617. Merge Two Binary Trees

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

Problem Description

You are asked to merge two binary trees in such a way that if two nodes from the two trees occupy the same position, their values are summed together to create a new node in the resulting binary tree. If at any position, only one of the trees has a node, the resulting tree should have a node with that same value. To merge the trees, you must start from their root nodes and continue merging the subtrees recursively following the same rule.

Intuition

To merge the two given binary trees, the solution employs a recursive approach. Each recursive call is responsible for handling the merging of two nodes - one from each of the trees. The first step is to determine what to do if one or both of the nodes being merged are null. If one of the nodes is null, the other node (which is not null) can be used directly in the merged tree since there's nothing to merge. If both nodes are null, the merged node is also null.

When both nodes are not null, a new node is created with a value that is the sum of the two nodes' values. Following that, the recursion is applied to both the left and the right children of the nodes being merged to handle the full depth of the trees. Thus, the process creates a new tree that is the result of merging the input trees by summing the overlapping nodes and copying the non-overlapping nodes.

Merging goes as deep as the deepest tree, and each level of recursion handles only one level of the tree. This way, the recursion stacks create the entire tree level by level.

Solution Approach

To implement the merging of two binary trees as described in the problem statement, the solution uses a recursive depth-first search algorithm.

Here's a step-by-step breakdown of the algorithm:

1. The base case of the recursive function handles the case when either `root1` or `root2` is `None`. If one of them is `None`, it returns the other node, because there is nothing to merge.

```
if root1 is None:
    return root2
if root2 is None:
    return root1
```

2. If both nodes are not `None`, it creates a new `TreeNode` with the sum of both nodes' values. This will be the current node in the merged tree.

```
node = TreeNode(root1.val + root2.val)
```

3. To merge the left subtree of both trees, the function makes a recursive call with the left child of both `root1` and `root2` and assigns the return value to the `left` attribute of the new node.

```
node.left = self.mergeTrees(root1.left, root2.left)
```

4. Similarly, to merge the right subtree, another recursive call is made with the right child of both nodes. The return value is assigned to the `right` attribute of the new node.

```
node.right = self.mergeTrees(root1.right, root2.right)
```

5. After recursively merging both the left and right subtrees, the function returns the `node`, which now represents the root of the merged subtree for the current recursive call.

```
return node
```

This depth-first recursive approach systematically merges all nodes at corresponding positions in each level of both input trees. By continuously dividing the problem into smaller subproblems (merging the left and right children, respectively), the algorithm efficiently constructs the merged binary tree in a bottom-up manner, returning the merged root node as the result.

The space complexity of this algorithm is O(n), where n is the smaller of the heights of the two input trees because it is the maximum depth of the recursion stack. The time complexity is also O(n), where n in this case is the total number of nodes that will be present in the new merged tree, as each node from both trees is visited exactly once.

Example Walkthrough

Let's consider two simple binary trees to demonstrate the solution approach.

Tree 1:

```
  1
 / \
3   2
```

Tree 2:

```
  2
 / \
1   3
 \
  4
```

We want to merge Tree 1 and Tree 2 into a single tree. Following the step-by-step algorithm:

1. Start at the root of both trees. Since none of them are `None`, we create a new root node for the merged tree with the sum of the values of Tree 1's root (1) and Tree 2's root (2), giving us a root node with value 3.
2. Next, we merge the left children of both trees. Tree 1's left child has a value of 3 and Tree 2's left child has a value of 1. Neither is `None`, so we create a new node with the sum of their values (3 + 1 = 4) which becomes the left child of the merged tree's root.
3. Tree 2's left child also has a non-null right child with a value of 4. Tree 1 doesn't have a corresponding node here, so the merged tree's left child will inherit this right child directly.
4. Merging the right children of the root nodes from Tree 1 and Tree 2 results in a new node with the sum of their values (2 + 3 = 5), which becomes the right child of the merged tree's root.

The merged binary tree now looks like this:

```
    3
   / \
  4   5
 / \
4   4
```

Here is a step-by-step mapping of the decisions made to reach this merged tree:

- Root node: Tree 1 has 1, Tree 2 has 2. The merged node has 3 (1 + 2).
- Left child of the root node: Tree 1 has 3, Tree 2 has 1. The merged node has 4 (3 + 1).
- Right child of root node: Tree 1 has 2, Tree 2 has 3. The merged node has 5 (2 + 3).
- Right child of the left node: Tree 1 does not have a corresponding node, only Tree 2 has 4. The merged node has 4.

This walkthrough illustrates how the recursive algorithm systematically combines the node values of overlapping positions while retaining the node values of non-overlapping positions from the given trees to construct a merged tree.

Solution Implementation

Python

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

class Solution:
    def mergeTrees(self, root1: TreeNode, root2: TreeNode) -> TreeNode:
        # If the first root is None, return the second root as the result of the merge.
        if root1 is None:
            return root2

        # If the second root is None, return the first root as the result of the merge.
        if root2 is None:
            return root1

        # If both roots are valid, create a new TreeNode with the sum of the values of root1 and root2.
        merged_node = TreeNode(root1.value + root2.value)

        # Recursively merge the left children of both trees and assign to the left of the merged node.
        merged_node.left = self.mergeTrees(root1.left, root2.left)

        # Recursively merge the right children of both trees and assign to the right of the merged node.
        merged_node.right = self.mergeTrees(root1.right, root2.right)

        # Return the merged tree root.
        return merged_node
```

Java

```java
/**
 * Definition for a binary tree node.
 */
class TreeNode {
    int value; // Variable to store the value of the node
    TreeNode left; // Reference to the left child node
    TreeNode right; // Reference to the right child node

    // Constructor to create a leaf node with a value
    TreeNode(int value) {
        this.value = value;
    }

    // Constructor to create a node with a value, left child, and right child
    TreeNode(int value, TreeNode left, TreeNode right) {
        this.value = value;
        this.left = left;
        this.right = right;
    }
}

public class Solution {
    /**
     * Merges two binary trees by adding values of overlapping nodes together.
     *
     * @param root1 The root node of the first binary tree.
     * @param root2 The root node of the second binary tree.
     * @return A new binary tree with nodes merged from root1 and root2.
     */
    public TreeNode mergeTrees(TreeNode root1, TreeNode root2) {
        // If the first root is null, return the second root as the result of merge
        if (root1 == null) {
            return root2;
        }

        // If the second root is null, return the first root as the result of merge
        if (root2 == null) {
            return root1;
        }

        // Create a new TreeNode with a value equal to the sum of both node's values
        TreeNode mergedNode = new TreeNode(root1.value + root2.value);

        // Recursively merge the left children of both roots and assign to the left of merged node
        mergedNode.left = mergeTrees(root1.left, root2.left);

        // Recursively merge the right children of both roots and assign to the right of merged node
        mergedNode.right = mergeTrees(root1.right, root2.right);

        // Return the merged tree node
        return mergedNode;
    }
}
```

C++

```cpp
#include <cstddef> //For NULL

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

    // Constructor to create a new node with no children
    TreeNode() : val(0), left(nullptr), right(nullptr) {}

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

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

class Solution {
public:
    // Merges two binary trees into one binary tree.
    // If nodes overlap, sum the values and create a new node with the sum.
    // If nodes do not overlap, clone the non-null node.
    TreeNode* mergeTrees(TreeNode* firstRoot, TreeNode* secondRoot) {
        // If the first tree is empty, return the second tree (no merge required).
        if (firstRoot == nullptr) return secondRoot;

        // If the second tree is empty, return the first tree (no merge required).
        if (!secondRoot) return firstRoot;

        // Create a new node with the sum of the values of overlapping nodes.
        TreeNode* mergedNode = new TreeNode(firstRoot->val + secondRoot->val);

        // Recursively merge the left children.
        mergedNode->left = mergeTrees(firstRoot->left, secondRoot->left);

        // Recursively merge the right children.
        mergedNode->right = mergeTrees(firstRoot->right, secondRoot->right);

        // Return the root of the newly merged tree.
        return mergedNode;
    }
};
```

TypeScript

```typescript
type TreeNodeOrNull = TreeNode | null;

// Definition for a binary tree node.
class TreeNode {
  val: number;
  left: TreeNodeOrNull;
  right: TreeNodeOrNull;

  constructor(val?: number, left?: TreeNodeOrNull = null, right?: TreeNodeOrNull = null) {
    this.val = val === undefined ? 0 : val;
    this.left = left;
    this.right = right;
  }
}

/**
 * Merges two binary trees into a new binary tree.
 * If two nodes overlap, the resulting node's value
 * is a sum of the overlapping nodes' values.
 * @param {TreeNodeOrNull} tree1 - The first binary tree.
 * @param {TreeNodeOrNull} tree2 - The second binary tree.
 * @returns {TreeNodeOrNull} - The merged binary tree.
 */
function mergeTrees(tree1: TreeNodeOrNull, tree2: TreeNodeOrNull): TreeNodeOrNull {
  // If both trees are null, return null.
  if (tree1 === null && tree2 === null) return null;
  // If one tree is null, return the other tree.
  if (tree1 === null) return tree2;
  if (tree2 === null) return tree1;

  // Merge the left and right subtrees recursively.
  const mergedLeft = mergeTrees(tree1.left, tree2.left);
  const mergedRight = mergeTrees(tree1.right, tree2.right);

  // Create a new tree node with the sum of tree1 and tree2 node values,
  // and the merged left and right subtrees as its children.
  const mergedNode = new TreeNode(tree1.val + tree2.val, mergedLeft, mergedRight);

  return mergedNode;
}
```

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

class Solution:
    def mergeTrees(self, root1: TreeNode, root2: TreeNode) -> TreeNode:
        # If the first root is None, return the second root as the result of the merge.
        if root1 is None:
            return root2

        # If the second root is None, return the first root as the result of the merge.
        if root2 is None:
            return root1

        # If both roots are valid, create a new TreeNode with the sum of the values of root1 and root2.
        merged_node = TreeNode(root1.value + root2.value)

        # Recursively merge the left children of both trees and assign to the left of the merged node.
        merged_node.left = self.mergeTrees(root1.left, root2.left)

        # Recursively merge the right children of both trees and assign to the right of the merged node.
        merged_node.right = self.mergeTrees(root1.right, root2.right)

        # Return the merged tree root node.
        return merged_node
```

Time and Space Complexity

The provided code defines a method for merging two binary trees. The function `mergeTrees` takes two binary trees (root nodes `root1` and `root2`) as input and returns a new binary tree that represents the sum of both input trees.

Time Complexity:

The time complexity of the function is $O(n)$, where n is the number of nodes present in the smaller of the two trees. This is because the function visits each node from both the trees exactly once (in the worst case, when both trees are of the same size). If the trees have a different number of nodes, the complexity is governed by the tree with fewer nodes since this is where the recursion will stop for each path.

Space Complexity:

The space complexity is also $O(n)$, considering the worst-case scenario where the tree is completely unbalanced and resembles a linked list. In this case, n is again the number of nodes in the smaller tree. This is because the function will have a number of recursive calls on the stack proportional to the height of the tree, which, in the worst case, could be the number of nodes in the tree if it's completely unbalanced. For a balanced tree, the space complexity would be $O(\log(n))$.

However, if we consider the space for the output tree, the space complexity would become $O(max(m, n))$, where m and n are the number of nodes in `root1` and `root2`, respectively, since a new tree of size at most $m+n$ is created. But often, the space used to create the output is not considered when analyzing the algorithm itself, since that space is required for the output data structure.