965. Univalued Binary Tree

Easy Tree Depth-First Search Breadth-First Search Binary Tree

Problem Description

The goal is to determine whether all the nodes in a binary tree have the same value. In other words, we are given the root node of a binary tree and need to check if every node in the tree has the same value as the root, which would make it a *uni-valued* binary tree. The function should return `true` if the tree is uni-valued, otherwise, it should return `false`. A binary tree is a data structure in which each node has at most two children, referred to as the left child and the right child.

Intuition

To check if a binary tree is uni-valued, one effective approach is to perform a Depth-First Search (DFS) starting from the root of the tree. During the DFS, we compare the value of each node we visit to the value of the root node. If at any point we find a node that has a different value than the root, we can conclude that the tree is not uni-valued and immediately return `false`. If the search completes without finding any differing value, then the tree is uni-valued and we return `true`.

In the provided solution, a recursive `dfs` function is defined which carries out the above logic. It checks if the current node (`node`) is `None`, signifying the end of a branch, and returns `true` in that case, since a non-existent node does not contradict the uni-valued property. If the node exists, the function checks whether its value matches the root's value and also recursively calls itself on the node's left and right children. The main function `isUnivalTree` initiates the DFS by calling `dfs` function on the root. If all nodes are consistent with the root's value, the `dfs` function will return `true`; otherwise, it will return `false` at some point during the recursion.

Solution Approach

The implementation of the solution involves a depth-first search (DFS) algorithm applied to a binary tree data structure. The DFS algorithm is a recursive strategy used to traverse all the nodes in a tree starting from some node (usually the root) and explores as far down the branches as possible before backtracking.

The core function `dfs` in the solution uses recursion to travel through the tree. At each node, it performs the following steps:

1. It checks if the current node is `None`. If yes, it returns `true`, indicating the end of this branch is reached without finding any contradicting node.
2. If the current node is not `None`, it compares the value of the current node (`node.val`) with the value of the root node (`root.val`):
   - If `node.val` is equal to `root.val`, the node matches the required condition for a uni-valued tree. The function then proceeds with the recursion and calls `dfs(node.left)` and `dfs(node.right)`. Both these calls must return `true` to confirm that both the left and right subtrees are also uni-valued.
   - If `node.val` is not equal to `root.val`, the function immediately returns `false`, indicating the tree is not uni-valued.
3. The result of the `dfs` function relies on the logical AND `&&` operator between the comparison check and the recursive calls. This ensures that every node must satisfy the condition to return `true` for the overall result to be `true`.

When `isUnivalTree` is called with the root of the tree, it starts the `dfs` algorithm by calling the `dfs` function for the first time. If the entire tree is explored without finding any differing values, the function will ultimately return `true`. Otherwise, the discovery of any node that does not match the root's value will result in an early termination of the `dfs` function with a `false` result.

Example Walkthrough

Let's consider a small binary tree as an example to illustrate the solution approach:

```
   1
  / \
 1   1
/ \
1   1
```

In this binary tree, we have the root node with a value of 1, and all child nodes also have the value 1. We will apply the `dfs` algorithm to determine if this is a uni-valued binary tree.

We begin by calling the `isUnivalTree` function with the root node:

1. The `dfs` function is called with the root node (value 1). Since the node is not `None` and its value matches the root's value, the function proceeds to check its children.
2. The `dfs` function is then recursively called on the left child of the root (value 1). Again, this node is not `None`, and its value matches the root's value. So we move on to its children:
   - The recursive call to its left child (value 1) returns `true` as there are no more children to check and it matches the root's value.
   - The recursive call to its right child (value 1) also returns `true` for the same reason.
3. Since both recursive calls for the left child of the root returned `true`, the `dfs` function progresses to the right child of the root.
4. The right child of the root (value 1) is not `None` and its value does match the root's value. Since it has no children, the recursive calls on its left and right will both return `true`.
5. At this point, all nodes have been checked, and they all have the same value as the root node. Therefore, every `dfs` call has returned `true`, making the binary tree uni-valued.

Finally, `isUnivalTree` receives a `true` result from the `dfs` function, indicating that the tree is indeed uni-valued. We can conclude that given the structure and values, the example binary tree is uni-valued and our function will correctly return `true`.

Solution Implementation

Python

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

class Solution:
    def isUnivalTree(self, root: TreeNode) -> bool:
        """
        Determines if a tree is a univalued tree (where all nodes have the same value)

        Args:
        root: The root of the binary tree.

        Returns:
        A boolean value indicating if the tree is a univalued tree.
        """

        # Helper function to perform depth-first search
        def depth_first_search(node):
            # If we reach a None, it means we are at a leaf node, or the tree is empty
            # Since a None is technically univalued, we return True here
            if node is None:
                return True

            # Check if the current node's value is same as the root's value
            # And recursively check the left and right subtrees
            return (node.val == root.val
                    and depth_first_search(node.left)
                    and depth_first_search(node.right))

        # Start the depth-first search from the root
        return depth_first_search(root)
```

Java

```java
/**
 * Definition for a binary tree node.
 */
public class TreeNode {
    int val; // Value of the node
    TreeNode left; // Left child
    TreeNode right; // Right child
    TreeNode() {} // Constructor without parameters
    TreeNode(int val) { this.val = val; } // Constructor with value parameter
    // Constructor with value and left, right children parameters
    TreeNode(int val, TreeNode left, TreeNode right) {
        this.val = val;
        this.left = left;
        this.right = right;
    }
}

class Solution {
    // Public method to check if a tree is a univalued tree
    public boolean isUnivalTree(TreeNode root) {
        // If the root is null, the tree is trivially univalued
        if (root == null) {
            return true;
        }
        // Use depth-first search starting with the root value to check univalued property
        return dfs(root, root.val);
    }

    // Private helper method to perform depth-first search on tree nodes
    private boolean dfs(TreeNode node, int val) {
        // If the current node is null, we've reached the end of a branch - return true
        if (node == null) {
            return true;
        }
        // Check if the current node value is equal to the given value
        // and recursively check both the left and right subtrees
        return node.val == val && dfs(node.left, val) && dfs(node.right, val);
    }
}
```

C++

```cpp
// Definition for a binary tree node.
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    // Constructor to initialize node with a value, and with left and right child pointers set to nullptr.
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
    // Constructor to initialize node with a value and explicit left and right child nodes.
    TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
};

class Solution {
public:
    // Public method to check if a tree is a unival tree, where all nodes' values are the same.
    bool isUnivalTree(TreeNode* root) {
        // Call the private DFS method, starting with the root's value, to check for unival.
        return dfs(root, root->val);
    }

private:
    // Private helper method to perform DFS and check if all nodes' values are equal to a given value.
    bool dfs(TreeNode* node, int val) {
        // If the current node is nullptr, return true because we have not encountered a different value.
        if (!node) return true;

        // Check if the current node's value equals the given value and
        // recursively call DFS for both left and right subtrees.
        // The tree is unival down the current path only if both subtrees are also unival.
        return node->val == val && dfs(node->left, val) && dfs(node->right, val);
    }
};
```

TypeScript

```typescript
// To check if a binary tree is a unival tree (all nodes have the same value)

// Function to check if the tree is a unival tree
function isUnivalTree(root: TreeNode | null): boolean {
    if (!root) {
        return true; // An empty tree is considered a unival tree
    }

    // Helper function to perform depth-first search
    function checkUnival(root: TreeNode | null, value: number): boolean {
        if (root === null) {
            return true; // Reached the end of a branch
        }
        if (root.val !== value) {
            return false; // Current node value doesn't match the value we're checking against
        }
        // Recursively check left and right subtrees
        return checkUnival(root.left, value) && checkUnival(root.right, value);
    }

    // Start checking univality from the root node using its value
    return checkUnival(root, root.val);
}

// Definition for a binary tree node
class TreeNode {
    val: number
    left: TreeNode | null;
    right: TreeNode | null;

    constructor(val?: number, left?: TreeNode | null, right?: TreeNode | null) {
        this.val = val !== undefined ? 0 : val;
        this.left = left === undefined ? null : left;
        this.right = right === undefined ? null : right;
    }
}

// Example usage:
// const tree = new TreeNode(1, new TreeNode(1), new TreeNode(1));
// console.log(isUnivalTree(tree)); // Output: true
```

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

class Solution:
    def isUnivalTree(self, root: TreeNode) -> bool:
        """
        Determines if a tree is a unival tree (where all nodes have the same value)

        Args:
        root: The root of the binary tree.

        Returns:
        A boolean value indicating if the tree is a unival tree.
        """

        # Helper function to perform depth-first search
        def depth_first_search(node):
            # If we reach a None, it means we are at a leaf node, or the tree is empty
            # Since a None is technically univalued, we return True here
            if node is None:
                return True

            # Check if the current node's value is same as the root's value
            # And recursively check the left and right subtrees
            return (node.val == root.val
                    and depth_first_search(node.left)
                    and depth_first_search(node.right))

        # Start the depth-first search from the root
        return depth_first_search(root)
```

Time and Space Complexity

Time Complexity

The time complexity of the provided code is `O(n)`, where `n` is the number of nodes in the binary tree. This is because each node in the tree is visited exactly once to check whether it matches the value of the root node.

Space Complexity

The space complexity of the code is `O(h)`, where `h` is the height of the binary tree. This is the space required by the call stack due to recursion. In the worst case, where the tree is skewed, the space complexity would be `O(n)`, corresponding to the number of recursive calls equal to the number of nodes in a skewed tree.