145. Binary Tree Postorder Traversal

Easy Stack Tree Depth-First Search Binary Tree Leetcode Link

Problem Description

Given a binary tree, the task is to perform a postorder traversal and return the values of the nodes. In postorder traversal, we visit the nodes in the following order:

1. Traverse the left subtree in postorder.
2. Traverse the right subtree in postorder.
3. Visit the root node.

The challenge is to write a function that systematically visits each node in this specific order and collects the node values along the way.

Intuition

To solve this problem, you can consider three main solution approaches: recursive traversal, iterative traversal using a stack, and Morris traversal. The code provided above uses the Morris traversal method.

Recursion is the most straightforward approach where the function calls itself to traverse left and right subtrees before processing the root. It is implemented easily but uses extra space due to the function call stack.

The iterative approach with a stack emulates the recursive behavior without the need for function calls by maintaining a stack to keep track of nodes. It requires carefully managing the stack to ensure nodes are visited in the postorder sequence.

Morris traversal is a more complex but space-efficient method, as it doesn't require a stack or recursion, thus using constant space. It uses a threaded binary tree concept by creating temporary links known as threads. The intuition here is to link each node's predecessor (its rightmost child in the left subtree) back to itself, allowing traversal of the tree without additional space for a stack or recursion.

In the code provided:

- The outer `while` loop is iterating over the tree nodes.
- If a node doesn't have a right child, we capture its value and move to its left child.
- If it has a right child, we find its predecessor and:
  - If the predecessor's `left` link is `None`, we link it back to the root (creating a temporary thread) and move to the right subtree.
  - If the predecessor's `left` link is already pointing to the root, we are visiting the root a second time, and hence, we break the temporary thread and move to the left subtree.

After exiting the loop, the `ans` list contains the values in a modified postorder sequence, and reversing it (`ans[::-1]`) gives the correct postorder traversal result.

This approach is more difficult to conceptualize, but it gives the elegance of an iterative solution without using additional space for a stack or the call stack, which is required in recursive solutions.

Solution Approach

The solution uses Morris traversal, which is a space-efficient traversal method:

1. Initialize an empty list `ans` to store the postorder traversal nodes' values.

2. Start with the `root` node of the tree. We will use a `while` loop that runs as long as the `root` is not `None`.

3. Inside the loop, if the `root`'s right child is `None`, we add the `root`'s value to the `ans` list, then we update the `root` to be its left child, as per the postorder sequence (left, right, root). Since there's no right subtree, we go left.

4. If the `root` has a right child, we need to find the `root`'s predecessor which would be the rightmost node of the left subtree.

   - We assign `next` to `root.right` and enter another `while` loop searching for a `next.left` that is not `None` and does not equal to `root`.
   - This loop is essentially moving `next` to the rightmost node in the left subtree of the `root`.

5. After finding the predecessor, we check its `left` attribute:

   - If `next.left` is not equal to `root`, it means we haven't set up a temporary thread from the predecessor back to the `root`.
     - We record the `root`'s value in `ans`.
     - Then we set `next.left` to `root`, creating a temporary thread.
     - Move `root` to its right subtree for the next iteration.
   - If `next.left` is equal to `root`, it means this is our second visit to the `root` after visiting its right subtree, and we must remove the temporary thread.
     - Set `next.left` back to `None` to restore the tree's structure.
     - Go to the next node by moving `root` to its left subtree.

6. This process continues until the `root` becomes `None`, signifying that we have visited all nodes.

7. The values stored in `ans` are in reverse order of the intended postorder sequence. To correct the order, we return `ans[::-1]`, a reverse of the list, which results in the correct postorder node values.

This implementation performs the postorder traversal without using recursion or a stack, using constant extra space, which makes it highly space-efficient. The patterns and operations used—like modifying the tree structure temporarily and then restoring it—are central to the Morris traversal algorithm.

The algorithm iteratively follows the postorder sequence but constructs the answer list in reverse. This algorithm depends heavily on exploiting the tree structure in a novel way, linking and unlinking nodes as it goes.

Example Walkthrough

Let's use a small binary tree as an example to illustrate the solution approach with Morris traversal for postorder:

Consider the following binary tree:

```
    A
   / \
  B   C
 / \
D   E
```

Now let's walk through the Morris postorder traversal:

1. We start at the root `A`. `A` has a right child, so we will look for the predecessor of `A` which is the rightmost node in the left subtree `(B)`.

2. We find that `B` has a right child `E`. We keep traversing to the right until we find that `E`'s right is `None` and `E` is not already linked back to `A`. We link `E` back to `A` (setting `E.left` to `A`) and move to the right child of `A`, which is `C`.

3. `C` has no right child, so we go straight to adding `C` to our list `ans` and traverse to its left, but since `C` is a leaf node, we jump back to `A` using the temporary thread from `E`.

4. Back at `A`, we now cut the thread by setting `E.left` back to `None`. We add `A` to `ans` and move to its left to `B`.

5. `B` also has a right child `E`, so we would go through creating a thread from `B's` successor `D` (since `D` is rightmost and has no right child) back to `B` and move to the right to `E`.

6. At `E`, we add it to `ans` (as `E` has no right child) and go left to `D`.

7. No right child for `D`, so we add `D` to `ans` and should move to the left, but `D` is a leaf node.

The sequence in `ans` at each step will be:

1. Starting with an empty list `ans = []`.
2. Visit `C` so `ans = [C]`.
3. Visit `A` so `ans = [C, A]`.
4. Visit `E` so `ans = [C, A, E]`.
5. Visit `D` so `ans = [C, A, E, D]`.

Now we reverse `ans` because we collected the values in a modified postorder:

- `ans[::-1]` will give us `[D, E, A, C]`.

However, there's one missing link here, which is `B`. This is not seen in the order above because of Morris traversal's nature of revisiting nodes; we only add nodes under certain conditions (e.g., when a right child doesn't exist or on a second visit to a node). A full implementation of the algorithm would successfully include `B` in the final list, ensuring that all nodes are traversed postorder. The correct reverse postorder ('ans' before reversing) will be `[C, E, D, B, A]`.

So, the final postorder traversal list after reversing is `[A, B, D, E, C]`, which aligns with the postorder sequence of left-right-root.

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 postorderTraversal(self, root: Optional[TreeNode]) -> List[int]:
        # Initialize an empty list to store the traversal result.
        traversal_result = []

        # Iterate while there are nodes to process.
        while root:
            # If there is no right child, process the current node and move to the left child.
            if root.right is None:
                traversal_result.append(root.val)
                root = root.left
            else:
                # Find the rightmost child of the left subtree or the leftmost previous node that we have already visited.
                predecessor = root.right
                while predecessor.left and predecessor.left != root:
                    predecessor = predecessor.left

                # If we have not set this relationship before, set it now and move to the right child.
                if predecessor.left != root:
                    traversal_result.append(root.val)
                    predecessor.left = root
                    root = root.right
                else:
                    # If we have previously visited this node (which indicates the link we created on `predecessor.left`),
                    # Restore the tree's structure by removing the temporary link and move to the left child.
                    predecessor.left = None
                    root = root.left

        # Since nodes are added to traversal_result in reverse postorder (root, right, left),
        # reverse the list to obtain the correct postorder (left, right, root) sequence.
        return traversal_result[::-1]

```

Java Solution

```java
class Solution {
    public List<Integer> postorderTraversal(TreeNode root) {
        // Initialize an empty linked list that will store the postorder traversal elements.
        // Using LinkedList with the addFirst method for efficient insertions at the beginning.
        LinkedList<Integer> result = new LinkedList<>();

        // Iterate while there are nodes to process
        while (root != null) {
            // If there is no right child, process the current node and go left
            if (root.right == null) {
                // Insert the current node's value at the beginning of the list
                result.addFirst(root.val);
                // Move to the left child
                root = root.left;
            } else {
                // Find the leftmost node of the right child
                TreeNode pre = root.right;
                while (pre.left != null && pre.left != root) {
                    pre = pre.left;
                }
                // Creating a temporary thread from right subtree's leftmost node back to root
                if (pre.left == null) {
                    // Add current node's value to the beginning of the list
                    result.addFirst(root.val);
                    // Make a temporary connection back to the root
                    pre.left = root;
                    // Move to the right child
                    root = root.right;
                } else {
                    // If there is already a temporary thread, remove it
                    pre.left = null;
                    // Move to the left child of the current node
                    root = root.left;
                }
            }
        }
        // Return the result of the postorder traversal
        return result;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <algorithm> // For 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:
    // Function to perform a postorder traversal of a binary tree
    std::vector<int> postorderTraversal(TreeNode* root) {
        std::vector<int> result; // Holds the postorder traversal result
        // Loop until there are no nodes to process
        while (root) {
            // If the right subtree does not exist, process the current node and move to the left subtree
            if (!root->right) {
                result.push_back(root->val);
                root = root->left;
            } else {
                // Find the rightmost node of the left subtree or the left child of the previous processed node
                TreeNode* next = root->right;
                while (next->left && next->left != root) {
                    next = next->left;
                }

                // Establish a temporary link so we can return to the current node after traversing its right subtree
                if (!next->left) {
                    result.push_back(root->val); // Process the current node
                    next->left = root; // Establish the temporary link
                    root = root->right; // Move to the right subtree
                } else {
                    next->left = nullptr; // Remove the temporary link
                    root = root->left; // Move to the left subtree since the right subtree has been processed
                }
            }
        }
        // Reverse the results because the nodes were visited in reverse postorder
        std::reverse(result.begin(), result.end());
        return result;
    }
};

```

Typescript Solution

```typescript
// A TreeNode class definition would normally be here,
// but per instructions, I've omitted the class definition.

function postorderTraversal(root: TreeNode | null): number[] {
    // If the tree is empty, return an empty array
    if (!root) return [];

    // Initialize a stack to keep track of nodes
    let stack: TreeNode[] = [];
    // Initialize an array to store the postorder traversal result
    let result: number[] = [];
    // Previous visited node
    let previous: TreeNode | null = null;

    // Iterate while there are still nodes to process
    while (root || stack.length > 0) {
        // Reach the leftmost node of the current subtree
        while (root) {
            stack.push(root);
            root = root.left;
        }
        // Peek at the node from the top of the stack
        root = stack.pop()!;

        // If the right subtree is already visited or doesn't exist,
        // process the current node
        if (!root.right || root.right === previous) {
            result.push(root.val);
            // Mark the current node as visited
            previous = root;
            // Reset root to null to indicate node processing is done
            root = null;
        } else {
            // If right subtree exists, push the current node back to
            // the stack, and move to the right subtree
            stack.push(root);
            root = root.right;
        }
    }

    // Return the result of the postorder traversal
    return result;
}

```

Time and Space Complexity

The given code is attempting to perform a postorder traversal of a binary tree without using recursion. Taking a closer look:

- The time complexity of the 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 finding the inorder predecessor and once to revert the structure of the tree. The traversal ensures that each node is also processed only once.

- The space complexity of the code is `O(1)`, assuming that the output list does not count towards the space complexity as it is part of the required output. This is because the algorithm utilizes the tree's existing structure to traverse it by temporarily modifying the nodes' `left` pointers and then restoring them to their original structure, meaning no additional significant space is required other than a few pointers for manipulating the nodes.

Note: If the output list is considered in the space complexity, then the space complexity becomes `O(n)`, since we need to store every node's value in the list.