590. N-ary Tree Postorder Traversal

Easy Stack Tree Depth-First Search Leetcode Link

Problem Description

The given problem requires us to conduct a postorder traversal on a given n-ary tree and return a list of the values of its nodes. In postorder traversal, we visit the nodes in a left-right-root order for binary trees, which extends to child1-child2-...-childN-root order in the case of n-ary trees, where N can be any number of children a node can have.

N-ary trees differ from regular binary trees as they can have more than two children. In the serialization of such a tree for level order traversal, each set of children is denoted, and then a `null` value is used to separate them, indicating the end of one level and the beginning of another.

Intuition

To solve this problem, we can utilize a stack to simulate the postorder traversal by considering the following points:

1. A stack data structure follows last in, first out (LIFO) principle, which can be leveraged to visit nodes in the required postorder.
2. We start by pushing the root node onto the stack.
3. When popping from the stack, we add the node's value to the answer list and then push all of its children onto the stack. This push operation is done in the original children order, which means when we pop them, they will be in reversed order because of the LIFO property of the stack.
4. The traversal continues until the stack is empty.
5. After we've traversed all nodes and have our answer list, it will be in root-childN-...-child1 order. To obtain the postorder traversal, we need to reverse this list before returning it. The reversal step is essential to have the nodes in the child1-child2-...-childN-root order.

By following the described steps, we visit each node's children before visiting the node itself (as per postorder traversal rules) by effectively leveraging the stack's characteristics to traverse the n-ary tree non-recursively.

Solution Approach

The implementation of the postorder traversal for an n-ary tree is done using a stack to keep track of the nodes. The following steps break down the solution approach:

1. Initialization:
   - We define a list named `ans` to store the postorder traversal.
   - Then, we check if the given `root` is `None` and immediately return `ans` if true, as there are no nodes to traverse.
2. Stack Preparation:
   - We initialize a stack named `stk` and push the root node into it.
3. Traversal:
   - We enter a `while` loop that continues as long as `stk` is not empty.
   - Inside the loop, we pop the topmost node from the stack and append its value to list `ans`.
   - We then iterate over the children of the popped node in their given order and push each onto the stack `stk`.
   - Since the stack follows LIFO order, children are appended to the list `ans` in reverse order with respect to how they were added to the stack.
4. Final Step:
   - Upon completion of the `while` loop (when the stack `stk` is empty), we have all the node values in `ans` but in the reverse order of the required postorder traversal.
   - To fix this, we simply reverse `ans` using `ans[::-1]` and then return it.

By using this approach, we avoid recursion, which can be helpful in environments where the call stack size is limited. The use of a stack allows us to control the processing order of nodes, thereby enabling the simulation of post-order traversal in an iterative fashion.

Example Walkthrough

Let's illustrate the solution approach with a simple n-ary tree example:

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

In this tree, node `1` is the root, node `2` has one child `5`, node `3` has no children, node `4` has two children `6` and `7`.

Following the solution steps:

1. **Initialization**:
   - Define a list `ans` to store the postorder traversal.
   - Check if the root is `None`. If it is, return `ans`.
2. **Stack Preparation**:
   - Initialize a stack `stk` and push the root node `1` onto it.
3. **Traversal**:
   - Start the while loop since `stk` is not empty.
   - Pop the topmost node from `stk`, which is node `1`. Add its value to list `ans`.
   - Push the children of node `1` (`4`, `3`, `2`) onto the stack. Note: Push in reverse order of the children as stated in the Intuition, so node `2` is on top.
   - Pop node `2` from `stk`, add its value to `ans`, and push its child onto the stack (`5`).
   - Pop node `5` from `stk`, add its value to `ans`. No children to add here.
   - Pop node `3` from `stk`, add its value to `ans`. No children to add here.
   - Pop node `4` from `stk`, add its value to `ans`, and push its children (`7`, `6`) onto the stack.
   - Pop node `7` from `stk`, add its value to `ans`. No children to add here.
   - Pop node `6` from `stk`, add its value to `ans`. No children to add here.
   - Now the `stk` is empty, exit the while loop.
4. **Final Step**:
   - At this point, `ans` is in reverse order of the required post-order traversal: `[1, 4, 7, 6, 3, 2, 5]`.
   - Reverse `ans` to get the correct postorder traversal: `[5, 2, 6, 7, 4, 3, 1]`.
   - Return the reversed list.

So using the stated steps, we successfully performed an iterative postorder traversal on the n-ary tree and the final output is `[5, 2, 6, 7, 4, 3, 1]`.

Python Solution

```python
class Node:
    def __init__(self, val=None, children=None):
        self.val = val
        self.children = children if children is not None else []


class Solution:
    def postorder(self, root: 'Node') -> list[int]:
        # List to store the postorder traversal result
        postorder_result = []

        # If the tree is empty, return an empty list
        if root is None:
            return postorder_result

        # Stack to keep track of nodes to visit
        stack = [root]

        # Continue processing nodes until stack is empty
        while stack:
            # Pop a node from the stack
            current_node = stack.pop()

            # Add the current node's value to the traversal result
            postorder_result.append(current_node.val)

            # Push all children of the current node to the stack
            # Note: We push children in the order as they are in the list so that they are processed
            # in reverse order when popped (because stack is LIFO), which is needed for postorder.
            for child in current_node.children:
                stack.append(child)

        # Return the result reversed, as we need to process children before their parent
        # This reverse operation gives us the correct order of postorder traversal
        return postorder_result[::-1]

```

Java Solution

```java
import java.util.Deque;       // Import the Deque interface
import java.util.LinkedList;  // Import the LinkedList class
import java.util.List;        // Import the List interface

class Solution {
    public List<Integer> postorder(Node root) {
        LinkedList<Integer> result = new LinkedList<>(); // Use 'result' instead of 'ans' for clarity
        if (root == null) {
            // If the root is null, return an empty list
            return result;
        }

        Deque<Node> stack = new ArrayDeque<>(); // Use 'stack' to represent the deque of nodes
        stack.offerLast(root); // Start with the root node

        while (!stack.isEmpty()) {
            // While there are nodes in the stack
            Node current = stack.pollLast(); // Get the last node
            result.addFirst(current.val); // Add the node value to the front of the result list (for postorder)

            // Iterate through children of the current node
            for (Node child : current.children) {
                stack.offerLast(child); // Add each child to the stack for processing
            }
        }
        return result; // Return the populated list as the result
    }
}

// Definition for a Node.
class Node {
    public int val;
    public List<Node> children;

    // Constructors
    public Node() {}

    public Node(int _val) {
        val = _val;
    }

    public Node(int _val, List<Node> _children) {
        val = _val;
        children = _children;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <stack>
#include <algorithm>

// Definition for a Node.
class Node {
public:
    int val;
    std::vector<Node*> children;

    Node() {}

    Node(int value) {
        val = value;
    }

    Node(int value, std::vector<Node*> childNodes) {
        val = value;
        children = childNodes;
    }
};

class Solution {
public:
    std::vector<int> postorder(Node* root) {
        // Initialize an empty vector to store the postorder traversal.
        std::vector<int> postorderTraversal;

        // Return empty vector if the root is null.
        if (!root) return postorderTraversal;

        // Use a stack to hold the nodes during traversal.
        std::stack<Node*> stack;
        stack.push(root);

        // Loop until the stack is empty.
        while (!stack.empty()) {
            // Get the top node from the stack.
            root = stack.top();
            stack.pop();

            // Append the node's value to the traversal vector.
            postorderTraversal.push_back(root->val);

            // Push all children of the current node to the stack.
            // We traverse from left to right, the stack is LIFO, so we need to reverse
            // the postorder by reversing the relation at the end.
            for (Node* child : root->children) {
                stack.push(child);
            }
        }
        // Reverse the order of the nodes to get the correct postorder.
        std::reverse(postorderTraversal.begin(), postorderTraversal.end());

        // Return the postorder traversal.
        return postorderTraversal;
    }
};

```

Typescript Solution

```typescript
// Definition for the Node class. (Do not create a Node class since the instruction is to define everything globally)
interface INode {
    val: number;
    children: INode[];
}

/**
 * Post-order traversal function which returns an array of values.
 * Post-order traversal means visiting the child nodes before the parent node.
 * @param root - The root node of the tree or null if the tree is empty.
 * @returns An array of node values in post-order.
 */
function postorder(root: INode | null): number[] {
    // An array to store the result of the post-order traversal
    const result: number[] = [];

    /**
     * Helper function for depth-first search post-order traversal of the node tree.
     * It recursively visits children first before the parent node then pushes the value to the result array.
     * @param node - The current node being visited.
     */
    const depthFirstSearchPostOrder = (node: INode | null) => {
        if (node == null) {
            // If the node is null, i.e., either the tree is empty or we've reached the end of a branch, return
            return;
        }

        // Loop through each child of the current node and recursively perform a post-order traversal
        for (const childNode of node.children) {
            depthFirstSearchPostOrder(childNode);
        }

        // After visiting the children, push the current node's value to the result array
        result.push(node.val);
    };

    // Initiate the post-order traversal from the root of the tree
    depthFirstSearchPostOrder(root);

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

```

Time and Space Complexity

The given code implements a post-order traversal of an n-ary tree iteratively using a stack. Analyzing the complexity:

- **Time Complexity:** Each node is visited exactly once, and all its children are added to the stack. Assuming the tree has `N` nodes, and let `C` be the maximum number of children a node can have. In the worst case, every node will be pushed and popped from the stack once, and all of its children (up to `C` children) will be processed. Therefore, the time complexity is `O(N * C)`. However, since every node is visited once, and C is a constant factor, the time complexity simplifies to `O(N)`.

- **Space Complexity:** The space complexity is determined by the size of the stack and the output list. In the worst case, if the tree is imbalanced, the stack could hold all nodes in one of its branches. Therefore, the space complexity is `O(N)` for storing the `N` nodes in the worst-case scenario. The list `ans` also stores `N` node values. Hence, when considering the output list, the total space complexity remains `O(N)`.

Therefore, the overall time complexity of the code is `O(N)` and the space complexity is `O(N)`.