538. Convert BST to Greater Tree

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

Problem Description

Given a Binary Search Tree (BST), the task is to transform it into a Greater Tree. In this new version of the tree, each node's value should be updated to be the sum of its original value plus the sum of all values of the nodes that have greater values in the BST. To clarify what constitutes a Binary Search Tree:

- The left subtree of a node only contains nodes with keys less than the node's key.
- The right subtree of a node only contains nodes with keys greater than the node's key.
- Each left and right subtree must also be a binary search tree by itself.

Intuition

The intuition behind the solution comes from two properties of BSTs: in-order traversal and the properties of tree structure. By performing a reverse in-order traversal (visiting the right subtree first, then the current node, and then the left subtree), we can visit the nodes in decreasing order. As we traverse, we maintain a running sum of all the nodes visited so far. Since we're visiting nodes in decreasing order, this running sum is the sum of all the nodes greater than the current node.

Here's the step-by-step thinking:

- Since a BST's in-order traversal yields sorted order, a reverse in-order traversal yields a sequence of nodes in non-increasing order.
- Begin the reverse in-order traversal from the root with a running sum set to 0. This sum will be used to store the sum of all the nodes which are greater than the currently visited node.
- For each node visited, add its value to the running sum. Then, update the current node's value to this running sum.
- Proceed to the left subtree (which contains nodes with smaller values) and repeat the process.
- By doing this, we've ensured that each node now contains the sum of its own value and all the nodes with greater value in the BST.

The solution provided uses a form of Morris Traversal (a tree traversal algorithm that doesn't use recursion or a stack) to achieve this without the extra space. Essentially, this algorithm makes use of the tree's leaf nodes' right children to create temporary links back to the node's ancestor (thereby avoiding the use of additional memory for traversal). Once done with the ancestor, these links are severed to revert the tree back to its original form.

Solution Approach

The provided solution uses the Morris Traversal technique, which is a clever way to traverse a binary tree without recursion and without a stack. This method takes advantages of the threaded binary trees concept, where a right NULL pointer is made to point to the inorder successor (if it exists).

Here is the breakdown of the Morris Traversal algorithm in the context of the problem:

- Start from the `root` node of the BST and initialize a variable `s` to 0, which will keep track of the running sum of node values.
- Iterate as long as the current node is not `None`.
- Within the loop, check if the current node has a right child. If it does not have a right child, this means we are at the rightmost node (which is the largest). We update the running sum `s` by adding the current node's value and then update the current node's value to the new sum `s`. We then move to the left child of the current node.
- If the current node has a right child, we will find the leftmost node of the right subtree (which is the inorder successor of the current node). We traverse to the right child of the current node and then keep moving to the left child until we find the leftmost node or until the left child is the current node itself.
- If the leftmost node (inorder successor) does not have its left child set yet, we set it to point back to the current node and move to the right child of the current node to continue the traversal.
- If the leftmost node already points back to the current node (creating a cycle), it means we have already visited this subtree. We reset the left child to `None` (breaking the cycle) which restores the tree structure. We then update the sum `s` and the current node's value as described above, and move to the left child of the current node to continue the traversal.
- Finally, when all nodes are visited in this reverse in-order manner, we end up with a tree where each node's value is equal to the original value plus the sum of all node values greater than itself. The original root of the BST is returned as the root of the Greater Tree.

By using Morris Traversal, the solution avoids additional space complexities that would come with using a stack for in-order traversal or recursion. Since the Morris Traversal is a constant space solution (O(1)), it proves to be an efficient way to resolve the problem at hand.

Example Walkthrough

To illustrate the Morris Traversal solution approach, let's consider a simple BST and transform it into a Greater Tree according to the method described.

Let's take the following BST:

```
        5
       / \
      2   13
```

- We start at the root (5). Since it has a right child (13), we'll find the inorder successor of 5.
- The inorder successor of 5 is 13 since 13 has no left child. We create a temporary link from the rightmost node in 13's left subtree (which is 13 itself, as it doesn't have a left subtree) back to 5.
- We then visit the node 13, update the sum `s = 0 + 13 = 13`, and update 13's value with this sum. Now our tree looks as follows:

```
        5
       / \
      2   13
          /
        (5)
```

(The parentheses indicate a temporary link)

- We go back to 5 via the temporary link and update it. The running sum `s = 13 + 5 = 18`, and we update 5's value with this sum:

```
        18
       /  \
      2    13
```

- We then remove the temporary link from 13 to 5 and proceed to the left subtree of 5 (node 2).
- Node 2 has no right child, so we update it directly. Running sum `s = 18 + 2 = 20` and we update 2's value to 20:

```
        18
       /  \
     20    13
```

- With no more nodes left to visit, the in-place transformation is complete and the original BST is now a Greater Tree:

```
        18
       /  \
     20    13
```

In this Greater Tree, each node's new value is the sum of its original value plus all values greater than it in the BST. The Morris Traversal allowed us to perform this transformation with O(1) additional space, respecting the constraints of the problem.

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 convertBST(self, root: TreeNode) -> TreeNode:
        # This variable will keep a running sum of node values as we traverse the tree.
        running_sum = 0
        # `current` will point to the current node being processed.
        current = root

        # Start Morris Traversal to convert the BST into a Greater Tree
        while current:
            # If there is no right child, process the current node and move to the left child.
            if current.right is None:
                running_sum += current.val
                current.val = running_sum
                current = current.left
            else:
                # Find the inorder predecessor of the current node
                predecessor = current.right
                while predecessor.left and predecessor.left != current:
                    predecessor = predecessor.left

                # There is no left child to the predecessor, make current the left child of predecessor.
                if predecessor.left is None:
                    predecessor.left = current
                    current = current.right
                else:
                    # Left child of predecessor exists which means we have processed right subtree.
                    # Update the current node with the running sum.
                    running_sum += current.val
                    current.val = running_sum
                    # Break the link to restore the tree structure.
                    predecessor.left = None
                    current = current.left

        # Return the modified tree.
        return root

```

Java Solution

```java
class Solution {
    // Convert the given BST to a Greater Tree where each key is changed to the original key plus
    // the sum of all keys greater than the original key in the BST.
    public TreeNode convertBST(TreeNode root) {
        // Initialize the sum which will hold the total of all nodes visited so far.
        int sum = 0;
        TreeNode currentNode = root;

        // Iterate through the tree using a modified in-order traversal.
        while (root != null) {
            // If there is no right child, visit this node and traverse to its left child.
            if (root.right == null) {
                sum += root.val;
                root.val = sum; // Update the value of the root with the sum.
                root = root.left;
            } else {
                // Find the in-order predecessor of the root.
                TreeNode predecessor = root.right;
                while (predecessor.left != null && predecessor.left != root) {
                    predecessor = predecessor.left;
                }

                // If the left child of the predecessor is null, make the current root
                // the left child of the predecessor and move to the right child of the root.
                if (predecessor.left == null) {
                    predecessor.left = root;
                    root = root.right;
                } else {
                    // If the left child of the predecessor is the current root, update the
                    // root's value with the sum and restore the original tree structure.
                    sum += root.val;
                    root.val = sum; // Update the value of the root with the sum.
                    predecessor.left = null; // Restore the tree structure.
                    root = root.left;
                }
            }
        }

        // Return the reference to the node which is now the root of the modified tree.
        return currentNode;
    }
}

```

C++ Solution

```cpp
#include <iostream>

struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode(int x) : val(x), left(NULL), right(NULL) {}
};

class Solution {
public:
    TreeNode* convertBST(TreeNode* root) {
        TreeNode* currentNode = root;
        int sum = 0; // This will keep track of the sum of all nodes processed so far

        // Iterate through the tree using Morris Traversal
        while (currentNode) {
            // If there is no right child, visit this node and go to its left child
            if (!currentNode->right) {
                sum += currentNode->val;
                currentNode->val = sum; // Update the value of currentNode with the cumulative sum
                currentNode = currentNode->left; // Move to the left child
            } else {
                // If there is a right child, find the leftmost child of the right subtree
                TreeNode* rightSubtree = currentNode->right;
                // Find the in-order predecessor of currentNode
                while (rightSubtree->left && rightSubtree->left != currentNode) {
                    rightSubtree = rightSubtree->left;
                }
                // Establish a temporary link from the in-order predecessor back to the currentNode
                if (!rightSubtree->left) {
                    rightSubtree->left = currentNode;
                    currentNode = currentNode->right; // Move to the right child
                } else {
                    // We have a temporary link which indicates that we've visited the left subtree of currentNode
                    sum += currentNode->val;
                    currentNode->val = sum; // Update the currentNode with the sum
                    rightSubtree->left = nullptr; // Remove the temporary link
                    currentNode = currentNode->left; // Move to the left child
                }
            }
        }
        return root; // Return the modified tree
    }
};

```

Typescript Solution

```typescript
// Define a TreeNode class to represent tree nodes
class TreeNode {
    val: number;
    left: TreeNode | null;
    right: TreeNode | null;

    constructor(x: number) {
        this.val = x;
        this.left = null;
        this.right = null;
    }
}

// Holds the current sum of all nodes processed so far
let sum: number = 0;

// Function to convert a binary search tree (BST) to a greater tree where every key of the original BST is changed to the original
function convertBST(root: TreeNode | null): TreeNode | null {
    let currentNode: TreeNode | null = root;

    // Iterate through the tree using Morris Traversal
    while (currentNode !== null) {
        // If there is no right child, visit this node and go to its left child
        if (currentNode.right === null) {
            sum += currentNode.val;
            currentNode.val = sum; // Update the value of the current node with the cumulative sum
            currentNode = currentNode.left; // Move to the left child
        } else {
            // If there is a right child, find the leftmost child of the right subtree
            let rightSubtree: TreeNode = currentNode.right;
            // Find the in-order predecessor of the current node
            while (rightSubtree.left !== null && rightSubtree.left !== currentNode) {
                rightSubtree = rightSubtree.left;
            }
            // Establish a temporary link from the in-order predecessor back to the current node
            if (rightSubtree.left === null) {
                rightSubtree.left = currentNode;
                currentNode = currentNode.right; // Move to the right child
            } else {
                // We have a temporary link which indicates that we've visited the left subtree of the current node
                sum += currentNode.val;
                currentNode.val = sum; // Update the current node with the sum
                rightSubtree.left = null; // Remove the temporary link
                currentNode = currentNode.left; // Move to the left child
            }
        }
    }
    return root; // Return the modified tree
}

// Example usage in TypeScript:
// let tree: TreeNode = new TreeNode(2);
// tree.left = new TreeNode(1);
// tree.right = new TreeNode(3);
// convertBST(tree);

```

Time and Space Complexity

The given code is a Morris traversal algorithm variant, which converts a Binary Search Tree (BST) into a Greater Tree such that the value of each node is replaced by the sum of all keys greater than the node key in BST.

Time Complexity

The time complexity of the algorithm is `O(n)`, where `n` is the number of nodes in the BST. This is because each node is visited at most twice. Once while navigating rightward to establish the threaded links, and once while revisiting nodes to accumulate the sum and remove the threads.

Space Complexity

The space complexity of the algorithm is `O(1)` excluding the input and output space as the algorithm uses a constant amount of space for pointers and variables (`s`, `node`, `next`). It does not make any additional allocations and hence does not depend on the number of nodes in the tree - achieving in-place conversion.