<https://leetcode.ca/all/1650.html>

Given two nodes of a binary tree p and q, return *their lowest common ancestor (LCA)*.

**Each node will have a reference to its parent node.** The definition for Node is below:

class Node {

    public int val;

    public Node left;

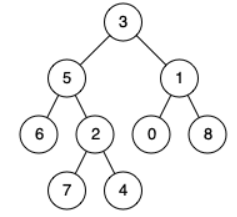
    public Node right;

    public Node parent;

}

According to the [definition of LCA on Wikipedia](https://en.wikipedia.org/wiki/Lowest_common_ancestor): "The lowest common ancestor of two nodes p and q in a tree T is the lowest node that has both p and q as descendants (where we allow **a node to be a descendant of itself**)."

Example 1:

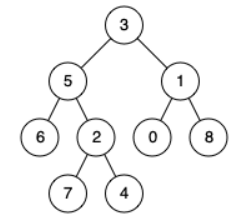


Input: root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 1

Output: 3

Explanation: The LCA of nodes 5 and 1 is 3.

Example 2:



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

Output: 5

Explanation: The LCA of nodes 5 and 4 is 5 since a node can be a descendant of itself according to the LCA definition.

Example 3:

Input: root = [1,2], p = 1, q = 2

Output: 1

Constraints:

The number of nodes in the tree is in the range [2, 10^5].

-10^9 <= Node.val <= 10^9

All Node.val are unique.

p != q

p and q exist in the tree.

**Note: The difference between L1650.Lowest Common Ancestor of a Binary Tree II and L236.Lowest Common Ancestor of a Binary Tree is L1650 NOT provide 'root' and provide 'parent' reference.**

**Attempt 1: 2022-12-28**

**Classic solution same as L236.Lowest Common Ancestor of a Binary Tree (TLE, have to find 'root' first)**

public class TreeSolution {

    private class TreeNode {

        public int val;

        public TreeNode left, right, parent;

        public TreeNode(int val) {

            this.val = val;

            this.left = this.right = this.parent = null;

        }

    }

    public static void main(String[] args) {

        /\*\*

        \*            1

        \*          / \

        \*          2  5

        \*        / \  \

        \*        3  4    6

        \*/

        TreeSolution s = new TreeSolution();

        TreeNode one = s.new TreeNode(1);

        TreeNode two = s.new TreeNode(2);

        TreeNode three = s.new TreeNode(3);

        TreeNode four = s.new TreeNode(4);

        TreeNode five = s.new TreeNode(5);

        TreeNode six = s.new TreeNode(6);

        //TreeNode seven = s.new TreeNode(7);

        one.left = two;

        one.right = five;

        two.left = three;

        two.right = four;

        two.parent = one;

        five.right = six;

        five.parent = one;

        three.parent = two;

        four.parent = two;

        six.parent = five;

        TreeNode lca = s.lowestCommonAncestor(three, five);

        System.out.println(lca.val);

    }

    private TreeNode lowestCommonAncestor(TreeNode p, TreeNode q) {

        TreeNode root = findRoot(p);

        return helper(root, p, q);

    }

    private TreeNode helper(TreeNode root, TreeNode p, TreeNode q) {

        if(root == null || root == p || root == q) {

            return root;

        }

        TreeNode left = helper(root.left, p, q);

        TreeNode right = helper(root.right, p, q);

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

            return root;

        }

        if(left != null) {

            return left;

        } else {

            return right;

        }

    }

    private TreeNode findRoot(TreeNode node) {

        // No need to reassign to a new TreeNode 'cur' since java pass by value

        // not by reference, the 'node' passed into function as 'a value refer

        // to the actual object physical address', during the recursion call,

        // in this example, no change on actual object the 'node' point to, the

        // change is only happening on 'node' itself as which object it point to,

        // the changed 'node' finally point to TreeNode 'root' physical address,

        // and this new value is what we want to return

//        TreeNode cur = node;

//        while(cur.parent != null) {

//            cur = cur.parent;

//        }

//        return cur;

        while(node.parent != null) {

            node = node.parent;

        }

        return node;

    }

}

Complexity Analysis

Time Complexity: O(N). Where N is the number of nodes in the binary tree.

In the worst case we might be visiting all the nodes of the binary tree.

Space Complexity: O(N). This is because the maximum amount of space utilized

by the recursion stack would be N since the height of a skewed binary tree could be N.

**Solution 1: Using parent node and calculate p, q depth first, then adjust to same depth to move back to same parent with same steps(60 min)**

class TreeSolution {

    private class TreeNode {

        public int val;

        public TreeNode left, right, parent;

        public TreeNode(int val) {

            this.val = val;

            this.left = this.right = this.parent = null;

        }

    }

    public static void main(String[] args) {

        /\*\*

        \*            1

        \*          / \

        \*          2  5

        \*        / \  \

        \*        3  4    6

        \*/

        TreeSolution s = new TreeSolution();

        TreeNode one = s.new TreeNode(1);

        TreeNode two = s.new TreeNode(2);

        TreeNode three = s.new TreeNode(3);

        TreeNode four = s.new TreeNode(4);

        TreeNode five = s.new TreeNode(5);

        TreeNode six = s.new TreeNode(6);

        //TreeNode seven = s.new TreeNode(7);

        one.left = two;

        one.right = five;

        two.left = three;

        two.right = four;

        two.parent = one;

        five.right = six;

        five.parent = one;

        three.parent = two;

        four.parent = two;

        six.parent = five;

        TreeNode lca = s.lowestCommonAncestor(three, five);

        System.out.println(lca.val);

    }

    private TreeNode lowestCommonAncestor(TreeNode p, TreeNode q) {

        int pDepth = findDepth(p);

        int qDepth = findDepth(q);

        // Update to the same depth

        while(pDepth > qDepth) {

            p = p.parent;

            pDepth--;

        }

        while(pDepth < qDepth) {

            q = q.parent;

            qDepth--;

        }

        // Same depth travel back together to the LCA

        while(p != q) {

            p = p.parent;

            q = q.parent;

        }

        return p;

    }

    private int findDepth(TreeNode node) {

        int depth = 0;

        while(node != null) {

            node = node.parent;

            depth++;

        }

        return depth;

    }

}

Complexity Analysis

Time Complexity: O(N). Where N is the number of nodes in the binary tree.

In the worst case we might be visiting all the nodes of the binary tree.

Space Complexity: O(N). This is because the maximum amount of space utilized

by the recursion stack would be N since the height of a skewed binary tree could be N.

**Refer to**

<https://blog.csdn.net/sinat_30403031/article/details/117254979>

**传统办法，第30（29/31 pass）个TC会TLE。先写出来吧，找LCA的算法，亚麻的OA考过。不同的是亚麻OA给了root**

class Solution:

    def lowestCommonAncestor(self, p: 'Node', q: 'Node') -> 'Node':

        root = self.findroot(p)

        print(root.val)

        res = self.helper(root, p, q)

        return res

    def findroot(self, node):

        while node.parent != None:

            node = node.parent

            self.findroot(node)

        return node

    def helper(self, root, p, q):

        if not root:return

        if root == p or root == q:return root

        left = self.helper(root.left, p, q)

        right = self.helper(root.right, p, q)

        if left and right:return root

        if left:return left

        if right:return right

**接下来，好好使用parent**

class Solution:

    def lowestCommonAncestor(self, p: 'Node', q: 'Node') -> 'Node':

        self.resp = []

        self.findroot(p, self.resp)

        while q:

            if q not in self.resp:

                q = q.parent

            else:

                return q

    def findroot(self, node, res):

        while node:

            res.append(node)

            node = node.parent

        return res

**Refer to**

<https://www.cnblogs.com/cnoodle/p/16456888.html>

这道题跟前两个版本的区别是多了一个 parent 节点。这样我们就可以从当前节点反过来往回找父节点是谁。既然还是找两个节点的最小公共父节点，那么我们就从两个节点分别开始找他们各自的父节点。这里我首先去看一下两个节点的深度分别是多少，并把他们的深度先调整成一样。当深度一样的时候，方便两个节点同时往他们各自的父节点走，这样他们可以同时到达他们共同的父节点。

/\*

// Definition for a Node.

class Node {

    public int val;

    public Node left;

    public Node right;

    public Node parent;

};

\*/

class Solution {

    public Node lowestCommonAncestor(Node p, Node q) {

        int pDepth = getDepth(p);

        int qDepth = getDepth(q);

        while (pDepth > qDepth) {

            pDepth--;

            p = p.parent;

        }

        while (pDepth < qDepth) {

            qDepth--;

            q = q.parent;

        }

        while (p != q) {

            p = p.parent;

            q = q.parent;

        }

        return p;

    }

    private int getDepth(Node node) {

        int depth = 0;

        while (node != null) {

            node = node.parent;

            depth++;

        }

        return depth;

    }

}

**Refer to**

[L235.Lowest Common Ancestor of a Binary Search Tree](note://3CB8ADE907F54FBE863D453CCE335C9E)

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

[L1644.Lowest Common Ancestor of a Binary Tree II](note://D87D2931D95B40A3BA98CA6C9386BC56)

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