2641. Cousins in Binary Tree II

Medium Tree Depth-First Search Breadth-First Search Hash Table Binary Tree

Leetcode Link

Problem Description

In this problem, we are given the root of a binary tree. A binary tree is a data structure where each node has at most two children, referred to as the left child and the right child. We are asked to modify the tree by replacing each node's value with the sum of the values of all its cousin nodes. Cousins are defined as nodes that are at the same level (have the same depth) but do not share the same parent. The depth of a node is measured by the number of edges from the root node to the given node.

The challenge is to perform this transformation in place, that is, without creating a new tree but rather by modifying the values of nodes in the original tree.

Intuition

The solution to this problem involves two main steps:

1. Calculate the sum of values at each depth in the tree.
2. Replace the value of each node with the sum of its cousin's values, which is the total sum at its depth minus the values of the node's children (since children are not considered cousins).

We can achieve this through two depth-first search (DFS) traversals:

- The first DFS (`dfs1`) traverses the tree to calculate the sum of values at each depth. We maintain an array `s` where `s[d]` represents the sum of values of all nodes at depth `d`. As we recurse, we pass the depth `d` and increment it when moving to the children.
- The second DFS (`dfs2`) uses the computed sums to replace each node's value with the correct sum of its cousins' values. For each node, we subtract the values of its children from the sum at its depth `s[d]` (if it has children) and assign this value to the node's children. We do not need to modify the value of the root node as it has no cousins.

The result is an updated tree where each node's value represents the sum of all its cousins' values, as required by the problem statement.

Solution Approach

The implementation of this problem can be broken down as follows:

- Firstly, we define two recursive functions `dfs1` and `dfs2` aimed at conducting depth-first search traversals. Depth-First Search (DFS) is an algorithm for traversing or searching tree or graph data structures. It starts at the root node and explores as far as possible along each branch before backtracking.
- The `dfs1` function is responsible for calculating the sum of values at each depth of the tree. It accepts two parameters: `root`, which is the current node being visited, and `d`, which represents the current depth. We utilize a list `s` where each `s[d]` corresponds to the sum of node values at depth `d`. When the function visits a node, it adds the node's value to `s[d]` and recursively calls itself for the left and right children, with the depth `d` increased by 1.
- After the first traversal with `dfs1`, we have the sum of values for each level of the tree stored in `s`. However, to find the sum of cousins, we must subtract the values of any children a node might have, as they are not its cousins.
- To carry out this subtraction and update the node values accordingly, we use the `dfs2` function. Like `dfs1`, `dfs2` also takes `root` and `d` as parameters. For each node, we calculate the sum of its children's values (if any) and subtract this sum from `s[d]` to find the sum of its cousins' values. We then recursively call `dfs2` for the left and right children, again increasing the depth `d` by 1.
- Note that we don't change the value of the root node itself, as it doesn't have any cousins. Therefore, we set `root.val` to 0 before calling `dfs2`.
- After the second traversal with `dfs2`, all nodes will contain the sum of their cousins' values.
- Lastly, we return the modified `root` of the tree.

The algorithm uses the DFS pattern to explore the nodes. It also makes use of recursion to navigate through the tree from the root to the leaves efficiently. By dividing the problem into two parts, we can achieve the solution without any need for complex data structures or additional storage beyond the tree and the array `s`.

Example Walkthrough

Let's consider the following binary tree for our example:

```
    1
   / \
  2   3
 / \   \
4   5   6
```

For simplicity, we'll refer to each node by its initial value.

Step 1: Run `dfs1` starting with the root node (node 1) at depth 0. This will calculate the sum of all node values at each depth.

- At depth 0, `s[0]` becomes 1 (value of root node).
- At depth 1, `s[1]` becomes 2 + 3 = 5 (values of node 2 and 3).
- At depth 2, `s[2]` becomes 4 + 5 + 6 = 15 (values of node 4, node 5, and node 6).

Step 2: Before running `dfs2`, set `root.val` to 0 since the root has no cousins.

Step 3: Run `dfs2` starting with node 1 (which now has a value of 0) at depth 0. This traversal will update each node with the sum of the cousins' values.

- Node 1's value remains 0.
- Node 2 is at depth 1 and its children's values sum up to 4 + 5 = 9. We calculate its new value as `s[1] - 9 = 5 - 9 = -4`.
- Node 3 is also at depth 1, but its child's value is 6. Its new value will be `s[1] - 6 = 5 - 6 = -1`.
- Node 4 is at depth 2, with no children. Its new value will be `s[2] = 15`.
- Node 5 is at depth 2, with no children. Its new value will also be `s[2] = 15`.
- Node 6 is at depth 2, with no children. Its new value will be `s[2] = 15`, just like nodes 4 and 5.

After running both traversals, the modified tree will look like:

```
    0
   / \
  -4  -1
 / \   \
15  15  15
```

Note that the sum values for the cousins are negative for nodes 2 and 3 because their children's values in the original tree exceeded the sum of values at that depth. This occurrence is based on the instructions given in the content and demonstrates that the algorithm processes the sum of node values at each depth and then properly subtracts the children's values to compute the final updated tree according to the DFS traversal.

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 replaceValueInTree(self, root: Optional[TreeNode]) -> Optional[TreeNode]:
        # Helper function to perform DFS and sum up the values at each depth.
        def dfs_sum_values(node, depth):
            if node is None:
                return

            # If we encounter a new depth, initialize sum for that depth.
            if len(depth_sums) == depth:
                depth_sums.append(0)
            # Add the current node's value to the corresponding depth's sum.
            depth_sums[depth] += node.val
            # Recursive calls for the left and right subtrees, with increased depth.
            dfs_sum_values(node.left, depth + 1)
            dfs_sum_values(node.right, depth + 1)

        # Helper function to perform DFS and replace the values of non-root nodes.
        def dfs_replace_values(node, depth):
            if node is None:
                return

            # Calculate the combined value of the current node's children.
            children_sum = (node.left.val if node.left else 0) + (node.right.val if node.right else 0)
            # Replace the values of the left child, if exists.
            if node.left:
                node.left.val = depth_sums[depth] - children_sum
            # Replace the values of the right child, if exists.
            if node.right:
                node.right.val = depth_sums[depth] - children_sum
            # Recursive calls for the left and right subtrees, with increased depth.
            dfs_replace_values(node.left, depth + 1)
            dfs_replace_values(node.right, depth + 1)

        # Initialize the list that will hold the sums of values at each depth.
        depth_sums = []
        # First DFS to calculate sums at each depth.
        dfs_sum_values(root, 0)
        # Set root's value to 0 since the question seems to be replacing non-root values.
        root.val = 0
        # Second DFS to replace values in the tree.
        dfs_replace_values(root, 1)

        return root
```

Java Solution

```java
class Solution {
    // Declare a list to store the sum of values at each level.
    private List<Integer> levelSums = new ArrayList<>();

    // Method to replace values in the tree with the calculated sums.
    public TreeNode replaceValueInTree(TreeNode root) {
        calculateLevelSums(root, 0); // Step 1: Calculate sums for all levels.
        root.val = 0;                // Set the root value to 0 as per the requirement.
        replaceValues(root, 1);      // Step 3: Start replacing the values from the first level.
        return root;                 // Return the modified tree.
    }

    // DFS helper method to calculate sum of all nodes at each depth level.
    private void calculateLevelSums(TreeNode node, int depth) {
        if (node == null) {
            return;
        }
        // If this is the first time we're visiting this depth level, add a new sum entry.
        if (levelSums.size() == depth) {
            levelSums.add(0);
        }
        // Update the current level sum with the value of the current node.
        levelSums.set(depth, levelSums.get(depth) + node.val);
        // Recurse on left and right children.
        calculateLevelSums(node.left, depth + 1);
        calculateLevelSums(node.right, depth + 1);
    }

    // DFS helper method to replace node values based on the previously calculated sums.
    private void replaceValues(TreeNode node, int depth) {
        if (node == null) {
            return;
        }
        // Compute the values to deduct (current node's children values if they aren't null).
        int leftValue = node.left == null ? 0 : node.left.val;
        int rightValue = node.right == null ? 0 : node.right.val;

        // If the left child exists, replace its value based on the sum at the current depth.
        if (node.left != null) {
            node.left.val = levelSums.get(depth) - leftValue - rightValue;
        }
        // If the right child exists, do the same for the right child.
        if (node.right != null) {
            node.right.val = levelSums.get(depth) - leftValue - rightValue;
        }

        // Recurse on left and right children to continue the process.
        replaceValues(node.left, depth + 1);
        replaceValues(node.right, depth + 1);
    }
}
```

C++ Solution

```cpp
#include <vector>
#include <functional>

// 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:
    TreeNode* replaceValueInTree(TreeNode* root) {
        // Create a vector to store the sum at each depth of the tree.
        std::vector<int> sumsByDepth;

        // First depth-first search to calculate the sum of values at each depth level.
        std::function<void(TreeNode*, int)> dfsSum = [&](TreeNode* node, int depth) {
            if (!node) {
                return;
            }
            // If the current depth is greater than the size of the vector, expand it.
            if (sumsByDepth.size() == depth) {
                sumsByDepth.push_back(0);
            }
            // Add the current node's value to the corresponding depth's sum.
            sumsByDepth[depth] += node->val;
            // Explore left and right subtrees.
            dfsSum(node->left, depth + 1);
            dfsSum(node->right, depth + 1);
        };

        // Second depth-first search to update the values in the tree.
        std::function<void(TreeNode*, int)> dfsUpdate = [&](TreeNode* node, int depth) {
            if (!node) {
                return;
            }
            // Store the values for the left and right children if they exist.
            int leftValue = node->left ? node->left->val : 0;
            int rightValue = node->right ? node->right->val : 0;

            // Update the left child's value if it exists.
            if (node->left) {
                node->left->val = sumsByDepth[depth] - leftValue - rightValue;
            }
            // Update the right child's value if it exists.
            if (node->right) {
                node->right->val = sumsByDepth[depth] - leftValue - rightValue;
            }
            // Explore left and right subtrees.
            dfsUpdate(node->left, depth + 1);
            dfsUpdate(node->right, depth + 1);
        };

        // Start the first DFS from the root at depth 0.
        dfsSum(root, 0);
        // Update the root's value to 0 according to the problem requirement.
        root->val = 0;
        // Start the second DFS from the root's children at depth 1.
        dfsUpdate(root, 1);

        // Return the modified tree root.
        return root;
    }
};
```

Typescript Solution

```typescript
// // A tree node may contain a value and pointers to left and right child nodes.
interface TreeNode {
    val: number;
    left: TreeNode | null;
    right: TreeNode | null;
}

// // This function performs a depth-first traversal to collect the sum of values at each depth.
function collectSumsByDepth(root: TreeNode | null, depth: number, sumsByDepth: number[]): void {
    if (!root) return;

    // Ensure the array is large enough to hold the sum for the current depth.
    if (sumsByDepth.length == depth) {
        sumsByDepth.push(0);
    }

    // Add the current node's value to the sum corresponding to its depth.
    sumsByDepth[depth] += root.val;

    // Traverse the left and right children.
    collectSumsByDepth(root.left, depth + 1, sumsByDepth);
    collectSumsByDepth(root.right, depth + 1, sumsByDepth);
}

// // This function replaces node values with the collected sum at their respective depth.
function replaceNodeValuesByDepth(root: TreeNode | null, depth: number, sumsByDepth: number[]): void {
    if (!root) return;

    // Get the values of the children nodes.
    const childSum = (root.left?.val ?? 0) + (root.right?.val ?? 0);

    // Replace the value of the left child if it exists.
    if (root.left) {
        root.left.val = sumsByDepth[depth] - childSum;
    }

    // Replace the value of the right child if it exists.
    if (root.right) {
        root.right.val = sumsByDepth[depth] - childSum;
    }

    // Continue the traversal for left and right children.
    replaceNodeValuesByDepth(root.left, depth + 1, sumsByDepth);
    replaceNodeValuesByDepth(root.right, depth + 1, sumsByDepth);
}

// // The main function that initiates the process of replacing node values by depth sum.
function replaceValueInTree(root: TreeNode | null): TreeNode | null {
    // An array to store the sum of values at each depth.
    const sumsByDepth: number[] = [];

    // First depth-first search to collect sums at each depth.
    collectSumsByDepth(root, 0, sumsByDepth);

    // Set the root value to 0 as per instructions.
    if (root) {
        root.val = 0;
    }

    // Second depth-first search to replace the node values by the collected sums at each depth.
    replaceNodeValuesByDepth(root, 1, sumsByDepth);

    return root;
}
```

Time and Space Complexity

The time complexity for the `dfs1` function is $O(n)$ where n is the number of nodes in the binary tree since it visits each node exactly once to calculate the sum at each depth. Similarly, `dfs2` also has a time complexity of $O(n)$ as it again visits each node exactly once to replace the values. Hence the combined time complexity of the entire `replaceValueInTree` method is also $O(n)$.

For space complexity, the code uses $O(h)$ extra space for recursion where h is the height of the binary tree, which in the worst case of a skewed tree can be $O(n)$. Additionally, the list `s` can grow up to $O(h)$ for storing the sum at each depth, again leading to $O(n)$ in the worst case. Therefore, the overall space complexity is $O(n)$ as well.