589. N-ary Tree Preorder Traversal

Easy Java Tree Depth-First Search

Leetcode Link

Problem Description

The problem requires writing a function that performs a preorder traversal of an n-ary tree and returns a list of the values of the nodes visited. In an n-ary tree, each node can have zero or more children, unlike binary trees where each node has at most two children. The 'preorder' part of the request means that the traversal should follow this order: visit the current node first, and then proceed recursively with the preorder traversal of the children, from left to right, if any. The input serialized n-ary tree representation uses level order traversal where each group of children is separated by the value `null`.

Intuition

The intuition behind solving a tree traversal problem, especially for preorder traversal, is to follow the pattern of visiting each node before its children. For an n-ary tree, we need to visit the current node, then visit each of the children in order. This approach naturally suggests using recursion or iteration with a stack to maintain the order of nodes to visit.

We choose an iterative approach using a stack to avoid potential issues with recursion, such as stack overflow when dealing with very deep trees. We start by pushing the root node onto the stack. Then, as long as the stack is not empty, we pop the top element from the stack, which is the current node to visit, and append its value to the result list. Next, we need to ensure that we visit the children of the current node in the correct order. Since a stack is a last-in, first-out (LIFO) structure, we reverse the children before pushing them onto the stack to maintain the left-to-right traversal order.

This way, when we continue to the next iteration of the while loop, we pop the nodes in preorder (parent before children, leftmost children first).

Solution Approach

The solution to the n-ary tree preorder traversal problem involves an iterative approach using a stack data structure. In computer science, a stack is a collection that follows the last-in, first-out (LIFO) principle. The algorithm proceeds as follows:

1. Initialize an empty list named `ans` to store the preorder traversal result.
2. If the root of the tree is `None`, return the empty list immediately, as there is nothing to traverse.
3. Start by creating a stack named `stk` and push the root node onto it.
4. While the stack is not empty, execute the following steps:
   - Pop the node at the top of the stack. This node is the current node being visited.
   - Append the value of the current node to the `ans` list.
   - Iterate through the children of the current node in reverse order. Reversing is crucial because we want the leftmost child to be processed first, and the stack, being LIFO, would process the last pushed item first.
   - For each child in the reversed children list, push the child onto the stack to visit it in subsequent iterations.

By repeating this process, we traverse the tree in a preorder fashion, visiting each node and its children in the correct order. The algorithm terminates when the stack is empty, which happens after we have visited all the nodes in the tree.

The key algorithm pattern used here is the iterative depth-first search (DFS). This pattern is particularly suited for preorder traversal because it allows us to visit the current node before exploring each subtree rooted at its children.

The time complexity of this algorithm is O(N), where N is the number of nodes in the n-ary tree, because every node and child relationship is examined exactly once. The space complexity is also O(N), assuming the worst-case scenario where the tree is heavily unbalanced and resembles a linked list, the stack would hold all nodes.

Here is the implementation of the algorithm referenced above:

```
class Node:
    def __init__(self, val=None, children=None):
        self.val = val
        self.children = children

class Solution:
    def preorder(self, root: 'Node') -> List[int]:
        ans = []
        if root is None:
            return ans
        stk = [root]
        while stk:
            node = stk.pop()
            ans.append(node.val)
            for child in node.children[::-1]:
                stk.append(child)
        return ans
```

Example Walkthrough

Let's consider a small n-ary tree example to illustrate how the solution approach works. Here we have an n-ary tree with the following structure:

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

The expected preorder traversal of this tree is [1, 2, 5, 6, 3, 7, 4].

According to our algorithm:

1. We initialize `ans` as an empty list.
2. The root is not `None`, so we proceed by creating a stack `stk` with the root node (1) in it.

Now the `stk` looks like `[1]`.

3. As long as `stk` is not empty, we continue with the following steps:

Step 1:

- We pop the top of the stack, so `node` is `1`.
- We append `1` to `ans`, so now `ans` is `[1]`.
- We push the children of node `1` which are `4`, `3`, and `2` onto the stack in reverse order. The reversal ensures that we will visit the leftmost child (node `2`) next.

Now `stk` looks like `[2, 3, 4]`.

Step 2:

- We pop the top of the stack, so `node` is `2`.
- We append `2` to `ans`, so now `ans` is `[1, 2]`.
- We push the children of node `2` which are `1` and `5` onto the stack in reverse order.

Now `stk` looks like `[5, 6, 3, 4]`.

Step 3:

- We pop the top of the stack, so `node` is `5`.
- We append `5` to `ans`, so now `ans` is `[1, 2, 5]`.
- Node `5` has no children, so we don't push anything onto the stack.

Now `stk` looks like `[6, 3, 4]`.

Step 4:

- We pop the top of the stack, so `node` is `6`.
- We append `6` to `ans`, so now `ans` is `[1, 2, 5, 6]`.
- Node `6` has no children, so we don't push anything onto the stack.

Now `stk` looks like `[3, 4]`.

Step 5:

- We pop the top of the stack, so `node` is `3`.
- We append `3` to `ans`, so now `ans` is `[1, 2, 5, 6, 3]`.
- We push the child of node `3` which is `7` onto the stack.

Now `stk` looks like `[7, 4]`.

Step 6:

- We pop the top of the stack, so `node` is `7`.
- We append `7` to `ans`, so now `ans` is `[1, 2, 5, 6, 3, 7]`.
- Node `7` has no children, so we don't push anything onto the stack.

Now `stk` looks like `[4]`.

Step 7:

- We pop the top of the stack, so `node` is `4`.
- We append `4` to `ans`, so now `ans` is `[1, 2, 5, 6, 3, 7, 4]`.
- Node `4` has no children, so no further action is necessary.

Finally, `stk` is empty, so we finish the loop. The `ans` list now contains the preorder traversal of our tree: `[1, 2, 5, 6, 3, 7, 4]`. This is exactly the order we expected.

Python Solution

```
class Node:
    """
    Definition for a node in an N-ary tree.
    Each node contains a value and a list of its children.
    """
    def __init__(self, val=None, children=None):
        self.val = val
        self.children = children if children is not None else []

class Solution:
    def preorder(self, root: 'Node') -> List[int]:
        """
        Perform preorder traversal on an N-ary tree and return a list of node values.

        :type root: Node
        :rtype: List[int]
        """
        # Initialize an empty list to hold the traversal result.
        traversal_result = []

        # If the tree is empty, return the empty result immediately.
        if not root:
            return traversal_result

        # Initialize a stack with the root node to begin traversal.
        stack = [root]

        # Process nodes from the stack until it's empty.
        while stack:
            # Pop the top node from the stack.
            current_node = stack.pop()

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

            # Reverse iterate through the current node's children and add them to the stack.
            # Reversing ensures that the children are processed in the original order for preorder traversal.
            stack.extend(reversed(current_node.children))

        # Once all nodes are processed, return the traversal result.
        return traversal_result
```

Java Solution

```
import java.util.ArrayDeque;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Deque;
import java.util.List;

// Definition for a Node in an N-ary tree.
class Node {
    public int val;
    public List<Node> children;

    public Node() {}

    // Constructor with node value
    public Node(int val) {
        this.val = val;
    }

    // Constructor with node value and list of children
    public Node(int val, List<Node> children) {
        this.val = val;
        this.children = children;
    }
};

class Solution {

    // Preorder Traversal for an N-ary tree
    public List<Integer> preorder(Node root) {
        // If the root is null, return an empty list
        if (root == null) {
            return Collections.emptyList();
        }

        // List to store the preorder traversal
        List<Integer> result = new ArrayList<>();

        // Use a stack to track the nodes for traversal
        Deque<Node> stack = new ArrayDeque<>();

        // Initialize the stack with the root node
        stack.push(root);

        // Iterate while the stack is not empty
        while (!stack.isEmpty()) {
            // Pop the current node from the stack
            Node currentNode = stack.pop();

            // Add the value of the current node to the result list
            result.add(currentNode.val);

            // Retrieve the children of the current node
            List<Node> childrenList = currentNode.children;

            // Iterate over the children in reverse order so that they
            // are visited in the correct order when subsequently popped from the stack
            for (int i = childrenList.size() - 1; i >= 0; --i) {
                stack.push(childrenList.get(i));
            }
        }

        // Return the final result containing the preorder traversal
        return result;
    }
}
```

C++ Solution

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

// Definition for a Node.
class Node {
public:
    int val; // Holds the value of the node.
    vector<Node*> children; // Vector of pointers to child nodes.

    // Constructor initializes the node with default values.
    Node() {}

    // Constructor initializes the node with a value.
    Node(int _val) {
        val = _val;
    }

    // Constructor initializes the node with a value and the children.
    Node(int _val, vector<Node*> _children) {
        val = _val;
        children = _children;
    }
};

class Solution {
public:
    // This method performs a preorder traversal of an n-ary tree.
    vector<int> preorder(Node* root) {
        if (!root) return {}; // If root is null, return an empty vector.

        vector<int> result; // The vector to store the preorder traversal.
        stack<Node* > nodes; // Stack to use for iterative traversal.
        nodes.push(root); // Start with the root node.

        // Iterate as long as there are nodes on the stack.
        while (!nodes.empty()) {
            Node* current = nodes.top(); // Take the top node from the stack.
            result.push_back(current->val); // Process the current node's value.
            nodes.pop(); // Remove the processed node from stack.

            // Reverse iterate over the node's children and add them to the stack.
            // This ensures that children are processed in the original order.
            for (int i = current->children.size() - 1; i >= 0; --i) {
                nodes.push(current->children[i]);
            }
        }

        return result; // Return the completed preorder traversal vector.
    }
};
```

Typescript Solution

```
// Node class type definition with value and children properties
class Node {
    val: number
    children: Node[];
}

/**
 * Performs a preorder traversal on an n-ary tree.
 * @param root - The root of the n-ary tree.
 * @returns An array of node values in preorder sequence.
 */
function preorder(root: Node | null): number[] {
    // The array to store the preorder traversal
    let traversalResult: number[] = [];

    /**
     * A helper function to perform a depth-first search (DFS) for the preorder traversal.
     * @param currentNode - The current node being traversed in the n-ary tree.
     */
    const dfs = (currentNode: Node | null) => {
        if (currentNode === null) {
            return;
        }
        // Add the current node's value to the result array
        traversalResult.push(currentNode.val);
        // Recursively call dfs for each of the children of the current node
        currentNode.children.forEach(child => {
            dfs(child);
        });
    };

    // Initiate the DFS with the root node of the tree
    dfs(root);
    // Return the result array containing the preorder traversal
    return traversalResult;
}
```

Time and Space Complexity

The time complexity of the given preorder traversal algorithm is $O(N)$, where N is the total number of nodes in the tree. This is because the algorithm visits each node exactly once.

The space complexity is also $O(N)$, which in the worst-case scenario is the space needed for the stack when the tree degenerates into a linked list (e.g., each node has only one child). However, in the average case where the tree is more balanced, the space complexity would be proportional to the height of the tree, which could be much less than N.