776. Split BST

Medium Tree Binary Search Tree Recursion Binary Tree Leetcode Link

Problem Description

Given a Binary Search Tree (BST) and an integer `target`, the problem asks us to split the tree into two separate subtrees based on the values relative to `target`. The first subtree should contain all the nodes that have values less than or equal to the target, and the second subtree should contain all the nodes that have values greater than the target. It's important that if a node was a parent to another node before the split, and both of them ended up in the same subtree after the split, then their parent-child relationship should remain unchanged.

Intuition

The intuition for solving this problem lies in understanding the properties of a BST. In a BST, for any given node, all the values in its left subtree are less than its value, and all the values in its right subtree are greater than its value. With this characteristic in mind, when we come across a node that is less than or equal to the target, we know that this node and its left subtree should definitely be a part of the subtree that contains nodes less than or equal to the target. However, we need to recursively adjust its right subtree to split it further according to the target.

Similarly, if a node's value is greater than the target, this node and its right subtree should be in the subtree that contains nodes greater than the target. Here, the left subtree must be checked recursively to separate the nodes correctly.

The recursive approach involves visiting each node and deciding whether to split its left or right subtree (or both), thereby constructing the two subtrees required while maintaining the original parent-child relationship whenever possible. The recursion eventually hits a base case when it reaches a null node, at which point it returns a pair of null nodes representing the end of a branch.

Hence, for each node, we return a pair: the first element of the pair is the root of the subtree containing nodes less than or equal to the target, and the second element is the root of the subtree containing nodes greater than the target.

Solution Approach

The solution implements a depth-first search (DFS) algorithm through a recursive function called `dfs`. This function takes a node of the tree as an input and splits it into two subtrees according to the `target`. `dfs` returns a list with two elements, where the first element is the root of the subtree with nodes less than or equal to the target, and the second is the root of the subtree with nodes greater than the target.

Here's a step-by-step breakdown of the `dfs` function in detail:

1. When the `dfs` function encounters a `None` node (the base case), it returns a pair of `None` nodes, indicating the end of a branch. This is represented by `return [None, None]`.
2. If the current node's value is less than or equal to the `target`, all nodes in its left subtree are guaranteed to also be less than or equal to the target (by the BST property). The function proceeds to split the right subtree. The recursive call `dfs(root.right)` will split the right subtree into two parts: those less than or equal to the target (`l`) and those greater than the target (`r`). Since the current node and its left subtree are all less than or equal to the target, the current node's right child should be updated to `l` (the part of the right subtree that's less than or equal to the target). It then returns `[root, r]`, where `root` is now the root of the subtree with data less than or equal to the target and `r` is the root of the subtree with data greater than the target.
3. If the current node's value is greater than the `target`, then by the BST property, all nodes in its right subtree are greater than the target. The function then recursively splits the left subtree with `dfs(root.left)`. It receives a pair of roots, `l` for the left split (all nodes less than or equal to the target) and `r` for the right split (all nodes greater than the target up to now). In this case, `root.left` should be updated to `r` (the part of the left subtree that's greater than the target). The function then returns `[l, root]` as the nodes that are less than or equal to the target are now in `l`, and those that are greater than the target are under `root`.

As the recursion unwinds, each node processes its left or right subtrees depending upon its value relative to the target. Since recursion visits all nodes once, the time complexity is `O(n)`, where `n` is the number of nodes in the BST. No extra space is used apart from the recursion stack, making the space complexity `O(h)`, where `h` is the height of the BST.

The main function `splitBST` simply starts this recursive process by calling `dfs` on the root of the tree and then returns the result. The returned value is a list that holds the two new roots of the subtrees that are the result of the split.

Example Walkthrough

Consider the following Binary Search Tree and the target value `3`:

```
    4
   / \
  2   6
 / \ / \
1  3 5  7
```

Our task is to split this BST into two subtrees, where the first subtree contains all nodes with values less than or equal to 3, and the second subtree contains all nodes with values greater than 3.

Let's apply the solution approach:

1. We begin with the root node (4). Since 4 is greater than 3, we'll need to split its left subtree to determine which nodes are less than or equal to the target, and the right subtree is already greater than 3.
2. Now we look at the left subtree:

   ```
     2
    / \
   1   3
   ```

   The root here is 2, which is less than or equal to 3. So, we keep 2 and its left subtree as is because all values must also be less than or equal to 3. The right child of 2 is 3, which is equal to the target, so it too remains in the subtree. The right subtree of this node is processed, but since it's a leaf node (3), it is just included without changes. So no further action is needed, and we keep the left subtree as it is.
3. Returning to the original root node (4), we have already verified that its entire left subtree is less than or equal to the target. We can simply include this left subtree in our result. For the right subtree, which starts with the root node (4), we don't need to process it since it is already in the subtree that contains nodes greater than the target. So, we attach the original right subtree (starting with node 6) to our node 4.

The resulting subtrees are:

```
Subtree less than or equal to 3:
    2
   / \
  1   3

Subtree greater than 3:
    4
     \
      6
     / \
    5   7
```

The final output of the `splitBST` function would be the roots of the two subtrees: `[2, 4]`. Node `2` is the root of the subtree with nodes less than or equal to 3, and node `4` is the root of the subtree with nodes greater than 3.

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 splitBST(self, root: Optional[TreeNode], target: int) -> List[Optional[TreeNode]]:
        # Helper function to recursively split the BST
        def dfs(node: Optional[TreeNode]) -> List[Optional[TreeNode]]:
            # Base case: if the current node is None, return a pair of None nodes
            if node is None:
                return [None, None]

            # If current node's value is less than or equal to the target,
            # split the right subtree
            if node.val <= target:
                left_of_right, right_of_right = dfs(node.right)
                # Link the left part of the split right subtree to the current node's right
                node.right = left_of_right
                # The current node becomes part of the left tree, the right part
                # of the split right subtree becomes the right tree
                return [node, right_of_right]
            else:
                # If current node's value is greater than the target,
                # split the left subtree
                left_of_left, right_of_left = dfs(node.left)
                # Link the right part of the split left subtree to the current node's left
                node.left = right_of_left
                # The left part of the split left subtree becomes the left tree,
                # the current node becomes part of the right tree
                return [left_of_left, node]

        # Call the helper function with the root of the BST
        return dfs(root)

```

Java Solution

```java
class Solution {
    private int targetValue;

    // Splits a BST into two trees based on the target value; each tree contains either all elements less than or equal to the target
    // or all elements greater than the target, maintaining the BST properties.
    public TreeNode[] splitBST(TreeNode root, int target) {
        targetValue = target;
        return split(root);
    }

    // A recursive helper function that splits the BST.
    private TreeNode[] split(TreeNode node) {
        if (node == null) {
            return new TreeNode[]{null, null};
        }
        if (node.val <= targetValue) {
            // If the current node's value is less than or equal to the target, split the right subtree.
            TreeNode[] splitRight = split(node.right);
            node.right = splitRight[0]; // The left part of the right subtree becomes the right child of the current node.
            splitRight[0] = node; // The current node becomes the rightmost node of the left split tree.
            return splitRight;
        } else {
            // If the current node's value is greater than the target, split the left subtree.
            TreeNode[] splitLeft = split(node.left);
            node.left = splitLeft[1]; // The right part of the left subtree becomes the left child of the current node.
            splitLeft[1] = node; // The current node becomes the leftmost node of the right split tree.
            return splitLeft;
        }
    }

    // Definition for a binary tree node.
    public class TreeNode {
        int val;
        TreeNode left;
        TreeNode right;

        TreeNode() {}

        TreeNode(int val) {
            this.val = val;
        }

        TreeNode(int val, TreeNode left, TreeNode right) {
            this.val = val;
            this.left = left;
            this.right = right;
        }
    }
}

```

C++ Solution

```cpp
/**
 * 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:
    int targetValue;

    // Splits the tree into two subtrees where one tree contains all elements less than or equal to target
    // and the other contains all elements greater than target.
    vector<TreeNode*> splitBST(TreeNode* root, int target) {
        targetValue = target;
        return split(root);
    }

    // Helper function to perform the split operation
    vector<TreeNode*> split(TreeNode* rootNode) {
        // If the current node is null, then return a pair of null pointers.
        if (!rootNode) {
            return {nullptr, nullptr};
        }

        // If the current node's value is less than or equal to the target, we process the right subtree.
        if (rootNode->val <= targetValue) {
            // Recursively split the right subtree.
            vector<TreeNode*> ans = split(rootNode->right);

            // Attach the 'less than or equal to target' part of the right subtree to the current node's right child.
            rootNode->right = ans[0];

            // The current node becomes part of the 'less than or equal to target' subtree.
            ans[0] = rootNode;

            // Return the updated pair of subtrees.
            return ans;
        } else {
            // If the current node's value is greater than the target, process the left subtree.
            vector<TreeNode*> ans = split(rootNode->left);

            // Attach the 'greater than target' part of the left subtree to the current node's left child.
            rootNode->left = ans[1];

            // The current node becomes part of the 'greater than target' subtree.
            ans[1] = rootNode;

            // Return the updated pair of subtrees.
            return ans;
        }
    }
};

```

Typescript Solution

```typescript
// Typing definition for a binary tree node
interface TreeNode {
  val: number;
  left: TreeNode | null;
  right: TreeNode | null;
}

/**
 * Splits a BST into two trees based on a target value
 * @param {TreeNode | null} root - The root of the BST to split
 * @param {number} target - The value to split the BST around
 * @return {TreeNode[]} An array containing two trees: one with all values <= target, one with all values > target
 */
const splitBST = (root: TreeNode | null, target: number): [TreeNode | null, TreeNode | null] => {
    // Initialized answer array where first element is the tree with values <= target,
    // and second element is the tree with values > target.
    let answer: [TreeNode | null, TreeNode | null] = [null, null];

    // If the root is null, return the answer array with both elements as null.
    if (!root) {
        return answer;
    }

    if (root.val <= target) {
        // If root value is less than or equal to target, split the right subtree.
        answer = splitBST(root.right, target);

        // Attach the left part of the split right subtree to the current root's right.
        root.right = answer[0];

        // Set the current root as the new root for the left part.
        answer[0] = root;
    } else {
        // If root value is greater than the target, split the left subtree.
        answer = splitBST(root.left, target);

        // Attach the right part of the split left subtree to the current root's left.
        root.left = answer[1];

        // Set the current root as the new root for the right part.
        answer[1] = root;
    }

    // Return the pair of trees as the result of the split.
    return answer;
};

```

Time and Space Complexity

The given Python code defines a method `splitBST` that splits a Binary Search Tree (BST) into two trees. It partitions the BST such that all elements less than or equal to the `target` are in the left tree, and all elements greater than the `target` are in the right tree.

Time Complexity

The time complexity of the `splitBST` function is `O(h)`, where `h` is the height of the BST. The algorithm works by performing a depth-first search (DFS) through the tree, splitting at each node based on the target. Since each node is visited at most once in a DFS traversal, and the depth of a DFS is proportional to the height of the tree, the overall time complexity is a function of the tree's height.

For a balanced tree, the height `h` would be `log(n)`, where `n` is the number of nodes, resulting in a time complexity of `O(log(n))`. For an unbalanced tree, in the worst case, the height could be `n`, leading to a time complexity of `O(n)`.

Space Complexity

The space complexity of the `splitBST` function is `O(h)`. In practice, this is due to the space used by the call stack during the recursive DFS traversal of the tree. As with the time complexity, the worst-case space complexity is when the tree is unbalanced, leading to a depth of `n`, and therefore a space complexity of `O(n)`.

For a balanced tree, since the height of the tree would be `log(n)`, the space complexity would be `O(log(n))`.

So, the complexities depend on the balance of the BST:

- For a balanced tree: `Time Complexity: O(log(n))`, `Space Complexity: O(log(n))`
- For an unbalanced tree: `Time Complexity: O(n)`, `Space Complexity: O(n)`