104. Maximum Depth of Binary Tree

Easy Tree Depth-First Search Breadth-First Search Binary Tree

Problem Description

In this LeetCode problem, we are given a binary tree, which is a tree data structure where each node has at most two children, referred to as the left child and the right child. The task is to find the maximum depth of the tree. The maximum depth is defined as the length of the longest path from the root node of the tree down to the farthest leaf node. The length of a path is measured by the number of nodes along that path. In this context, a leaf node is a node with no children, signifying that it's at the edge of the tree.

Intuition

To determine the solution to finding the maximum depth of a binary tree, we can use a strategy known as depth-first search (DFS).

The intuition behind the approach is quite straightforward:

- If we start at the root and the tree is empty, the maximum depth is zero.
- If the tree is not empty, we can recursively find the maximum depth of the left and right subtrees.
- The maximum depth of the tree would then be the larger of the two maximum depths found, plus one for the root node itself.

This recursive strategy hinges upon the idea that the depth of a tree is equal to the depth of its deepest subtree, plus a factor of one for the root. As we explore each subtree, we keep on asking the same question, 'what's the maximum depth from this node down?'. We keep on doing this recursively until we reach the leaf nodes, which have a depth of zero since there are no further nodes below them.

The solution leverages this idea and recursively descends through the tree, ensuring that the maximum depth is calculated for each subtree. Once the recursion reaches the leaf nodes, it begins to unwind, cumulatively calculating the depth by comparing the depths of the left and right subtrees at each step, and adding one to account for the current node's depth contribution. By the time the recursion unwinds back to the root, we would have found the maximum depth.

Solution Approach

The solution provided is an example of a depth-first search (DFS) algorithm implemented using recursion, which is a common pattern for traversing trees. The approach is simple and consists of the following steps:

1. Base Case Check: At the start of the `maxDepth` method, the base case checks if the current node, initially the root, is `None`. If it is, this means that we have hit the bottom of the tree (or the tree is empty to begin with), and we return `0` as the depth. Every leaf will eventually hit this base case.

```
if root is None:
    return 0
```

2. Recursive Calls: If the node is not `None`, we proceed to make recursive calls for the left child and right child of the current `root` node.

```
l, r = self.maxDepth(root.left), self.maxDepth(root.right)
```

   By calling `self.maxDepth` on `root.left` and `root.right`, we are asking for the maximum depth from each child node.

3. Calculating Depth: After receiving the maximum depths from the left and right subtrees (`l` and `r`), we calculate the maximum depth of the current tree by taking the `max` of `l` and `r`, and adding `1` to account for the current root node.

```
return 1 + max(l, r)
```

4. Climbing up the Recursion: This step essentially repeats for each node in the tree until all nodes have been visited, and at each step, we climb up the tree's layers, cumulatively calculating the maximum depth by comparing the depths from the left and right.

The choice of DFS and recursion in this case allows for an elegant and easily understandable solution to the problem. It's worth mentioning that every time a recursive call is made, the call stack keeps track of each node's state, enabling the process to 'remember' the paths taken once it needs to return and subsequently combine the depths found.

Overall, this implementation relies heavily on the nature of recursion to break down the problem into manageable pieces, solve each minor problem, and combine the results to arrive at the final solution, serving as a clear illustration of a divide-and-conquer strategy.

Example Walkthrough

Let's illustrate the solution approach using a small example of a binary tree. Assume we have the following binary tree:

```
1       1
2      / \
3     2   3
4    / \   \
5   4   5   6
6  /         \
7 7           8
```

In this tree, the root node is `1`, and its left child is `2`, and its right child is `3`. Continuing down the tree, `2` has two children `4` and `5`, and `3` has one child `6`. On the next level, `4` has one child `7` and `6` has one child `8`. The leaf nodes in this tree are `5`, `7`, and `8`.

Let's walk through the recursive solution to find the maximum depth of this tree:

1. **Starting at the root node (1)**:
   - The `maxDepth` method is called with the root node `1`. Since `1` is not `None`, we perform the recursive calls on its children (`2` and `3`).
2. **Explore the left subtree of node (2)**:
   - The `maxDepth` method is called on the left child (`2`), which is not `None`. Recursive calls are made on its children (`4` and `5`).
3. **Explore the left subtree of node (4)**:
   - Continuing the recursion, `maxDepth` is called on the child `4`. It has a left child `7`, so another recursive call is made.
4. **Leaf node (7)**:
   - The `maxDepth` method is called on `7`. With no children, `7` is a leaf node, reaching the base case. Hence, it returns `0`, indicating a node's depth is zero below it.
5. **Climb up from leaf node (7) to node (4)**:
   - Since `4` has no right child, it compares the depths of left (`0+1`) and (non-existent) right subtrees and returns `1`.
6. **Climb up to node (2)**:
   - Now we consider node `5`, which is a leaf node. It hits the base case and returns `0`.
   - Back at node `2`, we compare the maximum depths received from `4` (`1`) and `5` (`0`), and add `1` for the node `2` itself. So `2` returns `2`.
7. **Explore the right subtree of node (3)**:
   - The recursion calls `maxDepth` on node `3`. It has a right child `6`, so we call `maxDepth` on `6`.
8. **Explore the right subtree of node (6)**:
   - Node `6` has a right child `8`, and invoking `maxDepth` on `8` leads to a base case returning `0`.
9. **Climb up from leaf node (8) to node (6)**:
   - Node `6` has no left child, so it takes the maximum of left (non-existent) and right (`0+1`) depths, resulting in `1`.
10. **Climb up to node (3)**:
    - At node `3`, we compare the maximum depths from `6` (`1`), and since `3` has no left child, we conclude node `3`'s subtree has a maximum depth of `2` (`1` from `6` plus `1` for the `3` itself).
11. **Combine results at the root node (1)**:
    - Finally, at the root node `1`, we compare the depths received from `2` (`2`) and `3` (`2`), taking the maximum, which is `2`, and add `1` for the root node. Hence, we determine that the maximum depth of the tree is `3`.

This recursive process explored each subtree, remembered the maximum depth at each level, and combined the results to deliver the overall maximum depth.

Python Solution

```
# 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 maxDepth(self, root: TreeNode) -> int:
        # If the current node is None, it means this is an empty tree or we've reached the end of a branch.
        # Return 0 in this case as it contributes no depth.
        if root is None:
            return 0

        # Recursively find the depth of the left subtree.
        # This will traverse all the way down to the leftmost leaf node.
        left_depth = self.maxDepth(root.left)

        # Recursively find the depth of the right subtree.
        # This will traverse all the way down to the rightmost leaf node.
        right_depth = self.maxDepth(root.right)

        # The maximum depth of the current node will be 1 (for the current node) plus the maximum
        # of the depths of its left and right subtrees.
        # This ensures that the longest path from root to leaf is counted.
        return 1 + max(left_depth, right_depth)
```

Java Solution

```
// Definition for a binary tree node.
class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;

    // Constructor to create a leaf node.
    TreeNode(int val) {
        this.val = val;
    }

    // Constructor to create a node with specific children.
    TreeNode(int val, TreeNode left, TreeNode right) {
        this.val = val;
        this.left = left;
        this.right = right;
    }
}

class Solution {
    // Calculates the maximum depth of a binary tree.
    public int maxDepth(TreeNode root) {
        // If the root is null, the depth is 0.
        if (root == null) {
            return 0;
        }

        // Recursively compute the depth of the left subtree.
        int leftDepth = maxDepth(root.left);

        // Recursively compute the depth of the right subtree.
        int rightDepth = maxDepth(root.right);

        // The depth of the current node is the greater of its two children's depths plus one.
        return 1 + Math.max(leftDepth, rightDepth);
    }
}
```

C++ Solution

```
// Definition for a binary tree node.
struct TreeNode {
    int val; // value of the node
    TreeNode *left; // pointer to the left child
    TreeNode *right; // pointer to the right child

    // Constructor to initialize the node with a value, and nullptr for children
    TreeNode() : val(0), left(nullptr), right(nullptr) {}

    // Constructor to initialize the node with a value and no children
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}

    // Constructor to initialize the node with a value and left and right children
    TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
};

class Solution {
public:
    // Function to find the maximum depth of a binary tree.
    int maxDepth(TreeNode* root) {
        // Base case: if the current node is null, return 0 as the depth
        if (!root)
            return 0;

        // Recursively compute the depth of the left subtree
        int leftDepth = maxDepth(root->left);
        // Recursively compute the depth of the right subtree
        int rightDepth = maxDepth(root->right);

        // The maximum depth of the current node is 1 (for the current node) plus
        // the greater depth between the left and right subtrees
        return 1 + std::max(leftDepth, rightDepth);
    }
};
```

Typescript Solution

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

/**
 * This function computes the maximum depth of a binary tree.
 * The maximum depth is the number of nodes along the longest path
 * from the root node down to the farthest leaf node.
 *
 * @param {TreeNode | null} root - The root node of the binary tree.
 * @return {number} The depth of the binary tree.
 */
function maxDepth(root: TreeNode | null): number {
    // An empty tree has a depth of zero
    if (root === null) {
        return 0;
    }
    // Recursively compute the depth of the left and right subtree
    // and return the greater one incresed by 1 for the current node
    return 1 + Math.max(maxDepth(root.left), maxDepth(root.right));
}
```

Time and Space Complexity

Time Complexity: The time complexity of the code is `O(n)`, where `n` is the number of nodes in the tree. This is because the algorithm is a depth-first search, and it visits each node exactly once to determine the maximum depth.

Space Complexity: The space complexity of the code is `O(h)`, where `h` is the height of the tree. This space is used by the call stack during the recursion. In the worst case, if the tree is completely unbalanced, with all nodes on one side, the height of the tree `h` can be equal to `n`, leading to a space complexity of `O(n)`. In the best case, for a completely balanced tree, the height `h` would be `log(n)`, leading to a space complexity of `O(log(n))`.