# 111. Minimum Depth of Binary Tree

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

Leetcode Link

## Problem Description

In this problem, we are given a binary tree and we are required to determine its minimum depth. The minimum depth is defined as the number of nodes present along the shortest path from the root node to the nearest leaf node. Importantly, a leaf node is characterized as a node that does not have any children.

## Intuition

To solve this problem, the idea is to use a depth-first search (DFS) approach to recursively traverse the tree and find the depth of the shortest path from the root to a leaf node. Here are the steps we would follow:

- **Base Cases:** First, we check some base cases:
  - If the node we're looking at is `None`, we return 0 because we're dealing with a non-existent (empty) sub-tree.
  - If the node is a leaf node (both `left` and `right` children are `None`), we would return 1 because the path consists only of the current node.
- **Recursive Steps:** If the base cases aren't met, we proceed with the following logic:
  - If only one of the children is `None`, this means that the path must go through the side that has a child. We recursively calculate the depth of this valid side and add 1 for the current node.
  - If both left and right children are present, we recursively calculate the depth of both sides and choose the smaller one, then add 1 to account for the current node.

By applying this approach, we ensure that we find the shortest path that reaches a leaf node, satisfying the requirement to determine the minimum depth of the tree.

## Solution Approach

The implementation of the solution uses a straightforward recursive depth-first search (DFS) technique to traverse the binary tree. The following details outline the steps taken in the algorithm along with the relevant data structures and patterns:

- A recursive function `minDepth` is defined which takes a TreeNode as an argument.
- **Base Case Handling:** The algorithm first checks if the `root` node is `None` and returns 0, indicating an empty tree has a depth of 0.
- For a non-empty tree, two scenarios are handled separately:
  1. **One Child is `None`:** If either `root.left` or `root.right` is `None`, this implies that we do not need to consider that path because it doesn't lead to a leaf. The depth, in this case, is found by recursively calling `minDepth` on the non-`None` child and adding 1 for the current node.
  2. **Both Children Exist:** If both `root.left` and `root.right` are not `None`, the recursion is performed on both and the minimum of the two depths is taken. This minimum depth represents the shortest path to a leaf node because we are interested only in the smallest count of steps to reach a leaf.
- **Combining Recursive Call Results:** After considering the appropriate cases, the depth is incremented by 1 to account for the current `root` node and the result is returned.

No additional data structures are needed; the recursive stack implicitly built from the function calls keeps track of the recursion depth. The DFS pattern is suitable because we need to dive into the depths of the tree branches to find leaf nodes and evaluate their depth.

The Python code for the reference solution ensures that the branching occurs as necessary and handles each node in an efficient and succinct manner.

```python
class Solution:
    def minDepth(self, root: Optional[TreeNode]) -> int:
        if root is None:
            return 0
        if root.left is None:
            return 1 + self.minDepth(root.right)
        if root.right is None:
            return 1 + self.minDepth(root.left)
        return 1 + min(self.minDepth(root.left), self.minDepth(root.right))
```

This recursive approach is both elegant and effective for the problem at hand.

### Example Walkthrough

Let's illustrate the solution approach with a small example binary tree:

Let's say our binary tree looks like this (numbers represent the node values, not depth):

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

We need to find the minimum depth of this tree. We apply the solution approach as follows:

1. We start with the root node (1). Since it is not `None`, we move to the recursive steps.
2. The root node (1) has two children – nodes 2 and 3.
3. We then apply the DFS approach. We look at the left child first, which is node 2.
4. Node 2 is not a leaf since it has one child, node 4. Since its right child is `None`, we find the depth by recursively calling `minDepth` on its left child (node 4).
5. Node 4 is a leaf node because it has no children. Hence the base case is met, and we return 1.
6. Now, since node 2 had its right child as `None`, according to our recursive formula, we return 1 (depth of node 4) + 1 (node 2 itself), which equals 2.
7. Now back to the root node (1), we need to consider its right child, which is node 3.
8. Node 3 is a leaf node with no children. Hence, the base case is met, and we return 1.
9. Now, we compare the minimum depth between the left and right subtrees of the root. The left subtree has a depth of 2 (from node 2 to node 4), and the right subtree has a depth of 1 (node 3 alone).
10. We, therefore, choose the smaller depth, which is 1, and add 1 to account for the root. So, the minimum depth of the tree is 1 + 1 = 2, which represents the path [1 -> 3].

Here's how the solution approach effectively finds the minimum depth for this binary tree which is 2, denoting the shortest path from root to the nearest leaf node.

## Python Solution

```python
class TreeNode:
    """Definition for a binary tree node."""

    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right


class Solution:
    def minDepth(self, root: Optional[TreeNode]) -> int:
        """Calculate the minimum depth of the binary tree.

        The minimum depth is the number of nodes along the shortest path
        from the root node down to the nearest leaf node.
        """
        # If the root is None, the tree is empty, so return 0.
        if root is None:
            return 0

        # If there is no left child, compute the minDepth of the right subtree
        # and add 1 to account for the current node.
        if root.left is None:
            return 1 + self.minDepth(root.right)

        # If there is no right child, compute the minDepth of the left subtree
        # and add 1 to account for the current node.
        if root.right is None:
            return 1 + self.minDepth(root.left)

        # If the tree has both left and right children, compute the minDepth
        # of both subtrees, take the minimum, and add 1 for the current node.
        return 1 + min(self.minDepth(root.left), self.minDepth(root.right))
```

## 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;
    }
}

/**
 * This class contains a method to calculate the minimum depth of a binary tree.
 */
class Solution {

    /**
     * Finds the minimum depth of a binary tree recursively.
     * The minimum depth is the number of nodes along the shortest path from
     * the root node down to the nearest leaf node.
     *
     * @param root The root node of the binary tree.
     * @return The minimum depth of the tree.
     */
    public int minDepth(TreeNode root) {
        // If root is null, then there is no tree, and the depth is 0.
        if (root == null) {
            return 0;
        }

        // If the left child is null, recursively find the minDepth of the right subtree.
        // Here the right subtree's minDepth is used because without the left child,
        // the path through the right child is the only path to a leaf node.
        if (root.left == null) {
            return 1 + minDepth(root.right);
        }

        // If the right child is null, recursively find the minDepth of the left subtree.
        // Analogous to the previous condition, the left subtree's minDepth is used here.
        if (root.right == null) {
            return 1 + minDepth(root.left);
        }

        // If neither child is null, find the minDepth of both subtrees and
        // take the minimum of the two, adding 1 for the current node.
        return 1 + Math.min(minDepth(root.left), minDepth(root.right));
    }
}
```

## C++ Solution

```cpp
#include <algorithm> // To use the std::min function

/**
 * Definition for a binary tree node.
 */
struct TreeNode {
    int val; // Renamed 'val' to 'value' for clarity
    TreeNode *left;
    TreeNode *right;
    // Constructor initializes the node with a value and optional left and right children
    TreeNode(int val, TreeNode *leftChild = nullptr, TreeNode *rightChild = nullptr)
        : val(val), left(leftChild), right(rightChild) {}
};

class Solution {
public:
    // Function to find the minimum depth of a binary tree
    int minDepth(TreeNode* root) {
        // Base case: if tree is empty, return depth of 0
        if (!root) {
            return 0;
        }

        // If left child does not exist, recurse on right subtree and add 1 to the depth
        if (!root->left) {
            return 1 + minDepth(root->right);
        }

        // If right child does not exist, recurse on left subtree and add 1 to the depth
        if (!root->right) {
            return 1 + minDepth(root->left);
        }

        // If both left and right children exist, find the minimum of the two depths and add 1
        return 1 + std::min(minDepth(root->left), minDepth(root->right));
    }
};
```

## Typescript Solution

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

/**
 * Calculate the minimum depth of a binary tree.
 * The minimum depth is the number of nodes along the shortest path from the root node down to the nearest leaf node.
 *
 * @param {TreeNode | null} root - The root node of the binary tree.
 * @return {number} - The minimum depth of the tree.
 */
function minDepth(root: TreeNode | null): number {
    // Base case: if the tree is empty, return 0
    if (root === null) {
        return 0;
    }

    // Get the left and right child nodes
    const leftChild = root.left;
    const rightChild = root.right;

    // If the left child is null, recursively calculate the minimum depth of the right subtree and add 1 for the current node
    if (leftChild === null) {
        return 1 + minDepth(rightChild);
    }

    // If the right child is null, recursively calculate the minimum depth of the left subtree and add 1 for the current node
    if (rightChild === null) {
        return 1 + minDepth(leftChild);
    }

    // If both child nodes are present, calculate the minimum depth of both subtrees, take the lesser value, and add 1 for the currer
    return 1 + Math.min(minDepth(leftChild), minDepth(rightChild));
}
```

## Time and Space Complexity

The given code is a recursive solution to find the minimum depth of a binary tree. The minimum depth is the number of nodes along the shortest path from the root node down to the nearest leaf node.

### Time Complexity

The time complexity of the function depends on the structure of the tree. In the worst case, the tree is completely unbalanced, i.e., every node has only one child. This leads to a recursive call at each level, resulting in $O(n)$ time complexity, where $n$ is the number of nodes in the tree, as each node is visited once.

In the best case, the tree is completely balanced, and the number of nodes in the tree is 1 for the first level, 2 for the second level, 4 for the third, and so on, until $l$ levels. The total number of nodes $n$ in a balanced binary tree is $n = 2^l - 1$. The function calls are akin to a binary search in this scenario, the time complexity is $O(\log n)$ for a balanced binary tree.

Overall, the average and worst-case time complexities can be considered $O(n)$, since you cannot guarantee the tree is balanced.

### Space Complexity

The space complexity is mainly determined by the recursive call stack. In the worst case, where the tree is completely unbalanced (every node has only one child), there would be $n$ recursive calls, resulting in $O(n)$ space complexity.

In the best case, where the tree is balanced, the height of the tree is $\log n$ (where $n$ is the number of nodes). This would result in a space complexity of $O(\log n)$ due to the height of the recursive call stack.

Thus, the space complexity of the function varies from $O(\log n)$ for a balanced binary tree to $O(n)$ for an unbalanced binary tree where $n$ is the number of nodes.