988. Smallest String Starting From Leaf

Medium Tree Depth-First Search String Binary Tree Leetcode Link

Problem Description

In this problem, we are provided with the root of a binary tree where each node has a value within the range `[0, 25]`. These values represent the letters `'a'` to `'z'`. The task is to find the lexicographically smallest string that starts at a leaf of the tree and ends at the root. A leaf is defined as a node with no children. In terms of string comparison, a shorter prefix of a string is considered lexicographically smaller.

Intuition

The problem is essentially asking us to find the lexicographically smallest path from a leaf node to the root. Since the lexicographically smaller string could be due to a shorter length, we should aim to find shorter paths first.

We can solve this problem using a Depth-First Search (DFS) approach that will explore all paths from the root to the leaves and keep track of the lexicographically smallest path found at any point. Here's how we can proceed:

1. Traverse the tree starting from the root node and going down towards the leaves using DFS.
2. As we traverse, construct the string representing the path from the current leaf to the root by appending characters corresponding to node values at each step.
3. When a leaf node is reached, we reverse the constructed string (as we want it from leaf to root, but we are moving from root to leaf) and compare it with the currently found lexicographically smallest string. If it's smaller, we update the smallest string.
4. After the DFS is completed, we will have the lexicographically smallest string from any leaf to the root.

A crucial detail is to handle the string comparison efficiently. Using Python's built-in string comparison, we can directly compare the strings using the `min` function. It's also important to note that we need to reverse the constructed path only once when we reach a leaf to avoid unnecessary computation.

Solution Approach

The solution approach involves a Depth-First Search (DFS) algorithm that traverses the binary tree and efficiently constructs strings from leaves to root. During traversal, it keeps track of the smallest string found. Here's the detailed implementation described in steps:

1. **Character Conversion**: Since node values are integers in the range `[0, 25]`, the first step is to convert the node values into characters. This is done by adding the node value to the ASCII value of `'a'`, which is done using `chr(ord('a') + root.val)`.

2. **DFS Function (`dfs`)**: A recursive function `dfs` is defined, which takes the `root` node and the current `path` as arguments. The `path` argument is a list that stores the characters from the root to the current node. This function is responsible for DFS traversal and updates the smallest string.

3. **Base Case**: The base case for the DFS is when the `root` is `None`. In this case, the function returns without performing any action.

4. **Path Construction**: When the `root` is not `None`, the corresponding character for the node value is appended to the `path`. If the `root` is a leaf (both `root.left` and `root.right` are `None`), the complete path from the root to this leaf is evaluated for lexicographical order.

5. **Leaf String Reversal and Comparison**: Upon reaching a leaf node, the `path` is reversed to make the string start at the leaf and end at the root, and the lexicographically smallest string `ans` is updated by comparing the current string with the existing smallest string using `min(ans, ''.join(reversed(path)))`.

6. **DFS Recursion**: The function then recursively calls itself for the left and right children of the current node if they exist, enabling traversal of the entire tree.

7. **Backtracking**: After exploring both subtrees (left and right) from the current node, it's important to remove the current node's character from the `path`. This is to backtrack and ensure that when moving to a different branch, the path is correctly maintained - hence `path.pop()` is used.

8. **Global Smallest String (`ans`)**: A variable `ans` is initialized to a string value higher than any possible value from the tree (the ASCII value of `'z'` plus one), and it's defined as `nonlocal` within the `dfs` function to allow updating its value across recursive calls.

The specified approach takes advantage of the recursive nature of DFS to explore all paths from the root to leaf nodes and employs backtracking to efficiently update the paths during traversal. Additionally, Python's concise syntax and string comparison capabilities are leveraged to keep the implementation clean and readable.

Algorithm Complexity: The time complexity is O(N * M), where N represents the number of nodes in the tree and M is the maximum depth of the tree, since we need to compare strings of length M for each leaf node. The space complexity is O(M), which is required for the recursion stack and the `path` list in the worst case when the tree is skewed.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach. Consider a binary tree structured like this, where each node's value represents a character ('a' to 'z'):

```
   b
  / \
 a   d
    / \
   c   e
```

This tree can be represented as the following with numerical values `[0, 25]` (where 'a' = 0, 'b' = 1, ..., 'z' = 25):

```
   1
  / \
 0   3
    / \
   2   4
```

We want to find the lexicographically smallest string from a leaf to the root.

Step 1: Initializing the smallest string `ans` as "{" (ASCII value just above 'z') to ensure any valid path is smaller than `ans` initially.

Step 2: We start DFS from the root node 'b' ('b' has a numerical value of 1).

Step 3: The `dfs` function is called with the root node 'b'. We add 'b' to the current path.

Step 4: The root node has two children. The `dfs` function is recursively called on the left child 'a'.

Step 5: Now at node 'a'. It is a leaf node, so we reverse the current path which gives us "ab". We compare it with `ans` and update `ans` to "ab" as it is lexicographically smaller.

Step 6: Backtracking occurs, and we remove 'a' from the path and return to the root node 'b'.

Step 7: Next, the `dfs` is recursively called on the right child 'd'.

Step 8: At node 'd', we continue the process by moving to its left child 'c', which is also a leaf. The path here is "bdc". Reversed, we get "cdb". This string is not smaller than "ab", so `ans` remains "ab".

Step 9: We backtrack again and go to the right child 'e', encountering the path "bde". When reversed, we get "edb". Again, this is not smaller than "ab".

Step 10: DFS completes after we have visited all leaf nodes. `ans` is "ab", which is the lexicographically smallest string from leaf to root.

Thus, the lexicographically smallest string from a leaf to the root is "ab".

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 smallestFromLeaf(self, root: TreeNode) -> str:
        # Initialize the answer variable with a dummy string value that is higher than any other valid strings
        smallest_string = '{'  # '{' is the character immediately after 'z' in ASCII

        def dfs(node, path):
            nonlocal smallest_string
            if node:
                # Convert current node's value to corresponding lowercase letter and prepend to path
                path.append(chr(ord('a') + node.val))

                # Check if current node is a leaf node (no children)
                if node.left is None and node.right is None:
                    # Create string from leaf to root and update smallest_string if necessary
                    current_string = ''.join(reversed(path))
                    smallest_string = min(smallest_string, current_string)

                # Continue depth-first search in left and right subtrees
                dfs(node.left, path)
                dfs(node.right, path)

                # Backtrack: remove the current node's character from path
                path.pop()

        # Start depth-first search from the root node
        dfs(root, [])
        return smallest_string

```

Java Solution

```java
/**
 * 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;
    }
}

class Solution {
    private StringBuilder currentPath; // StringBuilder to store the current path from leaf to root.
    private String smallestPath; // String to keep track of the smallest path encountered.

    public String smallestFromLeaf(TreeNode root) {
        currentPath = new StringBuilder();
        // Initialize the smallest path with a string value greater than any possible leaf path.
        smallestPath = String.valueOf((char) ('z' + 1));
        // Start the depth-first search from the root.
        depthFirstSearch(root);
        // Return the smallest path from leaf to root.
        return smallestPath;
    }

    private void depthFirstSearch(TreeNode node) {
        if (node != null) {
            // Prepend the character representation of the current node's value to the path.
            currentPath.append((char) ('a' + node.val));

            // Check if it's a leaf node.
            if (node.left == null && node.right == null) {
                // Reverse the path to get the leaf to root string.
                String leafToRootPath = currentPath.reverse().toString();

                // Update the smallest path if the current path is smaller.
                if (leafToRootPath.compareTo(smallestPath) < 0) {
                    smallestPath = leafToRootPath;
                }

                // Reverse the path back to maintain the root to leaf order.
                currentPath.reverse();
            }

            // Continue the depth-first search to the left subtree.
            depthFirstSearch(node.left);
            // Continue the depth-first search to the right subtree.
            depthFirstSearch(node.right);

            // Backtrack: remove the last character representing the current node's value.
            currentPath.deleteCharAt(currentPath.length() - 1);
        }
    }
}

```

C++ Solution

```cpp
#include <algorithm> // Include for the reverse function

// 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:
    string smallestLeafString = ""; // Holds the lexicographically smallest string from leaf to root.

    // Entry point to find the smallest string from leaf to root.
    string smallestFromLeaf(TreeNode* root) {
        string path = "";          // Represents the current path as a string.
        dfs(root, path);           // Start the DFS traversal.
        return smallestLeafString; // Return the smallest string after traversing the whole tree.
    }

    // Depth First Search (DFS) to traverse the tree.
    void dfs(TreeNode* node, string& path) {
        if (!node) return; // Base case: if the current node is nullptr, end the recursion.

        // Append the current character represented by the node's value to the path.
        path += 'a' + node->val;

        if (!node->left && !node->right) { // Check if it's a leaf node.
            string pathReversed = path;    // Create a copy of the current path to reverse it.
            reverse(pathReversed.begin(), pathReversed.end()); // Reverse the string to get the path from leaf to root.
            // Set smallestLeafString to the pathReversed if it's smaller than the current smallestLeafString or if smallestLeafStr
            if (smallestLeafString.empty() || pathReversed < smallestLeafString) {
                smallestLeafString = pathReversed;
            }
        }

        // Continue the DFS traversal for both children.
        dfs(node->left, path);
        dfs(node->right, path);

        // Backtrack: Remove the last character as we return to the previous node.
        path.pop_back();
    }
};

```

Typescript Solution

```typescript
// Import necessary functionality for reversing strings.
import { reverse } from "algorithm";

// Definition for a binary tree node.
class TreeNode {
    val: number;
    left: TreeNode | null;
    right: TreeNode | null;

    constructor(val: number = 0, left: TreeNode | null = null, right: TreeNode | null = null) {
        this.val = val;
        this.left = left;
        this.right = right;
    }
}

// Holds the lexicographically smallest string from leaf to root.
let smallestLeafString: string = "";

// Entry point to find the smallest string from leaf to root.
function smallestFromLeaf(root: TreeNode | null): string {
    // Represents the current path as a string.
    let path: string = "";
    // Start the DFS traversal.
    dfs(root, path);
    // Return the smallest string after traversing the whole tree.
    return smallestLeafString;
}

// Depth First Search (DFS) to traverse the tree.
function dfs(node: TreeNode | null, path: string): void {
    // Base case: if the current node is null, end the recursion.
    if (!node) return;

    // Append the current character represented by the node's value to the path.
    path += String.fromCharCode('a'.charCodeAt(0) + node.val);

    // Check if it's a leaf node.
    if (!node.left && !node.right) {
        // Create a copy of the current path.
        let pathReversed = path;
        // Reverse the string to get the path from leaf to root.
        pathReversed = reverse(pathReversed);
        // Set smallestLeafString to the pathReversed if it's smaller than the current smallestLeafString or if smallestLeafString is
        if (!smallestLeafString || pathReversed < smallestLeafString) {
            smallestLeafString = pathReversed;
        }
    }

    // Continue the DFS traversal for both children.
    if (node.left) dfs(node.left, path);
    if (node.right) dfs(node.right, path);

    // Backtrack: Remove the last character as we return to the previous node.
    path = path.substr(0, path.length - 1);
}

```

Time and Space Complexity

Time Complexity

The time complexity of the code is `O(n)`, where `n` is the number of nodes in the tree. This is because the depth-first search (DFS) algorithm implemented by the `dfs` function visits each node exactly once. During each visit, operations are performed that take constant time, such as appending to and popping from the path list, and making a comparison to update the `ans` variable when a leaf is encountered.

Space Complexity

The space complexity of the code can be considered as `O(h)`, where `h` is the height of the tree. This space is used by the recursion stack during the depth-first traversal. In the worst case (e.g., a skewed tree), the height of the tree could be `n`, leading to a space complexity of `O(n)`.

There's an additional space complexity involved with storing the path and the construction of the string for comparison, which becomes `O(h)` for the path list itself as it stores the characters representing the path from the root to the current node. The `reversed(path)` operation creates a temporary list to hold the reversed path, which takes `O(h)` space as well. However, since these are temporary and do not grow with `n`, the dominant factor remains the recursion stack, so the additional space is not typically accounted for in the overall space complexity calculation.