1315. Sum of Nodes with Even-Valued Grandparent

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

Problem Description

The problem requires us to compute the sum of values of all nodes in a binary tree that have an even-valued grandparent. A grandparent of a node is defined as the parent of that node's parent, whenever such a relative exists. The binary tree is given by the `root` node, and we need to traverse the tree to find the nodes of interest and sum their values. If there are no nodes with an even-valued grandparent, the sum to be returned should be 0.

Intuition

To solve this problem, the solution uses a Depth-First Search (DFS) traversal approach. This means we explore as far as possible down each branch before backtracking. This approach is suitable for binary trees because it allows us to maintain state information as we traverse each level of the tree, which is essential for identifying grandparents and their grandchildren.

The intuition behind using DFS in this problem is that we can keep track of the grandparent and parent of the current node as we traverse the tree. Whenever we identify a grandparent with an even value, we know that we have to check its grandchildren (the children of the current parent node) and add their values to our sum, if they exist.

The algorithm involves starting at the root of the tree, then recursively going through each branch, keeping track of the current node, its parent, and grandparent. We pass the current parent to be the grandparent in the next level of the recursive call, and the current node's children to be the parent in the next level. Whenever the current grandparent's value is even, we add the value of the current parent's children to the result.

This method requires a closure or an external variable to keep the sum while the recursion is taking place, as illustrated in the solution code with `self.res`, which is initialized to `0`. As the recursion unfolds, `self.res` is updated whenever we are at nodes with an even-valued grandparent.

Solution Approach

The provided Python solution implements a Depth-First Search (DFS) algorithm to traverse the binary tree and identify the nodes that contribute to the sum based on the even-valued grandparent condition. Here's a step-by-step breakdown of how the code works, aligning with common practices in DFS algorithms:

1. **Initialization**: A variable `self.res` is defined at the class level to keep track of the cumulative sum of all nodes meeting the criteria (nodes with even-valued grandparents).
2. **DFS Function**: A nested function `dfs(g, p)` is defined within the `sumEvenGrandparent` function. This function takes two arguments: `g` for the grandparent node and `p` for the parent node. The key recursion happens within this helper function.
3. **Base Case**: Inside the `dfs` function, a base case checks if the parent node `p` is `None`. If it is, the function returns without performing any action. This means we have reached the end of a branch in the binary tree.
4. **Checking Grandparent's Value**: Before diving deeper into the tree, the function checks if the current grandparent node's value is even (`g.val % 2 == 0`). If true, and if the parent node has any children (`p.left` and `p.right`), their values are added to `self.res`.
5. **Recursive Calls**: The function then makes two recursive calls to continue the DFS traversal: one for the left child of the parent (`dfs(p, p.left)`) and one for the right child (`dfs(p, p.right)`). The current parent becomes the grandparent for the next level of the recursion, and its children become the new parent nodes.
6. **Starting the Traversal**: The initial calls to the `dfs` function start from the children of the root node since the root itself cannot be a grandchild (root's children have no grandparent and root's grandchildren are the children of `root.left` and `root.right`). Therefore, the DFS starts with `dfs(root, root.left)` and `dfs(root, root.right)` which effectively checks the trees rooted at the left and right child of the root node for eligible grandchildren.
7. **Result**: After the complete tree has been traversed, the function returns the sum stored in `self.res`.

By structuring the solution with recursion and maintaining state across recursive calls, the DFS algorithm effectively identifies and sums up all the nodes that fulfill the even-valued grandparent criterion. The use of a class variable to store the ongoing sum (`self.res`) avoids the need for passing accumulator variables through the recursive stack, simplifying the implementation.

Example Walkthrough

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

```
     6
    / \
   3   5
  / \   \
 2   11  9
    / \
   13  5
```

In this tree:

- The root node has a value of 6.
- The root's children are nodes with values 3 (left child) and 5 (right child).
- The grandchildren (from root) are nodes with values 2, 11 (left subtree) and 9 (right subtree).
- Node 11 has children with values 13 (left child) and 5 (right child).

Now, let's apply the solution approach as described in the content:

1. **Initialization**: We initialize `self.res` to `0`. This will hold the sum of all grandchild node values where the grandparent value is even.
2. **DFS Function**: The `dfs` function is ready to be invoked. It will be called with the grandparents and parents of each node.
3. **Base Case**: When `dfs` encounters a `None` node, it returns. This base case prevents the function from running indefinitely and ensures we only consider existing nodes.
4. **Checking Grandparent's Value**: As we call the `dfs` function, we check if the grandparent (`g`) node value is even. When we are at node 3 with grandparent 6, we add the values of node 3's children (2 and 11) to `self.res` since 6 is even. Similarly, when we are at node 5 with grandparent 6, we check node 5's child (9) and add its value to `self.res`.
5. **Recursive Calls**: We then recursively call `dfs` on node 3's children (2 and 11) and node 5's child (9), setting the current parent as the new grandparent for the next level.
6. **Traversal**: The DFS begins with `dfs(6, 3)` and `dfs(6, 5)`. These calls evaluate the left and right subtrees rooted at nodes 3 and 5, respectively.
7. **Result**: We sum the values of the nodes that have even-valued grandparents throughout the traversal. The nodes that satisfy this condition in our example are 2, 11, and 9. Thus, `self.res` becomes 2 + 11 + 9 = 22 by the end of the traversal.

After performing these steps, the final result (`self.res`) holds the sum of all nodes with even-valued grandparents. In this example, the output would be `22`.

Python Solution

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

class Solution:
    def sumEvenGrandparent(self, root: TreeNode) -> int:
        # Initialize the total sum of values.
        self.total_sum = 0

        def dfs(grandparent, parent):
            """
            Depth-first search to find and add values of grandchildren with even-valued grandparents.

            :param grandparent: Reference to the grandparent node.
            :param parent: Reference to the parent node.
            """
            # Base case to terminate recursion if we reach a None node.
            if parent is None:
                return

            # If the grandparent's value is even, add the grandchildren's values.
            if grandparent.val % 2 == 0:
                if parent.left:
                    self.total_sum += parent.left.val
                if parent.right:
                    self.total_sum += parent.right.val

            # Recur on the left and right child of the current parent node.
            dfs(parent, parent.left)
            dfs(parent, parent.right)

        # Handle the edge case if the tree doesn't contain any nodes.
        if root is None:
            return 0

        # Run DFS starting from the root's children to maintain the grandparent relationship.
        dfs(root, root.left)
        dfs(root, root.right)

        # Return the calculated total sum.
        return self.total_sum

```

Java Solution

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

    // Node Constructor without children
    TreeNode(int val) {
        this.val = val;
    }

    // Node Constructor with children
    TreeNode(int val, TreeNode left, TreeNode right) {
        this.val = val;
        this.left = left;
        this.right = right;
    }
}

public class Solution {
    private int totalSum;

    // Main method to calculate the sum of values of all nodes with even-valued grandparent
    public int sumEvenGrandparent(TreeNode root) {
        totalSum = 0;
        if (root != null) {
            // If the tree is not empty, perform depth-first search on both children
            depthFirstSearch(null, root, root.left);
            depthFirstSearch(null, root, root.right);
        }
        return totalSum;
    }

    // Helper method to perform depth-first search and accumulate the sum
    private void depthFirstSearch(TreeNode grandparent, TreeNode parent, TreeNode current) {
        if (current == null) {
            // If current node is null, stop the recursion
            return;
        }

        // If grandparent is not null and has an even value
        if (grandparent != null && grandparent.val % 2 == 0) {
            // Add the value of the current node to the total sum
            totalSum += current.val;
        }

        // Recurse for children of the current node
        // The current parent becomes the new grandparent, and the current node becomes the parent
        depthFirstSearch(parent, current, current.left);
        depthFirstSearch(parent, current, current.right);
    }
}

```

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:
    // Declare a class member to store the result
    int totalSum;

    // Function to initialize the summing process for finding the total sum of all nodes
    // with even-valued grandparents
    int sumEvenGrandparent(TreeNode* root) {
        // Initialize result to 0
        totalSum = 0;
        // Start Depth-First Search (DFS) from the children of the root node
        if (root) {
            dfs(root, root->left);
            dfs(root, root->right);
        }
        // Return the result after traversing the whole tree
        return totalSum;
    }

    // Depth-First Search function to traverse the tree and calculate the sum
    void dfs(TreeNode* grandparent, TreeNode* parent) {
        // If there is no parent node, return to stop the recursion
        if (!parent) return;

        // Check if the grandparent's value is even.
        // If it is, add the values of the parent's children to the result.
        if (grandparent->val % 2 == 0) {
            if (parent->left) totalSum += parent->left->val;
            if (parent->right) totalSum += parent->right->val;
        }

        // Continue the DFS for the children of the current parent node
        dfs(parent, parent->left);
        dfs(parent, parent->right);
    }
};

```

Typescript Solution

```typescript
// Define the tree node structure
class TreeNode {
    val: number;
    left: TreeNode | null;
    right: TreeNode | null;

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

// Variable to store the running total sum of all nodes with even-valued grandparents
let totalSum: number;

// Function to calculate the total sum of all nodes with even-valued grandparents
function sumEvenGrandparent(root: TreeNode | null): number {
    // Initialize the total sum to 0
    totalSum = 0;

    // Start the depth-first search if the root node exists
    if (root) {
        if (root.left) {
            dfs(root, root.left); // DFS on the left child of the root
        }
        if (root.right) {
            dfs(root, root.right); // DFS on the right child of the root
        }
    }

    // Return the calculated total sum
    return totalSum;
}

// Depth-first search function to traverse the tree
function dfs(grandparent: TreeNode, parent: TreeNode | null): void {
    // If the parent node does not exist, return to stop the recursion
    if (!parent) return;

    // Check if the grandparent's value is even
    if (grandparent.val % 2 === 0) {
        // If the grandparent's value is even, add the values of the parent's children to the total sum
        if (parent.left) {
            totalSum += parent.left.val;
        }

        if (parent.right) {
            totalSum += parent.right.val;
        }
    }

    // Recursively continue DFS for each child of the parent node
    if (parent.left) {
        dfs(parent, parent.left);
    }

    if (parent.right) {
        dfs(parent, parent.right);
    }
}

```

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. This is because the code performs a Depth-First Search (DFS) and visits each node exactly once.

Space Complexity: The space complexity of the code is `O(h)` where `h` is the height of the binary tree. This is due to the recursion stack during the DFS traversal. In the worst case (when the binary tree is skewed), the space complexity would be `O(n)`. In the best case (when the binary tree is perfectly balanced), it would be `O(log n)`.