1028. Recover a Tree From Preorder Traversal

Hard · Tree · Depth-First Search · String · Binary Tree — Leetcode Link

Problem Description

The task is to reconstruct a binary tree given the preorder traversal string where each node in the string is represented by a number of dashes equivalent to the node's depth in the tree followed immediately by the node's value. In the provided string, the depth of the root node is 0, meaning it starts with no dashes, and successive nodes have increasing numbers of dashes depending on their depth. Importantly, if a node has only one child, that child is always the left child.

Intuition

In the solution, a stack data structure is used to keep track of the nodes and their respective depths, leveraging the Last In, First Out (LIFO) property to manage the tree's structure as we iterate through the input string. We parse the string character by character, counting dashes to determine the depth of the next node and concatenating numbers to form the value of each node. Whenever we reach the end of a number (indicated by the next character being a dash or end of string), a new tree node is created.

The node's position in the tree is determined by comparing its depth with the size of the stack (which represents the current path down the tree). If the current depth is less than the size of the stack, we pop from the stack until the top of the stack is the parent of the new node, ensuring proper node placement according to the preorder sequence. If the stack is not empty, we link the new node as either the left or right child of the current top node on the stack, depending on the left child's presence. Then, we push the new node onto the stack, which allows us to pop back to it when its children are processed, ensuring the tree structure is correctly maintained. Finally, after processing the entire string, the stack's top holds the root of the rebuilt binary tree, which is returned.

Solution Approach

The solution approach uses a `stack` to retrace the steps of the preorder depth-first traversal that created the given string. Here's a step-by-step walk-through of the algorithm:

1. Initialize an empty stack `st` to keep track of the nodes as they are created, representing the current path through the tree based on the depth-first nature of the traversal. Additionally, define the variables `depth` and `num` to store the current depth and the value of the node being processed, respectively.
2. Iterate through each character `i` in the given string `S`. If the current character is a dash `'-'`, increment the depth counter `depth`. If the current character is a digit, update the node value `num` by shifting the existing digits to the left (multiplying by 10) and adding the current digit's value.
3. If the end of the node's value string is reached (either the end of `S` is reached or the next character is a dash), a node needs to be created with the accumulated `num` value. This is done as follows:
   - Create a new tree node `newNode` with the value `num`.
   - Modify the stack to correspond to the proper depth by popping nodes until `st.size()` equals the current `depth`. This ensures that the top of the stack is the parent node of `newNode`.
   - If the stack is not empty, attach `newNode` as a left child if the top node's left child is `nullptr`, otherwise as a right child. This aligns with the rule that singly childed nodes are left children.
   - Push `newNode` onto the stack. This node will become a parent to subsequent nodes in the traversal or will be popped if we backtrack to a lesser depth.
4. After processing all characters, pop all remaining nodes from the stack until it's empty. The last node popped is the root node of the tree (`res`), which is the correct return for a successful reconstruction of the tree according to the preorder traversal string.
5. Return the root node `res`.

This algorithm relies on understanding how preorder traversal works and simulating this process in reverse. This stack-based approach correctly aligns with the preorder traversal's property that parents are visited before their children, and it handles single-child subtrees by design, as the left child rule is directly enforced by the algorithm. The use of the stack allows us to track the path through the tree and ensures proper parent-child relationships.

Example Walkthrough

Let's use a small example to illustrate the solution approach.

Suppose we are given the following preorder traversal string of a binary tree: `"1-2--3--4-5--6--7"`. We need to reconstruct the corresponding binary tree from this string.

Steps to reconstruct the binary tree:

1. Initialize an empty stack `st`.
2. Begin with the first character in the string, which has no dashes in front, so it represents the root node with the value `1`. Create this node and push it onto the stack.
3. Move on to the next character. We encounter a single dash and then the value `2`. Since the depth indicated by dashes is `1`, we create the node with the value `2`, and since the stack's current depth is also `1`, this node is a left child of the top node on the stack. Attach node `2` as a left child of node `1` and push node `2` onto the stack.
4. The following four characters indicate a node with value `3` and a depth of `2`. We pop node `2` off the stack since its depth is more than `1`, and add node `3` as the left child of the new top of the stack, which is node `2`. Push node `3` onto the stack.
5. Similarly, for the next node with a depth of `2` and value `4`, we pop node `3` off the stack and attach node `4` as the right child of node `2` (since the left child is already present). Push node `4` onto the stack.
6. Proceed with characters `'-5--6--7'`. We encounter node `5` with a depth of `1`, pop nodes until the stack depth matches `1`, and then attach node `5` as the right child of the root node `1`. Push node `5` onto the stack.
7. Node `6` has a depth of `2`, so we pop node `5` to match the depth. Since node `5` has no left child, we attach node `6` as the left child. Push node `6` onto the stack.
8. Lastly, we handle node `7`, which also has a depth of `2`. We pop node `6` and attach node `7` as the right child of node `5`. After pushing node `7` onto the stack, we finish processing the string.

The final step is to pop all remaining nodes from the stack until it is empty. The last node popped is the root node `1`.

The binary tree that corresponds to the string `"1-2--3--4-5--6--7"` is now successfully reconstructed and is represented as:

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

By following this process as outlined in the solution approach, we have moved character by character through the input string, maintained a current path through the tree with the stack, and updated parent-child relationships, finally arriving at the original binary tree.

Python Solution

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

class Solution:
    def recoverFromPreorder(self, traversal: str) -> TreeNode:
        # Create a stack to hold nodes and initialize depth and value variables.
        nodes_stack = []
        current_depth = 0
        current_value = 0

        # Loop through each character in the traversal string.
        for i in range(len(traversal)):
            if traversal[i] == '-':
                # Increment the depth for every '-' character encountered.
                current_depth += 1
            else:
                # Calculate the node's value.
                current_value = 10 * current_value + (ord(traversal[i]) - ord('0'))

            # Check for the end of number or end of string.
            if i + 1 == len(traversal) or (traversal[i].isdigit() and traversal[i + 1] == '-'):
                # Create a new node with the computed value.
                new_node = TreeNode(current_value)

                # If the stack size is greater than the current depth, pop until the sizes match.
                while len(nodes_stack) > current_depth:
                    nodes_stack.pop()

                # If the stack is not empty, assign the new node to the appropriate child of the top node.
                if nodes_stack:
                    if nodes_stack[-1].left is None:
                        nodes_stack[-1].left = new_node
                    else:
                        nodes_stack[-1].right = new_node

                # Push the new node onto the stack.
                nodes_stack.append(new_node)

                # Reset current depth and value for the next node.
                current_depth = 0
                current_value = 0

        # The result is the root of the tree. It's the bottommost node in the stack.
        result = None
        while nodes_stack:
            result = nodes_stack.pop()

        # Return the reconstructed binary tree.
        return result

```

Java Solution

```java
import java.util.Stack;

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

    TreeNode(int x) {
        val = x;
        left = null;
        right = null;
    }
}

class Solution {
    public TreeNode recoverFromPreorder(String traversal) {
        Stack<TreeNode> nodesStack = new Stack<>();
        int currentDepth = 0;
        int currentValue = 0;

        for (int i = 0; i < traversal.length(); ++i) {
            if (traversal.charAt(i) == '-') {
                // Increment the currentDepth for each '-' character encountered
                currentDepth++;
            } else {
                // Calculate the currentValue of the current node
                currentValue = 10 * currentValue + (traversal.charAt(i) - '0');
            }

            // Check for the end of the number or the end of the string
            if (i + 1 == traversal.length() || (Character.isDigit(traversal.charAt(i)) && traversal.charAt(i + 1) == '-')) {
                TreeNode newNode = new TreeNode(currentValue);

                // If the nodesStack size is greater than the currentDepth, we pop nodes until they match
                while (nodesStack.size() > currentDepth) {
                    nodesStack.pop();
                }

                // If nodesStack is not empty, we link the newNode as a child to the node at the top of the stack
                if (!nodesStack.isEmpty()) {
                    if (nodesStack.peek().left == null) {
                        nodesStack.peek().left = newNode;
                    } else {
                        nodesStack.peek().right = newNode;
                    }
                }

                // Push the newNode onto the stack
                nodesStack.push(newNode);

                // Reset currentDepth and currentValue for the next node
                currentDepth = 0;
                currentValue = 0;
            }
        }

        // Result is the root of the binary tree, which is the bottommost node in the nodesStack
        TreeNode result = null;
        while (!nodesStack.isEmpty()) {
            result = nodesStack.peek();
            nodesStack.pop();
        }

        // Return the recovered binary tree
        return result;
    }
}

```

C++ Solution

```cpp
#include <cctype>
#include <stack>
#include <string>

// Definition for a binary tree node
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};

class Solution {
public:
    TreeNode* recoverFromPreorder(std::string traversal) {
        std::stack<TreeNode*> nodesStack;
        int currentDepth = 0;
        int currentValue = 0;

        for (int i = 0; i < traversal.length(); ++i) {
            if (traversal[i] == '-') {
                // Increment the depth for every '-' character encountered
                currentDepth++;
            } else {
                // Calculate the node's value
                currentValue = 10 * currentValue + (traversal[i] - '0');
            }

            // Check for end of number or end of string
            if (i + 1 == traversal.length() || (isdigit(traversal[i]) && traversal[i + 1] == '-')) {
                TreeNode* newNode = new TreeNode(currentValue);

                // If the stack size is greater than the current depth, pop until the sizes match
                while (nodesStack.size() > currentDepth) {
                    nodesStack.pop();
                }

                // If stack is not empty, assign the newNode to the appropriate child of the top node
                if (!nodesStack.empty()) {
                    if (nodesStack.top()->left == nullptr) {
                        nodesStack.top()->left = newNode;
                    } else {
                        nodesStack.top()->right = newNode;
                    }
                }

                // Push the new node onto the stack
                nodesStack.push(newNode);

                // Reset current depth and value for the next node
                currentDepth = 0;
                currentValue = 0;
            }
        }

        // The result is the root of the tree. It's the bottommost node in the stack.
        TreeNode* result = nullptr;
        while (!nodesStack.empty()) {
            result = nodesStack.top();
            nodesStack.pop();
        }

        // Return the recovered binary tree
        return result;
    }
};

```

Typescript Solution

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

    constructor(val: number) {
        this.val = val;
        this.left = null;
        this.right = null;
    }
}

// Function to recover a tree from a given preorder traversal string
function recoverFromPreorder(traversal: string): TreeNode | null {
    let nodesStack: TreeNode[] = [];
    let currentDepth = 0;
    let currentValue = 0;

    for (let i = 0; i < traversal.length; ++i) {
        if (traversal[i] === '-') {
            // Increment the depth for every '-' character encountered
            currentDepth++;
        } else {
            // Calculate the node's value
            currentValue = 10 * currentValue + parseInt(traversal[i]);
        }

        // Check for end of number or end of string
        if (i + 1 === traversal.length || (traversal[i] !== '-' && traversal[i + 1] === '-')) {
            let newNode = new TreeNode(currentValue);

            // If the stack size is greater than the current depth, pop until the sizes match
            while (nodesStack.length > currentDepth) {
                nodesStack.pop();
            }

            // If stack is not empty, assign the newNode to the appropriate child of the top node
            if (nodesStack.length > 0) {
                if (nodesStack[nodesStack.length - 1].left === null) {
                    nodesStack[nodesStack.length - 1].left = newNode;
                } else {
                    nodesStack[nodesStack.length - 1].right = newNode;
                }
            }

            // Push the new node onto the stack
            nodesStack.push(newNode);

            // Reset current depth and value for the next node
            currentDepth = 0;
            currentValue = 0;
        }
    }

    // The result is the root of the tree. It's the last node added to the stack.
    while (nodesStack.length > 1) {
        nodesStack.pop();
    }

    // We return the root of the binary tree which the stack should now contain
    return nodesStack.length > 0 ? nodesStack[0] : null;
}

```

Time and Space Complexity

The given code traverses the string `S` once to reconstruct a binary tree from its preorder traversal description. The length of the string `S` is denoted as `n`.

Time Complexity

The time complexity depends on the number of operations performed as the string is being processed:

- Each character is visited once, which results in `O(n)` time complexity for the traversal.
- The stack operations (pushing and popping nodes) occur once for each node added to the tree. In the worst case, it's possible for every node to be pushed and then popped, but that is bounded by the number of nodes in the tree, and hence by `n`. Therefore, stack operations contribute `O(n)` to time complexity.

Overall, the time complexity of the function is `O(n)`.

Space Complexity

The space complexity primarily depends on the stack used to reconstruct the tree:

- The maximum size of the stack is determined by the maximum depth of the tree. In the worst case (a skewed tree), the depth can be `O(n)` if each node except the last has only one child. Therefore, the stack can use `O(n)` space.
- The space for the tree itself, if considered, would also be `O(n)` since we are creating a new node for every value in the string `S`.

Considering the stack and the space for the new tree nodes, the overall space complexity of the function is `O(n)`.