1490. Clone N-ary Tree

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

Problem Description

An N-ary <u>tree</u> 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 <u>tree</u> is represented using a custom <u>Node</u> class. The <u>Node</u> class consists of two attributes:

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

• val: an integer representing the value of the node.

The intuition behind solving the deep copy problem for an N-ary tree lies in understanding tree traversal and the idea of creating

children: a list of child nodes.

tree. In other words, modifying the new tree should not affect the original tree.

- We are to perform the deep copy without altering the structure or values of the original <u>tree</u>.
- ntuition

new nodes as we traverse. Since we need to create a new structure that is identical to the original tree, we will use a <u>Depth-First</u> <u>Search</u> (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 <u>tree</u>. If the root is <u>None</u>, meaning the tree is empty, return <u>None</u> 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.

node's children and applying the cloning function on each of them.

Assign the list of cloned children to the new root node's children attribute.
 Once the recursion ends, we will have a new root node with its entire subtree cloned.

tree traversal, where we explore as far as possible along each branch before backtracking.

• It takes one parameter, root, which represents the root of the N-ary tree we want to clone.

• If the root is not None, the function creates a clone of the root node's list of children:

together through the children attribute to mimic the structure of the original tree.

The function is designed to return a new Node that is the root of the cloned tree.

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

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

The code snippet provided uses this recursive approach for cloning each node. The recursion naturally handles the depth-first

1. Definition of the cloneTree function:

Base Case:

The recursion starts by checking if the root is None. If so, it returns None because an empty tree cannot be copied.
 Recursive Case:

A new Node instance is created using the original node's value, root.val, and the list of cloned children, children. This line effectively

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

• The recursive call structure ensures that the DFS is executed correctly. It clones the nodes in a depth-first manner, starting at the root,

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

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

4. Data Structure Usage:

Recursion:

each subtree rooted at every child.

clones the current root node and its entire subtree.

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

down to the leaves, and then backtracking to cover all nodes.

3. 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:

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:

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.

Node(4)].

1 (new)

class Node:

class Solution:

Java

class Solution {

2 3 4 (new for each)

efficient deep copy of an N-ary tree.

self.value = value

Clones an N-arv tree.

if root is None:

return cloned_node

public Node cloneTree(Node root) {

for (Node child : root.children) {

children = new ArrayList<Node>();

def init__(self, value=None, children=None):

def cloneTree(self, root: 'Node') -> 'Node':

:param value: value of the node, defaulted to None

self.children = [] if children is None else children

:param root: The root node of the tree to clone.

cloned_node = Node(root.value, cloned_children)

ArrayList<Node> clonedChildren = new ArrayList<>();

clonedChildren.add(cloneTree(child));

return new Node(root.val, clonedChildren);

public int val; // Variable to store the node's value.

// Constructor to initialize the node with no children.

// Recursively clone all the children of the current node.

public List<Node> children; // List to store the node's children.

// Constructor to initialize the node with a value and no children.

// Define the structure for a Node in TS, which includes a value and an array of children Nodes.

// Recursively clone each child and add the cloned child to the clonedChildren array.

Node structure for N-ary tree with optional value and children list arguments.

:param children: list of child nodes, defaulted to empty list if None

If the root is None, return None to handle the empty tree case

cloned children = [self.cloneTree(child) for child in root.children]

Create a clone of the current node with the cloned children

:return: The root node of the cloned tree.

N-ary tree.

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

comprehensions and Node class instances ensure that the tree structure is preserved in the deep copy process.

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),

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

With both Node(5) and Node(6) cloned, we create a new instance Node(2) with the cloned children and return it.

Next, Node(3) is cloned. It has no children, so we simply create a new instance of Node(3) and return it.

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

Node structure for N-ary tree with optional value and children list arguments.

:param children: list of child nodes, defaulted to empty list if None

If the root is None, return None to handle the empty tree case

Create a clone of the current node with the cloned children

// This method creates a deep copy of a tree with nodes having an arbitrary number of children.

// Create a new node with the same value as the current node and the list of cloned children.

// If the current node is null, return null because there is nothing to clone.

Recursively call cloneTree on Node(6), also creating a new instance Node(6) and return it.

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

Solution Implementation

Python

Through the intuition and the steps highlighted above, one can grasp how the depth-first recursive approach facilitates an

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]

return null; } // Initialize a list to hold the cloned children nodes.

// Definition for a Node.

public Node() {

class Node {

if (root == null) {

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++ 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); **}**;

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

class Node:

class Solution:

TypeScript

class Node {

public val: number;

if (!root) {

return null;

public children: Node[];

this.val = val;

this.children = children;

let clonedChildren: Node[] = [];

for (let child of root.children) {

self.value = value

Clones an N-arv tree.

if root is None:

return None

constructor(val: number, children: Node[] = []) {

function cloneTree(root: Node | null): Node | null {

// This function clones an N-ary tree starting at the root node.

// Create an empty array to hold the cloned children.

// Iterate through each child of the root node.

clonedChildren.push(cloneTree(child));

def init__(self, value=None, children=None):

def cloneTree(self, root: 'Node') -> 'Node':

:param value: value of the node, defaulted to None

self.children = [] if children is None else children

:param root: The root node of the tree to clone.

cloned_node = Node(root.value, cloned_children)

:return: The root node of the cloned tree.

// If the root is null, there's nothing to clone; return null.

Time and Space Complexity

Time Complexity

Use list comprehension to recursively clone each subtree rooted at the children of the current node

function visits each node exactly once to create its clone.

tree, which also contains N nodes.

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

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