1008. Construct Binary Search Tree from Preorder Traversal

Medium Tools Tree Binary Search Tree Array Binary Tree Monotonic Stack

Leetcode Link

Problem Description

This problem requires constructing a Binary Search Tree (BST) from a given array of integers that represented the preorder traversal of the BST. A BST is a type of binary tree where each node satisfies the constraint that all the values in its left subtree are less than the node's value, and all the values in its right subtree are greater than the node's value.

Given the nature of preorder traversal, the first element in the traversal represents the root of the tree. The following elements from the root are then divided into two subarrays: the left subtree elements and the right subtree elements. The left subtree contains all elements less than the root's value, and the right subtree contains all elements greater than the root's value. These subarrays are used recursively to construct the left and right subtrees.

Intuition

To construct the BST from the preorder sequence, we make use of the above-mentioned property of BSTs: all left descendants are less than the node, and all right descendants are greater.

Here's the approach to solving the problem:

1. Treat the first value as the root node since we're given a preorder traversal array.
2. Split the remaining elements into two groups: one for the left subtree and one for the right subtree. The left subtree consists of all elements less than the root's value, whereas the right subtree includes all elements more significant than the root's value.
3. Recursively apply these steps to both the left and right groups to construct the entire tree.

The solution makes use of a recursive function, `dfs`, which creates a node for each call using the first element in the current array slice as the node's value. It then uses binary search to find the boundary between the left and right children nodes. Finally, it constructs the subtree by recursively calling the same function on the appropriate subarrays for left and right children.

By breaking down the problem in this way and using recursion, we can recreate the BST that corresponds to the given preorder traversal.

Solution Approach

The solution is implemented in Python, and it uses a recursive depth-first search (DFS) approach to reconstruct the Binary Search Tree (BST). The code block defines a helper function `dfs` that performs the main logic of constructing the BST.

Let's explore the key parts of the implementation:

Recursive DFS Function:

- The `dfs` function takes the `preorder` array slice representing the current subtree it is processing.
- It first checks if the `preorder` array is empty. If so, it returns `None`, as there are no more nodes to construct in this subtree.
- It then creates the root node of the current subtree using the first value of the `preorder` array as `TreeNode(preorder[0])`.

Binary Search to Split the Array:

- Next, within the `dfs` function, we use binary search to determine the boundary between the left and right children nodes.
- This boundary is the index of the first element in `preorder` that is greater than the root's value. We use a loop that performs a binary search to find this boundary efficiently.
- Once found, the elements from `preorder[1:left]` will be all the elements that belong to the left subtree (values less than the root's value), and the elements from `preorder[left:]` will be those that belong to the right subtree (values greater than the root's value).

Recursion to Build Subtrees:

- The `dfs` function calls itself twice recursively to construct the left and right subtrees. The left subtree is created using the elements before the found boundary, and the right subtree is created using the elements after the boundary.

Putting It All Together:

- The initial call to `dfs` is made with the entire `preorder` array, and it returns the root of the BST constructed.
- As the recursive calls proceed, smaller slices of the `preorder` array are passed down to construct each subtree until the entire BST is built.

The Recursive Process in Steps:

1. The function is initially called with the full `preorder` array. The first element is treated as the root node.
2. Use binary search within the remaining items to split them into left and right subtrees based on their value in relation to the root's value.
3. Recursively call `dfs` with the left subtree elements to construct the left subtree. This call will continue to split and build left and right subtrees until the base case (empty array slice) is reached.
4. Recursively call `dfs` with the right subtree elements to do the same for the right subtree.

Example Walkthrough

Let's illustrate the solution approach with a small example. Consider the `preorder` traversal array `[8, 5, 1, 7, 10, 12]`. We want to reconstruct the BST that results in this preorder sequence.

Step-by-Step Reconstruction:

1. Call `dfs` function with the full `preorder` array `[8, 5, 1, 7, 10, 12]`. The first element 8 is chosen as the root node.
2. We perform a binary search to split the array into elements that are less than 8 (`[5, 1, 7]`) for the left subtree, and elements that are greater (`[10, 12]`) for the right subtree.
3. The left subtree call to `dfs` uses `[5, 1, 7]`. Here, 5 is the root of the left subtree.
4. Binary search on `[1, 7]` splits it into `[1]` for the left subtree and `[7]` for the right, both relative to the new root node 5.
5. The recursion continues, with `dfs` called with `[1]`, making 1 a leaf node (as it has no further elements to process), and similarly, `dfs` called with `[7]` making 7 a leaf node on the right side of node 5.
6. Meanwhile, for the right subtree, `dfs` is called with `[10, 12]`. The root node for this subtree is 10, with no left children (since there are no elements less than 10) and a right node to be created from `[12]`.
7. `dfs` called with `[12]` makes 12 a leaf node, being the right child of node 10.

Final BST Structure:

The final BST, reconstructed from the preorder array `[8, 5, 1, 7, 10, 12]`, will look like this:

```
    8
   / \
  5   10
 / \   \
1   7   12
```

Each recursive call to `dfs` constructs a part of this tree, resulting in the BST that reflects the given preorder sequence.

Python Solution

```python
class TreeNode:
    # Definition for a binary tree node.
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Solution:
    def bstFromPreorder(self, preorder):
        """
        Construct a binary search tree from a list of values representing
        preorder traversal.

        :param preorder: List[int] - A list of integers representing the preorder traversal of a BST.
        :return: Optional[TreeNode] - The root node of the constructed BST.
        """

        def construct_bst_from_preorder(preorder_values):
            # Base case: if the list is empty, return None as there's no tree to construct.
            if not preorder_values:
                return None

            # The first value in the preorder list is the root of the BST.
            root = TreeNode(preorder_values[0])
            left_index, right_index = 1, len(preorder_values)

            # Find the boundary between left and right subtrees.
            while left_index < right_index:
                mid = (left_index + right_index) // 2  # Using floor division for Python 3.
                if preorder_values[mid] > preorder_values[0]:
                    # If the mid value is greater than the root's value,
                    # it belongs to the right subtree.
                    right_index = mid
                else:
                    # Otherwise, it belongs to the left subtree.
                    left_index = mid + 1

            # Recursively build the left and right subtrees.
            root.left = construct_bst_from_preorder(preorder_values[1:left_index])
            root.right = construct_bst_from_preorder(preorder_values[left_index:])

            # Return the root of the constructed subtree.
            return root

        # Call the recursive helper function to build the BST from preorder traversal.
        return construct_bst_from_preorder(preorder)
```

Java Solution

```java
class Solution {
    public TreeNode bstFromPreorder(int[] preorder) {
        // Start constructing binary search tree from the preorder array
        return constructBST(preorder, 0, preorder.length - 1);
    }

    /**
     * Helper method to recursively construct a BST given a preorder traversal range.
     *
     * @param preorder The array storing the preorder traversal of the BST.
     * @param startIndex The start index for the search in the current subtree.
     * @param endIndex The end index for the search in the current subtree.
     * @return The constructed subtree that is the root of this subtree.
     */
    private TreeNode constructBST(int[] preorder, int startIndex, int endIndex) {
        // Base case: when the start index is greater than end index or out of bounds
        if (startIndex > endIndex || startIndex >= preorder.length) {
            return null;
        }

        // The first element of the current range is the root of this subtree
        TreeNode root = new TreeNode(preorder[startIndex]);

        // Find the boundary between left and right subtrees
        // The first bigger element than the root will be the root of the right subtree
        int splitIndex = findSplitIndex(preorder, startIndex, endIndex, preorder[startIndex]);

        // Construct the left subtree
        root.left = constructBST(preorder, startIndex + 1, splitIndex - 1);

        // Construct the right subtree
        root.right = constructBST(preorder, splitIndex, endIndex);

        return root;
    }

    /**
     * Helper method to find the index of the first element greater than the root's value.
     *
     * @param preorder The array storing the preorder traversal.
     * @param startIndex The start index for the search.
     * @param endIndex The end index for the search.
     * @param rootValue The value of the root node.
     * @return The index of the first element bigger than rootValue, or the end of the range if not found.
     */
    private int findSplitIndex(int[] preorder, int startIndex, int endIndex, int rootValue) {
        int left = startIndex + 1;
        int right = endIndex + 1;
        while (left < right) {
            int mid = (left + right) >> 1; // Equivalent to (left + right) / 2 but faster
            if (preorder[mid] > rootValue) {
                right = mid;
            } else {
                left = mid + 1;
            }
        }
        // left is the index of the first element greater than rootValue, or endIndex + 1
        return left;
    }
}
```

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:
    // Constructs a binary search tree from preorder traversal values.
    TreeNode* bstFromPreorder(vector<int>& preorder) {
        return constructBST(preorder, 0, preorder.size() - 1);
    }

    // Helper function to construct BST from preorder traversal in range [start, end].
    TreeNode* constructBST(vector<int>& preorder, int start, int end) {
        // Base case: If start index is greater than end or start is out of bounds.
        if (start > end || start >= preorder.size()) return nullptr;

        // Create a new tree node with the current value.
        TreeNode* root = new TreeNode(preorder[start]);

        // Initialize the positions to partition the preorder vector.
        int leftTreeEnd = start + 1, rightTreeStart = end + 1;

        // Binary search to find the first value greater than root's value, to differentiate left and right subtrees.
        while (leftTreeEnd < rightTreeStart) {
            int mid = (leftTreeEnd + rightTreeStart) >> 1; // Equivalent to divide by 2
            if (preorder[mid] > preorder[start])
                rightTreeStart = mid; // Adjust the end of the left subtree
            else
                leftTreeEnd = mid + 1; // Adjust the start of the right subtree
        }

        // Recursively construct left and right subtrees.
        root->left = constructBST(preorder, start + 1, leftTreeEnd - 1);
        root->right = constructBST(preorder, leftTreeEnd, end);

        return root;
    }
};
```

Typescript Solution

```typescript
// Definition for a binary tree node.
class TreeNode {
    val: number
    left: TreeNode | null
    right: TreeNode | null
    constructor(val=0, left=null, right=null) {
        this.val = val
        this.left = left
        this.right = right
    }
}

// This function builds a binary search tree from a preorder traversal.
function bstFromPreorder(preorder: number[]): TreeNode | null {
    // Length of the preorder traversal
    const length = preorder.length;
    // Initialize the next array to track the next greater element's index
    const nextGreaterIndex = new Array(length);
    // Stack to help find the next greater element
    const stack: number[] = [];

    // Iterate backwards through the preorder array to populate nextGreaterIndex
    for (let i = length - 1; i >= 0; --i) {
        // Pop elements from the stack until the current element is greater
        while (stack.length > 0 && preorder[stack[stack.length - 1]] < preorder[i]) {
            stack.pop();
        }
        // Assign the index of the next greater element or the length if not found
        nextGreaterIndex[i] = stack.length > 0 ? stack[stack.length - 1] : length;
        // Push the current index onto the stack
        stack.push(i);
    }

    // Recursive function to build the tree
    const buildTree = (leftIndex: number, rightIndex: number): TreeNode | null => {
        // If the indices are the same, we've reached a leaf node, return null
        if (leftIndex === rightIndex) {
            return null;
        }
        // Create the root TreeNode with the value from preorder traversal
        // The left child is built from the elements immediately after the root
        // The right child is built from elements after the left subtree
        return new TreeNode(
            preorder[leftIndex],
            buildTree(leftIndex + 1, nextGreaterIndex[leftIndex]),
            buildTree(nextGreaterIndex[leftIndex], rightIndex)
        );
    };

    // Start building the tree from the first element
    return buildTree(0, length);
}

// Example usage:
// const preorder = [8, 5, 1, 7, 10, 12];
// const tree = bstFromPreorder(preorder);
// This will build the corresponding BST from the given preorder array
```

Time and Space Complexity

The provided code builds a binary search tree from its preorder traversal sequence. Let's analyze the time and space complexity.

Time Complexity

The time complexity of the code is $O(n \log n)$ in the average case and $O(n^2)$ in the worst case. Here's why:

- The `dfs()` function is invoked for each element of the `preorder` sequence once, and within each call, it performs a binary search to split the left and right subtrees, which takes $O(\log n)$ time in the best and average case due to the binary search condition `preorder[mid] < preorder[0]`.
- However, if the tree is skewed (i.e., every node only has one child), the binary search degrades into a linear scan at each insertion, as there will be no early termination of the binary search loop (`while left < right`). In this case, each insertion takes $O(n)$ time, giving us $O(n^2)$ complexity over n insertions.

Overall, assuming the tree is balanced or nearly balanced, we would expect average time complexity to be $O(n \log n)$. In the worst case, for a skewed tree, it would be $O(n^2)$.

Space Complexity

The space complexity can be considered in two parts: the recursion call stack space and the space used to store the tree itself.

- **Recursion Call Stack Space**: In the average case, the binary tree will be somewhat balanced, and the maximum depth of the recursion stack would be $O(\log n)$. In the worst case of a completely skewed tree (e.g., a linked list), the space complexity of the recursion would be $O(n)$.
- **Space for the Tree Itself**: Independently of the call stack, the space required to store the tree is $O(n)$ since we need to allocate a node for every element in the input array.

Thus, the overall space complexity is $O(n)$, including both the average case space to store the tree and the average scenario ($O(\log n)$) and the space used to store the tree nodes ($O(n)$). In the worst case, regarding the call stack, it could be $O(n)$, but the dominant term remains the tree storage, so we remain at a total space complexity of $O(n)$.