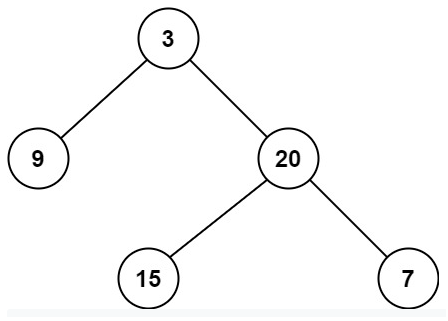
<https://leetcode.com/problems/minimum-depth-of-binary-tree/>

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 1:**



Input: root = [3,9,20,null,null,15,7]

Output: 2

**Example 2:**

Input: root = [2,null,3,null,4,null,5,null,6]

Output: 5

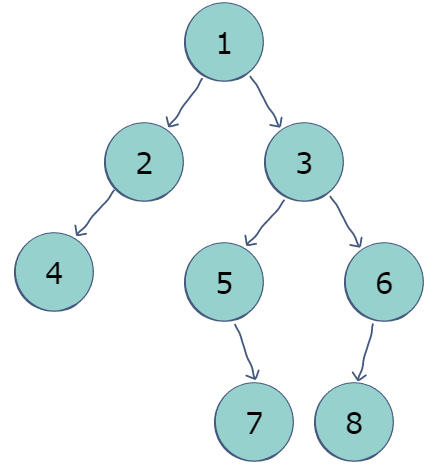
**Constraints:**

* The number of nodes in the tree is in the range [0, 105].
* -1000 <= Node.val <= 1000

**What is the minimum depth of a binary tree ?**

The **minimum depth of a binary tree** is the number of nodes from the root node to the **nearest leaf node**.

Consider the binary tree below:



The minimum depth of this tree is 33; it is comprised of nodes 1, 2, and 4.

**Attempt 1: 2022-11-09**

**Wrong Solution:**

**There is a big trap just change Maximum Depth of Binary Tree recursion logic by below:**

public int maxDepth(TreeNode root) {

if(root == null) {

return 0;

}

int left = maxDepth(root.left);

int right = maxDepth(root.right);

return Math.max(left, right) + 1;

}

**Only change last statement from Math.max(...) into Math.min(...) is TOTALLY WRONG !!!**

public int minDepth(TreeNode root) {

if(root == null) {

return 0;

}

int left = minDepth(root.left);

int right = minDepth(root.right);

return Math.min(left, right) + 1;

}

**Failure test: [1, 2]**

Input: [1,2]

Expected: 2 -----> Note: a little different than standard definition on tree depth, in standard definition tree root's depth is 0, but in example here tree root's depth is 1, that's why expect depth = 2 rather than 1

Ouput: 1

1

/

2

**Why failure ?**

To clarify the explanation - for the max length, the longest path will always choose a leaf node path over an empty path, even for edge cases like [1,2]. But if you substitute the same code from the max path for min path, whenever you have an edge case like [1,2], the code will pick the empty path length over the leaf node path length, in another word, **as the minimum depth of binary tree definition, the depth must get from root node to leaf node if we have rather than stop when encounter empty node(null), for given input [1,2] we have leaf node as 2 rather than null, so minimum path should be 1 -> 2 rather than 1 -> null**, if we just change Math.max(left, right) to Math.min(left, right) will pick up 1 > null,  which would result the wrong answer.

**How to fix ?**

To avoid wrongly stop at empty path, we have to add two conditions:

1. If current node's left child is empty, keep on checking its right child

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

2. If current node's right child is empty, keep on checking its left child

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

**Solution 1:  Divide and Conquer (10 min)**

/\*\*

\* Definition for a binary tree node.

\* public 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 {

public int minDepth(TreeNode root) {

if(root == null) {

return 0;

}

int left = minDepth(root.left);

int right = minDepth(root.right);

// If current node has no left subtree, try if right subtree exist

// to avoid stop find path because of empty left child node

if(root.left == null) {

return 1 + right;

}

// If current node has no right subtree, try if left subtree exist

// to avoid stop find path because of empty right child node

if(root.right == null) {

return 1 + left;

}

return Math.min(left, right) + 1;

}

}

**Solution 2:  Iterative level order traversal as BFS (10 min, much more efficient than Solution 1 DFS)**

/\*\*

\* Definition for a binary tree node.

\* public 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 {

public int minDepth(TreeNode root) {

if(root == null) {

return 0;

}

Queue<TreeNode> q = new LinkedList<TreeNode>();

q.offer(root);

int level = 1;

while(!q.isEmpty()) {

int size = q.size();

for(int i = 0; i < size; i++) {

TreeNode node = q.poll();

// Find a leaf node

if(node.left == null && node.right == null) {

return level;

}

if(node.left != null) {

q.offer(node.left);

}

if(node.right != null) {

q.offer(node.right);

}

}

level++;

}

return level;

}

}

**Refer to**

<https://www.enjoyalgorithms.com/blog/min-depth-of-binary-tree>

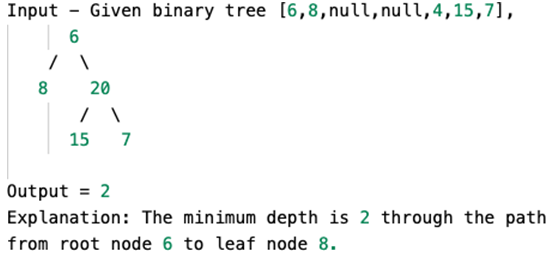
**Find Minimum Depth of Binary Tree**

**Key takeaway:** An excellent problem to learn problem-solving using both recursive and iterative tree traversal. This is also a question to understand efficient problem solving using BFS when the solution node is nearest to the root node.

**Let's understand the problem!**

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. The path has to end on a leaf node.

**Example**



**Discussed solution approaches**

Using depth-first Search: Recursive tree traversal

Using breadth-first search: Iterative tree traversal

**Using depth-first Search: Recursive tree traversal**

A binary tree is a recursive object, so we can think to solve this problem using recursive tree traversal. In other words, we can think to solve this problem using the solution of the two smaller subproblems: finding the min-depth of the left subtree and finding the min-depth of the right sub-tree. Here is an idea: If we know the min-depth of the left and right subtree, then min-depth of the tree = 1 + min ( min-depth of the left subtree, min-depth of the right subtree).

Suppose function treeMinDepth(root) will return the value of the min-depth of the overall tree.

if(root == Null), return 0. This is the base case which is the smallest version of the problem where recursion will backtrack and return the value directly.

if(root->left == Null), then we recursively call the same function for the right subtree and calculate the min-depth. In other words, there is no left sub-tree and min-depth of the tree = 1 + min-depth of the right subtree. **treeMinDepth(root) = 1 + treeMinDepth(root->right)**

if(root->right == Null), then we recursively call the same function for the left subtree and calculate the min-depth i.e. there is no right sub-tree and min-depth of the tree = 1 + min-depth of the left subtree. **treeMinDepth(root) = 1 + treeMinDepth(root->left)**

If both the children are not NULL, then we recursively call the same function for the left and right subtree, calculate the min-depth and get minimum of both to calculate the min-depth of the overall tree. treeMinDepth(root) = **1 + min (treeMinDepth(root->left), treeMinDepth(root->right))**

**Solution Pseudo code**

int treeMinDepth(TreeNode root) {

if(root == Null) return 0

if(root->left == Null) return 1 + treeMinDepth(root->right)

if(root->right == Null) return 1 + treeMinDepth(root->left)

return 1 + min(treeMinDepth(root->left), treeMinDepth(root->right))

}

**Time and space complexity analysis**

We are traversing each node of the tree only once. Time complexity = O(n) Space complexity=O(h) for recursion call stack. Space Complexity is equal to the maximum depth of the recursion tree which is equal to the height of the tree(h) where O(logn) < h < O(n)

**Using breadth-first search: Iterative tree traversal**

Now a critical question is: can we think to solve this problem iteratively using the BFS traversal? How do we calculate the min-depth using this idea? or, do we really need to calculate the min-depth for each node? Is there some information hidden in the problem which could help us to solve the problem efficiently? Think!

Here is an observation: the node with a minimum depth would be always the first leaf node if we traverse the tree level by level. In other words, if we traverse the tree level by level, then we need to return the depth of the first encountered leaf node. Think!

In a practical scenario, this idea would be much more efficient in comparison to the DFS approach because, in the DFS approach, we may end up with a complete traversal of the tree even when the topmost leaf is close to the root. But BFS traversal can access the node from top to bottom in level by order and we can get the topmost leaf quickly in fewer steps.

For example, If we have a tree where the left subtree has a depth of 100 and the right subtree has a depth of 1. The DFS will first traverse all the way down the 100 left subtree nodes before finally traversing the right subtree with a small depth of 1 and figuring out it as a min depth. But in BFS, instead of traversing 101 nodes to figure out the min depth, we can get the value in two steps. Now imagine the comparison of efficiency if there are thousands of nodes in the left subtree!

**Solution Pseudo code**

class QueueNode {

TreeNode node int depth

}

int treeMinDepth(TreeNode root) {

if (root == Null) return 0

else {

Queue Q

QueueNode tempQNode = QueueNode(root, 1)

Q.enqueue(temp)

while (Q.empty() == false) {

tempQNode = Q.dequeue()

TreeNode temp = tempQNode->node

int depth = tempQNode->depth

if (temp->left == Null && temp->right == Null) return depth

else {

if (temp->left != NULL) {

tempQNode->node = temp->left

tempQNode->depth = 1 + depth Q.enqueue(tempQNode)

}

if (temp->right != NULL) {

tempQNode->node = temp->right

tempQNode->depth = 1 + depth Q.enqueue(tempQNode)

}

}

}

}

return 0

}

**Time and space complexity analysis**

In the worst case, The total number of queue operations = 2n, because each node gets inserted and deleted only once in BFS traversal. Time complexity = O(n), where n is the total number of nodes. Space complexity = O(w) because BFS traversal requires queue size proportional to the max-width of the tree (w). (Think!)

**Critical ideas to think!**

Why do we need queue data structure in the level order traversal? What is the worst and best-case space complexity in the BFS approach? Can we solve this problem using all DFS traversals? Why do we prefer the BFS when the solution node is closer to the root? In which scenario, we prefer the DFS approach?

**Refer to**

<https://leetcode.com/problems/minimum-depth-of-binary-tree/discuss/36045/My-4-Line-java-solution/242269>

/\*\* Solution 1: DFS

\* Key point:

\* if a node only has one child -> MUST return the depth of the side with child, i.e. MAX(left, right) + 1

\* if a node has two children on both side -> return min depth of two sides, i.e. MIN(left, right) + 1

\* \*/

public int minDepth(TreeNode root) {

if (root == null) {

return 0;

}

int left = minDepth(root.left);

int right = minDepth(root.right);

if (left == 0 || right == 0) {

return Math.max(left, right) + 1;

}

else {

return Math.min(left, right) + 1;

}

}

/\*\* Solution 2: BFS level order traversal \*/

public int minDepth2(TreeNode root) {

if (root == null) {

return 0;

}

Queue<TreeNode> queue = new LinkedList<>();

queue.offer(root);

int level = 1;

while (!queue.isEmpty()) {

int size = queue.size();

for (int i = 0; i < size; i++) {

TreeNode curNode = queue.poll();

if (curNode.left == null && curNode.right == null) {

return level;

}

if (curNode.left != null) {

queue.offer(curNode.left);

}

if (curNode.right != null) {

queue.offer(curNode.right);

}

}

level++;

}

return level;

}