111. Minimum Depth of Binary Tree

Oct. 28, 2018 | 32.3K views

Average Rating: 4.17 (23 votes)

Given a binary tree, find its minimum depth.

The minimum depth is the number of nodes along the shortest path from the root node down to the nearest leaf node.

Note: A leaf is a node with no children.

Example:

Given binary tree `[3,9,20,null,null,15,7]`,

```
    3
   / \
  9  20
    /  \
   15   7
```

return its minimum depth = 2.

Solution

Tree definition

First of all, here is the definition of the `TreeNode` which we would use.

Java Python Copy

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

  TreeNode(int x) {
    val = x;
  }
}
```

Approach 1: Recursion

Algorithm

The intuitive approach is to solve the problem by recursion. Here we demonstrate an example with the DFS (Depth First Search) strategy.

Java Python Copy

```java
class Solution {
  public int minDepth(TreeNode root) {
    if (root == null) {
      return 0;
    }

    if ((root.left == null) && (root.right == null)) {
      return 1;
    }

    int min_depth = Integer.MAX_VALUE;
    if (root.left != null) {
      min_depth = Math.min(minDepth(root.left), min_depth);
    }
    if (root.right != null) {
      min_depth = Math.min(minDepth(root.right), min_depth);
    }

    return min_depth + 1;
  }
}
```

Complexity analysis

- Time complexity : we visit each node exactly once, thus the time complexity is $\mathcal{O}(N)$, where N is the number of nodes.
- Space complexity : in the worst case, the tree is completely unbalanced, *e.g.* each node has only one child node, the recursion call would occur N times (the height of the tree), therefore the storage to keep the call stack would be $\mathcal{O}(N)$. But in the best case (the tree is completely balanced), the height of the tree would be $\log(N)$. Therefore, the space complexity in this case would be $\mathcal{O}(\log(N))$.

Approach 2: DFS Iteration

We could also convert the above recursion into iteration, with the help of stack.

> The idea is to visit each leaf with the DFS strategy, while updating the minimum depth when we reach the leaf node.

So we start from a stack which contains the root node and the corresponding depth which is `1`. Then we proceed to the iterations: pop the current node out of the stack and push the child nodes. The minimum depth is updated at each leaf node.

Java Python Copy

```java
class Solution {
  public int minDepth(TreeNode root) {
    LinkedList<Pair<TreeNode, Integer>> stack = new LinkedList<>();
    if (root == null) {
      return 0;
    }
    else {
      stack.add(new Pair(root, 1));
    }

    int min_depth = Integer.MAX_VALUE;
    while (!stack.isEmpty()) {
      Pair<TreeNode, Integer> current = stack.pollLast();
      root = current.getKey();
      int current_depth = current.getValue();
      if ((root.left == null) && (root.right == null)) {
        min_depth = Math.min(min_depth, current_depth);
      }
      if (root.left != null) {
        stack.add(new Pair(root.left, current_depth + 1));
      }
      if (root.right != null) {
        stack.add(new Pair(root.right, current_depth + 1));
      }
    }
    return min_depth;
  }
```

Complexity analysis

- Time complexity : each node is visited exactly once and time complexity is $\mathcal{O}(N)$.
- Space complexity : in the worst case we could keep up to the entire tree, that results in $\mathcal{O}(N)$ space complexity.

Approach 3: BFS Iteration

The drawback of the DFS approach in this case is that all nodes should be visited to ensure that the minimum depth would be found. Therefore, this results in a $\mathcal{O}(N)$ complexity. One way to optimize the complexity is to use the BFS strategy. We iterate the tree level by level, and the first leaf we reach corresponds to the minimum depth. As a result, we do not need to iterate all nodes.

Java Python Copy

```java
class Solution {
  public int minDepth(TreeNode root) {
    LinkedList<Pair<TreeNode, Integer>> stack = new LinkedList<>();
    if (root == null) {
      return 0;
    }
    else {
      stack.add(new Pair(root, 1));
    }

    int current_depth = 0;
    while (!stack.isEmpty()) {
      Pair<TreeNode, Integer> current = stack.poll();
      root = current.getKey();
      current_depth = current.getValue();
      if ((root.left == null) && (root.right == null)) {
        break;
      }
      if (root.left != null) {
        stack.add(new Pair(root.left, current_depth + 1));
      }
      if (root.right != null) {
        stack.add(new Pair(root.right, current_depth + 1));
      }
    }
    return current_depth;
  }
```

Complexity analysis

- Time complexity : in the worst case for a balanced tree we need to visit all nodes level by level up to the tree height, that excludes the bottom level only. This way we visit $N/2$ nodes, and thus the time complexity is $\mathcal{O}(N)$.
- Space complexity : is the same as time complexity here $\mathcal{O}(N)$.

Rate this article:

Previous Next