998. Maximum Binary Tree II

Medium Tree Binary Tree

Leetcode Link

Problem Description

In this problem, we're given the root node of a maximum binary tree—a kind of binary tree where each node's value is greater than the values of every node in its subtree—and an integer `val`. Our task is to insert this integer into the maximum binary tree following a specific set of rules without having direct access to the array from which the tree was originally constructed.

To recall, a maximum binary tree is constructed as follows:

- If the array `a` is empty, return `null`.
- Find the largest element in `a`, let's say it's `a[i]`. This becomes the root node.
- The left child of the root node is another maximum tree constructed from the subarray to the left of `a[i]`, i.e., `a[0]` to `a[i - 1]`.
- The right child is constructed from the subarray to the right of `a[i]`, i.e., `a[i + 1]` to `a[a.length - 1]`.
- The node with the value `a[i]` becomes the root of this maximum binary tree.

The problem defines an array `b`, which is a copy of the original array `a` but with the new integer `val` added to the end. We must return a maximum binary tree constructed from this new array `b`, but as we do not have access to `a`, we must do this by modifying the given tree directly.

Intuition

The key to solving this problem lies in understanding the properties of the maximum binary tree and how the new value `val` can affect the structure of the tree:

1. The value `val` is being appended to the array, so in the resulting maximum binary tree for `b`, `val` will either be the new root or a child in the right subtree of the existing tree, as all values in `a` are to its left and thus cannot be its left child.
2. If `val` is greater than the current root's value, then `val` should become the new root, and the whole existing tree should become the left child of this new root.
3. If `val` is less than the current root's value, we need to traverse the right subtree recursively and perform a similar decision process considering `val` and the value of that subtree's root.
4. We continue this process recursively until we find the right position for the new value `val`. If we reach a point where `val` is larger than the root of a subtree, or we reach a leaf node (which would be `None`), then we insert `val` there.

The provided solution translates this intuition into a straightforward recursive function that correctly modifies the existing tree to include the new value.

Solution Approach

The solution provided uses a recursive function to insert the new value into the maximum tree. The implementation capitalizes on the recursive tree structure and the definition of a maximum tree to determine the appropriate place for the new value.

Here are the key steps and patterns used in the implementation:

1. **Base Cases**:
   - When the `root` passed to the `insertIntoMaxTree` function is `None`, this suggests we've reached a point where the new value `val` should be inserted. Hence, a new node with `val` is created and returned.
   - If `val` is greater than the `root.val`, according to the property of a maximum tree, `val` must be the new root, and the current tree should be the left child of this new node with `val`.
2. **Recursion**:
   - If the current `root`'s value is greater than `val`, we need to insert `val` into the right subtree of the current `root`. To do this, the function calls itself recursively with the right child of the root and the `val`.
3. **Tree Node Construction**:
   - A new tree node is created with the given `val`, when needed, by calling `TreeNode(val, root)` where `root` would become the left child if `val` becomes the new root.
4. **Linking Nodes**:
   - After the recursive call, the return value (which is the modified right subtree with the new value inserted) is linked back to the current `root`'s right child.

By applying these steps, the algorithm maintains the integrity of the maximum tree while inserting the new value in its correct place. Since the algorithm operates directly on the original tree nodes, no additional data structures are needed beyond those provided by the tree itself.

Here's an excerpt of the solution reflecting these steps:

```
if root is None or root.val < val:
    return TreeNode(val, root)
root.right = self.insertIntoMaxTree(root.right, val)
return root
```

This code checks whether we have to insert a new root or continue the traversal. The recursive call to `self.insertIntoMaxTree(root.right, val)` ensures we dive into the right subtree to explore further placement for `val`.

In this manner, every call of `insertIntoMaxTree` either creates a new node when it finds the appropriate place for `val` or navigates through the right subtree to find that place, thus ensuring the new value is added exactly where it should be to preserve the maximum tree property.

Example Walkthrough

Let's consider an example to illustrate the solution approach. Suppose we have a maximum binary tree constructed from the array `[3, 2, 1]`, and we want to insert the value `4`.

The existing maximum binary tree looks like this:

```
  3
 / \
2   1
```

In this tree, `3` is the root as it's the largest element in the array. The left subtree consists of `2`, and the right subtree consists of `1`.

Now, following our rules, we want to insert `4`. Since `4` is greater than the current root's value `3`, according to our solution approach, `4` should become the new root of the tree.

According to the solution approach step 1 for Base Cases:

1. `4` being greater than the existing root `3`, requires that a new root node be created with `4`. The existing tree becomes the left child of this new node.

The updated tree now looks like this:

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

With `4` as the new root, `3` now becomes the left child, and the previous left and right children (`2` and `1` respectively) of `3` remain unchanged.

If we were to insert a value less than `1`, for example `0`, the flow would be as follows according to our Intuition Step 3 and the Recursive process described in the solution approach:

1. Since `0` is not greater than the root node `4`, we move to the right subtree. However, since `3` (the left child of `4`) is also not the right place for `0` as our value needs to go to the right of `3`, we proceed to its right subtree.
2. Comparing `0` to `1` (the right child of `3`), `0` is still less, so we move to the right of `1`. Since `1` has no right child, we insert `0` as the right child of `1`.

Finally, the tree looks like this:

```
  4
 /
3
 \
  2
   \
    1
     \
      0
```

In each step of the above examples, the algorithm recursively traverses the tree to find the right spot for the insertion, always ensuring that the maximum tree property is preserved, which is that the value of each node is greater than all the values in its left subtree and less than any value in its right subtree.

Python Solution

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

class Solution:
    def insertIntoMaxTree(self, root: Optional[TreeNode], val: int) -> Optional[TreeNode]:
        """
        Inserts a new value into the Max Tree.
        A Max Tree is a tree where every node is greater than its children.
        The new value is always inserted as a leaf node.

        Parameters:
        root (TreeNode): The root of the original Max Tree.
        val (int): The value to be inserted.

        Returns:
        TreeNode: The root of the updated Max Tree.
        """
        # If the current node is None or the new value is greater than the current node,
        # then insert the new value above and to the right of the current node.
        if root is None or root.val < val:
            return TreeNode(val, root)

        # Recursively insert the new value into the right subtree.
        root.right = self.insertIntoMaxTree(root.right, val)

        # Return the root as it may be unchanged.
        return root
```

Java Solution

```
/**
 * Definition for a binary tree node.
 */
class TreeNode {
    int val;              // The value contained in the node
    TreeNode left;        // Reference to the left child
    TreeNode right;       // Reference to the right child

    // Constructor for creating a leaf node with a specific value
    TreeNode(int val) {
        this.val = val;
    }

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

class Solution {
    /**
     * Inserts a value into a maximum binary tree followed by the rules:
     * - Each tree node has a value greater than any of its children.
     * - Newly inserted value is always added as a new tree node in a position that maintains the maximum tree property.
     *
     * @param root the root of the current maximum binary tree
     * @param val the value to insert into the tree
     * @return the root of the modified maximum binary tree
     */
    public TreeNode insertIntoMaxTree(TreeNode root, int val) {
        // If the tree is empty or the value at the root is less than the value to be inserted,
        // create a new node with the inserted value and the current tree as its left subtree.
        if (root == null || root.val < val) {
            return new TreeNode(val, root, null);
        }

        // Recursively insert into the right subtree.
        // Because we are working with a max tree, we go down the tree to the right as long as
        // we do not find a node with a lesser value where our new node will be inserted.
        root.right = insertIntoMaxTree(root.right, val);

        // Return the root of the tree with the new value inserted while keeping its max tree structure.
        return root;
    }
}
```

C++ Solution

```
/**
 * Definition for a binary tree node.
 */
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode() : val(0), left(nullptr), right(nullptr) {}
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
    TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
};

class Solution {
public:
    /**
     * Inserts a new value into a Maximum Binary Tree and returns the updated tree.
     *
     * A Maximum Binary Tree is a tree where every node has a value greater than
     * any value of the nodes in its left subtree and any value of the nodes in its right subtree.
     *
     * @param root The root node of the Maximum Binary Tree.
     * @param val The value to be inserted into the Maximum Binary Tree.
     * @return Returns the updated tree after inserting the value.
     */
    TreeNode* insertIntoMaxTree(TreeNode* root, int val) {
        // If the root is null or the root value is less than the value to be inserted,
        // then a new node needs to be created with the value, where the current root becomes
        // the left child of the new node (as per Maximum Binary Tree properties).
        if (!root || root->val < val) {
            return new TreeNode(val, root, nullptr);
        }

        // If the value to be inserted is smaller than the root, traverse the right subtree
        // to find the spot to insert the new value, since all values in the right subtree
        // should be smaller than the root.
        root->right = insertIntoMaxTree(root->right, val);

        // Return the root as the unchanged part of the tree after the insertion.
        return root;
    }
};
```

Typescript Solution

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

// TreeNode constructors initializes a TreeNode with given values for val, left and right.
// If no value is provided, val defaults to 0 and left/right default to null.
const TreeNode = function(val = 0, left: TreeNode | null = null, right: TreeNode | null = null) {
    this.val = val;
    this.left = left;
    this.right = right;
};

// Function to insert a new value into a maximum binary tree.
// A maximum binary tree is a binary tree where every node has a value greater than any other value in its descendants.
//
// Parameters:
// root(Node): The root node of the current maximum binary tree.
// newValue: The new value to be inserted into the maximum binary tree.
//
// Returns: The root node of the binary tree after inserting the new value.
function insertIntoMaxTree(root: TreeNode | null, newValue: number): TreeNode | null {
    // If the root node is null or the value of the root is less than the new value,
    // create a new tree node with the new value, with the entire current tree as its left child.
    if (!root?.val || root.val < newValue) {
        return new TreeNode(newValue, root);
    }

    // Otherwise, recursively attempt to insert the new value into the right subtree.
    // The result of this recursive call will be assigned as the new right child of the current node.
    root.right = insertIntoMaxTree(root.right, newValue);

    // Return the root node as it retains its position.
    return root;
}
```

Time and Space Complexity

The time complexity of the code provided is $O(N)$, where N is the number of nodes in the binary tree. We are traversing the tree to find the right place to insert the new node with value `val`. In the worst case scenario, we insert the node as the rightmost child, which requires traversing all the way down to the bottom of the tree, which can be N nodes in a skewed tree.

The space complexity of the recursive implementation is also $O(N)$, where N is the height of the tree, because of the recursive call stack. In the worst case in a skewed tree (which behaves like a linked list), this would also be $O(N)$. In a balanced tree, the height H would be $O(\log N)$ so the space complexity would be $O(\log N)$. However, since the tree is formed by inserting nodes into the maximum tree, it can be heavily skewed, so the space complexity can also be considered $O(N)$ in the worst case.