on the stack at any one time.