563. Binary Tree Tilt

Problem Description

In this problem, we are given the root of a binary tree. Our task is to calculate the sum of the "tilt" of all the nodes in the tree. The tilt of a single node is defined as the absolute difference between the sum of all values in its left subtree and the sum of all values in its right subtree. Here, the left or right subtree may not exist; in that case, the sum for that subtree is considered to be 0. After calculating the tilt for each individual node, we need to add them all up to get the final result, which is the sum of all tilts in the tree.

Intuition

The intuition behind the solution lies in performing a post-order traversal of the tree (i.e., visit left subtree, then right subtree, and process the node last). When we are at any node, we need to know the sum of the values in its left subtree and its right subtree to calculate the node's tilt.

The solution uses a helper function `sum` to traverse the tree. During the traversal, the function calculates two things for every node:

1. The sum of all values in the subtree rooted at this node, which is needed by the parent of the current node to calculate its tilt.
2. The tilt of the current node, which is the absolute difference between the sum of values in the left and right subtrees.

As we are doing a post-order traversal, we first get the sum of values for left and right children (recursively calling `sum` function for them), calculate the current node's tilt, and add it to the overall `ans`, which is kept as a non-local variable so that it retains its value across recursive calls.

Finally, we return the sum of values of the subtree rooted at the current node, which is the value of the current node plus the sum of values from left and right subtree, which then can be used by the parent node to calculate its tilt.

The main function `findTilt` calls this helper function with the root of the tree, starts the recursive process, and once finished, returns the total tilt accumulated in `ans`.

Solution Approach

The solution's backbone is a recursive function that performs a post-order traversal of the binary tree. This traversal means that we process the left subtree, then the right subtree, and finally the node itself.

Here's how the approach works, explained in steps:

1. Define a recursive function called `sum`, which takes a node of the tree as an argument.
2. If the node is `None`, meaning we have reached beyond a leaf node, return `0`.
3. Recursively call the `sum` function on the left child of the current node and store the result as `left`.
4. Do the same for the right child and store the result as `right`.
5. Calculate the tilt for the current node by finding the absolute difference between `left` and `right`, which is `abs(left - right)`.
6. Add the tilt of the current node to the global sum `ans`, which is updated using a `nonlocal` variable. `nonlocal` is used so that nested `sum` function can access and modify the `ans` variable defined in the enclosing `findTilt` function's scope.
7. The `sum` function returns the total value of the subtree rooted at the current node, which includes the node's own value and the sum of its left and right subtrees: `root.val + left + right`.
8. Finally, the `findTilt` function initializes `ans` to `0` and calls the `sum` function with the root of the tree as the argument to kick-off the process.
9. Once the recursive calls are finished (the entire tree has been traversed), `findTilt` returns the total tilt that has been accumulated in `ans`.

This implementation is efficient since each node in the tree is visited only once. The `sum` function calculates both the sum of subtree values and the tilt on the fly. It is a fine example of a depth-first search (DFS) algorithm, where we go as deep as possible down one path before backing up and checking other paths for the solution.

Example Walkthrough

To illustrate the solution approach, let's consider a binary tree with the root node having a value of `4`, a left child with value `2`, and a right child with value `7`. The left child of the root has its own children with values `1` and `3`, respectively, and the right child of the root has children with values `6` and `9`. Here's a visual representation of the tree:

```
    4
   / \
  2   7
 / \ / \
1  3 6  9
```

We want to calculate the sum of the tilt of all the nodes in the tree. Let's walk through the approach step by step:

1. We define the recursive `sum` function and initialize the traversal from the root of the tree (`4`).
2. We start the post-order traversal by going to the left subtree. The recursion makes us first go to the left child `2`.
3. For `2`, we go to its left child `1`. Since `1` is a leaf node, its left and right children are `None`, and the recursion returns `0` for both, with a tilt of `0`. The sum of values for node `1` is just `1`.

```
1  Tilt of 1 = |0 - 0| = 0
2  Sum of values for node 1's subtree = 1
```

4. We go back up to `2` and then to its right child `3`, which is also a leaf node. The tilt is `0` and the sum is `3`, just like for `1`.

```
1  Tilt of 3 = |0 - 0| = 0
2  Sum of values for the node 3's subtree = 3
```

5. Now, we have both the left and right sum for the node `2`, so we can calculate its tilt. The sum for the left is `1`, and for the right is `3`.

```
1  Tilt of 2 = |1 - 3| = 2
2  Sum of values for the node 2's subtree = 2 (itself) + 1 (left) + 3 (right) = 6
```

6. Similarly, we traverse the right subtree of the root starting with node `7`, going to its left `6` and right `9` nodes, all leaf nodes, thus having their tilts as `0`. Then we calculate the tilt for node `7`.

```
1  Tilt of 7 = |6 - 9| = 3
2  Sum of values for node 7's subtree = 7 (itself) + 6 (left) + 9 (right) = 22
```

7. Finally, with both subtrees computed, we calculate the tilt for the root `4`:

```
1  Tilt of 4 = |6 - 22| = 16
2  Sum of values for the node 4's subtree = 4 (itself) + 6 (left) + 22 (right) = 32
```

8. The `sum` function adds up all the tilts as it calculates them using the nonlocal variable `ans`, which is initially set to `0`. So, we have:

```
1  ans = Tilt of 1 + Tilt of 3 + Tilt of 2 + Tilt of 6 + Tilt of 9 + Tilt of 7 + Tilt of 4
2  ans = 0 + 0 + 2 + 0 + 0 + 3 + 16
3  ans = 21
```

9. The `findTilt` function then returns `ans`, which is `21` in this case.

So the sum of all tilts in the tree is `21`. The solution provided does this in a depth-first manner, ensuring that each node is visited only once, which is quite efficient.

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 find_tilt(self, root: TreeNode) -> int:
        # Initialize the total tilt of the tree
        total_tilt = 0

        def calculate_subtree_sum(node):
            # Base case: if the node is None, return a sum of 0
            if not node:
                return 0

            # Using the nonlocal keyword to update the total_tilt variable
            nonlocal total_tilt

            # Recursively calculate the sum of values for the left subtree
            left_sum = calculate_subtree_sum(node.left)
            # Recursively calculate the sum of values for the right subtree
            right_sum = calculate_subtree_sum(node.right)

            # Update the total tilt using the absolute difference between left and right subtree sums
            total_tilt += abs(left_sum - right_sum)

            # Return the sum of values for the current subtree
            return node.val + left_sum + right_sum

        # Start the recursion from the root node
        calculate_subtree_sum(root)

        # After the recursion, total_tilt will have the tree's tilt
        return total_tilt
```

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 {
    // Variable to store the total tilt of all nodes.
    private int totalTilt;

    /**
     * Find the tilt of the binary tree.
     * The tilt of a tree node is defined as the absolute difference between the sum of all left subtree node values
     * and the sum of all right subtree node values.
     *
     * @param root The root of the binary tree.
     * @return The total tilt of the whole binary tree.
     */
    public int findTilt(TreeNode root) {
        totalTilt = 0;
        calculateSum(root);
        return totalTilt;
    }

    /**
     * Recursive helper function to calculate the sum of all nodes under the current node, including itself.
     * It also updates the total tilt during the process.
     *
     * @param node The current node from which we are calculating the sum and updating the tilt.
     * @return The sum of all node values under the current node, including itself.
     */
    private int calculateSum(TreeNode node) {
        // Base case: if the node is null, there's no value to sum or tilt to calculate.
        if (node == null) {
            return 0;
        }

        // Recursive call to calculate the sum of values in the left subtree.
        int leftSubtreeSum = calculateSum(node.left);
        // Recursive call to calculate the sum of values in the right subtree.
        int rightSubtreeSum = calculateSum(node.right);

        // Calculate the tilt at this node and add it to the total tilt.
        totalTilt += Math.abs(leftSubtreeSum - rightSubtreeSum);

        // Return the sum of values under this node, which includes its own value and the sums from both subtrees.
        return node.val + leftSubtreeSum + rightSubtreeSum;
    }
}
```

C++ Solution

```cpp
// Definition for a binary tree node.
struct TreeNode {
    int val;            // value of the node
    TreeNode *left;     // pointer to the left child node
    TreeNode *right;    // pointer to the right child node

    // Constructor to initialize a node with value and no children
    TreeNode(int x)
        : val(x), left(nullptr), right(nullptr) {}

    // Constructor to initialize a node with a given value
    TreeNode(int x)
        : val(x), left(nullptr), right(nullptr) {}

    // Constructor to initialize a node with a value and given left and right children
    TreeNode(int x, TreeNode *left, TreeNode *right)
        : val(x), left(left), right(right) {}
};

class Solution {
public:
    int totalTilt;  // To store the total tilt of the entire tree

    // Public method to find the tilt of the entire binary tree
    // The tilt of a tree node is the absolute difference between the sum of all left subtree node values and the sum of all right
    // The tilt of the whole tree is the sum of all nodes' tilts.
    int findTilt(TreeNode* root) {
        totalTilt = 0;
        computeSubtreeSum(root);  // Start the recursive sum computation
        return totalTilt;
    }

private:
    // Helper method to calculate subtree sum.
    // Recursively calculates the sum of values of all nodes in a subtree rooted at 'root',
    // while updating the total tilt of the tree.
    int computeSubtreeSum(TreeNode* root) {
        if (!root) return 0;  // Base case: if the current node is null, return 0

        // Recursively compute the sum of the left and right subtrees
        int leftSubtreeSum = computeSubtreeSum(root->left);
        int rightSubtreeSum = computeSubtreeSum(root->right);

        // Update the total tilt by the absolute difference between left and right subtree sums
        totalTilt += abs(leftSubtreeSum - rightSubtreeSum);

        // Return the sum of the current node's value and its left and right subtrees
        return root->val + leftSubtreeSum + rightSubtreeSum;
    }
};
```

Typescript Solution

```typescript
// Define a binary tree node interface.
interface TreeNode {
    val: number;        // Value of the node.
    left: TreeNode | null;  // Pointer to the left child node.
    right: TreeNode | null;  // Pointer to the right child node.
}

// Global variable to store the total tilt of the entire tree.
let totalTilt: number = 0;

// Function to find the tilt of the entire binary tree.
// The tilt of a tree node is defined as the absolute difference between the sum of all left subtree node values and the sum of all r
// The tilt of the whole tree is the sum of the tilts of all nodes.
function findTilt(root: TreeNode | null): number {
    totalTilt = 0;
    computeSubtreeSum(root);  // Start the recursive sum computation.
    return totalTilt;
}

// Helper function to calculate the sum of values in a subtree rooted at 'root'
// Recursively calculates this sum and updates the total tilt of the tree.
function computeSubtreeSum(root: TreeNode | null): number {
    if (root === null) return 0;  // Base case: if the current node is null, the sum is 0.

    // Recursively compute the sum of values in the left and right subtrees.
    let leftSubtreeSum = computeSubtreeSum(root.left);
    let rightSubtreeSum = computeSubtreeSum(root.right);

    // Calculate the tilt for the current node and add it to the total tilt of the tree.
    totalTilt += Math.abs(leftSubtreeSum - rightSubtreeSum);

    // Return the sum of the current node's value and the sums from its left and right subtrees.
    return root.val + leftSubtreeSum + rightSubtreeSum;
}
```

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 auxiliary function `sumTroot` is a recursive function that visits each node exactly once to compute the sum of values and tilt of each subtree.

Space Complexity

The space complexity of the given code is `O(h)`, where `h` is the height of the binary tree. This accounts for the recursive call stack that goes as deep as the height of the tree in the worst case (when the tree is completely unbalanced). For a balanced binary tree, the height `h` would be `log(n)`, resulting in `O(log(n))` space complexity due to the balanced nature of the call stack. However, in the worst case (a skewed tree), the space complexity can be `O(n)`.