404. Sum of Left Leaves

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

Problem Description

The problem provides us with the root of a binary tree and asks us to calculate the sum of all left leaves in the tree. In the context of this problem, a *leaf* is defined as a node with no children and a *left leaf* is specifically a leaf that is also the left child of its parent node. Our goal is to traverse the binary tree and find all the nodes that satisfy the condition of being a left leaf, and then sum their values to return the final result.

Intuition

The solution to the problem is rooted in a classic tree traversal approach. We must navigate through all the nodes of the tree to identify which ones are left leaves. Since a leaf node is one with no children, we can determine a left leaf if the following conditions are met:

- The node is the left child of its parent (`root.left`).
- The node itself does not have any children (`root.left.left` is `None` and `root.left.right` is `None`).

The process involves a recursive function that inspects each node. We can accumulate the sum incrementally during the traversal. When we hit a `None` node (indicating the end of a branch), we return 0 since it doesn't contribute to the sum. If we find a left leaf, we add its value to the result. After inspecting a node for being a left leaf, we continue the traversal for both its left and right children because they may have their own left leaves further down the tree. The sum of left leaf values for the entire tree is the result of aggregating the values of all left leaves found during the traversal.

The recursive nature of binary tree traversal warrants a termination condition. In this case, we return a sum of 0 when a `None` node (signifying a non-existent child of a leaf node) is encountered. This is the base case for our recursive calls.

Using this intuition, the function `sumOfLeftLeaves` correctly sums only the left leaves of the binary tree, adhering strictly to the given definitions and constraints without unnecessary computation.

Solution Approach

The problem is solved using a recursive approach that includes the Depth-First Search (DFS) pattern. Let's break down the implementation:

1. **Base Case**: If the current node is `None`, which means we have reached beyond the leaf nodes, we just return `0` as there is no contribution to the sum from a non-existent leaf.
2. **Identifying a Left Leaf**:
   - When the recursive call is made for a left child node, we check: a. If the left child itself is a leaf node by confirming that both the `left` and `right` children of the left child node are `None`. b. If the above condition is satisfied, we've identified a left leaf, and we add its value to the `res` (the running total of the sum of left leaves).
3. **Recursive Calls**:
   - We make a recursive call on the `left` child of the current node to continue searching for left leaves in the left subtree.
   - The result of this call (which is the sum of left leaves found in the left subtree) is added to `res`.
   - We do not stop after checking the left child. To ensure all left leaves are accounted for, we also make a recursive call on the `right` child of the current node. However, this time any leaf we find will not be a left leaf (because it comes from a right child node), so we won't add their values unless they have left leaves in their own subtrees.
4. **Returning the Result**:
   - After adding values from left leaves found in both the left and right subtrees, the final `res` value is returned, which represents the sum of all left leaves in the binary tree.

By following this approach, all nodes in the binary tree are visited once, and left leaves are identified and summed up. The use of recursion makes the code easier to understand and succinctly handles the tree traversal and left leaf summing in one coherent process. Data structures aren't explicitly used besides the inherent recursive stack that manages the sequence of function calls.

The solution is efficient as it has a time complexity of O(n), where n is the number of nodes in the tree (since each node is visited once), and space complexity is O(h), where h is the height of the tree, which corresponds to the height of the recursive call stack.

Here is the part of the code that is critical for the understanding of the recursive approach:

```
if root.left and root.left.left is None and root.left.right is None:
    res += root.left.val
res += self.sumOfLeftLeaves(root.left)
res += self.sumOfLeftLeaves(root.right)
```

This snippet includes the check for a left leaf (the `if` statement), and the recursive calls to traverse the left and right subtrees (the subsequent two lines). The summing up of leaf nodes' values happens naturally with each recursive return, allowing for an elegant and effective solution.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach:

Suppose we have the following binary tree:

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

In this tree, we have two left leaves: 9 and 15.

1. We start with the `root` node which has a value of 3. It's not a leaf, so we proceed to check its children.
2. We check the left child of the root, which is `9`. It has no left or right children, making it a left leaf node. We add its value to `res`, so `res = 9`.
3. Next, we explore the right child of the root, which is `20`. It's not a leaf, so we need to traverse its children.
4. We check the left child of node `20`, which is `15`. Again, it has no left or right children, qualifying it as a left leaf. We add its value to `res`, so now `res = 9 + 15 = 24`.
5. We continue and check the right child of node `20`, which is `7`. Even though `7` is a leaf, it is not a left leaf (since it's the right child of its parent), so we do not add its value to `res`.
6. Our traversal is complete, and we have successfully identified all left leaves in the binary tree. The final sum of all left leaves is `24`.

Applying the given solution to this example, the following is how the recursive function `sumOfLeftLeaves` processes the nodes:

- The base case returns `0` for non-existent nodes, which is not visible in this example but is crucial for recursion.
- The identification of a left leaf is made when we check node `9` and node `15`. For node `9`, `root.left.left` and `root.left.right` are both `None`, satisfying the left leaf conditions. The same applies to node `15`.
- Recursive calls allow us to explore all subtrees of the binary tree. In our example, recursive calls are made to nodes `9`, `20`, `15`, and `7`, but only `9` and `15` contribute to the sum `res`.
- The result is aggregated throughout the recursive calls and finally returns the sum of all left leaves when all nodes have been visited.

By following these steps, our recursive function would successfully return the sum of all left leaves for any binary tree given to it.

Python Solution

```python
class TreeNode:
    # Initialization of a Tree Node
    def __init__(self, value):
        self.val = value  # Assign the value to the node
        self.left = None  # Initialize left child as None
        self.right = None  # Initialize right child as None


class Solution:
    # Function to calculate the sum of all left leaves in a binary tree
    def sumOfLeftLeaves(self, root: TreeNode) -> int:
        # If the root is None, then return 0 as there are no leaves
        if root is None:
            return 0

        # Initialize result as 0
        sum_left_leaves = 0

        # Check if the left child exists and it's a leaf node
        if root.left and root.left.left is None and root.left.right is None:
            sum_left_leaves += root.left.val  # Add its value to the sum

        # Recursively find the sum of left leaves in the left subtree
        sum_left_leaves += self.sumOfLeftLeaves(root.left)

        # Recursively find the sum of left leaves in the right subtree
        sum_left_leaves += self.sumOfLeftLeaves(root.right)

        # Return the total sum of left leaves
        return sum_left_leaves

```

Java Solution

```java
/**
 * Definition for a binary tree node.
 */
class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;

    TreeNode(int x) {
        val = x;
    }
}

public class Solution {

    /**
     * Calculates the sum of all left leaves in a binary tree.
     *
     * @param root the root of the binary tree
     * @return the sum of all left leaves' values
     */
    public int sumOfLeftLeaves(TreeNode root) {
        // Base case: If the current node is null, return 0 since there are no leaves.
        if (root == null) {
            return 0;
        }

        // Initialize sum to keep track of the left leaves sum.
        int sum = 0;

        // Check if the current node has a left child and the left child is a leaf node.
        if (root.left != null && isLeaf(root.left)) {
            // If it's a left leaf node, add its value to the sum.
            sum += root.left.val;
        }

        // Recursive call to traverse the left subtree and add any left leaves found to the sum.
        sum += sumOfLeftLeaves(root.left);
        // Recursive call to traverse the right subtree but left leaves in this subtree are not added.
        sum += sumOfLeftLeaves(root.right);

        // Return the total sum of left leaves found.
        return sum;
    }

    /**
     * Helper method to check if a given node is a leaf node.
     *
     * @param node the node to check
     * @return true if the node is a leaf node, false otherwise
     */
    private boolean isLeaf(TreeNode node) {
        return node.left == null && node.right == null;
    }
}

```

C++ Solution

```cpp
#include <iostream>

// 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:
    // Helper function to perform a depth-first search on the binary tree to find the sum of left leaves.
    int depthFirstSearch(TreeNode* root, bool isLeft) {
        // Base case: If the current node is null, return 0.
        if (root == nullptr) {
            return 0;
        }

        // Check if the current node is a leaf node.
        if (root->left == nullptr && root->right == nullptr) {
            // If it is a left child, return its value; otherwise, return 0.
            return isLeft ? root->val : 0;
        }

        // Recursively sum the values of left leaves from the left and right subtrees.
        int leftSum = depthFirstSearch(root->left, true);
        int rightSum = depthFirstSearch(root->right, false);
        return leftSum + rightSum;
    }

    // Function to calculate the sum of all left leaves in a binary tree.
    int sumOfLeftLeaves(TreeNode* root) {
        // Call the depth-first search starting from the root.
        // The initial call is not a left child, so isLeft is false.
        return depthFirstSearch(root, false);
    }
};

int main() {
    // Example of usage:
    // Construct a binary tree.
    TreeNode* root = new TreeNode(3);
    root->left = new TreeNode(9);
    root->right = new TreeNode(20, new TreeNode(15), new TreeNode(7));

    Solution solution;
    // Calculate the sum of all left leaves in the binary tree.
    std::cout << "Sum of left leaves: " << solution.sumOfLeftLeaves(root) << std::endl;

    // Don't forget to delete allocated memory to avoid memory leaks.
    // This is just a quick example and does not delete the entire tree.
    delete root->right->left;
    delete root->right->right;
    delete root->right;
    delete root->left;
    delete root;

    return 0;
}

```

Typescript Solution

```typescript
// Function to perform a depth-first search on the binary tree to find the sum of left leaves.
// root: The current node we are visiting.
// isLeft: A boolean value indicating whether the current node is a left child.
const depthFirstSearch = (root: TreeNode | null, isLeft: boolean): number => {
    // Base case: If the current node is null, return 0.
    if (!root) {
        return 0;
    }

    // Destructuring to get the value, left child, and right child of the current node.
    const { val, left, right } = root;

    // Check if the current node is a leaf node.
    if (!left && !right) {
        // If it is a left child, return its value; otherwise, return 0.
        return isLeft ? val : 0;
    }

    // Recursively sum the values of left leaves from the left and right subtrees.
    return depthFirstSearch(left, true) + depthFirstSearch(right, false);
};

// Function to calculate the sum of all left leaves in a binary tree.
// root: The root of the binary tree.
function sumOfLeftLeaves(root: TreeNode | null): number {
    // Call the depth-first search starting from the root.
    // The initial call is not a left child, so isLeft is false.
    return depthFirstSearch(root, false);
}

```

Time and Space Complexity

The provided Python code defines a function `sumOfLeftLeaves` that calculates the sum of all left leaves in a given binary tree. The function is a recursive implementation that traverses the entire tree to find and sum all the left leaf nodes.

Time Complexity

The time complexity of the `sumOfLeftLeaves` function is `O(n)`, where `n` is the number of nodes in the binary tree. This is because the algorithm must visit each node exactly once to check if it is a left leaf node or not.

Space Complexity

The space complexity of the `sumOfLeftLeaves` function can be considered as `O(h)`, where `h` is the height of the binary tree. This space is used for the recursive call stack. In the worst-case scenario (e.g., a completely unbalanced tree), the height of the tree can be `n` (the same as the number of nodes), and thus the space complexity can be `O(n)`. However, in the best case (a completely balanced tree), the height of the tree is `log(n)`, which results in a space complexity of `O(log(n))`.