Problem Description

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 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

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

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

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

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).

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:

3. Start by creating a stack named stk and push the root node onto it.

Solution Approach

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.

o 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.

2. If the root of the tree is None, return the empty list immediately, as there is nothing to traverse.

1. Initialize an empty list named ans to store the preorder traversal result.

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.

heavily unbalanced and resembles a linked list, the stack would hold all nodes.

Here is the implementation of the algorithm referenced above:

def preorder(self, root: 'Node') -> List[int]:

for child in node.children[::-1]:

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

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

self.children = children

ans.append(node.val)

stk.append(child)

class Solution:

14

15

16

17

ans = []

return ans

Example Walkthrough

following structure:

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

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

if root is None: return ans stk = [root] while stk: node = stk.pop() 13

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

• 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

```
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].
```

Step 1:

Step 2:

Step 3:

Step 4:

Step 5:

leftmost child (node 2) next. Now stk looks like [2, 3, 4].

We pop the top of the stack, so node is 1.

We pop the top of the stack, so node is 2.

We append 1 to ans, so now ans is [1].

 We append 2 to ans, so now ans is [1, 2]. • We push the children of node 2 which are 6 and 5 onto the stack in reverse order.

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

 Node 5 has no children, so we don't push anything onto the stack. Now stk looks like [6, 3, 4].

We pop the top of the stack, so node is 6.

We pop the top of the stack, so node is 3.

We pop the top of the stack, so node is 7.

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.

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.

We append 6 to ans, so now ans is [1, 2, 5, 6].

We pop the top of the stack, so node is 5.

We append 5 to ans, so now ans is [1, 2, 5].

Now stk looks like [3, 4].

Node 6 has no children, so we don't push anything onto the stack.

Step 6:

Now stk looks like [7, 4].

Now stk looks like [4].

is exactly the order we expected.

Python Solution

class Node:

class Solution:

 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.

9

10

12

13

14

15

16

17

18

26

27

28

29

32

33

34

35

36

37

38

Java Solution

class Node {

import java.util.ArrayDeque;

import java.util.Collections;

// Definition for a Node in an N-ary tree.

public List<Node> children;

2 import java.util.ArrayList;

import java.util.Deque;

import java.util.List;

public int val;

public Node() {}

Step 7:

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 []

:type root: Node

:rtype: List[int]

stack = [root]

while stack:

traversal_result = [] 20 21 22 # If the tree is empty, return the empty result immediately. 23 if root is None: 24 return traversal_result 25

Pop the top node from the stack.

current_node = stack.pop()

Process nodes from the stack until it's empty.

traversal_result.append(current_node.val)

Initialize an empty list to hold the traversal result.

Initialize a stack with the root node to begin traversal.

Append the current node's value to the traversal result.

Reverse iterate through the current node's children and add them to the stack.

def preorder(self, root: 'Node') -> List[int]:

Perform preorder traversal on an N-ary tree and return a list of node values.

for child in reversed(current_node.children): 39 stack.append(child) # Once all nodes are processed, return the traversal result. 43 return traversal_result 44

Reversing ensures that the children are processed in the original order for preorder traversal.

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

```
13
       // Constructor with node value
14
16
17
```

10

11

12

```
public Node(int val) {
           this.val = val;
18
       // Constructor with node value and list of children
19
       public Node(int val, List<Node> children) {
20
           this.val = val;
22
           this.children = children;
23
24 }
25
   class Solution {
27
28
       // Preorder traversal for an N-ary tree
29
       public List<Integer> preorder(Node root) {
           // If the tree is empty, return an empty list
           if (root == null) {
32
               return Collections.emptyList();
33
34
           // List to store the preorder traversal
35
36
           List<Integer> result = new ArrayList<>();
37
           // Use a stack to track the nodes for traversal
38
           Deque<Node> stack = new ArrayDeque<>();
39
40
           // Initialize the stack with the root node
41
42
           stack.push(root);
43
           // Iterate while the stack is not empty
44
45
           while (!stack.isEmpty()) {
               // Pop the top node from the stack
46
               Node currentNode = stack.pop();
47
48
49
               // Add the value of the current node to the result list
                result.add(currentNode.val);
50
51
               // Retrieve the children of the current node
52
53
               List<Node> childrenList = currentNode.children;
54
55
               // Iterate over the children in reverse order so that they
56
               // are visited in the correct order when subsequently popped from the stack
57
               for (int i = childrenList.size() - 1; i >= 0; --i) {
                    stack.push(childrenList.get(i));
58
59
60
61
           // Return the final result containing the preorder traversal
62
           return result;
63
64 }
65
C++ Solution
```

38 Node* current = nodes.top(); // Take the top node from the stack. 39 result.push_back(current->val); // Process the current node's value. nodes.pop(); // Remove the processed node from stack. 40 42 // Reverse iterate over the node's children and add them to the stack. // This ensures that children are processed in the original order. 43

Typescript Solution

interface Node {

1 #include <vector>

using namespace std;

Node() {}

class Solution {

public:

// Definition for a Node.

Node(int _val) {

val = _val;

val = _val;

children = _children;

vector<int> preorder(Node* root) {

while (!nodes.empty()) {

int val; // Holds the value of the node.

Node(int _val, vector<Node*> _children) {

vector<Node*> children; // Vector of pointers to child nodes.

// Constructor initializes the node with a value and the children.

// This method performs a preorder traversal of an n-ary tree.

nodes.push(root); // Start with the root node.

// Iterate as long as there are nodes on the stack.

nodes.push(current->children[i]);

// Node class type definition with value and children properties

if (!root) return {}; // If root is null, return an empty vector.

vector<int> result; // The vector to store the preorder traversal.

for (int i = current->children.size() - 1; i >= 0; --i) {

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

stack<Node*> nodes; // Stack to use for iterative traversal.

// Constructor initializes the node with default values.

// Constructor initializes the node with a value.

2 #include <stack>

6 class Node {

7 public:

10

11

12

13

14

15

16

17

18

19

20

21

23

25

28

29

30

31

32

33

34

35

36

37

45

46

47

48

49

51

50 };

24 };

```
val: number;
       children: Node[];
 6
   /**
    * 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
13
       let traversalResult: number[] = [];
14
15
       /**
16
        * 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.
19
        */
       const dfs = (currentNode: Node | null) => {
20
           if (currentNode === null) {
21
               return;
23
           // Add the current node's value to the result array
25
           traversalResult.push(currentNode.val);
           // Recursively call dfs for each of the children of the current node
26
           currentNode.children.forEach(childNode => {
               dfs(childNode);
28
           });
29
30
       };
31
32
       // Initiate the DFS with the root node of the tree
33
       dfs(root);
       // Return the result array containing the preorder traversal
34
       return traversalResult;
35
36 }
37
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.