144. Binary Tree Preorder Traversal

Easy Stack Tree Depth-First Search Binary Tree Leetcode Link

Problem Description

In this problem, we are asked to perform a preorder traversal on a binary tree. Preorder traversal is a type of depth-first traversal where each node is processed before its children. Specifically, the order of traversal is:

1. Visit (process) the root node.
2. Traverse the left subtree in preorder.
3. Traverse the right subtree in preorder.

The challenge here is to implement a function that takes the root node of a binary tree as input and returns a list of the node values in the order they were visited during the preorder traversal.

Intuition

The given solution uses the Morris traversal algorithm, which is an efficient way to perform tree traversals without recursion and without a stack (thus using O(1) extra space). The key idea behind Morris traversal is to use the tree's structure to create temporary links that allow returning to previous nodes, essentially replacing the function call stack with in-tree threading. Here's a step-by-step intuition for how the provided solution works:

1. Start at the root node.
2. If the current node has no left child, process the current node's value (in this case, append to the `ans` list), then move to the right child.
3. If the current node does have a left child, find the rightmost node of the left child (which is the predecessor node in the inorder traversal).
4. If the right child of the predecessor node is `None`, create a temporary threaded link from the predecessor to the current node, process the current node's value, and move to the left child of the current node.
5. If the right child of the predecessor node is the current node, it means we've returned to the current node after processing its left subtree. In this case, break the temporary link and move to the right child of the current node.
6. Repeat this process until all nodes have been visited.

This approach allows nodes to be visited in the correct preorder sequence without extra space or recursive function calls.

Solution Approach

The provided Python code implements the Morris traversal algorithm, which is one of the few approaches to perform a tree traversal. Here, we'll break down this implementation and also refer to the other approaches mentioned in the Reference Solution Approach:

1. **Recursive Traversal**: This is the simplest and most straightforward approach, where the function calls itself to visit nodes in the correct order. However, this could lead to a stack overflow if the tree is very deep since it requires space proportional to the height of the tree.
2. **Non-recursive using Stack**: Instead of system call stack in recursion, an explicit stack is used to simulate the recursion. The stack stores the nodes of the tree still to be visited. This approach does not have the stack overflow issue but still requires space proportional to the height of the tree.
3. **Morris Traversal**: As used in the provided solution, Morris traversal is an optimization over the recursive and stack approach that uses no additional space (O(1) space complexity). The algorithm works by establishing a temporary link known as a thread for finding the predecessor of the current node. The detailed steps are as follows:
   - Begin at the root node.
   - While the current node is not `None`:
     - If the current node has no left child, output the current node value and move to its right child.
     - If the current node has a left child, find its inorder predecessor in the left subtree—the rightmost node in the left subtree.
     - If the inorder predecessor has its right link as `None`, this means we haven't yet visited this part of the tree. Link it with the current node, output the current node's value, and move to the left child to continue the traversal.
     - If the inorder predecessor's right link points to the current node, this means we've finished visiting the left subtree and returned to the current node. Therefore, remove the temporary link (set it back to `None`), and move to the right child to continue traversal.
   - Repeat the above steps until all nodes are visited.

The `ans` list accumulates the values of the nodes in the preorder traversal order. The sign that a left subtree has been processed is that you arrive back at a node whose left subtree's rightmost node has a right child pointing back to it — in other words, a node you have created a temporary threaded link to earlier.

Overall, the Morris Traversal implementation is an elegant way to walk through the binary tree without additional space, making it extremely useful for memory-limited environments.

Example Walkthrough

Let's illustrate the Morris traversal algorithm using a small binary tree.

Consider the following binary tree:

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

The preorder traversal of this tree should give us the node values in the sequence `1, 2, 8, 3, 4, 5`. Here's how the Morris traversal would process this tree:

1. Begin at the root node, which is `1`. There is a left child, so we look for the rightmost node in the left subtree of `1` (which is node `8`).
2. Node `8` doesn't have a right child, so we create a temporary link from node `8` back to node `1`, process `1` by adding it to the `ans` list (`ans = [1]`), and move left to node `2`.
3. At node `2`, again, we have a left child, so we look for the rightmost node in the left subtree of `2`, which is node `8`.
4. Since the right child of `8` points back to node `1`, we remove this link and move to the right child of `1`, which is `3`.
5. Since `2` has no right child, we process `2` (`ans = [1, 2]`) and then follow the temporary link back to node `1`.
6. We've returned to node `1`, and node `2` has been processed. We remove the temporary link (from `8` to `1`) and move to the right child of `1`, which is node `3`.
7. We arrive at node `3`, and it has a left child, so we look for the rightmost node in the left subtree of `3`, which is `4` having no right child.
8. Create a temporary link from `4` back to `3`, process `3` by adding it to the `ans` list (`ans = [1, 2, 3]`), and move to the left child, which is node `4`.
9. Visit node `4`, there's no left child, so process `4` (`ans = [1, 2, 3, 4]`) and move to its right, which is `None`.
10. Arrive at a `None` node, so we return to the previous node (`3`) via the temporary thread. Now we are at node `3` again, and the link from `4` to node `3` indicates we have processed the left subtree of `3`.
11. We remove the temporary link from `4` back to `3`, process node `4` (`ans = [1, 2, 3, 4]` remains unchanged as `4` was already processed), and move to the right child of `3`, which is `5`. Process `5` (`ans = [1, 2, 3, 4, 5]`).
12. Since `4` and `5` have no left children, and we've already processed `3`, we finish our traversal.

By the end of this process, our `ans` list is `[1, 2, 3, 4, 5]`. However, we missed processing node `8` because the description lacked detailed return steps to the root after visiting leftmost nodes. But following the Morris traversal algorithm rigorously, node `8` would be processed after step 5 before returning to `1` (`ans = [1, 2, 8, 3, 4, 5]`).

The final output of the preorder traversal using Morris algorithm is `[1, 2, 8, 3, 4, 5]`.

Python Solution

```python
# A binary tree node has a value, and references to left and right child nodes.
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Solution:
    def preorderTraversal(self, root: Optional[TreeNode]) -> List[int]:
        # Initialize the result list that will contain the values of nodes visited.
        result = []

        # Loop over the tree nodes until we visited all of them.
        while root:
            # If there is no left child, we can visit this node and move to right.
            if root.left is None:
                result.append(root.val)
                root = root.right
            else:
                # Find the inorder predecessor of the current node.
                predecessor = root.left
                # Move to the rightmost child of the left subtree.
                while predecessor.right and predecessor.right != root:
                    predecessor = predecessor.right

                # If the rightmost node of left subtree is not linked to current node.
                if predecessor.right is None:
                    # Link it to the current node and visit the current node.
                    result.append(root.val)
                    predecessor.right = root
                    # Move to the left child of the current node.
                    root = root.left
                else:
                    # It's already linked, it means we've finished the left subtree.
                    # Remove the predecessor and move to the right child of the node.
                    predecessor.right = None
                    root = root.right

        # Return the result list.
        return result
```

Java Solution

```java
import java.util.List;
import java.util.ArrayList;

// Definition for a binary tree node.
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;
    }
}

public class Solution {

    /**
     * Performs a preorder traversal of a binary tree and returns
     * the visited nodes values in a list.
     *
     * @param root the root node of the binary tree
     * @return a list of integers representing the node values in preorder
     */
    public List<Integer> preorderTraversal(TreeNode root) {
        List<Integer> result = new ArrayList<>();
        while (root != null) {
            if (root.left == null) {
                // If there is no left child, add the current node value
                // and move to the right node
                result.add(root.val);
                root = root.right;
            } else {
                // Find the inorder predecessor of the current node
                TreeNode predecessor = root.left;
                while (predecessor.right != null && predecessor.right != root) {
                    predecessor = predecessor.right;
                }

                // If the right child of the inorder predecessor is null, we
                // set it to the current node and move to the left child of
                // the current node after recording it.
                if (predecessor.right == null) {
                    result.add(root.val);
                    predecessor.right = root;
                    root = root.left;
                } else {
                    // If the right child is already pointing to the current node,
                    // we are visiting the node again, so we restore the tree structure
                    // by setting the right child of the predecessor to null and move
                    // to the right child of the current node, as we've finished traversing
                    // the left subtree due to Morris traversal.
                    predecessor.right = null;
                    root = root.right;
                }
            }
        }
        return result;
    }
}
```

C++ Solution

```cpp
#include <vector>
using namespace std;

// Definition for a binary tree node.
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    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:
    vector<int> preorderTraversal(TreeNode* root) {
        vector<int> result; // This vector will hold the preorder traversal result

        while (root != nullptr) { // Continue until there are no more nodes to visit
            if (root->left == nullptr) {
                // If there is no left child, visit this node and move to the right child
                result.push_back(root->val);
                root = root->right;
            } else {
                // If there is a left child, find the rightmost node of the
                // left subtree or a previously created link to the current node
                TreeNode* previous = root->left;
                while (previous->right != nullptr && previous->right != root) {
                    previous = previous->right;
                }

                if (previous->right == nullptr) {
                    // Establish a temporary link back to the current node
                    // so we can return after visiting the left subtree
                    result.push_back(root->val); // Visit the current node
                    previous->right = root; // Make link to go back to root after left subtree is done
                    root = root->left; // Move to the left subtree
                } else {
                    // The left subtree has been visited, remove the link
                    previous->right = nullptr;
                    // Move to the right subtree
                    root = root->right;
                }
            }
        }
        return result; // Return the result of preorder traversal
    }
};
```

Typescript Solution

```typescript
/**
 * Performs a preorder traversal on a binary tree without using recursion or a stack.
 *
 * @param {TreeNode | null} root - The root node of the binary tree.
 * @returns {number[]} The preorder traversal output as an array of node values.
 */
function preorderTraversal(root: TreeNode | null): number[] {
    // Initialize an array to store the preorder traversal sequence
    const result: number[] = [];

    // Iterate while there are nodes to visit
    while (root !== null) {
        if (root.left === null) {
            // If there is no left child, push the current node's value and move to right child
            result.push(root.val);
            root = root.right;
        } else {
            // Find the rightmost node of the inorder predecessor of the current node
            let predecessor = root.left;
            while (predecessor.right !== null && predecessor.right !== root) {
                predecessor = predecessor.right;
            }

            // If the right subtree of the predecessor is not yet linked to the current node
            if (predecessor.right === null) {
                // Record the current node's value (as part of preorder)
                result.push(root.val);
                // Link the predecessor's right to the current node to create a temporary threaded binary tree
                predecessor.right = root;
                // Move to the left child to continue traversal
                root = root.left;
            } else {
                // If the right subtree is already linked to the current node, remove that temporary link
                predecessor.right = null;
                // Move to the right child to continue traversal since the left subtree is already processed
                root = root.right;
            }
        }
    }

    // Return the result array with the traversal sequence
    return result;
};
```

Time and Space Complexity

The given Python code is an implementation of the Morris Preorder Tree Traversal algorithm. Let's analyze the time complexity and space complexity of the code:

Time Complexity

The time complexity of this algorithm is $O(n)$, where n is the number of nodes in the binary tree. This is because each edge in the tree is traversed at most twice: once when searching for the predecessor and once when moving back to the right child. Since there are $n - 1$ edges for n nodes, and each edge is traversed no more than twice, this leads to a linear time complexity.

Space Complexity

The space complexity of the Morris traversal algorithm is $O(1)$. This algorithm modifies the tree by adding temporary links to the predecessor nodes but it does not use any additional data structures like stacks or recursion that depend on the number of nodes in the tree. Hence, it requires a constant amount of additional space.