Documentation on N-ary Tree Level Order Traversal

Table of Contents

1. Problem Statement

Given an **N-ary tree**, return the **level order traversal** of its nodes' values. An **N-ary tree** is a tree in which a node can have at most **N children**.

Example 1

Input Tree:

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

Input Representation: root = [1, null, 3, 2, 4, null, 5, 6]

Output: [[1], [3, 2, 4], [5, 6]]

Example 2

Input Tree:

```
   1
 / | | \
2  3 4  5
 / \   |
6   7  8
    |
    9
```

Output: [[1], [2, 3, 4, 5], [6, 7, 8], [9]]

2. Intuition

- The problem requires **level-order traversal**, which means **visiting nodes level by level**.
- The best way to achieve this is by using **Breadth-First Search (BFS)** since it naturally processes nodes in level order.

3. Key Observations

- **Nodes at the same level must be grouped together** in the output list.
- A **queue (FIFO structure)** is ideal for **processing nodes level by level**.
- Each node's **children should be enqueued** before moving to the next level.
- **Edge cases like an empty tree** should be handled.

4. Approach

i. **Base Case:** If the tree is empty (root == None), return an empty list [].
ii. **Initialize a queue** and add the root node.
iii. **Loop until the queue is empty**:
   a. **Process all nodes at the current level**:

- Remove a node from the front of the queue.
- Store its value in a list.
- Add all its children to the queue.

b. Append the level's list to the final result.

iv. **Return the final result list**.

5. Edge Cases

- **Empty Tree:** If root is None, return [].
- **Single Node:** If the tree has only root, return [[root.val]].
- **Tree with Varying Child Counts:** Some nodes may have **no children**, while others have **many children**.
- **Deep Trees:** The height can be up to 1000, so we need to ensure **no stack overflow**.
- **Large Number of Nodes:** The number of nodes can be 10^4, so **efficient traversal is necessary**.

6. Complexity Analysis

Time Complexity

- Each node is visited **once**, and each child is processed **once**.
- Thus, the complexity is **O(N)**, where N is the total number of nodes.

Space Complexity

- The queue **stores at most one level** of nodes at a time.
- In the worst case (when the tree is balanced), the last level has **approximately N/k nodes** (where k is the average branching factor).
- **Worst-case space complexity:** O(N).

7. Alternative Approaches

1. Recursive Approach (DFS)

- Perform **Depth-First Search (DFS)** and store values level-wise.
- Requires **extra recursion depth**, leading to **stack overflow** for deep trees.
- **Time Complexity:** O(N)
- **Space Complexity:** O(H) (tree height)

2. Using a Dictionary for Levels

- Instead of a queue, use a **dictionary {level: [nodes]}** to store nodes at each level.
- Requires **two passes** (one for traversal, one for extraction).
- **Less efficient** than BFS.

8. Test Cases

Test Case 1: Basic Example

Tree: [1, null, 3, 2, 4, null, 5, 6]
Expected Output: [[1], [3, 2, 4], [5, 6]]

Test Case 2: Deep Tree

Tree: [1, null, 2, 3, 4, 5, null, 6, 7, null, 8, null, 9, 10, null, null, 11, null, 12, null, 13, null, null, 14]
Expected Output: [[1], [2,3,4,5], [6,7,8,9,10], [11,12,13], [14]]

Test Case 3: Single Node

Tree: [1]
Expected Output: [[1]]

Test Case 4: Empty Tree

Tree: []
Expected Output: []

9. Final Thoughts

- BFS is the **best approach** for level-order traversal due to **queue efficiency**.
- DFS can be used, but **not ideal** for deep trees due to **recursion depth limits**.
- Always consider **edge cases** such as **empty trees** and **large depth**.
- The problem is **straightforward but requires careful implementation** to handle large inputs efficiently.

🚀 **Final Verdict:** BFS + Queue is the **optimal** way to solve this problem efficiently!