1660. Correct a Binary Tree

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

Problem Description

In this problem, we're given a binary tree that has one specific defect. There is one node in the tree that has its right child incorrectly pointing to a node that is not actually within its subtree, but to another node that is located at the same depth and further to its right in the tree. We are asked to correct this binary tree by removing the invalid node and all nodes beneath it, except for the node that it incorrectly points to.

It's essential to understand that a valid binary tree does not have any cross connections—each child should only be connected to one parent, and there should not be any connections that could create a cycle within the tree. The presence of such a defect can skew traversal algorithms and produce wrong results. The goal here is to prune the tree in such a way that it becomes a valid binary tree again.

Intuition

The intuition behind the solution involves performing a depth-first traversal on the tree. During this traversal, we will keep track of nodes that have been visited. The defect in the tree makes it so that a node's right child might point to a node that's already been visited, which is the indication that such a node is the one with the incorrect right child reference.

We start by creating a set named `vis` to keep track of visited nodes. We then perform a depth-first search (DFS) using a function called `dfs`. Within this function, if we encounter a node that is `None` or a node whose right child is already in the `vis` set, we know that we have found the invalid node or we're at the end of a path in the tree. We return `None` in this case to signify that this branch should be pruned (i.e., removed from the tree).

During the DFS, after checking for the invalid node, we add the current node to the visited set `vis`, and we recursively continue our search on the right and left children. If the search returns valid (non-`None`) nodes, we link them back to the current node's right and left pointers, respectively.

The `correctBinaryTree` function initiates this process starting from the root of the tree. The DFS function is called with the root node, and through recursive calls, invalid nodes are pruned out. Ultimately, the function returns the corrected tree starting from the `root`.

In essence, the solution leverages the property of DFS and a visited set to detect the incorrect node connection and remove the invalid portion of the tree effectively.

Solution Approach

The solution uses a typical DFS (Depth-First Search) algorithm on the binary tree to traverse nodes. It also utilizes a set called `vis` to keep track of the visited nodes. Here's a step-by-step explanation of how the solution approach is implemented:

1. Define a function `dfs` that takes a node of the tree as an input. This function will be used to perform the DFS and it is defined inside the `correctBinaryTree` method.
2. Inside the `dfs` function, check the base case where the current node is `None`. If it is `None`, or the right child of the current node is already present in the visited set `vis`, return `None`. This signifies that no valid tree node should occupy this position in the corrected tree.
3. If the current node is valid (i.e., not leading to an invalid subtree), add this node to the `vis` set since it's being visited in the DFS process. This helps in identifying if a node is incorrectly pointing to an already visited node (which is essentially the defect in the tree).
4. Recursively call the `dfs` function on the right child of the current node and assign the return value to the current node's right child. This ensures that if the right subtree contains the defect, it's removed from the current node's right child.
5. Similar to the step above, recursively call the `dfs` function on the left child and update the current node's left child with the return value.
6. Return the current node as it is now part of the corrected binary tree, with any invalid children replaced by `None`.

The critical component here is using the set `vis` to identify the defective node. Since the defect involves a node pointing to a right node at the same depth but further to its right, during the DFS, we find that the right node of a current node is already visited which is not possible in a correct binary tree.

The `correctBinaryTree` method initializes the `vis` set and kicks off the DFS with the root node. The corrected tree is returned, starting from the root, after the `dfs` call processes the entire tree and effectively handles the defect.

No additional mathematical formulas or advanced algorithmic patterns are needed in this straightforward implementation. The basic utilities of DFS and the set data structure are enough to deliver a correct solution.

Example Walkthrough

Let's illustrate the solution approach with an example. Consider the following defective binary tree:

```
    1
   / \
  2   3
     / \
    4---5
```

In this example, the node with value 5 is incorrectly pointing to the node with value 4, which is at the same depth but to its right in the tree. According to the rules of a valid binary tree, this is not acceptable.

Now, let's walk through the solution step by step:

1. We define a `dfs` function that performs a depth-first search. The correction process will start by calling `dfs` on the root node (node with value 1).
2. As we start the depth-first search, we initially arrive at node 1. Since node 1 is not `None` and has not been visited yet, we add it to the `vis` set.
3. We now move to the right child of node 1, which is node 3. We add it to the `vis` set and look at its right child, node 5.
4. Before adding node 5 to the tree, we check if its right child (node 4) is in the `vis` set. Since node 4 is already present in the `vis` set, this indicates that node 5 is the incorrect node.
5. At this point, we identify node 5 and all nodes beneath it (if any) as the part of the tree to be pruned. We return `None` to the parent of node 5 (which is node 3) to remove the invalid connection.
6. Continuing the DFS, we also visit the left child of node 3 (which is node 4). We add node 4 to the `vis` set and since it has no children, we return node 4 to its parent, which keeps it in the tree.
7. The `dfs` function is also called on the left child of node 1, which is node 2. Node 2 has no children, so it's simply added to the `vis` set and we move back up the tree.

After the DFS has completed, the resulting corrected tree will look like this:

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

This is a valid binary tree with the incorrect node removed. Thus, our function has successfully corrected the tree by following the depth-first search approach and utilizing a set to track visited nodes.

Python Solution

```python
# Definition for a binary tree node.
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Solution:
    def correctBinaryTree(self, root: TreeNode) -> TreeNode:
        # Initialize a set to keep track of visited nodes
        visited_nodes = set()

        # Helper function to perform DFS on the binary tree
        def dfs(node):
            # Base condition: if the node is None or the right child is already visited
            if node is None or node.right in visited_nodes:
                return None

            # Mark current node as visited
            visited_nodes.add(node)

            # Recursively correct the right subtree
            node.right = dfs(node.right)

            # Recursively correct the left subtree
            node.left = dfs(node.left)

            # Return the node after correction
            return node

        # Use the DFS helper function starting from the root
        return dfs(root)

```

Java Solution

```java
import java.util.HashSet;
import java.util.Set;

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

    TreeNode() {}

    TreeNode(int val) {
        this.val = val;
    }

    TreeNode(int val, TreeNode left, TreeNode right) {
        this.val = val;
        this.left = left;
        this.right = right;
    }
}

class Solution {
    // A HashSet to keep track of visited nodes to detect a node with ref to its ancestor
    private Set<TreeNode> visited = new HashSet<>();

    /**
     * This function initiates the correction of a binary tree in which any node's
     * right child points to any node in the subtree of its ancestors.
     * @param root - The root of the binary tree.
     * @return The corrected binary tree's root.
     */
    public TreeNode correctBinaryTree(TreeNode root) {
        return dfs(root);
    }

    /**
     * This is a depth-first search function that corrects the tree by removing
     * the invalid right child if it points to an ancestor node.
     * @param node - The current node being visited.
     * @return The current node after correction or null if it is to be pruned.
     */
    private TreeNode dfs(TreeNode node) {
        // Base case: if the node is null or its right child points to a visited node (ancestor), prune it.
        if (node == null || visited.contains(node.right)) {
            return null;
        }
        // Mark the current node as visited before exploring its children.
        visited.add(node);
        // Recursively correct the right child then the left child.
        node.right = dfs(node.right);
        node.left = dfs(node.left);
        // Return the node itself after correction.
        return node;
    }
}

```

C++ Solution

```cpp
#include <unordered_set>
using namespace std;

/**
 * Definition for a binary tree node.
 */
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;

    // Constructor for the node with no children.
    TreeNode() : val(0), left(nullptr), right(nullptr) {}

    // Constructor for a node with a value and no children.
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}

    // Constructor for a node with a value and specified left and right children.
    TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
};

class Solution {
public:
    // Function to correct the binary tree so that no right child points to any node in its subtree
    TreeNode* correctBinaryTree(TreeNode* root) {
        // Use a hash set to record nodes that are visited during DFS.
        unordered_set<TreeNode*> visited;

        // Define the DFS function using a lambda expression.
        function<TreeNode*(TreeNode*)> dfs = [&](TreeNode* node) -> TreeNode* {
            // If the node is null or the right child has been visited, return null.
            if (!node || visited.count(node->right)) {
                return nullptr;
            }

            // Mark this node as visited.
            visited.insert(node);

            // Recursively correct the right subtree
            node->right = dfs(node->right);

            // Recursively correct the left subtree
            node->left = dfs(node->left);

            // Return the node itself after correcting both subtrees.
            return node;
        };

        // Call the DFS function on the root to start the correction process.
        return dfs(root);
    }
};

```

Typescript Solution

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

    constructor(val: number = 0, left: TreeNode | null = null, right: TreeNode | null = null) {
        this.val = val;
        this.left = left;
        this.right = right;
    }
}

/**
 * Corrects the binary tree by making sure that no right node that appears
 * in the set of visited nodes continues to be part of the tree.
 *
 * @param {TreeNode | null} rootNode - The root of the tree to correct.
 * @return {TreeNode | null} - The root of the corrected tree.
 */
const correctBinaryTree = (rootNode: TreeNode | null): TreeNode | null => {
    /**
     * Depth-first search traversal to remove incorrect nodes.
     *
     * @param {TreeNode | null} node - Current node being visited.
     * @return {TreeNode | null} - The new subtree without the incorrect nodes.
     */
    const deepFirstSearch = (node: TreeNode | null): TreeNode | null => {
        if (!node || visitedNodes.has(node.right)) {
            return null;
        }
        visitedNodes.add(node);
        node.right = deepFirstSearch(node.right);
        node.left = deepFirstSearch(node.left);
        return node;
    };

    // A set to keep track of visited nodes during the DFS.
    const visitedNodes: Set<TreeNode | null> = new Set();

    // Starting DFS from the root node.
    return deepFirstSearch(rootNode);
};

// Note: Do not invoke `correctBinaryTree` in this global context as it is meant to be used within a specific context where a `TreeNc

```

Time and Space Complexity

The given Python function `correctBinaryTree()` uses a depth-first search (DFS) algorithm to traverse and correct a binary tree based on a specific rule.

The time complexity of this function is `O(N)`, where `N` is the number of nodes in the tree. This is because each node in the tree is visited exactly once in the worst case. The check for `root.right in vis` is `O(1)` thanks to Python's set data structure, which provides average constant time complexity for lookup, so the overall complexity remains linear with respect to the number of nodes.

The space complexity of the function is also `O(N)`. This is for two reasons: the recursion stack that will grow as deep as the height of the tree, which is `O(H)` where `H` is the height of the tree, and the set `vis`, which in the worst-case will contain all `N` nodes if there is no correction made to the tree. Since a binary tree can degenerate into a linked list structure (in the worst case, where `H = N`), the space complexity can also be considered `O(N)` in the worst case.