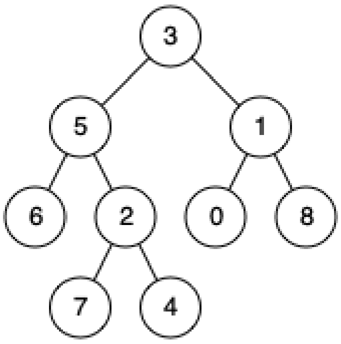
https://leetcode.com/problems/lowest-common-ancestor-of-a-binary-tree/

Given a binary tree, find the lowest common ancestor (LCA) of two given nodes in the tree.
According to the definition of LCA on Wikipedia: “The lowest common ancestor is defined between two nodes p and q as the lowest node in
T that has both p and q as descendants (where we allow **a node to be a descendant of itself**).”

Example 1:

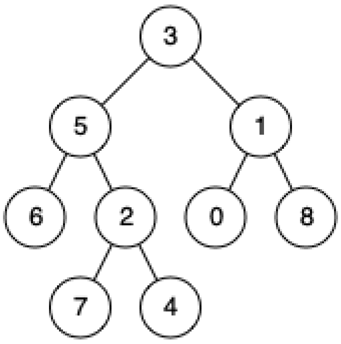


Input: root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 1

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: 5t

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 will exist in the tree.

Attempt 1: 2022-12-03

Solution 1: Divide and Conquer (30 min)

```java
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
    public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
        if(root == null || root == p || root == q) {
            return root;
        }
        TreeNode left = lowestCommonAncestor(root.left, p, q);
        TreeNode right = lowestCommonAncestor(root.right, p, q);
        if(left != null && right != null) {
            return root;
        }
        if(left != null) {
            return left;
        } else {
            return right;
        }
    }
}
```

```

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://segmentfault.com/a/1190000003509399

深度优先标记

复杂度

时间 O(h) 空间 O(h) 递归栈空间

思路

我们可以用深度优先搜索，从叶子节点向上，标记子树中出现目标节点的情况。如果子树中有目标节点，标记为那个目标节点，如果没有，标记为null。显然，如果左子树、右子树都有标记，说明就已经找到最小公共祖先了。如果在根节点为p的左右子树中找p、q的公共祖先，则必定是p本身。
换个角度，可以这么想：如果一个节点左子树有两个目标节点中的一个，右子树没有，那这个节点肯定不是最小公共祖先。如果一个节点右子树有两个目标节点中的一个，左子树没有，那这个节点肯定也不是最小公共祖先。只有一个节点正好左子树有，右子树也有的时候，才是最小公共祖先。

代码

```
public class Solution {
    public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
        //发现目标节点则通过返回值标记该子树发现了某个目标结点
        if(root == null || root == p || root == q) return root;
        //查看左子树中是否有目标结点，没有为null
        TreeNode left = lowestCommonAncestor(root.left, p, q);
        //查看右子树是否有目标节点，没有为null
        TreeNode right = lowestCommonAncestor(root.right, p, q);
        //都不为空，说明左右子树都有目标结点，则公共祖先就是本身
        if(left!=null&&right!=null) return root;
        //如果发现了目标节点，则继续向上标记为该目标节点
        return left == null ? right : left;
    }
}
```

Refer to
https://leetcode.com/problems/lowest-common-ancestor-of-a-binary-tree/discuss/1405170/4-STEPS-SOLUTION-or-Easy-Heavily-EXPLAINED-with-COMPLEXITIES

EXPLANATION

- We'll do just normal tree traversal of the given binary tree recursivly.
- For finding LCA (lowest common ancestor) we've following conditions for every node in the tree,
- But before that, this solutions works under the assumption that both Node 'p' & Node 'q' will present in the tree...
- if single one of the node is present in the tree, it'll not work or simply return null.

CONDITIONS: -

1. if current node is same as 'p' OR 'q'.
2. if one of it's subtrees contains 'p' and other 'q' (subtrees means, left_sub_tree and right_sub_tree).
3. if one of it's subtree contains both 'p' & 'q'.
4. if none of it's subtrees contains any of 'p' & 'q'.

- Note: that's a tricky implementation, but works well under the assumption that 'p' & 'q' will be definitely present.

EFFICIENT SOLUTION

- Runtime: 15ms [C++]

```
TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
 if(root == NULL) return NULL;
 if(root->val == p->val || root->val == q->val) return root;       // 👉 FIRST CONDITION...

        TreeNode* lca1 = lowestCommonAncestor(root->left, p, q);          // traverse on the left part of the tree
        TreeNode* lca2 = lowestCommonAncestor(root->right, p, q);         // traverse on the right part of the tree

 if(lca1 != NULL && lca2 != NULL) return root;                     // 👉 SECOND CONDITION... (IF BOTH SUB-TREE CONTAINS 'p' & 'q' RESPECTIVELY)
 if(lca1 != NULL) return lca1;                                     // 👉 THIRD CONDITION...
 return lca2;                                        // 👉 FOURTH CONDITION...
}
```

TIME COMPLEXITY :

O(N),Where N : total number of nodes in the BT

SPACE COMPLEXITY :

O(H) or O(N) (Worse Case), Where H : total height of tree for recursion stack

---

Solution 2: Promote Divide and Conquer with flag when both p and q in same left subtree to skip redundant scanning in right subtree (30 min)

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
    boolean found = false;
    public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
        if(found) {
            return null;
        }
        if(root == null || root == p || root == q) {
            return root;
        }
        TreeNode left = lowestCommonAncestor(root.left, p, q);
        TreeNode right = lowestCommonAncestor(root.right, p, q);
        if(left != null && right != null) {
            found = true;
            return root;
        }
        if(left != null) {
            return left;
        } else {
            return right;
        }
    }
}

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://leetcode.com/problems/lowest-common-ancestor-of-a-binary-tree/discuss/65226/My-Java-Solution-which-is-easy-to-understand/112901

This is a good solution but un-necessarily does the extra work of checking the whole tree if we have already found the ancestor in the left subtree.

https://leetcode.com/problems/lowest-common-ancestor-of-a-binary-tree/discuss/65226/My-Java-Solution-which-is-easy-to-understand/184794

You can add some flags when you've already found both p q under a same subtree, if you want to.

https://leetcode.com/problems/lowest-common-ancestor-of-a-binary-tree/discuss/65226/My-Java-Solution-which-is-easy-to-understand/195686

```
boolean found = false;
public TreeNode helper(TreeNode root, TreeNode p, TreeNode q)
{
    if(found||root==null) return null;
    TreeNode left = helper(root.left, p, q);
    TreeNode right = helper(root.right, p, q);

    if(left!=null&&right!=null)
    {
        found = true;
        return root;
    }
    if(root.val==p.val||root.val==q.val)
        return root;
    else if(left!=null)
        return left;
    else if(right!=null)
        return right;

    return null;
}
```

Test Case:

```
/**
* e.g
*           3
*         /   \
*        9     20
*       / \   /  \
*      8  10 15   7
*
* Test with 8 and 10 both under left subtree, after adding flag it will skip scanning right subtree
*/


class Solution {
    public static void main(String[] args) {
       Test b = new Test();
       TreeNode three = b.new TreeNode(3);
       TreeNode nine = b.new TreeNode(9);
       TreeNode tweeten = b.new TreeNode(20);
       TreeNode fifteen = b.new TreeNode(15);
       TreeNode seven = b.new TreeNode(7);
       TreeNode eight = b.new TreeNode(8);
       TreeNode ten = b.new TreeNode(10);

       three.left = nine;
       three.right = tweeten;
       tweeten.left = fifteen;
       tweeten.right = seven;
       nine.left = eight;
       nine.right = ten;
       TreeNode result = b.lowestCommonAncestor(three, eight, ten);
       System.out.println(result);
    }

    boolean found = false;
    public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
        if(found) {
            return null;
        }
        if(root == null || root == p || root == q) {
            return root;
        }
        TreeNode left = lowestCommonAncestor(root.left, p, q);
        TreeNode right = lowestCommonAncestor(root.right, p, q);
        if(left != null && right != null) {
            found = true;
            return root;
        }
        if(left != null) {
            return left;
        } else {
            return right;
        }
    }
}
```

Solution 3: BFS iterative traversal (30 min)

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
    public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
        // {child -> parent}
        Map<TreeNode, TreeNode> map = new HashMap<TreeNode, TreeNode>();
        Queue<TreeNode> queue = new LinkedList<TreeNode>();
        map.put(root, null);
        queue.offer(root);
        while(!map.containsKey(p) || !map.containsKey(q)) {
            TreeNode node = queue.poll();
            if(node.left != null) {
                map.put(node.left, node);
                queue.offer(node.left);
            }
            if(node.right != null) {
                map.put(node.right, node);
                queue.offer(node.right);
            }
        }
        Set<TreeNode> p_parents = new HashSet<TreeNode>();
        while(p != null) {
            p_parents.add(p);
            p = map.get(p);
        }
        while(!p_parents.contains(q)) {
            q = map.get(q);
        }
        return q;
    }
}

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). In the worst case space utilized by the stack(queue), the parent pointer dictionary and the ancestor set, would be N each, since the height of a skewed binary tree could be N.
```

Refer to

https://leetcode.com/problems/lowest-common-ancestor-of-a-binary-tree/discuss/65236/JavaPython-iterative-solution

To find the lowest common ancestor, we need to find where is p and q and a way to track their ancestors. A parent pointer for each node found is good for the job. After we found both p and q, we create a set of p's ancestors. Then we travel through q's ancestors, the first one appears in p's is our answer.

Iterative Algorithm

1.traverse tree iteratively with stack (queue) to look for p and q

2.use HashMap<TreeNode, TreeNode> parent to record <child, parent> relation.

3.once both p and q found (child, parent relation for both p and q found)

4.add p's all ancestor to a Set

5.traverse q's ancestors in order, and first shared ancestor is the shared LCA

```java
public class Solution {
    public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
        Map<TreeNode, TreeNode> parent = new HashMap<>();
        Deque<TreeNode> stack = new ArrayDeque<>();
        parent.put(root, null);
        stack.push(root);
        while (!parent.containsKey(p) || !parent.containsKey(q)) {
            TreeNode node = stack.pop();
            if (node.left != null) {
                parent.put(node.left, node);
                stack.push(node.left);
            }
            if (node.right != null) {
                parent.put(node.right, node);
                stack.push(node.right);
            }
        }
        Set<TreeNode> ancestors = new HashSet<>();
        while (p != null) {
            ancestors.add(p);
            p = parent.get(p);
        }
        while (!ancestors.contains(q))
            q = parent.get(q);
        return q;
    }
}
```

Instead of Stack, BFS more prefer Queue to traversal

Refer to

https://leetcode.com/problems/lowest-common-ancestor-of-a-binary-tree/discuss/65236/JavaPython-iterative-solution/66954

```
TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
    unordered_map<TreeNode*, TreeNode*> parents;
    parents[root] = nullptr;
    queue<TreeNode*> qu;
    qu.push(root);
    while (!parents.count(p) || !parents.count(q)) {
        int qsize = (int)qu.size();
        for (int i = 0; i < qsize; ++i) {
            auto node = qu.front();
            qu.pop();
            if (node -> left) {
                parents[node -> left] = node;
                qu.push(node -> left);
            }
            if (node -> right) {
                parents[node -> right] = node;
                qu.push(node -> right);
            }
        }
    }
    unordered_set<TreeNode*> ancestors;
    while (p) ancestors.insert(p), p = parents[p];
    while (q && !ancestors.count(q)) q = parents[q];
    return q;
}
```

Refer to

L235.Lowest Common Ancestor of a Binary Search Tree

L865.Smallest Subtree with all the Deepest Nodes (Ref.L236)