<https://leetcode.com/problems/smallest-subtree-with-all-the-deepest-nodes/description/>

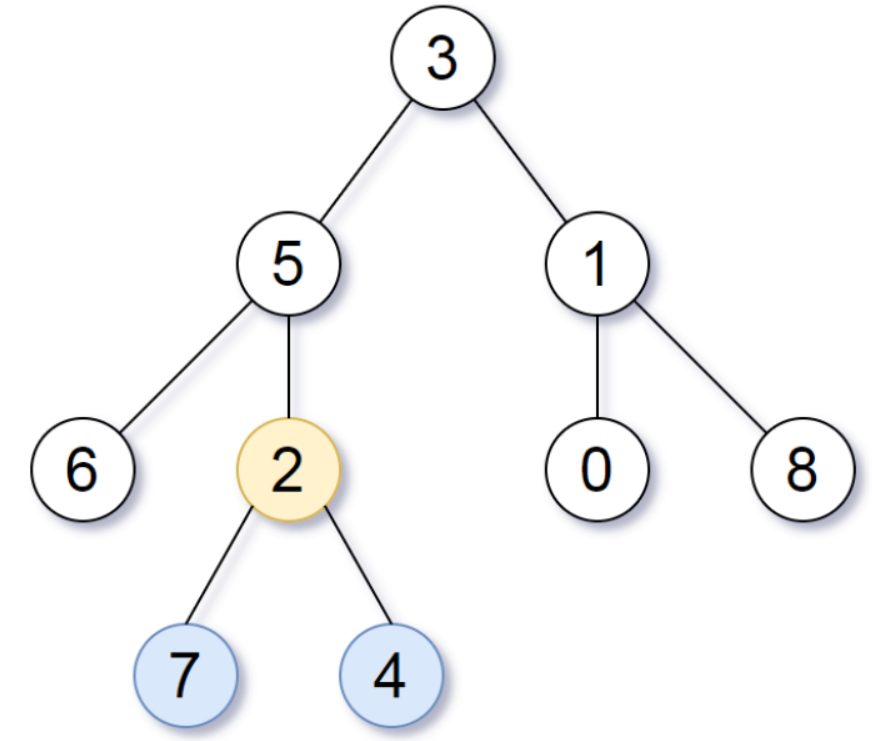
Given the root of a binary tree, the depth of each node is **the shortest distance to the root**.

Return *the smallest subtree* such that it contains **all the deepest nodes** in the original tree.

A node is called **the deepest** if it has the largest depth possible among any node in the entire tree.

The **subtree** of a node is a tree consisting of that node, plus the set of all descendants of that node.

**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 nodes of the tree.

Notice that nodes 5, 3 and 2 contain the deepest nodes in the tree but node 2 is the smallest subtree among them, so we return it.

**Example 2:**

Input: root = [1]

Output: [1]

Explanation: The root is the deepest node in the tree.

**Example 3:**

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

Output: [2]

Explanation: The deepest node in the tree is 2, the valid subtrees are the subtrees of nodes 2, 1 and 0 but the subtree of node 2 is the smallest.

**Constraints:**

* The number of nodes in the tree will be in the range [1, 500].
* 0 <= Node.val <= 500
* The values of the nodes in the tree are **unique**.

**Note:** This question is the same as 1123: <https://leetcode.com/problems/lowest-common-ancestor-of-deepest-leaves/>

**Attempt 1: 2022-12-30**

**Solution 1: Two pass recursion with standard find LCA and map recording {node, depth} help (30min)**

/\*\*

\* 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 {

Map<TreeNode, Integer> map;

int maxDepth;

public TreeNode subtreeWithAllDeepest(TreeNode root) {

map = new HashMap<TreeNode, Integer>();

// Root depth start with 0, its parent set as NULL and depth -1

map.put(null, -1);

// Mapping depth to node (start with root)

findDepth(root, null);

maxDepth = 0;

for(int depth : map.values()) {

if(depth > maxDepth) {

maxDepth = depth;

}

}

// Find LCA for node has maximum depth

return findLCA(root);

}

// Classic recursive traversal (Top Down DFS) void return

private void findDepth(TreeNode root, TreeNode parent) {

// Base

if(root == null) {

return;

}

// Process on current level

map.put(root, map.get(parent) + 1);

// Separate into smaller issue

findDepth(root.left, root);

findDepth(root.right, root);

}

// Divide and Conquer (Bottom Up DFS) return type TreeNode

private TreeNode findLCA(TreeNode root) {

if(root == null) {

return null;

}

// If the node in question has maximum depth (leaves), it is the answer.

if(map.get(root) == maxDepth) {

return root;

}

// Standard way to find LCA

TreeNode left = findLCA(root.left);

TreeNode right = findLCA(root.right);

// If both the left and right child of a node have a deepest

// descendant, then the answer is this parent node.

if(left != null && right != null) {

return root;

}

// Otherwise, if some child has a deepest descendant,

// then the answer is that child.

if(left != null) {

return left;

} else {

return right;

}

}

}

Time Complexity: O(n)

Space Complexity: O(n)

**Refer to**

<https://leetcode.com/problems/smallest-subtree-with-all-the-deepest-nodes/solutions/146663/smallest-subtree-with-all-the-deepest-nodes>

#### **Approach 1: Paint Deepest Nodes**

**Intuition**

We try a straightforward approach that has two phases.

The first phase is to identify the nodes of the tree that are deepest. To do this, we have to annotate the depth of each node. We can do this with a depth first search.

Afterwards, we will use that annotation to help us find the answer:

* If the node in question has maximum depth, it is the answer.
* If both the left and right child of a node have a deepest descendant, then the answer is this parent node.
* Otherwise, if some child has a deepest descendant, then the answer is that child.
* Otherwise, the answer for this subtree doesn't exist.

**Algorithm**

In the first phase, we use a depth first search dfs to annotate our nodes.

In the second phase, we also use a depth first search answer(node), returning the answer for the subtree at that node, and using the rules above to build our answer from the answers of the children of node.

Note that in this approach, the answer function returns answers that have the deepest nodes of the *entire* tree, not just the subtree being considered.

class Solution {

Map<TreeNode, Integer> depth;

int max\_depth;

public TreeNode subtreeWithAllDeepest(TreeNode root) {

depth = new HashMap();

depth.put(null, -1);

dfs(root, null);

max\_depth = -1;

for (Integer d: depth.values())

max\_depth = Math.max(max\_depth, d);

return answer(root);

}

public void dfs(TreeNode node, TreeNode parent) {

if (node != null) {

depth.put(node, depth.get(parent) + 1);

dfs(node.left, node);

dfs(node.right, node);

}

}

public TreeNode answer(TreeNode node) {

if (node == null || depth.get(node) == max\_depth)

return node;

TreeNode L = answer(node.left),

R = answer(node.right);

if (L != null && R != null) return node;

if (L != null) return L;

if (R != null) return R;

return null;

}

}

**Complexity Analysis**

* Time Complexity: O(N), where N is the number of nodes in the tree.
* Space Complexity: O(N)

**Solution 2: Divide and Conquer one pass recursion (30min)**

**Style 1: With helper class to return both node and depth at the same time**

/\*\*

\* 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 subtreeWithAllDeepest(TreeNode root) {

return helper(root).node;

}

private Node helper(TreeNode root) {

// Base

if(root == null) {

return new Node(null, 0);

}

// Divide

Node left = helper(root.left);

Node right = helper(root.right);

// Process & Conquer

if(left.depth == right.depth) {

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

} else if(left.depth > right.depth) {

return new Node(left.node, left.depth + 1);

} else {

return new Node(right.node, right.depth + 1);

}

}

}

class Node {

TreeNode node;

int depth;

public Node(TreeNode node, int depth) {

this.node = node;

this.depth = depth;

}

}

Time Complexity: O(n)

Space Complexity: O(n)

**Refer to**

<https://leetcode.com/problems/smallest-subtree-with-all-the-deepest-nodes/solutions/146808/c-java-python-one-pass>

Write a sub function deep(TreeNode root).Return a pair(int depth, TreeNode subtreeWithAllDeepest)

In sub function deep(TreeNode root):

**if root == null**, return pair(0, null)

**if left depth == right depth**, deepest nodes both in the left and right subtree, return pair (left.depth + 1, root)

**if left depth > right depth**, deepest nodes only in the left subtree, return pair (left.depth + 1, left subtree)

**if left depth < right depth**, deepest nodes only in the right subtree, return pair (right.depth + 1, right subtree)

public TreeNode subtreeWithAllDeepest(TreeNode root) {

return deep(root).getValue();

}

public Pair<Integer, TreeNode> deep(TreeNode root) {

if (root == null) return new Pair(0, null);

Pair<Integer, TreeNode> l = deep(root.left), r = deep(root.right);

int d1 = l.getKey(), d2 = r.getKey();

return new Pair(Math.max(d1, d2) + 1, d1 == d2 ? root : d1 > d2 ? l.getValue() : r.getValue());

}

**Refer to**

<https://leetcode.com/problems/smallest-subtree-with-all-the-deepest-nodes/solutions/146663/smallest-subtree-with-all-the-deepest-nodes>

#### **Approach 2: Recursion**

**Intuition**

We can combine both depth first searches in *Approach #1* into an approach that does both steps in one pass. We will have some function dfs(node) that returns both the answer for this subtree, and the distance from node to the deepest nodes in this subtree.

**Algorithm**

* The Result (on some subtree) returned by our (depth-first search) recursion will have two parts:

1. Result.node: the largest depth node that is equal to or an ancestor of all the deepest nodes of this subtree.
2. Result.dist: the number of nodes in the path from the root of this subtree, to the deepest node in this subtree.

* We can calculate these answers disjointly for dfs(node):

To calculate the Result.node of our answer:

1. If one childResult has deeper nodes, then childResult.node will be the answer.
2. If they both have the same depth nodes, then node will be the answer.

* The Result.dist of our answer is always 1 more than the largest childResult.dist we have.

class Solution {

public TreeNode subtreeWithAllDeepest(TreeNode root) {

return dfs(root).node;

}

// Return the result of the subtree at this node.

public Result dfs(TreeNode node) {

if (node == null) return new Result(null, 0);

Result L = dfs(node.left),

R = dfs(node.right);

if (L.dist > R.dist) return new Result(L.node, L.dist + 1);

if (L.dist < R.dist) return new Result(R.node, R.dist + 1);

return new Result(node, L.dist + 1);

}

}

/\*\*

\* The result of a subtree is:

\* Result.node: the largest depth node that is equal to or

\* an ancestor of all the deepest nodes of this subtree.

\* Result.dist: the number of nodes in the path from the root

\* of this subtree, to the deepest node in this subtree.

\*/

class Result {

TreeNode node;

int dist;

Result(TreeNode n, int d) {

node = n;

dist = d;

}

}

**Complexity Analysis**

* Time Complexity: O(N), where N is the number of nodes in the tree.
* Space Complexity: O(N)

**Wrong Solution for Style 2 which try to more intuitive with helper class only return depth and use global variable to return node, since not able to equal 'maxDepth' with 'leftDepth' or 'rightDepth',  let alone return TreeNode**

/\*\*

\* 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 {

int maxDepth = 0;

TreeNode result = null;

public TreeNode subtreeWithAllDeepest(TreeNode root) {

helper(root);

return result;

}

private int helper(TreeNode root) {

if(root == null) {

return 0; // not return depth

}

// Divide

int leftDepth = helper(root.left);

int rightDepth = helper(root.right);

// Process

int curDepth = Math.max(leftDepth, rightDepth) + 1;

maxDepth = Math.max(maxDepth, curDepth);

// Wrong statement since never able to reach because leftDepth or rightDepth

// never able to equal maxDepth since curDepth always + 1 on leftDepth or

// rightDepth, and maxDepth will update to curDepth, so maxDepth always larger

// than leftDepth or rightDepth here

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

result = root;

}

// Conquer

return curDepth;

}

}

**Style 2: More intuitive with helper class only return depth and use global variable to return node, bottom level return 'depth' since '+1' operation not in DFS three steps (divide -> process -> conquer) but only happen on parameter that passed in recursion function, since no actual operation to update 'depth' during DFS, to reflect change happen on 'depth' in the parameter, requires return 'depth' to pass in next recursion (Refer to L104.Maximum Depth of Binary Tree)**

/\*\*

\* 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 {

int maxDepth = 0;

TreeNode result = null;

public TreeNode subtreeWithAllDeepest(TreeNode root) {

helper(root, 0);

return result;

}

// Additional parameter 'depth'

private int helper(TreeNode root, int depth) {

if(root == null) {

return depth; // not return 0

}

// Divide

int leftDepth = helper(root.left, depth + 1); // + 1 on parameter that passed in recursion function, not in divide -> process -> conquer steps

int rightDepth = helper(root.right, depth + 1); // + 1 on parameter that passed in recursion function, not in divide -> process -> conquer steps

// Process (node under processing is on bottom level resepect

// to current recursion, that's why so called Bottom Up)

int curDepth = Math.max(leftDepth, rightDepth);

maxDepth = Math.max(maxDepth, curDepth);

// If left depth and right depth both equal to deepest depth,

// then deepest nodes in the left and right subtree, the answer

// node is the current root for both left and right subtree

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

result = root;

}

return curDepth;

}

}

**Refer to**

<https://leetcode.com/problems/smallest-subtree-with-all-the-deepest-nodes/solutions/146868/simple-java-dfs-recursion-function-with-explanation>

First Root to Leaf: return the deep level of every node

Then Leaf to Root: only when the its left node and right node both have the deepest level, update the result node

class Solution {

int deepestLevel = 0;

TreeNode res = null;

public TreeNode subtreeWithAllDeepest(TreeNode root) {

dfs(root, 0);

return res;

}

private int dfs(TreeNode root, int level) {

if (root == null) return level;

int leftLevel = dfs(root.left, level + 1);

int rightLevel = dfs(root.right, level + 1);

int curLevel = Math.max(leftLevel, rightLevel);

deepestLevel = Math.max(deepestLevel, curLevel);

if (leftLevel == deepestLevel && rightLevel == deepestLevel)

res = root;

return curLevel;

}

}