814. Binary Tree Pruning

Depth-First Search

Binary Tree

Problem Description

Medium

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

doesn't contain a 1. 3. Perform the same operation for the right child.

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.

- 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.

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

result of pruning its subtrees.

6. If the current node should remain, we return the current node with its potentially pruned left and right subtrees.

Solution Approach

• Post-Order Traversal: This pattern is used where we process the children nodes before the parent node, which is essential in this problem

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

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

None.

that node).

The implementation of the solution employs the following concepts:

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

• Recursion: The core mechanism for traversing and modifying the tree is the recursive function pruneTree.

- Recursively call pruneTree(root) with the original root of the tree. If the root is None, it returns None, signifying an empty
- tree. pruneTree function is called on root.left, pruning the left subtree. The assignment root.left The

self.pruneTree(root.left) ensures that the current node's left pointer is updated to the result of pruning its left subtree -

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

either a pruned subtree or None. The same operation is performed on root right to prune the right subtree.

left and right children of the nodes by returning either a pruned subtree or None.

appropriately based on the state of their descendants.

with the value 1, so we keep this subtree intact.

After the pruning operation, our tree looks like this:

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

The use of recursion here allows the solution to smoothly handle the nested/tree-like structure of the problem and prune nodes

Example Walkthrough

structure:

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

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

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,

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

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

traversal method to achieve this.

now None.

and it becomes None.

1 in the subtree, we prune this node as well. 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

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

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.

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

- The pruned tree now only contains subtrees which have at least one node with the value 1 as required by the problem statement.
- # Definition for a binary tree node. class TreeNode: def init (self, val=0, left=None, right=None): self.val = val self.left = left

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

return None

TreeNode left; // left child

return null;

// Definition for a binary tree node.

int val; // Value of the node

// it will remove that subtree.

TreeNode* pruneTree(TreeNode* root) {

// Recursively prune the left subtree

// Recursively prune the right subtree

root->right = pruneTree(root->right);

root->left = pruneTree(root->left);

if (!root) return nullptr;

TreeNode *left; // Pointer to the left child

TreeNode *right; // Pointer to the right child

TreeNode() : val(0), left(nullptr), right(nullptr) {}

TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}

// If the current node is null, return null (base case)

return root;

TreeNode right; // right child

TreeNode() {} // default constructor

Solution Implementation

self.right = right

if root is None:

return None

def prune tree(self, root: TreeNode) -> TreeNode:

root.left = self.prune_tree(root.left)

root.right = self.prune_tree(root.right)

If the current node is None, no pruning needed, return None.

Prune the left subtree and assign the result to the left child.

Prune the right subtree and assign the result to the right child.

TreeNode(int value) { this value = value; } // constructor with value

// If the current node's value is zero and it doesn't have any children,

// Constructor initializes the node with a default value and null child pointers

// Constructor initializes the node with a given value and null child pointers

// Function to prune a binary tree. If a subtree does not contain the value '1'

* Recursively prunes a binary tree, removing all subtrees that contain only 0s.

if (node.val === 0 && node.left === null && node.right === null) {

Prune the right subtree and assign the result to the right child.

if root.val == 0 and root.left is None and root.right is None:

Check if the current node is a leaf with value 0, prune it by returning None.

root.right = self.prune tree(root.right)

return None

* @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.

// 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) {}

if (root.value == 0 && root.left == null && root.right == null) {

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

// it is a leaf node with value 0, thus should be pruned (returned as null)

// constructor with value, left child, and right child

if root.val == 0 and root.left is None and root.right is None:

Check if the current node is a leaf with value 0, prune it by returning None.

Python

class Solution:

```
/**
* Definition for a binary tree node.
* public class TreeNode {
* int value: // TreeNode's value
```

Java

```
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);
```

};

public:

C++

struct TreeNode {

class Solution {

val: number;

*/

left: TreeNode | null:

return null;

right: TreeNode | null;

// 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** // Definition for a binary tree node. interface TreeNode {

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 the current node has a value of 1 or has any children, return the node itself. return node; # 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)

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 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 0(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 0(log n), but for a skewed tree, it could be 0(n).