2773. Height of Special Binary Tree

Medium

Leetcode Link

Problem Description

You are given the root node of a special binary tree which is defined by nodes numbered from `1` to `n`. Unlike a regular binary tree, this special binary tree has a unique property for its leaves. These leaves are numbered `b_1` through `b_k`, representing their order. Here's what makes the leaves special:

- If a leaf node `b_i` has a right child, then this right child will be the next leaf in the order, `b_(i+1)`, unless `b_i` is the last leaf (`i = k`), in which case its right child loops back to the first leaf, `b_1`.
- Conversely, if a leaf node `b_i` has a left child, that child is the previous leaf in the order, `b_(i-1)`, unless `b_i` is the first leaf (`i = 1`), in which case its left child is the last leaf, `b_k`.

Your task is to calculate the height of this binary tree. The height is the length of the longest path from the root node to any other node in the tree.

Intuition

To find the height of the tree, we need to determine the longest path from the root node down to the farthest leaf node. The solution involves a depth-first search (DFS) algorithm to traverse the tree. The intuition behind using DFS is that we can explore as far down a branch as possible before backtracking, thus naturally finding the longest path.

With DFS, we start from the root and go as deep as possible along each branch before backtracking, which allows us to calculate the depth (`d`) for each node. We keep track of the maximum depth we encounter during our traversal using a variable, `ans`, that is initialized to 0. As we dive into each node, we increment the depth.

We have two important conditions to check at each node:

- For the left child of a node, we ensure that it is not coming back to the current node via the special property of right child equals parent node (`root.left.right != root`), because in that case, it would not be a valid path down the tree; it would be moving back up and shouldn't be considered for depth calculation.
- The same logic applies to the right child (`root.right.left != root`).

By ensuring these conditions, we accurately calculate the depth only for paths that go down the tree. During DFS, every time we move down a level to a child node, we increase `d` by 1. Finally, when the DFS is completed, `ans` will hold the maximum depth value—that is, the height of the tree.

Solution Approach

The implementation of the solution uses a recursive depth-first search (DFS) algorithm to traverse the binary tree and find its height. The main steps of the solution are as follows:

1. Define a helper function `dfs` that takes two arguments: `root`, which is the current node, and `d`, which is the current depth from the root node to the current node.

2. Initialize a variable `ans` in the outer scope of the `dfs` function (using `nonlocal` in Python) to keep track of the maximum depth encountered during the traversal of the tree.

3. In the `dfs` function, update `ans` to be the maximum of its current value or the depth `d`.

4. Check the left child of the current node. If the current node's left child exists and its right child is not the current node itself (`root.left.right != root`), then recurse on the left child with an increased depth `d + 1`.

5. Check the right child of the current node. If the current node's right child exists and its left child is not the current node itself (`root.right.left != root`), then recurse on the right child with an increased depth `d + 1`.

6. The recursion will eventually visit all the nodes in the binary tree while respecting the special property of the leaves. Since `ans` is updated at each node with the maximum depth, by the end of the recursion, it will hold the value of the height of the tree.

7. Call the `dfs` function initially with the `root` of the tree and a starting depth of `0`.

8. After the `dfs` function has completed the traversal, return the value of `ans`, which is the height of the given binary tree.

The `dfs` helper function is necessary to perform the depth-first search, and the use of the `nonlocal` keyword allows us to modify the `ans` variable defined outside of the function scope.

This approach ensures that all pathways down the tree are explored, and only valid pathways that follow the specific properties of this special binary tree are considered when calculating the maximum depth.

By using this recursive DFS strategy, we are able to calculate the height of the tree efficiently without having to separately store or manage the nodes visited.

Example Walkthrough

Suppose we have a special binary tree with `5` nodes where `3` is the root, and the leaves are ordered as follows: `1`, `4`, and `5`. The special property indicates that leaf `1` has a right child which is leaf `4` and similarly, leaf `4` has a right child which is leaf `5`. Leaf `5`, being the last leaf, has its right child linked back to the first leaf, `1`. According to the special property, leaves shouldn't have the parent node as an immediate right or left child.

Here is our example tree for reference:

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

Now let's walk through the solution approach:

1. Define the helper function `dfs` with the root node `3` and the current depth `d` which is `0`.

2. Initialize `ans` to `0`. This will keep track of the maximum depth.

3. Inside `dfs`, compare the current depth (starting with `0` for the root) with `ans`, and update `ans` if the depth is greater.

4. Check the left child of node `3`, which is node `2`. It does not violate the special property, so we perform `dfs` on this node with a depth of `1` (`d + 1`).

5. Node `2` has a left child `1`. Recurse `dfs` with node `1`. The current depth is `2`. Since this is a leaf, and it doesn't have a left child linking back to its parent (`3`), we update `ans` to `2`.

6. Node `2` (left child of the root) also has a right child `4`, but we don't recurse here since `4` has a right child which is `5`, not `2`.

7. Now go back to the root node `3` and check its right child, which is another node `2`. Follow the same process as in steps 4 and 5. Since node `2` does not have a left child, we move to its right child `5`.

8. We recurse on node `5` with a depth of `2`. Since node `5` does not have a left child that links back to its parent (`2`), we check further down. We arrive at node `1`, which is leaf `5`'s right child, but we don't recurse here because its right child loops back to the starting leaf, thus forming a cycle, not a downward path.

9. After this, finish the traversal, and `ans` now holds the maximum depth we encountered, which is `2`.

10. We return `ans` as the final answer, so in this case, the height of the tree is `2`.

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 heightOfTree(self, root: Optional[TreeNode]) -> int:
        # Helper function to perform a depth-first search to calculate the height.
        def dfs(node: Optional[TreeNode], depth: int):
            # Access the outer variable 'max_height' to keep track of the tree's height.
            nonlocal max_height
            # Update the maximum height reached so far.
            max_height = max(max_height, depth)
            # Recurse on the left child if it exists and is not creating a cycle.
            if node.left and node.left.right != node:
                dfs(node.left, depth + 1)
            # Recurse on the right child if it exists and is not creating a cycle.
            if node.right and node.right.left != node:
                dfs(node.right, depth + 1)

        # Initialize the maximum height to 0.
        max_height = 0
        # Start the DFS from the root node at depth 0.
        dfs(root, 0)
        # Return the maximum height of the tree.
        return max_height

```

Java Solution

```java
class Solution {
    private int maxDepth; // renaming 'ans' to 'maxDepth' for better clarity

    // Method to find the maximum depth of a binary tree
    public int heightOfTree(TreeNode root) {
        // Start the depth-first search from the root with an initial depth of 0
        dfs(root, 0);
        // After DFS is complete, `maxDepth` will contain the height of the tree
        return maxDepth;
    }

    // Helper method to perform a depth-first search on the tree
    private void dfs(TreeNode node, int depth) {
        // Update the maximum depth reached so far
        maxDepth = Math.max(maxDepth, depth);

        // Increment the depth because we're going one level deeper in the tree
        depth++;

        // Recursively call the DFS method on the left child, if it's not null
        // and it doesn't incorrectly point back to the current node
        if (node.left != null && node.left.right != node) {
            dfs(node.left, depth);
        }

        // Similarly, recursively call the DFS method on the right child with the same checks
        if (node.right != null && node.right.left != node) {
            dfs(node.right, depth);
        }
    }
}

// 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;
    }
}

```

C++ Solution

```cpp
#include <functional> // for std::function

// 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:
    // Function to find the height of a binary tree.
    int heightOfTree(TreeNode* root) {
        // Variable to store the final answer - the height of the tree.
        int maxDepth = 0;

        // Lambda function to perform a depth-first search on the tree.
        // It captures the maxDepth by reference.
        std::function<void(TreeNode*, int)> dfs = [&](TreeNode* node, int depth) {
            // Update maxDepth based on the current depth.
            maxDepth = std::max(maxDepth, depth);

            // Increment depth for the next level.
            ++depth;

            // If there's a left child and it's not pointing back to the current node (to avoid cycles),
            // recurse into the left subtree.
            if (node->left && node->left->right != node) {
                dfs(node->left, depth);
            }

            // Do the same for the right child.
            if (node->right && node->right->left != node) {
                dfs(node->right, depth);
            }
        };

        // Invoke the DFS starting from the root at depth 0.
        dfs(root, 0);

        // Return the maximum depth, which is the height of the tree.
        return maxDepth;
    }
};

```

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)
        this.left = (left === undefined ? null : left)
        this.right = (right === undefined ? null : right)
    }
}

/**
 * Computes the height of a binary tree.
 * @param {TreeNode | null} root - The root node of the binary tree.
 * @return {number} The height of the tree.
 */
function heightOfTree(root: TreeNode | null): number {
    // Initialize the answer to zero
    let maxHeight = 0;

    /**
     * Depth-first search recursive helper function to determine the height of the tree.
     * @param {TreeNode | null} node - The current node.
     * @param {number} depth - The current depth of the node in the tree.
     */
    const dfs = (node: TreeNode | null, depth: number) => {
        if (node === null) {
            // If the node is null, we are at the end of a path, so update the answer if necessary
            maxHeight = Math.max(maxHeight, depth);
            return;
        }

        // If the left child is not null and doesn't point back to the current node (to avoid cycles)
        if (node.left !== null && node.left !== node) {
            dfs(node.left, depth + 1);
        }

        // If the right child is not null and doesn't point back to the current node (to avoid cycles)
        if (node.right !== null && node.right !== node) {
            dfs(node.right, depth + 1);
        }
    };

    // Start the DFS traversal with root node and initial depth of 0
    dfs(root, 0);

    // After traversal, return the found maximum height of the tree
    return maxHeight;
}

```

Time and Space Complexity

Time Complexity

The provided code performs a depth-first search (DFS) on a tree. During the DFS, each node is visited exactly once. For a tree with `n` nodes, the time complexity is `O(n)`, since every node is checked to determine its height.

However, the code also includes additional conditional checks that are intended to avoid moving in cycles (like a check to see if `root.left.right != root` and `root.right.left != root`). But since this is a binary tree, these checks are unnecessary and do not impact the overall time complexity. They are supposed to validate that we do not move back to the parent, yet by the nature of binary trees, this condition is redundant; hence the time complexity remains `O(n)`.

Space Complexity

The space complexity of the code is primarily affected by the recursive DFS, which uses space on the call stack proportional to the height of the tree for its execution context. In the worst case (a completely unbalanced tree), the space complexity would be `O(n)`. However, in a balanced tree, the space complexity would be `O(log n)` due to the reduced height of the tree. The variable `ans` used to maintain the maximum depth does not significantly contribute to space complexity.

Moreover, the space complexity is also influenced by the environment in which python functions execute. The usage of the `nonlocal` keyword allows the `DFS` internal function to modify a variable in its enclosing scope (`ans` in this case), but it does not add to space complexity.

To summarize, the space complexity is `O(n)` in the worst case and `O(log n)` in the average or best case (balanced tree).