112. Path Sum

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

Problem Description

The problem presents a binary tree where each node contains an integer value. The goal is to find out if there is a path from the root node down to any leaf node such that the sum of the values of all the nodes along the path is equal to the given integer `targetSum`. A leaf node is defined as a node that does not have any children. If such a path exists, the function should return `true`; otherwise, it should return `false`.

Intuition

The solution to this problem is based on the Depth-First Search (DFS) algorithm. DFS is a common tree traversal method that can be used efficiently to search all possible paths in a tree from the root node down to the leaves.

The intuition behind using DFS in this particular problem is the need to explore each possible path fully before moving to an alternative path. We traverse down one branch of the tree, adding the values of each node we pass through, until we reach a leaf. Once at a leaf, we check if the accumulated sum is equal to `targetSum`.

If we hit a leaf and the sum does not equal `targetSum`, we backtrack and try a different path. This backtracking is naturally handled by the call stack in a recursive DFS approach.

Thus, the solution approach is to start from the root and explore each branch recursively, carrying forwards the cumulative sum. If at any point we reach a leaf and the sum matches `targetSum`, we return `true`. Otherwise, if we exhaust all paths and none meet the criteria, we return `false`.

As part of the recursion, we ensure two things:

1. If the current node is `None` (i.e., we've reached beyond a leaf), this path is not valid, and we return `false`.
2. When we reach a leaf node (both its children are `None`), we check if the cumulative sum equals `targetSum` and return the result of this comparison.

The provided code defines a nested `dfs` function inside the main function `hasPathSum` that handles the recursive traversal and the accumulation of the sum.

Solution Approach

The solution utilizes a straightforward recursive DFS strategy. By defining a helper function `dfs(root, s)`, we can keep track of the current sum `s` as we traverse down the tree. The base case checks if the current node `root` is `None`, meaning that the path has reached beyond a leaf node, and hence, it returns `False` since there cannot be a valid path through a non-existing node.

When the recursive function `dfs` is called, it performs the following steps:

1. It first checks if the current node is `None`. If true, the recursion stops and returns `False`.
2. If the current node is not `None`, it adds the node's value to the ongoing sum `s`.
3. Next, it checks if the current node is a leaf (both `root.left` and `root.right` are `None`). If it is a leaf, and `s` equals `targetSum`, it indicates that a valid path has been found and returns `True`.
4. If the current node is not a leaf, the function proceeds to call itself recursively for both the left and right children of the current node, if they exist (`dfs(root.left, s)` and `dfs(root.right, s)`).
5. The result of these recursive calls is combined using logical OR (`or`), which means if either the left or the right subtree has a valid path, the function will return `True`.

The `dfs` function encapsulates the logic for searching the tree and determining if a valid path exists that sums up to `targetSum`. It is initiated by passing the root of the tree and an initial sum of `0` to the `dfs` function.

Here is how the `dfs` function is implemented within the context of the provided solution code snippet:

```python
def dfs(root, s):
    if root is None:  # Base case: empty node, return False
        return False
    s += root.val  # Increment the sum by the current node's value
    # If it's a leaf and the sum matches targetSum, return True
    if root.left is None and root.right is None and s == targetSum:
        return True
    # Recursively search the left and right subtrees
    return dfs(root.left, s) or dfs(root.right, s)
```

And the `hasPathSum` function then calls this `dfs` function with the initial parameters:

```python
return dfs(root, 0)
```

In summary, the recursive function explores all possible paths from the root to each leaf, checking for a match against `targetSum`. It returns `True` as soon as a valid path is found, or `False` if no such path exists, effectively implementing a depth-first search for the problem at hand.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach. Assume we have the following binary tree and `targetSum` of 22.

```
        5
       / \
      4   8
     /   / \
    11  13  4
   /  \      \
  7    2      1
```

Now, let's follow the `dfs` function to see how it works:

1. Begin at the root (node with value 5), with a starting sum `s` of 0. Call `dfs(root=5, s=0)`.
2. At root (5), `s` becomes 5 (`0 + 5`). This node is not a leaf, so we make recursive calls for left and right children.
   - Call `dfs(root=4, s=5)` to continue on the left child.
3. At node 4, `s` becomes 9 (`5 + 4`). This is also not a leaf. Recursive call on left child.
   - Call `dfs(root=11, s=9)`.
4. At node 11, `s` becomes 20 (`9 + 11`). Not a leaf, we have two children, so we make two recursive calls.
   - First, call `dfs(root=7, s=20)`.
   - Then we will call `dfs(root=2, s=20)`, but let's follow the first call for now.
5. At node 7, `s` becomes 27 (`20 + 7`). This is a leaf, but `s` is not equal to the `targetSum` of 22. Returns `False`.
   - Backtrack to node 11 and proceed with the second call, `dfs(root=2, s=20)`.
6. At node 2, `s` becomes 22 (`20 + 2`). This is a leaf and `s` matches `targetSum`. Returns `True`.

Once `True` is encountered, the function does not need to explore any further branches. The propagation of `True` through the recursive calls will eventually lead to the `hasPathSum` function returning `True`, indicating that there is a path that adds up to the `targetSum` of 22.

The binary tree has successfully been searched with a depth-first approach to find a path sum that matches the target. In this case, the path with values `[5, 4, 11, 2]` yields the desired sum, and so our problem is solved with the function returning `True`.

Python Solution

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

class Solution:
    def hasPathSum(self, root: Optional[TreeNode], target_sum: int) -> bool:
        """
        Determines if the tree has a root-to-leaf path such that
        adding up all the values along the path equals the given sum.

        :param root: TreeNode, the root of the binary tree
        :param target_sum: int, the target sum to be achieved
        :return: bool, True if such a path exists, False otherwise
        """

        def dfs(node, current_sum):
            """
            Depth-first search helper function that traverses the tree
            to find a root-to-leaf path that sums to the target_sum.

            :param node: TreeNode, the current node being traversed
            :param current_sum: int, the sum of the values from the root node up to the current node
            :return: bool, True if a path is found, False otherwise
            """
            if node is None:
                # Base case: if the current node is None, no path exists
                return False

            # Update the current_sum by adding the current node's value
            current_sum += node.val

            # Check if the current node is a leaf and matches the target_sum
            if node.left is None and node.right is None and current_sum == target_sum:
                return True

            # Recursively search in the left and right subtrees for the path
            # If either subtree returns True, a valid path has been found
            return dfs(node.left, current_sum) or dfs(node.right, current_sum)

        # Start DFS traversal from the root with an initial sum of 0
        return dfs(root, 0)

```

Java Solution

```java
/**
 * Definition for a binary tree node.
 */
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 {
    /**
     * Returns true if the tree has a root-to-leaf path such that adding up all the values along the path equals the given target sum
     *
     * @param root The root of the binary tree.
     * @param targetSum The target sum to find.
     * @return True if such a path exists, otherwise false.
     */
    public boolean hasPathSum(TreeNode root, int targetSum) {
        return hasPathSumDFS(root, targetSum);
    }

    /**
     * Helper method to perform depth-first search to find the path sum.
     *
     * @param node The current node being visited.
     * @param currentSum The sum accumulated so far.
     * @return True if a path with the given sum is found, otherwise false.
     */
    private boolean hasPathSumDFS(TreeNode node, int currentSum) {
        // If the node is null, we've hit a dead end and should return false.
        if (node == null) {
            return false;
        }

        // Subtract the value of current node from current sum.
        currentSum -= node.val;

        // Check if the current node is a leaf and the current sum equals zero,
        // which means we've found a path with the required sum.
        if (node.left == null && node.right == null && currentSum == 0) {
            return true;
        }

        // Recursively check the left and right subtrees for the remaining sum.
        // If either subtree returns true, a path has been found.
        return hasPathSumDFS(node.left, currentSum) || hasPathSumDFS(node.right, currentSum);
    }
}

```

C++ Solution

```cpp
// Definition for a binary tree node
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    // Constructor to initialize a tree node with a value and optional left and right children
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
    // Constructor to initialize a tree node with a value and both children
    TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
};

class Solution {
public:
    // Checks if the binary tree has a root-to-leaf path that sums up to the targetSum
    bool hasPathSum(TreeNode* root, int targetSum) {
        // Lambda function that performs depth-first search on the tree to find path sum
        // [&] captures local variables by reference
        std::function<bool(TreeNode*, int)> dfs = [&](TreeNode* node, int sumSoFar) -> bool {
            // If the node is null, return false as we've reached beyond a leaf node
            if (!node) return false;

            // Add the current node's value to the running sum
            sumSoFar += node->val;

            // Check if the current node is a leaf and the sum equals the targetSum
            if (!node->left && !node->right && sumSoFar == targetSum) {
                return true;
            }

            // Continue searching in the left and right subtrees
            // If any of the recursive calls return true, the whole expression will be true
            // Utilize the short-circuit evaluation of the '||' operator
            return dfs(node->left, sumSoFar) || dfs(node->right, sumSoFar);
        };

        // Start the depth-first search from the root node with an initial sum of 0
        return dfs(root, 0);
    }
};

```

Typescript Solution

```typescript
// Typescript type definition for a binary tree node.
type TreeNode = {
  val: number;
  left: TreeNode | null;
  right: TreeNode | null;
};

/**
 * Determines if the binary tree has a root-to-leaf path that sums up to the given target sum.
 *
 * @param {TreeNode | null} root - The root node of the binary tree.
 * @param {number} targetSum - The target sum to find the path for.
 * @returns {boolean} - True if such a path exists, otherwise False.
 */
function hasPathSum(root: TreeNode | null, targetSum: number): boolean {
  // If the root is null, there is no path, return false.
  if (root === null) {
    return false;
  }

  // Destructure the current node to get its value and its children.
  const { val, left, right } = root;

  // If the node is a leaf (no left or right children),
  // Check if subtracting the node's value from the target sum equals 0.
  if (left === null && right === null) {
    return targetSum - val === 0;
  }

  // Recursively check the left and right subtrees reducing the target sum by the current node's value.
  // Return true if either subtree has a path that sums up to the adjusted target sum.
  return hasPathSum(left, targetSum - val) || hasPathSum(right, targetSum - val);
}

```

Time and Space Complexity

Time Complexity

The time complexity of the given code is `O(n)`, where `n` is the number of nodes in the binary tree. The reason for this is that the code implements a depth-first search (DFS) algorithm, which visits each node exactly once in the worst-case scenario.

Space Complexity

The space complexity of the code is `O(h)`, where `h` is the height of the binary tree. This space is used by the execution stack for recursive calls. In the worst-case scenario where the tree is completely unbalanced, the space complexity will be `O(n)`. But on a balanced tree, the height `h` can be considered as `O(log n)`, making the average space complexity `O(log n)`.