# 1038. Binary Search Tree to Greater Sum Tree

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

## Problem Description

The given problem presents a Binary Search Tree (BST) and requires transforming it into a "Greater Tree." The transformation should be such that each node's value is updated to the sum of its original value and the values of all nodes with greater keys in the BST.

In a BST, the properties are as follows:

- The left subtree of a node has nodes with keys less than the node's key.
- The right subtree has nodes with keys greater than the node's key.
- Both left and right subtrees adhere to the BST rules themselves.

The challenge lies in updating each node with the sum of values greater than itself while maintaining the inherent structure of the BST.

## Intuition

The solution approach leverages the properties of the BST, specifically the in-order traversal property where visiting nodes in this order will access the nodes' values in a sorted ascending sequence. However, to obtain a sum of the nodes greater than the current node, we need to process the nodes in the reverse of in-order traversal, which means we should begin from the rightmost node (the highest value) and traverse to the leftmost node (the lowest value).

We'll maintain a running sum `s` that accumulates the values of all nodes visited so far during our reverse in-order traversal. For each node `n`, we will:

- Recursively visit the right subtree to add all greater values to `s`.
- Update `n`'s value by adding `s` to `n`'s original value.
- Add `n`'s updated value (which is inclusive of its original value) to `s` before moving to the left subtree.

This process ensures that each node's value is replaced by the sum of its original value and all greater values in the BST. Our solution does this iteratively with the help of a Morris traversal-like algorithm that reduces the space complexity to O(1) by temporarily modifying the tree structure during traversal and then restoring it before moving to the left subtree.

## Solution Approach

The solution provided utilizes an iterative approach with a Morris traversal pattern, which aims to traverse the tree without additional space for recursion or a stack. Morris traversal takes advantage of the thread, a temporary link from a node to its predecessor, to iterate through the tree. Here's the breakdown of the approach:

1. Initialize a variable `s` to store the running sum and a variable `node` to keep the reference of the original root.
2. Start iterating from the `root` of the BST. Continue until the `root` is not null, as this indicates the traversal is complete.
3. For each node, there are two cases to consider:
   - If there is no right child, add the node's value to `s`. Then, update the node's value with `s` and move to the left child.
   - If there is a right child, find the leftmost child of this right subtree (`next`), which will act as the current node's predecessor.
4. If the leftmost child (`next`) doesn't have a left child (indicating that we haven't processed this subtree yet), make the current node its left child (creating a thread) and move to the right child of the current node, deferring its update until after the greater values have been incorporated into `s`.
5. If the leftmost child (`next`) already has a left child (meaning the current node has been threaded and it's time to update it), remove the thread, add the current node's value to `s`, update the current node's value with `s`, and move to the left child.
6. The loop continues until all nodes have been visited in reverse in-order, which updates all nodes with the sum of all greater node values.

This Morris traversal-based algorithm effectively improves space complexity to O(1) as it doesn't use any auxiliary data structure. The time complexity remains O(n), where 'n' is the number of nodes in the BST since each node is visited at most twice.

### Example Walkthrough

Let's illustrate the solution with a simple BST example:

Consider a BST with the following structure:

```
    4
   / \
  1   6
```

We want to transform it into a "Greater Tree" using the described Morris Traversal approach.

1. We start with the `root` node which has the value 4. We initialize `s` to 0.
2. Since node 4 has a right child (6), we look for the leftmost node in node 4's right subtree. Node 6 has no left child, so this step is skipped.
3. Since node 6 has no right child, we process it by adding its value to `s`. Now `s = 0 + 6 = 6` and update node 6 to the new value `s`. The BST now looks like this:

```
    4
   / \
  1   6
```

Node 6 has been transformed into a "Greater Node" containing the sum of values greater than itself (which in this simple case is just its own value because it's the highest).

4. Returning to node 4, we now should add its value to `s` (`s = 6 + 4 = 10`) and update it to the new value `s`. The tree structure at this moment is:

```
    10
   / \
  1   6
```

Node 4 is now a "Greater Node" having the sum of all nodes greater than itself.

5. Now we consider the left child of node 4, which is node 1. Since node 1 does not have a right child, we add its value to `s` (`s = 10 + 1 = 11`) and update its value:

```
    10
   / \
  11  6
```

Node 1 is now a "Greater Node," which includes the sum of all nodes greater than it.

6. Since node 1 is the leftmost node and it has no left child, our traversal and the transformation are complete for this simple tree.

The final "Greater Tree" is:

```
    10
   / \
  11  6
```

Followed by the Morris Traversal approach discussed, each node's value has been updated with the sum of all greater node values while maintaining the BST structure.

## 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 bstToGst(self, root: TreeNode) -> TreeNode:
        total_sum = 0  # This will store the running sum of nodes

        # Start with the root node
        current_node = root

        # Traverse the tree
        while current_node:

            # If there is no right child, update the current node's value with total_sum
            # and move to the left child
            if current_node.right is None:
                total_sum += current_node.val
                current_node.val = total_sum
                current_node = current_node.left
            else:
                # If the right child is present, find the leftmost child in the right subtree
                predecessor = current_node.right
                while predecessor.left and predecessor.left != current_node:
                    predecessor = predecessor.left

                # Establish a temporary link between current_node and its predecessor
                if predecessor.left is None:
                    predecessor.left = current_node
                    current_node = current_node.right
                else:
                    # When leftmost child is found and a cycle is detected (temporary link exists),
                    # revert the changed tree structure and update the current node
                    total_sum += current_node.val
                    current_node.val = total_sum
                    predecessor.left = None
                    current_node = current_node.left

        # Return the modified tree
        return root

```

## Java Solution

```java
class Solution {
    public TreeNode bstToGst(TreeNode root) {
        int sum = 0; // This variable keeps track of the accumulated sum
        TreeNode currentNode = root; // Save the original root to return the modified tree later

        // Iteratively traverse the tree using the reverse in-order traversal
        // (right -> node -> left) because this order visits nodes from the largest to smallest
        while (root != null) {
            // If there is no right child, update the current node's value with the sum and go left
            if (root.right == null) {
                sum += root.val; // Accumulate the node's value into sum
                root.val = sum; // Update the node's value with the accumulated sum
                root = root.left; // Move to the left child
            } else {
                // Find the inorder successor, the smallest node in the right subtree
                TreeNode inorderSuccessor = root.right;
                // Keep going left on the successor until we reach the bottom left most node
                // that is not the current root
                while (inorderSuccessor.left != null && inorderSuccessor.left != root) {
                    inorderSuccessor = inorderSuccessor.left;
                }

                // When we find the leftmost child of the inorder successor
                if (inorderSuccessor.left == null) {
                    // Set a temporary link back to the current root node and move to the right child
                    inorderSuccessor.left = root;
                    root = root.right;
                } else {
                    // The temporary link already exists, and we've visited the right subtree
                    sum += root.val; // Update the sum with the current value
                    root.val = sum; // Update current root's value with accumulated sum
                    inorderSuccessor.left = null; // Remove the temporary link
                    root = root.left; // Move to the left child
                }
            }
        }
        // Return the modified tree starting from the original root node
        return currentNode;
    }
}

```

## C++ Solution

```cpp
/**
 * Definition for a binary tree node.
 */
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    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* bstToGst(TreeNode* root) {
        int sum = 0; // Initialize sum to keep track of the accumulated values.
        TreeNode* node = root; // Keep track of the original root node

        // Traverse the tree.
        while (root != nullptr) {
            // If there is no right child, process current node and move to left child.
            if (root->right == nullptr) {
                sum += root->val;
                root->val = sum; // Update the value to be the greater sum.
                root = root->left;
            } else {
                // Find the in-order predecessor of the current node.
                TreeNode* predecessor = root->right;
                while (predecessor->left != nullptr && predecessor->left != root) {
                    predecessor = predecessor->left;
                }

                // If the left child of the predecessor is null, set it to the current node.
                if (predecessor->left == nullptr) {
                    predecessor->left = root;
                    root = root->right; // Move to the right child.
                } else {
                    // If the left child of the predecessor is the current node, process current node.
                    sum += root->val;
                    root->val = sum; // Update the node's value to be the greater sum.

                    // Restore the tree structure by removing the temporary link.
                    predecessor->left = nullptr;
                    root = root->left; // Move to the left child.
                }
            }
        }
        // Return the modified tree which now represents the Greater Sum Tree.
        return node;
    }
};

```

## Typescript Solution

```typescript
// Function to transform a Binary Search Tree into a Greater Sum Tree
function bstToGst(root: TreeNode | null): TreeNode | null {
    let current = root;
    let totalSum = 0;

    // Loop over the tree using Morris Traversal approach
    while (current != null) {
        let rightNode = current.right;

        // Case where there is no right child
        if (rightNode == null) {
            totalSum += current.val;       // Update the total sum with current value
            current.val = totalSum;        // Modify the current node's value to total sum
            current = current.left;        // Move to the left subtree
        } else {
            // Find the leftmost node in the current's right subtree
            let leftMost = rightNode;
            while (leftMost.left != null && leftMost.left != current) {
                leftMost = leftMost.left;
            }

            // First time visiting this right subtree, make a thread back to current
            if (leftMost.left == null) {
                leftMost.left = current;
                current = rightNode;
            } else {  // Second time visiting - the thread is already there
                leftMost.left = null;     // Remove the thread
                totalSum += current.val;  // Update the total sum with current value
                current.val = totalSum;   // Modify the current node's value to the total sum
                current = current.left;   // Move to the left subtree
            }
        }
    }

    // Return the modified tree root
    return root;
}

```

## Time and Space Complexity

The provided code implements a variation of the Morris traversal algorithm to convert a Binary Search Tree (BST) to a Greater Sum Tree (GST), where every key of the original BST is changed to the original key plus the sum of all keys greater than the original key in BST.

### Time Complexity

The time complexity of the code is `O(n)`, where `n` is the number of nodes in the tree. This is because each node is visited at most twice—once when the right child is connected to the current node during the transformation to threaded trees and once when it is reverted. There is no recursion stack or separate data structure which keeps track of the visited nodes. The while loop and nested while loop both ensure that each node is processed.

### Space Complexity

The space complexity of the code is `O(1)` if we do not consider the space required for the output structure - it modifies the tree nodes in place with a constant number of pointers. There is no use of recursion, nor is there any allocation of proportional size to the number of nodes. However, if the function call stack is taken into account then that will not increase our space complexity because the recursion stack is not being used here. The operation is done by manipulating the right pointers of the original tree.