669. Trim a Binary Search Tree

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

Problem Description

The problem presents a binary search tree (BST) and requires one to trim the tree such that all the nodes contain values between two boundaries: `low` and `high` (inclusive). After trimming, the resulting tree should still be a valid BST, and the relative structure of the remaining nodes should be the same. This means that if a node is a descendant of another in the original tree, it should still be a descendant in the trimmed tree. Importantly, the values that remain in the tree should fall within the specified range, which may also result in a new root being chosen if the original root does not meet the criteria.

Intuition

To arrive at the solution, we need to leverage the properties of a BST, where the left child is always less than the node, and the right child is always greater than the node. We perform a depth-first search (DFS) traversal of the tree and make decisions at each node:

- If the current node's value is less than `low`, it means that this node and all nodes in its left subtree are not needed because their values will also be less than `low`. Thus, we recursively trim the right subtree because there can be valid values there.
- Conversely, if the current node's value is greater than `high`, the current node and all nodes in its right subtree won't be needed since they'll be greater than `high`. So, we recursively trim the left subtree.
- If the current node's value falls within the range `[low, high]`, it is part of the solution tree. We then apply the same trimming process to its left and right children, updating the left and right pointers as we do so.

The recursive process will traverse the tree, and when it encounters nodes that fall outside the `[low, high]` range, it effectively 'removes' them by not including them in the trimmed tree. Once the trimming is complete, the recursive function will return the root of the now-trimmed tree, which may or may not be the same as the original root, depending on the values of `low` and `high`.

Solution Approach

The solution uses a recursive approach to implement the depth-first search (DFS) algorithm for the binary search tree. The goal of the DFS is to visit every node and decide whether to keep it in the trimmed tree based on the `low` and `high` boundaries. The base case for the recursion is when a `None` (null) node is encountered, indicating that we have reached a leaf's child or a node has been removed because it's outside `[low, high]`.

Upon visiting each node, we perform the following steps:

- First, we check if the current node's value is greater than `high`. If it is, we know that we won't need this node or any nodes in its right subtree, so we only return the result of the recursive call on the node's left child.
- If the current node's value is less than `low`, we perform the opposite action. We return the result of the recursive call on the node's right child because the left subtree would contain values that are also too low.
- For nodes that have values within the `[low, high]` range, we apply the recursive trimming process to both children, and then we return the current node, as it satisfies the range conditions.

Here's the implementation of the algorithm:

```python
def dfs(root):
    if root is None:
        return root
    if root.val > high:
        return dfs(root.left)
    if root.val < low:
        return dfs(root.right)
    root.left = dfs(root.left)
    root.right = dfs(root.right)
    return root
```

The `dfs` function is defined within the main class method `trimBST`. We then call `dfs(root)` to begin the recursive trimming process from the root of the given tree.

This recursive function significantly simplifies the process of traversing and modifying the tree, as it leverages the call stack to keep track of the nodes being processed and utilizes the recursion's inherent backtracking mechanism to connect the remaining nodes in the correct structure.

The time complexity of this algorithm is O(N), where N is the number of nodes in the binary search tree, since in the worst case, we may have to visit all nodes. The space complexity is O(H), where H is the height of the tree, due to the space used by the recursion stack during the depth-first search (it may become O(N) in the case of a skewed tree).

Example Walkthrough

Let's consider a small example to illustrate the solution approach. We have the following BST and we want to trim it so that all node values are between `low = 1` and `high = 3`:

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

We start at the root:

1. The root node has a value `4`, which is greater than `high`. According to the algorithm, we won't need this node or any in its right subtree, so we now move to its left child `2`.
2. Node `2` falls within our range `[low, high]`. Therefore, we need to check both of its children.
   - The left child `1` is within the range and ends up being a leaf in our trimmed tree.
   - The right child `3` is also within the range and is kept.

Since `2` falls within the range and both of its children are kept, `2` becomes the new root of our trimmed BST.

The trimmed BST looks like this:

```
    2
   / \
  1   3
```

All nodes are within the range `[1, 3]`, and the tree structure for these nodes is maintained relative to each other. Thus, the trimming process is complete. The recursive calls have assessed and made decisions at every node, leading to this valid trimmed BST.

Python Solution

```python
class TreeNode:
    # Initializer for the TreeNode class
    def __init__(self, value=0, left=None, right=None):
        self.value = value
        self.left = left
        self.right = right

class Solution:
    def trimBST(self, root: Optional[TreeNode], low: int, high: int) -> Optional[TreeNode]:
        """
        Trims the binary search tree such that all values are between the given low and high range (inclusive).
        Removes nodes that are not within the range, and adjusts the tree accordingly.

        :param root: The root of the binary search tree.
        :param low: The lower bound of the range to keep.
        :param high: The upper bound of the range to keep.
        :return: The root of the trimmed binary search tree.
        """
        # Helper function to perform the trimming using depth-first search.
        def trim(root):
            # If we have reached a null node, return it
            if root is None:
                return root

            # If the current node's value is greater than the high cutoff, move to the left subtree
            if root.value > high:
                return trim(root.left)

            # If the current node's value is less than the low cutoff, move to the right subtree
            if root.value < low:
                return trim(root.right)

            # Otherwise, the current node's value is within the range [low, high]
            # We recursively trim the left and right subtrees
            root.left = trim(root.left)
            root.right = trim(root.right)

            # Return the node after potentially trimming its subtrees
            return root

        # Start the trimming process from the root
        return trim(root)

```

Java Solution

```java
// Class definition for a binary tree node.
class TreeNode {
    int val; // Value of the node.
    TreeNode left; // Reference to the left subtree.
    TreeNode right; // Reference to the right subtree.

    // Constructor for creating a tree node without children.
    TreeNode() {}

    // Constructor for creating a tree node with a specific value.
    TreeNode(int val) { this.val = val; }

    // Constructor for creating a tree node with a specific value and children.
    TreeNode(int val, TreeNode left, TreeNode right) {
        this.val = val;
        this.left = left;
        this.right = right;
    }
}

public class Solution {

    /**
     * Trims the binary search tree (BST) so that all its elements lie between the
     * 'low' and 'high' values, inclusive.
     * If the node value is less than 'low', it trims the left subtree.
     * If the node value is greater than 'high', it trims the right subtree.
     *
     * @param root The root of the binary search tree.
     * @param low The lower limit of the range of values to be kept in the BST.
     * @param high The upper limit of the range of values to be kept in the BST.
     * @return The root of the trimmed binary search tree.
     */
    public TreeNode trimBST(TreeNode root, int low, int high) {
        // Base case: if the root is null, return null since there's nothing to trim.
        if (root == null) {
            return null;
        }

        // If the current node's value is greater than 'high', trim the left subtree,
        // as none of the nodes in the right subtree can have value <= 'high'.
        if (root.val > high) {
            return trimBST(root.left, low, high);
        }

        // If the current node's value is less than 'low', trim the right subtree,
        // as none of the nodes in the left subtree can have value >= 'low'.
        if (root.val < low) {
            return trimBST(root.right, low, high);
        }

        // Recursively trim the left and right subtrees.
        root.left = trimBST(root.left, low, high);
        root.right = trimBST(root.right, low, high);

        // Return the trimmed subtree rooted at the current node.
        return root;
    }
}

```

C++ Solution

```cpp
/**
 * Definition for a binary tree node.
 */
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;

    // Constructor with no arguments initializes value to zero and child pointers to nullptr
    TreeNode() : val(0), left(nullptr), right(nullptr) {}

    // Constructor that initializes value with argument x and child pointers to nullptr
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}

    // Constructor that initializes the node with the given value and left and right subtree
    TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
};

class Solution {
public:
    /* Function to trim nodes of a binary search tree such that all its elements
     * lie in the range [low, high]. Values outside the range are removed
     * and the tree is modified accordingly.
     */
    TreeNode* trimBST(TreeNode* root, int low, int high) {
        // If the node is null, just return null
        if (!root) return root;

        // If the node's value is greater than the high boundary, trim the right subtree
        if (root->val > high) return trimBST(root->left, low, high);

        // If the node's value is less than the low boundary, trim the left subtree
        if (root->val < low) return trimBST(root->right, low, high);

        // Node value is within range, recursively trim the left and right subtrees
        root->left = trimBST(root->left, low, high);
        root->right = trimBST(root->right, low, high);

        // Return the node after trimming subtrees as it is within the [low, high] range
        return root;
    }
};

```

Typescript Solution

```typescript
// Define the structure for a binary tree node.
interface TreeNode {
  val: number;
  left: TreeNode | null;
  right: TreeNode | null;
}

/**
 * Trims a binary search tree to contain only values in the [low, high] range.
 * @param root - The root node of the binary search tree.
 * @param low - The lower bound of the allowed value range.
 * @param high - The upper bound of the allowed value range.
 * @returns The root node of the trimmed binary search tree.
 */
function trimBST(root: TreeNode | null, low: number, high: number): TreeNode | null {
    // Helper method that performs a depth-first search to trim the tree.
    function depthFirstSearch(node: TreeNode | null): TreeNode | null {
        if (node === null) {
            return null;
        }

        // If the current node's value is less than low, trim the left subtree.
        if (node.val < low) {
            return depthFirstSearch(node.right);
        }
        // If the current node's value is greater than high, trim the right subtree.
        else if (node.val > high) {
            return depthFirstSearch(node.left);
        }

        // Otherwise, recursively trim the left and right subtrees and attach to the current node.
        node.left = depthFirstSearch(node.left);
        node.right = depthFirstSearch(node.right);
        return node;
    }

    // Kick off the depth-first search with the root node.
    return depthFirstSearch(root);
}

```

Time and Space Complexity

Time Complexity

The time complexity of this function is `O(n)`, where `n` is the number of nodes in the binary tree. This is because, in the worst case, each node of the tree will be visited once when the function `dfs` is called recursively. Even though pruning occurs for nodes that are outside the `[low, high]` range, the function will still visit each node to check if it meets the conditions to be included in the trimmed binary tree.

Space Complexity

The space complexity of this function is `O(h)`, where `h` is the height of the binary tree. This space is used on the call stack because the function is recursive. In the best case, the binary tree is balanced and the height `h` would be `log(n)`, which results in a space complexity of `O(log(n))`. However, in the worst case (if the tree is extremely unbalanced), the height could be `n`, which would result in a space complexity of `O(n)`.