1490. Clone N-ary Tree

Medium Tree Depth-First Search Breadth-First Search Hash Table Leetcode Link

Problem Description

In this problem, we are given the root of an N-ary tree and asked to create a deep copy of it. A deep copy means that we should create an entirely new tree, where each node is a new instance with the same values as the corresponding nodes in the original tree. In other words, modifying the new tree should not affect the original tree.

An N-ary tree is a tree in which a node can have zero or more children. This differs from a binary tree where each node has at most two children. Each node in an N-ary tree holds a value and a list of nodes that represent its children.

The tree is represented using a custom `Node` class. The `Node` class consists of two attributes:

- `val`: an integer representing the value of the node.
- `children`: a list of child nodes.

We are to perform the deep copy without altering the structure or values of the original tree.

Intuition

The intuition behind solving the deep copy problem for an N-ary tree lies in understanding tree traversal and the idea of creating new nodes as we traverse. Since we need to create a new structure that is identical to the original tree, we will use a Depth-First Search (DFS) traversal. DFS allows us to visit each node, starting from the root, and proceed all the way down to its children recursively before backtracking.

Here's the breakdown of the approach:

1. Start from the root of the tree. If the root is `None`, meaning the tree is empty, return `None` as there is nothing to copy.
2. For a non-empty tree, create a new root node for the cloned tree with the same value as the original root.
3. Recursively apply the same clone process for all the children of the root node. Create a list of cloned children by iterating through the original node's children and applying the cloning function on each of them.
4. Assign the list of cloned children to the new root node's `children` attribute.
5. Once the recursion ends, we will have a new root node with its entire subtree cloned.

The code snippet provided uses this recursive approach for cloning each node. The recursion naturally handles the depth-first traversal of the tree and cloning of sub-trees rooted at each node's children.

Solution Approach

The provided code snippet implements the deep copy of an N-ary tree using a straightforward recursive strategy, which aligns with the DFS (Depth-First Search) method indicated in the Reference Solution Approach. DFS is a fundamental algorithm used in tree traversal, where we explore as far as possible along each branch before backtracking.

Here's a walk-through of the solution approach, highlighting the algorithm, data structures, and patterns involved:

1. Definition of the `cloneTree` function:
   - It takes one parameter, `root`, which represents the root of the N-ary tree we want to clone.
   - The function is designed to return a new `Node` that is the root of the cloned tree.
2. Base Case:
   - The recursion starts by checking if the `root` is `None`. If so, it returns `None` because an empty tree cannot be copied.
3. Recursive Case:
   - If the `root` is not `None`, the function creates a clone of the `root` node's list of children:
   - `children = [self.cloneTree(child) for child in root.children]`
     - This line iterates over each child of the current `root` node and applies the `cloneTree` function recursively, creating a deep copy of each subtree rooted at every child.
   - A new `Node` instance is created using the original node's value, `root.val`, and the list of cloned children, `children`. This line effectively clones the current `root` node and its entire subtree.
4. Data Structure Usage:
   - A `list` comprehension is used to succinctly clone all the children for a given node and construct a list of these cloned children.
   - The `Node` class is central to the solution. New instances of `Node` are created to form the cloned tree, and they are dynamically linked together through the `children` attribute to mimic the structure of the original tree.
5. Recursion:
   - The recursive call structure ensures that the DFS is executed correctly. It clones the nodes in a depth-first manner, starting at the root, down to the leaves, and then backtracking to cover all nodes.
6. Pattern:
   - The pattern here is a typical DFS recursion pattern tailored to handle N-ary trees as opposed to binary trees.

By iteratively applying this cloning process to each node, starting from the root and moving depth-first into each branch of the tree, the algorithm successfully constructs a deep copy of the entire N-ary tree. The recursive approach, while elegant and simple, takes advantage of the call stack to keep track of the nodes yet to be visited and the children list to maintain the tree structure in the cloned tree.

Example Walkthrough

Let's illustrate the solution approach using a small example of an N-ary tree. Consider the following tree structure:

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

Here, node 1 is the root with three children 2, 3, and 4. Node 2 has two children 5 and 6. The node class for each of these would look something like this:

- Node(1, [Node(2), Node(3), Node(4)])
- Node(2, [Node(5), Node(6)])

Now, let's walk through the steps in our algorithm to create a deep copy of this tree:

1. We call `cloneTree(root)` where `root` is Node(1).

2. Since `root` is not `None`, we proceed to clone its children. We iterate over the children [Node(2), Node(3), Node(4)].

3. First, we clone Node(2). Since it has children [Node(5), Node(6)], we:

   - Recursively call `cloneTree` on Node(5), which has no children. We create a new instance Node(5) and return it.
   - Recursively call `cloneTree` on Node(6), also creating a new instance Node(6) and return it.
   - With both Node(5) and Node(6) cloned, we create a new instance Node(2) with the cloned children and return it.
4. Next, Node(3) is cloned. It has no children, so we simply create a new instance of Node(3) and return it.

5. Finally, Node(4) is also cloned without children, resulting in a new Node(4).

6. Now that all children of the root have been cloned, we create a new root Node(1) with the cloned children [Node(2), Node(3), Node(4)].

7. The cloning process of the entire tree is complete, and we return the new root Node(1). This root points to the entirely cloned N-ary tree.

The cloned tree structure is now as follows, and modifying this new tree has no impact on the original tree:

```
      1 (new)
    / | \
   2  3  4 (new for each)
  / \
 5   6 (new for each)
```

At each step, recursive calls ensure that an entirely new node instance is created for every node in the original tree. The list comprehensions and Node class instances ensure that the tree structure is preserved in the deep copy process.

Through the intuition and the steps highlighted above, one can grasp how the depth-first recursive approach facilitates an efficient deep copy of an N-ary tree.

Python Solution

```python
class Node:
    def __init__(self, value=None, children=None):
        """
        Node structure for N-ary tree with optional value and children list arguments.
        :param value: value of the node, defaulted to None
        :param children: list of child nodes, defaulted to empty list if None
        """
        self.value = value
        self.children = [] if children is None else children


class Solution:
    def cloneTree(self, root: 'Node') -> 'Node':
        """
        Clones an N-ary tree.
        :param root: The root node of the tree to clone.
        :return: The root node of the cloned tree.
        """
        # If the root is None, return None to handle the empty tree case
        if root is None:
            return None

        # Use list comprehension to recursively clone each subtree rooted at the children of the current node
        cloned_children = [self.cloneTree(child) for child in root.children]

        # Create a clone of the current node with the cloned children
        cloned_node = Node(root.value, cloned_children)

        return cloned_node

```

Java Solution

```java
class Solution {
    // This method creates a deep copy of a tree with nodes having an arbitrary number of children.
    public Node cloneTree(Node root) {
        // If the current node is null, return null because there is nothing to clone.
        if (root == null) {
            return null;
        }

        // Initialize a list to hold the cloned children nodes.
        ArrayList<Node> clonedChildren = new ArrayList<>();

        // Recursively clone all the children of the current node.
        for (Node child : root.children) {
            clonedChildren.add(cloneTree(child));
        }

        // Create a new node with the same value as the current node and the list of cloned children.
        return new Node(root.val, clonedChildren);
    }
}

// Definition for a Node.
class Node {
    public int val; // Variable to store the node's value.
    public List<Node> children; // List to store the node's children.

    // Constructor to initialize the node with no children.
    public Node() {
        children = new ArrayList<Node>();
    }

    // Constructor to initialize the node with a value and no children.
    public Node(int val) {
        this.val = val;
        children = new ArrayList<Node>();
    }

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

```

C++ Solution

```cpp
class Solution {
public:
    // This function clones an N-ary tree starting at the root node.
    Node* cloneTree(Node* root) {
        // If the root is nullptr, there's nothing to clone; return nullptr.
        if (!root) {
            return nullptr;
        }

        // Create an empty vector to hold the cloned children.
        std::vector<Node*> clonedChildren;

        // Iterate through each child of the root node.
        for (Node* child : root->children) {
            // Recursively clone each child and add the cloned child to the clonedChildren vector.
            clonedChildren.push_back(cloneTree(child));
        }

        // Create and return a new node with the cloned value and cloned children.
        return new Node(root->val, clonedChildren);
    }
};

```

Typescript Solution

```typescript
// Define the structure for a Node in TS, which includes a value and an array of children Nodes.
class Node {
    public val: number;
    public children: Node[];

    constructor(val: number, children: Node[] = []) {
        this.val = val;
        this.children = children;
    }
}

// This function clones an N-ary tree starting at the root node.
function cloneTree(root: Node | null): Node | null {
    // If the root is null, there's nothing to clone; return null.
    if (!root) {
        return null;
    }

    // Create an empty array to hold the cloned children.
    let clonedChildren: Node[] = [];

    // Iterate through each child of the root node.
    for (let child of root.children) {
        // Recursively clone each child and add the cloned child to the clonedChildren array.
        clonedChildren.push(cloneTree(child));
    }

    // Create and return a new Node with the cloned value and cloned children.
    return new Node(root.val, clonedChildren);
}

```

Time and Space Complexity

Time Complexity

The time complexity of the cloneTree function is `O(N)`, where `N` is the total number of nodes in the tree. This is because the function visits each node exactly once to create its clone.

Space Complexity

The space complexity of the function is also `O(N)` in the worst case. This space is required for the call stack due to recursion, which could go as deep as the height of the tree in the case of a skewed tree. Additionally, space is needed to store the cloned tree, which also contains `N` nodes.