428. Serialize and Deserialize N-ary Tree

Leetcode Link

Problem Explanation:

We need to serialize and then deserialize an N-ary tree. Here serialization means to convert the tree structure into a string so we can store it into a file or transmit it across a network, and then bring it back into its original structure through deserialization.

For example, let us consider a 3-ary tree:

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

5 6
```

We may serialize this into string: "1 3 2 4 5 6"

Now, the task is to convert this string back into its original tree structure through deserialization.

The problem states that there is no restriction on how the serialization/deserialization algorithm should work.

The only constraints are:

- N lies in the range of 1 to 1000.
- We should not use class member/global/static variables to store states.

Solution Approach:

The solution given here uses Breadth-First Search (BFS) to traverse the tree while serializing and deserializing the tree. We start with the root of the tree and form a queue to store the nodes. We pop nodes from the queue to check the children of the node and push the children into the queue. This is done until the queue is empty (i.e., all nodes are traversed).

Let's go through the example mentioned above:

- Serialize:

1. We start with the root node and add it to a queue
2. We keep on popping nodes from the queue, check its children, and add children into the queue. As we go through each node, we add its value into the serialized string. If a node has no children, we just add "n" to the string.
3. This is repeated until the queue is empty and all nodes are traversed.

- Deserialize:

1. Start by creating the root node from the first element from the serialized string and add it into a queue.
2. Next, go through each child of the node, create a node for each child and add it to the node's children.
3. Each child node is also added to the queue to be checked for its children.
4. This is repeated until the serialized string is completely traversed.

To see this in a more detailed form, we present a solution in Python below.

Python Solution:

```python

python
class Node(object):
    def __init__(self, val=0, children=0):
        self.val = val
        self.children = children

class Codec:
    def serialize(self, root):
        if root is None:
            return []

        queue = [root]
        result = [root.val]  # add root value

        while queue:
            node = queue.pop(0)
            if node is None:
                continue

            for child in node.children:
                queue.append(child)
            result.append(len(node.children))  # add count of children
            result.extend([child.val for child in node.children])  # add children values

        return result

    def deserialize(self, data):
        if not data:
            return None

        root = Node(data[0])  # get root from first index
        data = data[1:]  # remove root from data
        queue = [root]

        while queue:
            node = queue.pop(0)
            if node is None:
                continue

            for _ in range(data.pop(0)):  # check children count
                child = Node(data.pop(0))  # get child value
                node.children.append(child)
                queue.append(child)

        return root
```

In the above code, we first create a queue and push the root into the queue. While the queue is not empty, we pop a node from the queue and push the children of the current node into the queue. Here we also add the important information related to the node like its value and the count of children to the result array.

While deserializing, we start by popping nodes from the data and creating nodes for them. We keep popping as many nodes as the children count for the current node and add them to the node's children and the queue.

We repeat this process until all nodes are processed.## JavaScript Solution:

```
javascript
class Node {
  constructor(val, children) {
    this.val = val;
    this.children = children;
  }
}

class Codec {
  serialize(root) {
    if (root == null) return [];
    let queue = [root];
    let result = [root.val];
    while (queue.length > 0) {
      let node = queue.shift();
      if (node == null) continue;
      for (let child of node.children) {
        queue.push(child);
      }
      result.push(node.children.length);
      result.push(...node.children.map(child => child.val));
    }
    return result;
  }

  deserialize(data) {
    if (data.length == 0) return null;
    let root = new Node(data[0], []);
    data = data.slice(1);
    let queue = [root];
    while (queue.length > 0) {
      let node = queue.shift();
      if (node == null) continue;
      let count = data.shift();
      for (var i = 0; i < count; i++) {
        let child = new Node(data.shift(), []);
        node.children.push(child);
        queue.push(child);
      }
    }
    return root;
  }
}
```

This JavaScript code mimics the approach of the Python solution. It uses a queue to store nodes and performs serialization and deserialization operations.

Java Solution:

```
java
import java.util.*;

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

class Codec {
    public String serialize(Node root) {
        if (root == null) return null;
        Queue<Node> queue = new LinkedList<>();
        queue.offer(root);
        StringBuilder sb = new StringBuilder();
        sb.append(root.val).append(" ");
        while (!queue.isEmpty()) {
            Node node = queue.poll();
            if (node == null) continue;
            for (Node child : node.children) {
                queue.offer(child);
            }
            sb.append(node.children.size()).append(" ");
            for (Node child : node.children) {
                sb.append(child.val).append(" ");
            }
        }
        return sb.toString();
    }

    public Node deserialize(String data) {
        if (data == null) return null;
        String[] arr = data.split(" ");
        Node root = new Node(Integer.parseInt(arr[0]), new ArrayList<>());
        Queue<Node> queue = new LinkedList<>();
        queue.offer(root);
        int i = 1;
        while (!queue.isEmpty()) {
            Node node = queue.poll();
            int count = Integer.parseInt(arr[i++]);
            for (int j = 0; j < count; j++) {
                Node child = new Node(Integer.parseInt(arr[i++]), new ArrayList<>());
                node.children.add(child);
                queue.offer(child);
            }
        }
        return root;
    }
}
```

In the Java solution, we use the StringBuilder class for string concatenation to improve program performance. The rest of the logic is similar to the Python and JavaScript approach, using a queue to process the nodes and using the dequeuing operation to serialize/deserialize the tree.