1379. Find a Corresponding Node of a Binary Tree in a Clone of That Tree

Easy Tree Depth-First Search Breadth-First Search Binary Tree

Leetcode Link

Problem Description

The problem presents two identical binary trees, `original` and `cloned`, meaning that the `cloned` tree is an exact copy of the `original` one. Along with these trees, you are given a reference to a node, `target`, within the `original` tree. The task is to find and return a reference to the node in the `cloned` tree that corresponds exactly to the `target` node in the `original` tree. It's important to note that you cannot alter either of the trees or the `target` node in any way; the method must solely locate the corresponding node in the cloned tree.

Intuition

To find a node that corresponds to the `target` node in the `cloned` tree, we need to mirror the traversal done in the `original` tree. Since the `cloned` tree is an exact copy, every left or right move that leads us to the `target` node in the `original` tree will lead us to the corresponding node in the `cloned` tree.

Therefore, a simple method to solve this problem is to use Depth-First Search (DFS). The `dfs` function is a recursive method that takes as arguments the current node being examined in `original` (`root1`) and the corresponding node in `cloned` (`root2`). If `root1` is `None`, we have hit a leaf node, and we return `None` since there is no corresponding node in `cloned`. If `root1` is the `target` node, we return `root2` since this is the corresponding node in `cloned` that we are looking for. If we haven't found the `target`, the search continues recursively in both the left and right children. The `or` operator is used to return the non-`None` node – if the `target` node is not in the left subtree, the search will continue on the right.

This method ensures that we do a thorough search through both trees simultaneously, comparing nodes from `original` to the `target` and from `cloned` to the identified node in the `original` tree. When we find the `target` in `original`, the same position in `cloned` will be the node we want to return.

Solution Approach

The reference solution relies on a classic recursive traversal similar to a Depth-First Search (DFS) pattern to navigate through the original and cloned trees. Let's dissect this approach:

1. **DFS Algorithm**: At the core of the solution is the recursive DFS algorithm. The algorithm starts from the root node and explores as far as possible along each branch before backtracking. This process naturally fits tree structures and is useful for tree comparison.
2. **Simultaneous Traversal**: The solution explores both the `original` and `cloned` trees in parallel, passing in corresponding nodes to the recursive `dfs` function. This ensures that at any recursive call level, `root1` from the `original` tree and `root2` from the `cloned` tree are always nodes at the same position within their respective trees.
3. **Base Cases**:
   - If `root1` is `None`, `root2` will also point to `None` in the `cloned` tree, and the function returns `None`. This case handles reaching the end of a branch in both trees.
   - If `root1` is equal to `target`, we have found the analogous node in the `original` tree, so we return `root2`, which is the corresponding node in the `cloned` tree.
4. **Recursive Calls**:
   - If the `target` node is not found, the algorithm continues to search recursively in both the left and right children of the current `root1` and `root2` nodes.
   - The function performs an "or" operation between the returned values of the recursive calls to `dfs(root1.left, root2.left)` and `dfs(root1.right, root2.right)`. Since Python's `or` short-circuits, it'll return the first truthy value it encounters. This mechanism is effectively used here to return the non-`None` node reference once the `target` node has been found in either the left or right subtree.
5. **Data Structures**: The data structure used to carry the nodes during the traversal is the call stack, which is a natural consequence of the recursive approach. There is no need for any additional data structures like lists or dictionaries.
6. **Function Call**:
   - The `getTargetCopy` function initiates the process by calling `dfs` with the `original` and `cloned` root nodes, and it returns whatever node the `dfs` function finds, which will be the node corresponding to the `target` node but in the `cloned` tree.

By taking advantage of the structure and properties of binary trees, as well as the recursive nature of DFS, the solution performs an efficient and straightforward search to find the corresponding node in the cloned tree without additional space complexity outside of the recursive call stack.

Example Walkthrough

Let's consider a very simple example with the following binary trees:

```
Original Tree:      Cloned Tree:
      1                   1
     / \                 / \
    2   3               2   3
```

Assume `target` is the node with value `2` in the `original` tree.

We will now walk through the application of the Depth-First Search (DFS) to find the corresponding node in the `cloned` tree using the steps described in the solution approach:

1. **DFS Algorithm**: We initiate a DFS traversal starting from the `root` node of both the `original` and the `cloned` trees.
2. **Simultaneous Traversal**: We are at nodes `1` in both the `original` and the `cloned` trees. They are corresponding nodes, and neither of them is the `target`. So we continue the traversal.
3. **Base Cases**: The base cases are not met as the `root` is not `None` and the `root` is not equal to the `target`.
4. **Recursive Calls**: We make a recursive call to the left child of the `root` node:
   - `dfs(root1.left, root2.left)` is called with `root1.left` being node `2` in the original tree and `root2.left` being node `2` in the cloned tree.
5. **Data Structures**: The state (current node) is maintained simply within the recursive call stack.
6. **Function Call**: Upon the recursive call:
   - We check the base cases again. This time, `root1` is equal to `target` (since both have the value `2`), so we return `root2`, which is the corresponding node in the `cloned` tree.

By following each step, the algorithm finds the `target` in the `original` tree and the corresponding node in the `cloned` tree without the need for any additional data structures or alterations to the original structures. In this case, the node with value `2` in the `cloned` tree is returned as the correct answer.

Python Solution

```python
# Definition for a binary tree node.
class TreeNode:
    def __init__(self, val):
        self.val = val
        self.left = None
        self.right = None


class Solution:
    def getTargetCopy(self, original: TreeNode, cloned: TreeNode, target: TreeNode) -> TreeNode:
        """
        Finds and returns the node in the cloned tree that corresponds to the target node in the original tree.

        Parameters:
        original : TreeNode
            The root of the original binary tree.
        cloned : TreeNode
            The root of the cloned binary tree, which is an exact copy of the original binary tree.
        target : TreeNode
            The target node that exists in the original tree.

        Returns:
        TreeNode
            The corresponding node in the cloned tree that matches the target node from the original tree.
        """

        # Helper function to perform Depth First Search (DFS) on both trees simultaneously.
        def dfs(node_original: TreeNode, node_cloned: TreeNode) -> TreeNode:
            # Base case: if reached the end of the tree, return None.
            if node_original is None:
                return None

            # If the current node in the original tree matches the target, return the corresponding node from cloned tree.
            if node_original == target:
                return node_cloned

            # Recursively search in the left subtree and if not found, then right subtree.
            found_left = dfs(node_original.left, node_cloned.left)
            if found_left:
                return found_left
            return dfs(node_original.right, node_cloned.right)

        # Call dfs with the root nodes of both the original and cloned trees.
        return dfs(original, cloned)
```

Java Solution

```java
// Definition for a binary tree node.
class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;

    // Constructor for tree node
    TreeNode(int val) {
        this.val = val;
    }
}

class Solution {
    private TreeNode targetNode;

    /**
     * Finds and returns the node in the cloned tree that corresponds to the target node
     * from the original tree.
     *
     * @param original The root node of the original binary tree.
     * @param cloned The root node of the cloned binary tree, which is an exact copy of the original binary tree.
     * @param target The target node that needs to be found in the cloned tree.
     * @return The corresponding node in the cloned tree.
     */
    public final TreeNode getTargetCopy(final TreeNode original, final TreeNode cloned, final TreeNode target) {
        // Assign the target node to the global variable for reference in DFS.
        this.targetNode = target;
        // Start DFS traversal with both trees.
        return dfs(original, cloned);
    }

    /**
     * A helper method to perform DFS on both trees simultaneously.
     *
     * @param nodeOriginal The current node in the original tree during the DFS traversal.
     * @param nodeCloned The current node in the cloned tree during the DFS traversal.
     * @return The corresponding node in the cloned tree if the target node is found, else null.
     */
    private TreeNode dfs(TreeNode nodeOriginal, TreeNode nodeCloned) {
        // Base case: if the current node in the original tree is null, return null.
        if (nodeOriginal == null) {
            return null;
        }
        // Check if the current node in the original tree is the target node.
        if (nodeOriginal == targetNode) {
            // If the target node is found, return the corresponding node from the cloned tree.
            return nodeCloned;
        }
        // Recursively search in the left subtree.
        TreeNode result = dfs(nodeOriginal.left, nodeCloned.left);
        // If the result is not found in the left subtree, search in the right subtree.
        return result == null ? dfs(nodeOriginal.right, nodeCloned.right) : result;
    }
}
```

C++ Solution

```cpp
/**
 * Definition for a binary tree node.
 */
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};

class Solution {
public:
    // This method takes the original and cloned trees, along with the target node from the original tree,
    // and returns the corresponding node from the cloned tree.
    TreeNode* getTargetCopy(TreeNode* original, TreeNode* cloned, TreeNode* target) {

        // A depth-first search (DFS) lambda function to traverse both trees simultaneously
        std::function<TreeNode*(TreeNode*, TreeNode*)> dfs = [&](TreeNode* nodeOriginal, TreeNode* nodeCloned) -> TreeNode* {
            // If the original node is null, return null as the search has reached the end of a branch
            if (nodeOriginal == nullptr) {
                return nullptr;
            }

            // If the original node is the target node we are searching for, return the corresponding cloned node
            if (nodeOriginal == target) {
                return nodeCloned;
            }

            // Search in the left subtree
            TreeNode* leftSubtreeResult = dfs(nodeOriginal->left, nodeCloned->left);

            // If the left subtree did not contain the target, search in the right subtree;
            // otherwise, return the result from the left subtree.
            return leftSubtreeResult == nullptr ? dfs(nodeOriginal->right, nodeCloned->right) : leftSubtreeResult;
        };

        // Call the DFS function with the original and cloned tree roots to start the search
        return dfs(original, cloned);
    }
};
```

Typescript Solution

```typescript
// Definition for a binary tree node.
interface TreeNode {
    val: number;
    left: TreeNode | null;
    right: TreeNode | null;
}

/**
 * Finds and returns the node with the same value in the cloned tree as the target node in the original tree.
 *
 * @param original - The root node of the original binary tree
 * @param cloned - The root node of the cloned binary tree
 * @param target - The target node that we want to find in the cloned tree
 * @returns TreeNode | null - The node in the cloned tree that corresponds to the target node in the original tree
 */
function getTargetCopy(
    original: TreeNode | null,
    cloned: TreeNode | null,
    target: TreeNode | null
): TreeNode | null {
    // Helper function to perform a depth-first search (DFS)
    const depthFirstSearch = (nodeOriginal: TreeNode | null, nodeCloned: TreeNode | null): TreeNode | null => {
        // If the current node in the original tree is null, return null since we can't find a corresponding node
        if (!nodeOriginal) {
            return null;
        }
        // If the current node in the original tree is the target, return the corresponding node in the cloned tree
        if (nodeOriginal === target) {
            return nodeCloned;
        }
        // Recursively search in the left subtree, and if not found, search in the right subtree.
        return depthFirstSearch(nodeOriginal.left, nodeCloned.left) || depthFirstSearch(nodeOriginal.right, nodeCloned.right);
    };

    // Start DFS from the root nodes of both trees
    return depthFirstSearch(original, cloned);
}
```

Time and Space Complexity

Time Complexity

The time complexity of the provided code is `O(N)` where `N` is the total number of nodes in the tree. This is because the algorithm traverses each node of the binary tree exactly once in the worst case (when the target node is the last one visited or not present at all).

The `dfs` function is a recursive depth-first search that goes through all nodes of the tree, and for each node, it performs constant-time operations (checking if the node is the target and returning the corresponding node from the cloned tree).

Space Complexity

The space complexity of the provided code is `O(H)` where `H` is the height of the binary tree. This is because the maximum amount of space used by the call stack will be due to the depth of the recursive calls, which corresponds to the height of the tree in the worst case.

For a balanced tree, this would be `O(log N)`, where `N` is the total number of nodes in the tree, because the height of a balanced tree is logarithmic with respect to the number of nodes. However, in the worst case of a skewed tree (i.e., when the tree is a linked list), the space complexity would be `O(N)`.