1325. Delete Leaves With a Given Value

Medium Tree Depth-First Search Binary Tree Leetcode Link

Problem Description

In this problem, we're given a binary tree where each node contains an integer value. Our task is to remove all the leaf nodes that have a value equal to a given target integer. A leaf node is defined as a node that doesn't have left or right children. However, this operation has a cascade effect: if, after removing a leaf node, its parent node becomes a leaf node and its value is equal to the target, then we should also remove that parent node. We need to continue this process recursively until there are no more nodes that meet the criteria for deletion.

In summary, we're tasked with pruning the binary tree by removing target-valued leaf nodes and continuing to remove their respective parent nodes if they become target-valued leaves as a result.

Intuition

To solve this problem effectively, we can make use of recursion, which lends itself naturally to trees' hierarchical structure. The intuition is as follows: we will perform a post-order traversal of the tree. This means that we will first look at the child nodes before dealing with their parent nodes. This ordering is crucial because it allows us to decide whether the parent should be deleted after we've already considered and possibly deleted its children.

Here is a step-by-step process of the recursive solution:

1. If the current node is `None` (or in other words, if we're looking at an empty spot in the tree), we simply return `None`, as there is nothing to delete.
2. We recursively call the function on the left child of the current node. This will prune the left subtree and return the new left child for the current node.
3. We do the same for the right child of the current node.
4. After we have the results of the recursive calls for both children, we check the current node's value. If the current node has no children left (both are `None` after the recursive calls) and its value is equal to `target`, we will return `None`. This effectively deletes the current node because when the recursion rolls back up to the parent, this node will not be reattached to the tree.
5. If the current node isn't a leaf or its value doesn't match `target`, we return the current node itself, potentially with modified children, resulting from the recursive deletion process.

By following this approach, we can ensure that all nodes that are or become leaf nodes with value `target` are removed from the tree.

Solution Approach

The implementation of the solution uses a basic pattern of tree traversal called post-order traversal. This approach processes a node's children before the node itself, which is ideal for this problem since we need to look at the leaf nodes before making decisions about their parents.

Here's a detailed explanation of the algorithms, data structures, or patterns used in the solution:

1. **Recursive Function:** The solution defines a recursive function `removeLeafNodes` that takes a `TreeNode root` and an integer `target` as input. The base case of this recursive function is when the `root` is `None`. If this is the case, we return `None` because there's nothing to do on an empty subtree.

2. **Post-order Traversal:** The function calls itself to handle the left and right subtrees `root.left = self.removeLeafNodes(root.left, target)` and `root.right = self.removeLeafNodes(root.right, target)`. These recursive calls will first prune the subtrees before looking at the current node. This is the essence of post-order traversal—visit left, then right, then process the node.

3. **Leaf Node Check and Deletion:** After the recursive calls, the function checks whether the current node is now a leaf node `if root.left is None and root.right is None`. If it's a leaf node and its value equals the `target and root.val == target`, we return `None` to delete this node.

4. **Returning Pruned Tree:** If the current node is not a target leaf node, it simply returns the current node `return root`. This might be a node with one or both children pruned, or it might be unchanged if neither child was a target leaf.

Here's what the main parts of the function look like in code:

- Recursive call to traverse and prune the left subtree:

```
root.left = self.removeLeafNodes(root.left, target)
```

- Recursive call to traverse and prune the right subtree:

```
root.right = self.removeLeafNodes(root.right, target)
```

- Checking if the current node is a leaf and should be deleted:

```
if root.left is None and root.right is None and root.val == target:
    return None
```

- Returning the node if it shouldn't be pruned:

```
return root
```

By combining these steps, the solution prunes the tree in one pass by leveraging the call stack inherent in recursion, allowing us to visit nodes in the precise order necessary to solve the problem efficiently.

Example Walkthrough

Let's demonstrate how the solution approach works with the following small example:

Suppose we have a binary tree where the target value we want to remove is `1`. The initial tree looks like this:

```
    1
   / \
  2   1
 /   / \
2   1   2
   /
  1
```

Following the solution approach, here's what happens step-by-step:

1. Start with the root node with the value `1`.
2. Recursively traverse to the left child which has a value of `2`.
   1. Since the left child of `2` is also `2`, recur on it.
      - It's a leaf and not equal to our target `1`, so it will not be pruned and is returned as is.
   2. The right child of the first `2` is `None`, so no action is needed.
   3. Check the first `2`, it's not a leaf node because it still has a left child, so it's also returned as is.
3. Recursively traverse to the right child of the root which has a value of `1`.
   1. The left child of this `1` is another `1`.
      - It has a leaf `1` as a left child, which is equal to our target and thus pruned and returned as `None`.
      - The right child is `None`, so after pruning its left child, this `1` has become a leaf itself and since its value is the target, it is pruned too and returned as `None`.
   2. The right child of the first `1` on the right side has a value of `2`, which is a leaf but not equal to our target, so it remains as is.
4. Now, the root node has its left subtree unchanged and right subtree modified (as the `1` has been pruned). The right subtree is now:

```
1
 \
  2
```

5. The root `1` is not a leaf, so it stays. The final pruned tree looks like:

```
    1
   / \
  2   1
 /     \
2       2
```

By following the post-order traversal, we efficiently pruned all leaves with the target value `1` and also those becoming leaves after pruning their children, with one sweep of recursion.

Python Solution

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

class Solution:
    def removeLeafNodes(self, root: Optional[TreeNode], target: int) -> Optional[TreeNode]:
        """
        Recursively removes all leaf nodes from the binary tree that have the given target value.

        Parameters:
        root (TreeNode): The root of the binary tree.
        target (int): The target value for which leaf nodes should be removed.

        Returns:
        TreeNode: The root of the modified binary tree after removing the target leaf nodes.
        """
        # Return None if the current node is None
        if root is None:
            return None

        # Recursively remove target leaf nodes from the left subtree
        root.left = self.removeLeafNodes(root.left, target)

        # Recursively remove target leaf nodes from the right subtree
        root.right = self.removeLeafNodes(root.right, target)

        # If the current node is a leaf and its value matches the target,
        # return None to remove it
        if root.left is None and root.right is None and root.val == target:
            return None

        # Return the current node if it is not a target leaf node
        return root

```

Java Solution

```java
// Definition for a binary tree node.
class TreeNode {
    int val; // Value of the node
    TreeNode left; // Reference to the left child node
    TreeNode right; // Reference to the right child node

    // Constructor to initialize the node with no children
    TreeNode() {}

    // Constructor to initialize the node with a value
    TreeNode(int val) { this.val = val; }

    // Constructor to initialize the node with a value and references to left and right children
    TreeNode(int val, TreeNode left, TreeNode right) {
        this.val = val;
        this.left = left;
        this.right = right;
    }
}

class Solution {
    // Removes all leaf nodes with a specified value from a binary tree.
    public TreeNode removeLeafNodes(TreeNode root, int target) {
        // If the root is null, the tree is empty, and we return null as there are no nodes to remove.
        if (root == null) {
            return null;
        }

        // Recursively remove leaf nodes from the left subtree.
        root.left = removeLeafNodes(root.left, target);
        // Recursively remove leaf nodes from the right subtree.
        root.right = removeLeafNodes(root.right, target);

        // Check if the current node is a leaf node with the target value.
        if (root.left == null && root.right == null && root.val == target) {
            // If so, remove this node by returning null to the parent call.
            return null;
        }

        // Return the possibly updated root to the previous recursive call.
        // If no changes were made, the original node is returned.
        return root;
    }
}

```

C++ Solution

```cpp
#include <cstddef> // for nullptr

// Forward declaration for the TreeNode struct.
struct TreeNode {
    int val; // Value of the node.
    TreeNode *left; // Pointer to the left child node.
    TreeNode *right; // Pointer to the right child node.

    // Constructor for a tree node with a default value of 0 and no children.
    TreeNode() : val(0), left(nullptr), right(nullptr) {}

    // Constructor for creating a leaf node with a specific value.
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}

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

class Solution {
public:
    // Function to remove all leaf nodes in a binary tree that have the specified target value.
    TreeNode* removeLeafNodes(TreeNode* node, int target) {
        // If the current node is null, we have reached the end of a branch and return null.
        if (!node) {
            return nullptr;
        }

        // Recursively remove target leaves from the left subtree.
        node->left = removeLeafNodes(node->left, target);

        // Recursively remove target leaves from the right subtree.
        node->right = removeLeafNodes(node->right, target);

        // If the current node is a leaf (has no children) and its value equals the target,
        // delete this node by returning null.
        if (!node->left && !node->right && node->val == target) {
            delete node; // Free the memory of the node to avoid memory leaks.
            return nullptr;
        }

        // Return the potentially updated current node.
        return node;
    }
};

```

Typescript Solution

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

  constructor(val?: number, left?: TreeNode | null, right?: TreeNode | null) {
    this.val = val === undefined ? 0 : val; // Assign the given value, or default to 0 if no value is provided.
    this.left = left === undefined ? null : left; // Assign the given left child, or default to null if none.
    this.right = right === undefined ? null : right; // Assign the given right child, or default to null if none.
  }
}

/**
 * Removes all leaf nodes with the specified target value from the binary tree.
 * If, after removing a leaf, the parent also becomes a leaf with the target value, it gets removed as well, and so on.
 * @param {TreeNode | null} node - The current node of the binary tree.
 * @param {number} target - The value of the target leaf nodes that need to be removed.
 * @return {TreeNode | null} The modified tree with specified leaf nodes removed.
 */
function removeLeafNodes(node: TreeNode | null, target: number): TreeNode | null {
  // Base case: if the current node is null, simply return null.
  if (!node) {
    return null;
  }

  // Recursively remove leaf nodes in the left subtree.
  node.left = removeLeafNodes(node.left, target);
  // Recursively remove leaf nodes in the right subtree.
  node.right = removeLeafNodes(node.right, target);

  // Check if the current node has become a leaf node with the value equal to target.
  // If so, remove this node by returning null; otherwise, return the current node.
  if (node.left === null && node.right === null && node.val === target) {
    return null;
  } else {
    return node;
  }
}

```

Time and Space Complexity

Time Complexity

The given code visits each node of the binary tree exactly once. The operations performed per node (checking if a node is a leaf and whether it carries the target value) are constant time operations. Therefore, the time complexity is `O(N)`, where `N` is the number of nodes in the tree.

Space Complexity

The space complexity is affected by the recursive calls to `removeLeafNodes`. In the worst case, the tree could be skewed, meaning each level contains a single node. In this case, there would be `O(N)` recursive calls on the call stack at the same time. Therefore, the worst-case space complexity is `O(N)`. However, in the average case, where the tree is balanced, the height of the tree would be `O(log N)`, leading to a space complexity of `O(log N)` due to the call stack.