814. Binary Tree Pruning

Medium Tree Depth-First Search Binary Tree Leetcode Link

Problem Description

In this problem, we are given the root of a binary tree. Our task is to modify the tree by removing all subtrees that do not have at least one node with the value `1`. A subtree, as defined in the problem, is composed of a node and all of its descendants. The goal is to return the same tree, but with certain nodes pruned so that any remaining subtree contains the value `1`.

Intuition

The intuition behind the solution is based on a post-order traversal of the tree, where we visit the child nodes before dealing with the parent node. The recursive function `pruneTree` works bottom-up:

1. If the current node is `None`, there is nothing to do, so we return `None`.
2. Recursively call `pruneTree` on the left child of the current node. The return value will be the pruned left subtree or `None` if the left subtree doesn't contain a `1`.
3. Perform the same operation for the right child.
4. Once both left and right subtrees are processed, check if the current node's value is `0` and that it has no left or right children that contain a `1` (as they would have been pruned if that was the case).
5. If the current node is a `0` node with no children containing a `1`, it is pruned as well by returning `None`.
6. If the current node should remain, we return the current node with its potentially pruned left and right subtrees.

This recursive approach ensures that we remove all subtrees which do not contain a `1` by considering each node's value and the result of pruning its subtrees.

Solution Approach

The implementation of the solution employs the following concepts:

- **Recursion:** The core mechanism for traversing and modifying the tree is the recursive function `pruneTree`.
- **Post-Order Traversal:** This pattern is used where we process the children nodes before the parent node, which is essential in this problem because the decision to prune a parent node depends on the status of its children.
- **Tree Pruning:** Nodes are removed from the tree based on a certain condition (in this case, the absence of the value `1` in the subtree rooted at that node).

Here's a step-by-step walkthrough of how the algorithm works using the provided Python code:

1. Recursively call `pruneTree(root)` with the original root of the tree. If the root is `None`, it returns `None`, signifying an empty tree.

2. The `pruneTree` function is called on `root.left`, pruning the left subtree. The assignment `root.left = self.pruneTree(root.left)` ensures that the current node's left pointer is updated to the result of pruning its left subtree - either a pruned subtree or `None`.

3. The same operation is performed on `root.right` to prune the right subtree.

4. After both subtrees are potentially pruned, we check the current node. If `root.val` is `0`, and both `root.left` and `root.right` are `None` (meaning they were pruned or originally empty), this node must also be pruned. Therefore, the function returns `None`.

5. If the current node is not pruned (i.e., it has the value `1` or it has a non-pruned subtree), it is returned as is.

The result of the recursive function is a reference to the root of the pruned tree since each recursive call potentially modifies the left and right children of the nodes by returning either a pruned subtree or `None`.

The use of recursion here allows the solution to smoothly handle the nested/tree-like structure of the problem and prune nodes appropriately based on the state of their descendants.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach. Consider a binary tree represented by the following structure:

```
        0
       / \
      0   1
     /   / \
    0   0   1
       /
      1
```

We need to prune this tree by removing all subtrees that do not have at least one node with the value `1`. We will use a post-order traversal method to achieve this.

1. We start at the leftmost node, which has the value `0`. Since it has no children, it does not contain a `1`, so we prune this node, and it becomes `None`.

2. Moving up to its parent, which also has a value of `0`. It has no right child and its left child was pruned in step 1. As there is no `1` in the subtree, we prune this node as well.

3. Now, we visit the left child of the root, which again has a value of `0`. Its left child (subtree from steps 1 and 2) has already been pruned. This node does not have a right child. Therefore, we prune this node, and the entire left subtree of the root is now `None`.

4. We move to the right subtree of the root. The left child of the root's right child has the value `0`. This node has a single child with the value `1`, so we keep this subtree intact.

5. Moving to the right child of the root's right child, it has the value `1`, so we keep this node.

6. Looking at the parent of the nodes from steps 4 and 5, which is the right child of the root with the value `1`, since it contains `1` and its subtrees (which are its children nodes) also contain the value `1` or are `None`, we keep this subtree intact.

7. Finally, we move to the root of the tree. Since its right subtree contains a `1`, we keep the right subtree. The left subtree was pruned earlier. The root itself has a value `0`, but because it has a non-empty right subtree that contains a `1`, we do not prune the root.

After the pruning operation, our tree looks like this:

```
        0
         \
          1
         / \
        0   1
       /
      1
```

The pruned tree now only contains subtrees which have at least one node with the value `1` as required by the problem statement.

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 prune_tree(self, root: TreeNode) -> TreeNode:
        # If the current node is None, no pruning needed, return None.
        if root is None:
            return None

        # Prune the left subtree and assign the result to the left child.
        root.left = self.prune_tree(root.left)

        # Prune the right subtree and assign the result to the right child.
        root.right = self.prune_tree(root.right)

        # Check if the current node is a leaf with value 0, prune it by returning None.
        if root.val == 0 and root.left is None and root.right is None:
            return None

        # Return the current node as it is valid in the pruned tree.
        return root

    # Alias the method to the originally provided method name to comply with LeetCode interface.
    pruneTree = prune_tree

```

Java Solution

```java
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int value; // TreeNode's value
 *     TreeNode left; // left child
 *     TreeNode right; // right child
 *     TreeNode() {} // default constructor
 *     TreeNode(int value) { this.value = value; } // constructor with value
 *     // constructor with value, left child, and right child
 *     TreeNode(int value, TreeNode left, TreeNode right) {
 *         this.value = value;
 *         this.left = left;
 *         this.right = right;
 *     }
 * }
 */
class Solution {
    /**
     * Prunes a binary tree by removing all subtrees that do not contain the value 1.
     * A subtree rooted at node containing 0s only is pruned.
     *
     * @param root The root of the binary tree.
     * @return The pruned binary tree's root.
     */
    public TreeNode pruneTree(TreeNode root) {
        // Base case: if the node is null, return null (i.e., prune the null subtree)
        if (root == null) {
            return null;
        }

        // Recursively prune the left subtree
        root.left = pruneTree(root.left);

        // Recursively prune the right subtree
        root.right = pruneTree(root.right);

        // If the current node's value is zero and it doesn't have any children,
        // it is a leaf node with value 0, thus should be pruned (returned as null)
        if (root.value == 0 && root.left == null && root.right == null) {
            return null;
        }

        // Otherwise, return the current (possibly pruned) node.
        return root;
    }
}

```

C++ Solution

```cpp
// Definition for a binary tree node.
struct TreeNode {
    int val; // Value of the node
    TreeNode *left; // Pointer to the left child
    TreeNode *right; // Pointer to the right child

    // Constructor initializes the node with a default value and null child pointers
    TreeNode() : val(0), left(nullptr), right(nullptr) {}

    // Constructor initializes the node with a given value and null child pointers
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}

    // Constructor initializes the node with a given value and given left and right children
    TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
};

class Solution {
public:
    // Function to prune a binary tree. If a subtree does not contain the value '1'
    // it will remove that subtree.
    TreeNode* pruneTree(TreeNode* root) {
        // If the current node is null, return null (base case)
        if (!root) return nullptr;

        // Recursively prune the left subtree
        root->left = pruneTree(root->left);

        // Recursively prune the right subtree
        root->right = pruneTree(root->right);

        // If the current node's value is 0 and both left and right subtrees are null (pruned away),
        // then remove this node as well by returning null
        if (root->val == 0 && !root->left && !root->right) return nullptr;

        // If the current node is not pruned, return the node
        return root;
    }
};

```

Typescript Solution

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

/**
 * Recursively prunes a binary tree, removing all subtrees that contain only 0s.
 * @param {TreeNode | null} node - The root node of the binary tree or subtree being pruned.
 * @returns {TreeNode | null} - The pruned tree's root node, or null if the entire tree is pruned.
 */
function pruneTree(node: TreeNode | null): TreeNode | null {
    // Base case: if the current node is null, just return it.
    if (node === null) {
        return node;
    }

    // Recursively prune the left and right subtrees.
    node.left = pruneTree(node.left);
    node.right = pruneTree(node.right);

    // If the current node's value is 0 and it has no children, prune it by returning null.
    if (node.val === 0 && node.left === null && node.right === null) {
        return null;
    }

    // If the current node has a value of 1 or has any children, return the node itself.
    return node;
}

```

Time and Space Complexity

The time complexity of the code is `O(n)`, where `n` is the number of nodes in the binary tree. This is because the algorithm must visit each node exactly once to determine whether it should be pruned.

The space complexity of the code is `O(h)`, where `h` is the height of the binary tree. This represents the space taken up by the call stack due to recursion. In the worst-case scenario, the function could be called recursively for each level of the tree, resulting in a stack depth equal to the height of the tree. For a balanced tree, this would be `O(log n)`, but for a skewed tree, it could be `O(n)`.