623. Add One Row to Tree

Medium Tree Depth-First Search Breadth-First Search Binary Tree Leetcode Link

Problem Description

The goal of this problem is to add a new row to a binary tree, with all new nodes having the same value. The new row should be added at a specified depth in the tree, where the root node starts at depth 1. Here's how the process should work:

- If `depth` equals 1, a new root node with the given `val` should be created, and the entire original tree becomes the left subtree of this new root.
- For depths greater than 1, we look for all nodes at `depth - 1`. For each of these nodes, we create two new children with the given `val`.
  - The new left child becomes the parent of the original left subtree of the node.
  - The new right child becomes the parent of the original right subtree of the node.
- This process effectively inserts a row of new nodes at the specified `depth`, pushing the existing nodes at that depth (if any) to become children of the newly added nodes.

Intuition

The intuition behind the solution is to traverse the tree and locate the nodes at `depth - 1`. For each of these nodes, we then attach new children nodes with the given `val`. The essential steps we follow are:

- If `depth` is 1, we don't need to traverse the tree, because we simply create a new root with the given `val` and link the entire tree as the left subtree of this new root node.
- If `depth` is greater than 1, we use depth-first traversal (DFS) to reach the nodes at `depth - 1`.
  - During the traversal, we keep track of the current depth.
  - Once we reach the required level (`depth - 1`), we perform the insertion of new nodes.
    - This involves creating two new tree nodes with `val` as their value.
    - The new left node takes the current node's left subtree, and the new right node takes the current node's right subtree.
- After performing the insertions at the required depth, we ensure the rest of the tree remains unchanged by only applying changes where necessary.

In this approach, we modify the tree in place without creating a separate structure, and we only create new nodes where the row is supposed to be added.

Solution Approach

The provided solution utilizes recursion for a depth-first search approach to solve the problem efficiently. Here's a step-by-step explanation of how the algorithm operates:

1. The `dfs` function defined within the `Solution` class recursively explores the binary tree.
2. The `dfs` function takes two parameters: `root` which represents the current node in the binary tree and `d` which indicates the current depth of the recursive call.
3. The base case checks if `root` is `None`, in which case the function simply returns without performing any action, as we've reached a leaf node's child.
4. If the current depth `d` is equal to `depth - 1`, it means we've reached the level above where the new row should be inserted. We perform the following insertions in this case:
   - Create a new `TreeNode` with a value of `val` and set its left child to the current node's original left subtree (`root.left`). The new node is then assigned to `root.left`.
   - Similarly, create another new `TreeNode` with a value of `val` for the right side and assign the current node's original right subtree (`root.right`) to the new node's right child. This new node is then assigned to `root.right`.
5. If the current depth `d` is not yet at `depth - 1`, the function makes recursive calls to continue the search down the left and right subtrees, respectively, incrementing the depth `d` by 1.
6. The main part of the `addOneRow` method checks if `depth` equals 1. If so, a new `TreeNode` is created with the specified `val` and the entire original tree as its left subtree. This new node becomes the new root.
7. If `depth` is greater than 1, the recursive `dfs` call is initiated with `root` and a starting depth of 1.
8. After the recursive calls complete, the original `root` of the tree is returned with the modifications in place (unless a new root was created, in which case that is returned).

The algorithm effectively leverages the call stack as its primary data structure, storing the state of each node's exploration during the recursion. The overall time complexity of this solution is O(n), where n is the number of nodes in the tree, since in the worst case, every node is visited once.

Example Walkthrough

To illustrate the solution approach, let's consider a binary tree and the task of adding a row of nodes with value `v` at a given depth `k`.

Assume we have the following binary tree:

```
     4
   /   \
  2     6
 / \   / \
3   1 5   7
```

And we want to add a row of nodes with value `5` at depth `3`.

Starting with `addOneRow`, we check if the depth is 1. It's not, since we want to add the row at depth `3`. Therefore, we proceed to call the `dfs` function passing the root of the tree and the initial depth `1`.

The `dfs` function begins to traverse the tree. At the initial depth, none of the conditions to insert a node are met, so the function recursively calls itself for the left child `2` and right child `6` of the root `4`, with depth `2`.

For both child nodes, `2` and `6`, we are still not at the target depth (`depth - 1` which is `2`) for insertion, so the function recursively calls itself for their children, with depth incremented to `3`, which is our target for insertion.

Node `2` has children `3` and `1`, and node `6` has children `5` and `7`. Now that `d` equals `depth - 1` at this level, we perform the insertions:

- The original left child of node `2` (which is `3`) becomes the left child of a new node with `val 5`, and this new node is then assigned as the new left child of node `2`.
- Similarly, the original right child of node `2` (which is `1`) becomes the left child of another new node with `val 5`, which is then assigned as the new right child of node `2`.

The same process occurs for node `6` and its children.

With insertions complete, the function returns to its caller at the higher level and ultimately back to `addOneRow`, which returns the original root of the binary tree.

After the row addition, the modified binary tree looks like this:

```
     4
   /   \
  2     6
 / \   / \
5   5 5   5
/     /     \
3     1      7
       \
        5
```

In this example, only the nodes that needed new children were changed—`2` and `6`—and no other part of the tree was modified unnecessarily.

Python Solution

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

class Solution:
    def addOneRow(self, root: Optional[TreeNode], value: int, depth: int) -> Optional[TreeNode]:
        # Helper function to perform Depth-First Search (DFS) on the binary tree
        def depth_first_search(node, current_depth):
            # If node is None (the base case), we have reached a leaf's child and we return
            if node is None:
                return

            # If we have reached the desired depth, we add the new row with value
            if current_depth == depth - 1:
                # Create new nodes with the given value and link to the previous children
                node.left = TreeNode(value, left=node.left, right=None)
                node.right = TreeNode(value, left=None, right=node.right)
                return

            # Recursively call DFS on the left and right children, incrementing the depth
            depth_first_search(node.left, current_depth + 1)
            depth_first_search(node.right, current_depth + 1)

        # Special case when the new row needs to be added at the root
        if depth == 1:
            # Create a new root with the given value and set the original root as its left child
            return TreeNode(value, left=root)

        # Begin DFS with the original root at the starting depth of 1
        depth_first_search(root, 1)
        return root

```

Java Solution

```java
// Definition for a binary tree node.
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 {
    // Instance variables to store the value to be added and the target depth
    private int value;
    private int targetDepth;

    // Main method to add a new row to the tree
    public TreeNode addOneRow(TreeNode root, int value, int depth) {
        // Handling the special case where the new row is to be added as the new root
        if (depth == 1) {
            return new TreeNode(value, root, null);
        }
        // Initialize the instance variables
        this.value = value;
        this.targetDepth = depth;
        // Start the depth-first search (DFS) from the root
        depthFirstSearch(root, 1);
        return root;
    }

    // Helper method to perform depth-first search
    private void depthFirstSearch(TreeNode node, int currentDepth) {
        // If the node is null, there is nothing to do; return immediately
        if (node == null) {
            return;
        }
        // Check if we reached the parent level of the target depth
        if (currentDepth == targetDepth - 1) {
            // Create new nodes with the given value and make them children of the current node
            TreeNode leftChild = new TreeNode(value, node.left, null);
            TreeNode rightChild = new TreeNode(value, null, node.right);
            // Update the current node's children to the newly created nodes
            node.left = leftChild;
            node.right = rightChild;
            // No need to traverse further as we have added the row at the target depth
            return;
        }
        // Recursively search the left and right subtrees, increasing the depth by 1
        depthFirstSearch(node.left, currentDepth + 1);
        depthFirstSearch(node.right, currentDepth + 1);
    }
}

```

C++ Solution

```cpp
/**
 * 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 v, TreeNode *leftNode, TreeNode *rightNode) : val(v), left(leftNode), right(rightNode) {}
};

class Solution {
public:
    // Funtion to add one row to the tree at a given depth with the given value
    TreeNode* addOneRow(TreeNode* root, int value, int depth) {
        // If the depth is 1, create a new node with the given value and make the existing tree its right child
        if (depth == 1) {
            return new TreeNode(value, root, nullptr);
        }

        // Set class variables to use in recursive calls
        targetValue_ = value;
        targetDepth_ = depth;

        // Start the depth-first search (DFS)
        depthFirstSearch(root, 1);

        return root;
    }

private:
    int targetValue_; // value to be added
    int targetDepth_; // depth at which to add the new row

    // Recursively traverse the tree to find the proper insertion depth
    void depthFirstSearch(TreeNode* node, int currentDepth) {
        // Base case: if the node is null, stop recursion
        if (!node) {
            return;
        }

        // When the target depth is reached, insert new nodes with targetValue_
        if (currentDepth == targetDepth_ - 1) {
            // Insert new left and right nodes between the current node and its children
            TreeNode *newLeftNode = new TreeNode(targetValue_, node->left, nullptr);
            TreeNode *newRightNode = new TreeNode(targetValue_, nullptr, node->right);

            // Update the current node's children to point to the new nodes
            node->left = newLeftNode;
            node->right = newRightNode;

            return; // No need to go deeper as we have already added the new row at this depth
        }

        // If not at the target depth yet, keep going deeper into the tree
        depthFirstSearch(node->left, currentDepth + 1);
        depthFirstSearch(node->right, currentDepth + 1);
    }
};

```

Typescript Solution

```typescript
// Function to add a new row at the given depth with the specified value in a binary tree.
function addOneRow(root: TreeNode | null, val: number, depth: number): TreeNode | null {
    // If the depth is 1, create a new root node with the current root as its left child.
    if (depth === 1) {
        return new TreeNode(val, root);
    }

    // Initialize a queue to perform level order traversal.
    const queue: (TreeNode | null)[] = [root];

    // Traverse the tree until the level before the desired depth is reached.
    for (let currentDepth = 1; currentDepth < depth - 1; currentDepth++) {
        const levelSize = queue.length; // Number of nodes at the current level.
        for (let i = 0; i < levelSize; i++) {
            // Remove the first node from the queue.
            const currentNode = queue.shift();
            // Add the left child to the queue if it exists.
            if (currentNode?.left) queue.push(currentNode.left);
            // Add the right child to the queue if it exists.
            if (currentNode?.right) queue.push(currentNode.right);
        }
    }

    // For each node at the target depth, add new nodes as their left and right children.
    for (const parentNode of queue) {
        if (parentNode) {
            // Insert the new left child with the existing left child as its left subtree.
            parentNode.left = new TreeNode(val, parentNode.left);
            // Insert the new right child with the existing right child as its right subtree.
            parentNode.right = new TreeNode(val, null, parentNode.right);
        }
    }

    // Return the original root as the new tree with the added row.
    return root;
}

```

Time and Space Complexity

The provided code defines a solution to add a row of nodes with a specific value at a given depth in a binary tree. To analyze the time and space complexity, let's consider `n` to be the total number of nodes in the binary tree.

Time Complexity:

The time complexity of the code can be determined by the number of nodes the algorithm visits. The function `dfs` is a recursive function that visits each node exactly once when the depth is equal to or greater than the depth of insertion (`d >= depth`).

In the worst-case scenario, which occurs when the new row is added at the maximum depth of the tree, the algorithm must visit all `n` nodes to determine their depth and to potentially add the new nodes.

Therefore, the time complexity of the code is `O(n)`.

Space Complexity:

Space complexity comes from the recursive stack space used in the depth-first search (DFS). In the worst-case scenario, the depth of the recursive call stack is proportional to the height of the tree.

- In the case of a balanced binary tree, the height of the tree is logarithmic, and the space complexity would be `O(log n)`.
- For a skewed binary tree (a tree in which every node has only one child), the height could be as high as `n`, and the worst-case space complexity would be `O(n)`.

Therefore, the overall space complexity is `O(h)` where `h` is the height of the tree, which ranges from `O(log n)` for a balanced tree to `O(n)` for a skewed tree.