1305. All Elements in Two Binary Search Trees

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

Problem Description

In this problem, we are given two binary search trees, `root1` and `root2`. A binary search tree (BST) is a tree where each node has at most two children, known as the left child and the right child. In a BST, the left subtree of a node contains only nodes with keys less than the node's key, while the right subtree contains only nodes with keys greater than the node's key. Our task is to retrieve all the integers from both trees and return them as a list sorted in ascending order.

Intuition

Given that we're dealing with binary search trees, we can take advantage of their inherent properties to solve this problem efficiently:

1. In-Order Traversal: A key insight is that performing an in-order traversal (Left, Node, Right) on a BST will visit the nodes in ascending order. Therefore, if we perform this traversal on both BSTs, we will get two sorted lists of values. In the given solution, the `dfs` function is a recursive function that conducts an in-order traversal and outputs the visited values into a list.
2. Merging Sorted Lists: With two sorted lists from the BSTs, the next step is to merge these lists into one sorted list. This can be done efficiently by maintaining pointers at the front of each list and taking the smaller value from the front of either list at each step. The `merge` function in the given solution employs a two-pointer approach to combine the lists from the two BSTs into a single sorted list.

By breaking down the problem into two simpler subproblems—one focusing on tree traversal to obtain sorted elements and the other on merging sorted lists—we effectively construct a solution that leverages the characteristics of BSTs to achieve the desired result.

Solution Approach

The solution approach is divided into two main parts: performing in-order traversal on both trees to obtain sorted lists, and then merging these two sorted lists into one.

1. **In-Order Traversal:** The first step involves writing a `dfs` (Depth-First Search) function to perform an in-order traversal of a binary search tree. The function takes a node (initially the root of a BST) and a list as arguments. If the current node is null, it returns immediately, as it signifies the end of a branch. Otherwise, it recursively calls itself for the left child, appends the node's value to the list, and then recursively calls itself for the right child. This approach ensures that each tree's values are added to their respective lists (`t1` and `t2`) in ascending order. The traversal is defined by the following recursive algorithm:

```python
def dfs(root, t):
    if root is None:
        return
    dfs(root.left, t)
    t.append(root.val)
    dfs(root.right, t)
```

2. **Merging Sorted Lists:** After obtaining the sorted lists, the second part of the approach involves merging these two lists into one. The `merge` function introduces two pointers (`i` and `j`), starting at 0, pointing to the beginnings of `t1` and `t2` respectively. It compares the values at `t1[i]` and `t2[j]` and appends the lesser value to the `ans` list, incrementing the corresponding pointer (`i++` or `j++`). If one list is exhausted before the other (one pointer reaches the end of its list), the remaining elements of the other list are appended to `ans`. This merging process yields a new sorted list resulting from combining `t1` and `t2`. Here's the merging logic:

```python
def merge(t1, t2):
    ans = []
    i = j = 0
    while i < len(t1) and j < len(t2):
        if t1[i] <= t2[j]:
            ans.append(t1[i])
            i += 1
        else:
            ans.append(t2[j])
            j += 1
    while i < len(t1):
        ans.append(t1[i])
        i += 1
    while j < len(t2):
        ans.append(t2[j])
        j += 1
    return ans
```

The algorithm comes together in the `getAllElements` method. It initializes two empty lists, `t1` and `t2`, and then calls `dfs` on both `root1` and `root2`, filling `t1` and `t2` with each tree's elements. Afterward, it calls `merge` on these lists to produce a single sorted list containing all the elements from both BSTs, which is finally returned.

By systematically dividing the problem into manageable subproblems, applying appropriate data structures (lists) and algorithms (DFS for tree traversal, merging for combining lists), the solution efficiently merges the elements from two BSTs into a sorted order.

Example Walkthrough

Let's illustrate the solution approach using a small example. Consider the following two binary search trees:

```
Tree 1       Tree 2
  3            2
 / \          / \
1   4        1   7
```

1. **In-Order Traversal:**

First, we perform an in-order traversal on Tree 1:

- Visit left subtree: 1
- Visit root: 3
- Visit right subtree: 4

The sorted list `t1` for Tree 1 is thus `[1, 3, 4]`.

Next, we perform the same traversal on Tree 2:

- Visit left subtree: 1
- Visit root: 2
- Visit right subtree: 7

The sorted list `t2` for Tree 2 is `[1, 2, 7]`.

2. **Merging Sorted Lists:**

Now we have two sorted lists, `t1 = [1, 3, 4]` and `t2 = [1, 2, 7]`. We merge them into one list as follows:

- Initialize `ans = []`, `i = 0`, `j = 0`
- Compare `t1[i]` and `t2[j]`: `1 <= 1`, append `t1[i]` to `ans`, increment `i`. Now `ans = [1]`.
- Compare new `t1[i]` and `t2[j]`: `3 > 1`, append `t2[j]` to `ans`, increment `j`. Now `ans = [1, 1]`.
- Compare new `t1[i]` and `t2[j]`: `3 > 2`, append `t2[j]` to `ans`, increment `j`. Now `ans = [1, 1, 2]`.
- Compare new `t1[i]` and `t2[j]`: `3 <= 7`, append `t1[i]` to `ans`, increment `i`. Now `ans = [1, 1, 2, 3]`.
- Compare new `t1[i]` and `t2[j]`: `4 <= 7`, append `t1[i]` to `ans`, increment `i`. Now `ans = [1, 1, 2, 3, 4]`.
- `i` has reached the end of `t1`, so we append the remaining elements of `t2` (`[7]`) to `ans`. Now `ans = [1, 1, 2, 3, 4, 7]`.

The merged sorted list `ans` is `[1, 1, 2, 3, 4, 7]`, and that will be our final output. The solution approach leverages efficient in-order traversal to generate sorted lists from each tree, followed by a streamlined merge process, thus solving the problem effectively.

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 getAllElements(self, root1: TreeNode, root2: TreeNode) -> List[int]:
        # Helper function to perform in-order traversal of the binary tree
        # and store the elements in a list.
        def in_order_traversal(root: TreeNode, elements: List[int]):
            if root is None:
                return
            in_order_traversal(root.left, elements)
            elements.append(root.val)
            in_order_traversal(root.right, elements)

        # Helper function to merge two sorted lists into a single sorted list.
        def merge_lists(sorted_list1: List[int], sorted_list2: List[int]) -> List[int]:
            merged_list = []
            i, j = 0, 0
            # Interleave elements from both lists in a sorted manner.
            while i < len(sorted_list1) and j < len(sorted_list2):
                if sorted_list1[i] <= sorted_list2[j]:
                    merged_list.append(sorted_list1[i])
                    i += 1
                else:
                    merged_list.append(sorted_list2[j])
                    j += 1
            # Append any remaining elements from the first list
            while i < len(sorted_list1):
                merged_list.append(sorted_list1[i])
                i += 1
            # Append any remaining elements from the second list
            while j < len(sorted_list2):
                merged_list.append(sorted_list2[j])
                j += 1
            return merged_list

        # Initialize lists to store the elements from each tree.
        tree1_elements, tree2_elements = [], []
        # Perform in-order traversal on both trees to get sorted elements.
        in_order_traversal(root1, tree1_elements)
        in_order_traversal(root2, tree2_elements)
        # Merge the sorted lists and return the result.
        return merge_lists(tree1_elements, tree2_elements)

```

Java Solution

```java
class Solution {
    // This method returns a list of all elements from two binary tree roots.
    public List<Integer> getAllElements(TreeNode root1, TreeNode root2) {
        List<Integer> tree1Elements = new ArrayList<>();
        List<Integer> tree2Elements = new ArrayList<>();
        // Perform in-order traversal to get elements in non-descending order.
        inorderTraversal(root1, tree1Elements);
        inorderTraversal(root2, tree2Elements);
        // Merge two sorted lists into one sorted list.
        return mergeSortedLists(tree1Elements, tree2Elements);
    }

    // Helper method to perform a DFS in-order traversal.
    private void inorderTraversal(TreeNode root, List<Integer> elements) {
        if (root == null) {
            return;
        }
        inorderTraversal(root.left, elements); // Visit left subtree
        elements.add(root.val); // Visit current node
        inorderTraversal(root.right, elements); // Visit right subtree
    }

    // Helper method to merge two sorted lists.
    private List<Integer> mergeSortedLists(List<Integer> list1, List<Integer> list2) {
        List<Integer> mergedList = new ArrayList<>();
        int i = 0, j = 0;
        // Merge elements while there are elements in both lists.
        while (i < list1.size() && j < list2.size()) {
            if (list1.get(i) <= list2.get(j)) {
                mergedList.add(list1.get(i++)); // Add from list1 and increment i
            } else {
                mergedList.add(list2.get(j++)); // Add from list2 and increment j
            }
        }
        // Add any remaining elements in list1 to the merged list.
        while (i < list1.size()) {
            mergedList.add(list1.get(i++));
        }
        // Add any remaining elements in list2 to the merged list.
        while (j < list2.size()) {
            mergedList.add(list2.get(j++));
        }
        return mergedList;
    }
}

```

C++ Solution

```cpp
#include <vector>

// 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:
    // Retrieves all elements from both trees and returns them in a sorted vector.
    vector<int> getAllElements(TreeNode* root1, TreeNode* root2) {
        vector<int> elementsTree1;
        vector<int> elementsTree2;

        // Traverse both trees and collect their values
        depthFirstSearch(root1, elementsTree1);
        depthFirstSearch(root2, elementsTree2);

        // Merge the collected values into a single sorted vector
        return merge(elementsTree1, elementsTree2);
    }

    // Helper function to perform in-order depth-first search on the tree
    // and collect values in a vector.
    void depthFirstSearch(TreeNode* root, vector<int>& elements) {
        if (!root) return;
        depthFirstSearch(root->left, elements);  // Visit left subtree
        elements.push_back(root->val);           // Visit node
        depthFirstSearch(root->right, elements); // Visit right subtree
    }

    // Helper function to merge two sorted vectors into one sorted vector.
    vector<int> merge(const vector<int>& tree1Elements, const vector<int>& tree2Elements) {
        vector<int> mergedElements;
        int i = 0, j = 0;

        // Merge the two vectors until one is fully traversed.
        while (i < tree1Elements.size() && j < tree2Elements.size()) {
            if (tree1Elements[i] <= tree2Elements[j]) {
                mergedElements.push_back(tree1Elements[i++]);
            } else {
                mergedElements.push_back(tree2Elements[j++]);
            }
        }

        // Add the remaining elements from the first vector if any.
        while (i < tree1Elements.size()) {
            mergedElements.push_back(tree1Elements[i++]);
        }

        // Add the remaining elements from the second vector if any.
        while (j < tree2Elements.size()) {
            mergedElements.push_back(tree2Elements[j++]);
        }

        return mergedElements;
    }
};

```

Typescript Solution

```typescript
function getAllElements(root1: TreeNode | null, root2: TreeNode | null): number[] {
    // Initialize the result array which will contain all the elements in sorted order.
    const result: number[] = [];

    // Use two stacks to perform inorder traversal on both trees simultaneously.
    const stacks: [TreeNode[], TreeNode[]] = [[], []];

    // Continue the traversal as long as there are nodes to be processed in either stack or trees.
    while (root1 !== null || stacks[0].length > 0 || root2 !== null || stacks[1].length > 0) {
        // Inorder traversal on the first tree.
        while (root1 !== null) {
            stacks[0].push(root1);
            root1 = root1.left;  // Move to the left child.
        }

        // Inorder traversal on the second tree.
        while (root2 !== null && (stacks[0].length === 0 || root2.val < stacks[0][stacks[0].length - 1].val)) {
            stacks[1].push(root2);
            root2 = root2.left;  // Move to the left child.
        }

        // Determine which tree's node value to take (the smaller one), and move to the right subtree.
        if (stacks[0].length === 0 ||
            (stacks[1].length > 0 && stacks[0][stacks[0].length - 1].val > stacks[1][stacks[1].length - 1].val)) {
            const { val, right } = stacks[1].pop()!;
            result.push(val);  // Add the value of the node to the result array.
            root2 = right;     // Update root2 to the right child.
        } else {
            const { val, right } = stacks[0].pop()!;
            result.push(val);  // Add the value of the node to the result array.
            root1 = right;     // Update root1 to the right child.
        }
    }

    // Return the result array containing all the elements from both trees in a sorted order.
    return result;
}

```

Time and Space Complexity

The above code consists of two major parts: the Depth-First Search (DFS) for in-order traversal and the Merge step to combine two sorted arrays. Let's analyze each part separately.

DFS In-order Traversal

The DFS function `dfs()` is called for each of the two binary trees. It traverses all nodes exactly once in a recursive in-order fashion (left, node, right).

- **Time Complexity for each tree**: It's `O(n)` for `root1` and `O(m)` for `root2`, where `n` and `m` represent the number of nodes in `root1` and `root2`, respectively.
- **Space Complexity for each tree**: Due to the recursion stack, the space complexity is `O(h1)` for `root1` and `O(h2)` for `root2`, where `h1` and `h2` are the heights of the trees.

Merge Function

The `merge()` function merges two sorted arrays `t1` and `t2`.

- **Time Complexity**: As each element from both lists is considered exactly once, the time complexity of this operation is `O(n + m)`.
- **Space Complexity**: We're creating a new list `ans` to store the merged arrays, so the space complexity is `O(n + m)` as we are storing all `n + m` elements.

Overall Complexity

The total time complexity of `getAllElements()` is the sum of the complexities of both parts, dominated by the `O(n + m)` complexity of the merge step.

- **Time Complexity**: `O(n + m)`
  - To be precise, it can be broken down to `O(n) + O(m) + O(n + m)`, which simplifies to `O(n + m)`.

The total space complexity is given by the space used to store the traversal results and the merge results, together with the recursion stack space.

- **Space Complexity**: `O(n + m) + max(O(h1), O(h2))`
  - Here, `O(n + m)` is for the space needed for `t1`, `t2`, and `ans`, and `max(O(h1), O(h2))` accounts for the recursive call stack space which will be at most the height of the taller tree at any instance.