**Same question as 865. Smallest Subtree with all the Deepest Nodes**

<https://leetcode.com/problems/lowest-common-ancestor-of-deepest-leaves/description/>

Given the root of a binary tree, return **the lowest common ancestor of its deepest leaves**.

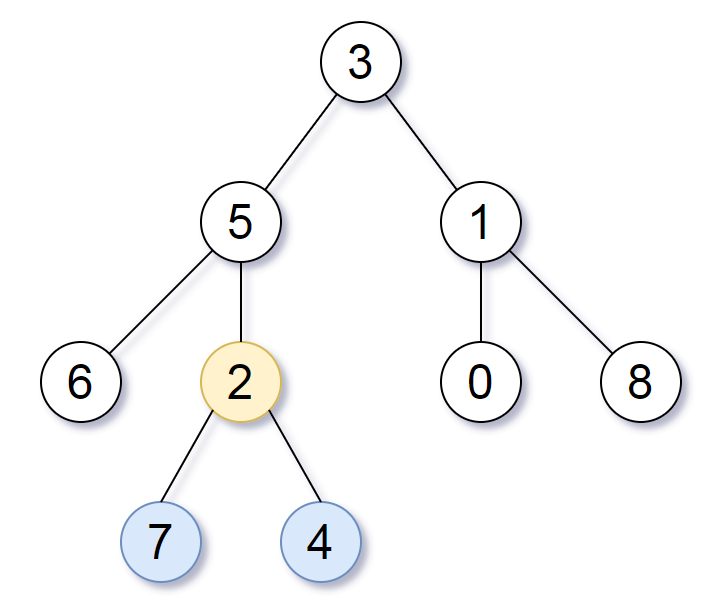
Recall that:

The node of a binary tree is a leaf if and only if it has no children

The depth of the root of the tree is 0. if the depth of a node is d, the depth of each of its children is d + 1.

The lowest common ancestor of a set S of nodes, is the node A with the largest depth such that every node in S is in the subtree with root A.

**Example 1:**



**Input:** root = [3,5,1,6,2,0,8,null,null,7,4]

**Output:** [2,7,4]

**Explanation:** We return the node with value 2, colored in yellow in the diagram.The nodes coloured in blue are the deepest leaf-nodes of the tree.Note that nodes 6, 0, and 8 are also leaf nodes, but the depth of them is 2, but the depth of nodes 7 and 4 is 3.

**Example 2:**

**Input:** root = [1]

**Output:** [1]

**Explanation:** The root is the deepest node in the tree, and it's the lca of itself.

**Example 3:**

**Input:** root = [0,1,3,null,2]

**Output:** [2]

**Explanation:** The deepest leaf node in the tree is 2, the lca of one node is itself.

**Attempt 1: 2024-8-13**

**Solution 1: Get Subtree Height and LCA (60min)**

**The style is "Process subtree nodes first, then process current node"**

**相对于参考答案，更合理的部分是用 height 来代替 depth，因为按照先处理 subtree nodes 后处理 current node 的逻辑，我们是从 leaf 所在的高度算起，而不是从 root 所在的深度算起**

/\*\*

 \* 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 TreeNode lcaDeepestLeaves(TreeNode root) {

        return helper(root).treeNode;

    }

    private Node helper(TreeNode root) {

        // Empty node height should be -1, leaf node height should be 0

        if(root == null) {

            return new Node(root, -1);

        }

        Node left = helper(root.left);

        Node right = helper(root.right);

        if(left.height > right.height) {

            // Important: The 'treeNode' used for holding the deapest

            // leaf node during travesal so far, not the current node

            return new Node(left.treeNode, left.height + 1);

        } else if(left.height < right.height) {

            return new Node(right.treeNode, right.height + 1);

        } else {

            return new Node(root, left.height + 1);

        }

    }

}

class Node {

    TreeNode treeNode;

    int height;

    public Node(TreeNode treeNode, int height) {

        this.treeNode = treeNode;

        this.height = height;

    }

}

**Refer to**

<https://leetcode.com/problems/lowest-common-ancestor-of-deepest-leaves/solutions/334577/java-c-python-two-recursive-solution/>

**Complexity**

Both solution are one pass.

Time O(N) for one pass

Space O(H) for recursion management

**Solution 1: Get Subtree Height and LCA**

helper function return the subtree height and lca and at the same time.

null node will return depth 0,

leaves will return depth 1.

class Solution {

public TreeNode lcaDeepestLeaves(TreeNode root) {

return dfs(root).lca;

}

private Result dfs(TreeNode node) {

if (node == null) {

return new Result(null, 0);

}

Result leftResult = dfs(node.left);

Result rightResult = dfs(node.right);

if (leftResult.depth > rightResult.depth) {

return new Result(leftResult.lca, leftResult.depth + 1);

} else if (rightResult.depth > leftResult.depth) {

return new Result(rightResult.lca, rightResult.depth + 1);

} else {

return new Result(node, leftResult.depth + 1);

}

}

private static class Result {

TreeNode lca;

int depth;

Result(TreeNode lca, int depth) {

this.lca = lca;

this.depth = depth;

}

}

}

**Solution 2: Get Subtree Deepest Depth (60min)**

**The style is "Process current node first, then process subtree nodes"**

**Style 1: Calculate max depth first then find the LCA match the max depth**

/\*\*

 \* 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 {

    private int maxDepth;

    private TreeNode lca;

    public TreeNode lcaDeepestLeaves(TreeNode root) {

        maxDepth = findMaxDepth(root);

        findLCA(root, 0);

        return lca;

    }

    // First, determine the maximum depth of the tree

    private int findMaxDepth(TreeNode node) {

        // The empty node depth is -1, the root node depth is 0

        if (node == null) {

            return -1;

        }

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

    }

    // Second, process the current node first, then the subtrees

    private int findLCA(TreeNode node, int depth) {

        // When the empty node reached, the actual 'depth' should

        // not include '+ 1' part added when enter current recursion

        // on findLCA(..., depth + 1) method, the 'depth' should

        // only mapping to empty node's parent non-empty node, so

        // we remove the '+ 1' by '- 1'

        if (node == null) {

            return depth - 1;

        }

        if (depth == maxDepth) {

            // If current node is at the maximum depth, it might be part of LCA

            lca = node;

            return maxDepth;

        }

        int leftDepth = findLCA(node.left, depth + 1);

        int rightDepth = findLCA(node.right, depth + 1);

        // If the deepest leaves are found in both left and right subtrees

        if (leftDepth == maxDepth && rightDepth == maxDepth) {

            lca = node;

        }

        // Return the maximum depth encountered

        return Math.max(leftDepth, rightDepth);

    }

}

**Refer to chatGPT**

**Q1: The original reference is below, but in our final solution, its a slightly modification on how to deal with empty node's depth**

class Solution {

private int maxDepth;

private TreeNode lca;

public TreeNode lcaDeepestLeaves(TreeNode root) {

maxDepth = findMaxDepth(root);

findLCA(root, 0);

return lca;

}

// First, determine the maximum depth of the tree

private int findMaxDepth(TreeNode node) {

if (node == null) {

return 0;

}

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

}

// Second, process the current node first, then the subtrees

private int findLCA(TreeNode node, int depth) {

if (node == null) {

return depth;

}

if (depth == maxDepth - 1) {

// If current node is at the maximum depth, it might be part of LCA

lca = node;

return depth + 1;

}

int leftDepth = findLCA(node.left, depth + 1);

int rightDepth = findLCA(node.right, depth + 1);

// If the deepest leaves are found in both left and right subtrees

if (leftDepth == maxDepth && rightDepth == maxDepth) {

lca = node;

}

// Return the maximum depth encountered

return Math.max(leftDepth, rightDepth);

}

}

This solution is a slightly modified version of the previous approach where you process the current node first and then the subtrees. The main difference lies in how the depth of nodes is handled, particularly with the treatment of null nodes. Let's break it down step by step.

**Key Modifications and Their Purpose**

Depth Calculation for **null Nodes**:

Original Approach: When encountering a null node, the base depth returned was 0.

Modified Approach: The depth of a null node is treated as -1, and when this depth is returned from a recursion, it's adjusted by subtracting 1. This is done to ensure that when processing the actual nodes, the depth calculations correctly reflect their position relative to their parent nodes.

Handling the Maximum Depth:

In this version, the maxDepth is calculated using the findMaxDepth method, where the depth of an empty node is treated as -1, and the depth of the root node is 0.

This adjustment ensures that when the depth comparison is made, it aligns correctly with how depths are defined for null nodes and actual nodes.

Depth Management in **findLCA Method**:

When traversing the tree in the findLCA method, the depth passed into recursive calls is adjusted so that when a null node is reached, the depth reflects the correct level of its parent node. This is done by subtracting 1 when a null node is encountered (return depth - 1).

This adjustment prevents the depth from being incorrectly incremented due to the presence of a null node, which doesn't actually contribute to the depth of the tree.

**Detailed Explanation**

findMaxDepth **Method**:

This method is used to determine the maximum depth of the tree, starting with -1 for null nodes, ensuring that the root node is at depth 0.

For a null node, the depth is -1, and for any other node, it's 1 + max(depth of left child, depth of right child).

findLCA **Method**:

Here, you start by checking if the current node is at the maxDepth. If it is, this node is a candidate for the LCA.

The method then recursively checks the left and right children, adjusting the depth by adding 1 before the recursive call.

If both the left and right subtrees have nodes at the maxDepth, then the current node is marked as the LCA.

The depth returned is adjusted to ensure that when comparing the left and right subtree depths, the correct depth is returned.

**Why This Modification?**

Handling Edge Cases:

By treating the depth of null nodes differently and adjusting the depth returned from recursive calls, the code is better suited to handle edge cases where a node might have only one child or no children.

More Accurate Depth Tracking:

This adjustment helps maintain accurate tracking of depths when determining the LCA, especially in cases where the binary tree might not be balanced.

**Final Thoughts**

This modified approach ensures that the depth calculations are more precise and that the process of determining the LCA of the deepest leaves is robust, even in trees with various structures. The key idea is to prevent over-counting depth when encountering null nodes, which could lead to incorrect LCA identification. By subtracting 1 for null nodes, the algorithm ensures that only actual nodes contribute to the depth calculation, leading to more accurate results.

**Q2: For above solution, if i remove below statement:**

if (leftDepth == maxDepth && rightDepth == maxDepth) {

lca = node;

}

**it will error out on input as:**

Input: root =[3,5,1,6,2,0,8,null,null,7,4]

Output: [4]

Expected output: [2], since [2] is LCA of [7] and [4] as both are deepest leaf nodes

If you remove the following statement from the solution:

if (leftDepth == maxDepth && rightDepth == maxDepth) {

lca = node;

}

**The solution would no longer correctly identify the lowest common ancestor (LCA) of the deepest leaves when there are multiple deepest leaves, as seen in the example input:**

Input: root = [3,5,1,6,2,0,8,null,null,7,4]

**Why the Statement is Crucial:**

Purpose of the Statement:

**The statement checks whether both the left and right subtrees of the current node contain nodes that are at the maximum depth. If both subtrees contain the deepest leaves, then the current node must be the LCA of those deepest leaves.**

**The condition ensures that when both the left and right subtrees have the deepest leaves, the node being processed is recognized as the lowest common ancestor.**

Impact of Removing the Statement:

**Without this check, the code will not correctly update the lca variable when both the left and right subtrees contain deepest leaves.**

**Instead, the lca will be incorrectly set to one of the deepest leaf nodes rather than their actual ancestor.**

**Example Breakdown:**

Given the input:

3

/ \

5 1

/ \ / \

6 2 0 8

/ \

7 4

Deepest Leaves: Nodes 7 and 4, both at depth 3.

Expected LCA: Node 2, because it is the lowest common ancestor of Nodes 7 and 4.

**What Happens Without the Check:**

The modified code (without the if statement) would set lca to Node 4 when processing Node 2, as Node 4 is the deepest leaf encountered at that point. However, it would fail to recognize that Node 2 is the common ancestor of both Nodes 7 and 4.

**Proper Execution With the Check:**

With the if statement, when the algorithm reaches Node 2, it correctly identifies that both its left and right subtrees contain nodes at the maximum depth (3). Hence, it correctly sets lca to Node 2.

**Summary:**

Correctness: The if statement ensures the correct identification of the LCA when there are multiple deepest leaves in different subtrees.

Error Without It: Removing it leads to incorrect results because the code would no longer correctly track the common ancestor of all deepest leaves, especially when those leaves are in different subtrees.

**Conclusion:**

To correctly solve the problem, particularly for cases where there are multiple deepest leaves, the check if (leftDepth == maxDepth && rightDepth == maxDepth) is essential. It ensures that the LCA is identified correctly when processing a node that is the ancestor of the deepest leaves found in both the left and right subtrees.

**Style 2: Calculate max depth first and LCA match the max depth at the same time**

class Solution {

    private TreeNode lca;

    int maxDepth = 0;

    public TreeNode lcaDeepestLeaves(TreeNode root) {

        findDepthAndLCA(root, 0);

        return lca;

    }

    // This method returns the depth and also updates the LCA

    private int findDepthAndLCA(TreeNode node, int depth) {

        // Move 'node == null' check logic before compare with 'maxDepth'

        // to make sure get correct depth and then assign to 'maxDepth'

        if(node == null) {

            // Subtract 1 to correct the depth for null nodes

            depth = depth - 1;

            return depth;

        }

        maxDepth = Math.max(maxDepth, depth);

        //if (node == null) {

        //    return depth;

        //}

        // Recursively get the depth of the left and right subtrees

        int leftDepth = findDepthAndLCA(node.left, depth + 1);

        int rightDepth = findDepthAndLCA(node.right, depth + 1);

        // If both subtrees have nodes at the maximum depth, update LCA

        // Note: even getting the final result on one branch (either left

        // or right), the only LCA node has both left and right subtree

        // node as null, the left and right subtree depth will be same

        // by running into base condition 'depth = depth - 1' which reflect

        // the current deepest leaf node depth, and both depth will be

        // same as 'maxDepth', below condition still work

        if (leftDepth == maxDepth && rightDepth == maxDepth) {

            lca = node;

        }

        // Return the maximum depth of the subtrees

        return Math.max(leftDepth, rightDepth);

    }

}

**Refer to**

<https://leetcode.com/problems/lowest-common-ancestor-of-deepest-leaves/solutions/334577/java-c-python-two-recursive-solution/>

**Solution 2: Get Subtree Deepest Depth**

helper function return the deepest depth of descendants.

deepest represent the deepest depth.

We use a global variable lca to represent the result.

One pass, Time O(N) Space O(H)

int deepest = 0;

TreeNode lca;

public TreeNode lcaDeepestLeaves(TreeNode root) {

helper(root, 0);

return lca;

}

private int helper(TreeNode node, int depth) {

deepest = Math.max(deepest, depth);

if (node == null) {

return depth;

}

int left = helper(node.left, depth + 1);

int right = helper(node.right, depth + 1);

if (left == deepest && right == deepest) {

lca = node;

}

return Math.max(left, right);

}

**Q3: Can we merge the findMaxDepth() method and findLCA() method into one method ? And i don't like the depth handling on Solution 2 above**

**Refer to chatGPT**

The modified solution introduces a correction that ensures the correct depth is calculated and then properly compared with maxDepth. Here’s how the logic works and why the changes lead to the correct solution:

**Key Changes and Their Impact:**

Moved **node == null Check Before maxDepth Comparison**:

The critical change is moving the check for null nodes (if(node == null)) before any depth comparison or update to maxDepth.

This ensures that when a null node is encountered, the current depth is accurately adjusted (by subtracting 1) and then returned. This depth is not included in the maxDepth calculation, which prevents incorrect depth values from affecting the LCA determination.

Correct **maxDepth Assignment**:

After ensuring the depth is correctly adjusted for null nodes, maxDepth is updated using the accurate depth information. This avoids issues where maxDepth could be incorrectly set to a lower value than it should be.

LCA Update Logic:

The logic for updating the LCA remains the same: the LCA is updated when both the left and right subtrees of a node contain nodes at the maxDepth.

**How the Logic Works:**

**Initial Setup:**

maxDepth: This variable keeps track of the deepest level encountered so far during the tree traversal.

lca: This variable is used to store the lowest common ancestor of the deepest leaves.

**Traversal and Depth Calculation:**

The method findDepthAndLCA is called recursively for each node in the tree.

Base Case:

If the node is null, the method returns the adjusted depth (depth - 1). This adjustment ensures that the depth calculations are correct and prevents null nodes from contributing to maxDepth.

Depth Calculation:

For non-null nodes, the maxDepth is updated by comparing it with the current depth, ensuring that maxDepth always holds the deepest level encountered so far.

**LCA Determination:**

Checking Subtree Depths:

After calculating the depths of the left and right subtrees, the method checks if both subtrees contain nodes at the maxDepth.

If they do, it means that the current node is the lowest common ancestor of those deepest nodes, and lca is updated to this node.

Returning Maximum Depth:

The method returns the maximum depth between the left and right subtrees, which helps propagate the correct depth information up the recursive stack.

**Example Breakdown:**

Let's consider the example tree:

3

/ \

5 1

/ \ / \

6 2 0 8

/ \

7 4

Initial Call: The function is called on the root (node 3) with a starting depth of 0.

Traversal: The function recursively traverses down to the deepest leaves (7 and 4), calculating and updating maxDepth as it goes.

LCA Update: When the function processes node 2, it sees that both its left and right subtrees contain nodes at maxDepth (depth 3), so it updates lca to node 2.

Final Result: After completing the traversal, the function returns lca, which correctly points to node 2.

**Conclusion:**

The corrected solution ensures that depth calculations are accurate by properly handling null nodes before updating maxDepth. This prevents erroneous depth values from influencing the LCA determination. The result is that the method correctly identifies the lowest common ancestor of the deepest leaves in the tree.

**Why a single deepest leaf node can be an LCA ?**

**For input below**

1

/ \

2 3

/

4

**What should be the result for Lowest Common Ancestor of Deepest Leaves ?**

**The answer is node 4, because node 4 is the deepest leaf node in the tree, and even node 1 sounds like a LCA node, but node 4 is the actual answer, the deepest leaf node itself can be the LCA node, if no same depth nodes from other branch. This idea will explain the doubt as when left.height and right.height NOT equal, why we can pick up the leaf node from one branch.**

**From chatGPT:**

**Deepest Leaves:**

The deepest leaf node in this tree is **Node 4**, which is at depth 2.

**Expected Result:**

The task is to find the **Lowest Common Ancestor (LCA)** of the deepest leaves.

Since **Node 4** is the only deepest leaf, the LCA of the deepest leaves is **Node 4** itself.

**Explanation:**

In this tree, Node 4 is the deepest node, and there are no other nodes at the same depth as Node 4. Therefore, the LCA of the deepest leaves is simply Node 4.

**Output:**

The result for this input should be **Node 4**.

So, the method should return **Node 4** as the LCA of the deepest leaves.

**Refer to**

[L236.Lowest Common Ancestor of a Binary Tree (Ref.L865,L235)](note://E191ABBC6A9B4A3C989AF0136CABCFA4)

[L104.Maximum Depth of Binary Tree (Ref.L222,L366)](note://DFEAE5072F224B7F85934A11F3381C70)

[L865.Smallest Subtree with all the Deepest Nodes (Ref.L236,L104)](note://4579E996593B44E79D1E73701DB5DCA1)

[What is the difference between tree depth and height](note://BDC7B166ABD743F29E3B4BC31D6FA0B1)