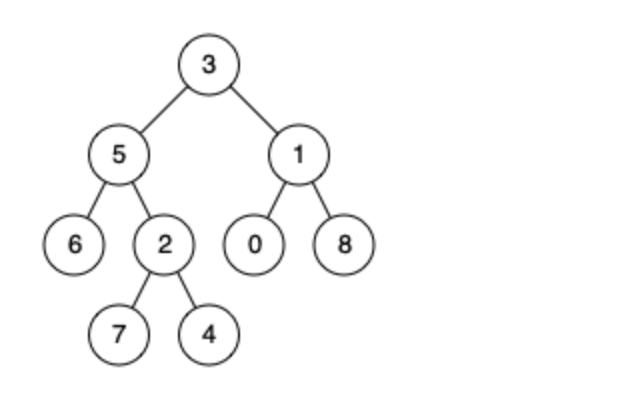
# 题目

给定一棵二叉树的根节点 root 和 TreeNode 类对象的数组（列表） nodes，返回 nodes 中所有节点的最近公共祖先（LCA）。数组（列表）中所有节点都存在于该二叉树中，且二叉树中所有节点的值都是互不相同的。

我们扩展二叉树的最近公共祖先节点在维基百科上的定义：“对于任意合理的 i 值， n 个节点 p1 、 p2、...、 pn 在二叉树 T 中的最近公共祖先节点是后代中包含所有节点 pi 的最深节点（我们允许一个节点是其自身的后代）”。一个节点 x 的后代节点是节点 x 到某一叶节点间的路径中的节点 y。

示例 1:



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

输出: 2

解释: 节点 4 和 7 的最近公共祖先是 2。

示例 2:

形状, 圆圈

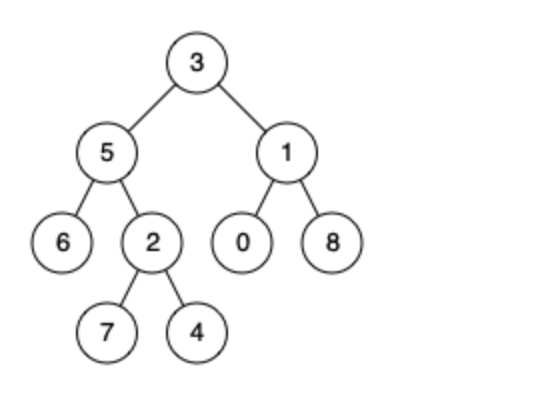
描述已自动生成

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

输出: 1

解释: 单个节点的最近公共祖先是该节点本身。

示例 3:

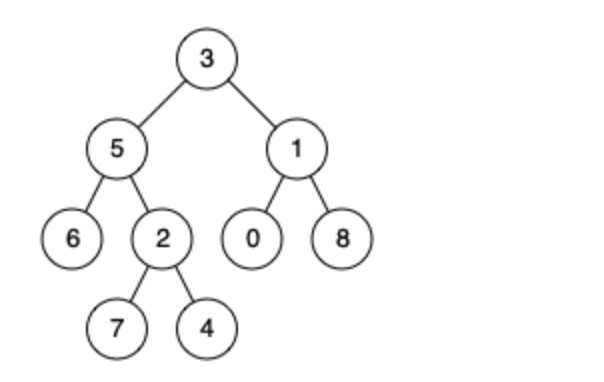


输入: root = [3,5,1,6,2,0,8,null,null,7,4], nodes = [7,6,2,4]

输出: 5

解释: 节点 7、6、2 和 4 的最近公共祖先节点是 5。

示例 4:



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

输出: 3

解释: 树中所有节点的最近公共祖先是根节点。

提示:

树中节点个数的范围是 [1, 104] 。

-109 <= Node.val <= 109

所有的 Node.val 都是互不相同的。

所有的 nodes[i] 都存在于该树中。

所有的 nodes[i] 都是互不相同的。

# 分析

要解决“寻找二叉树中多个节点的最近公共祖先（LCA）”问题，核心思路是扩展两节点LCA的逻辑：通过后序遍历检查每个节点的子树是否包含目标节点集合，利用“子树包含目标节点的数量”或“是否包含所有目标节点”的判断，定位最深的、包含所有目标节点的节点，即为多节点的LCA。

思路分析

1、核心定义扩展：多节点的LCA是“包含所有目标节点作为后代的最深节点”（允许节点自身是目标节点）。

2、后序遍历的优势：后序遍历（左→右→根）能先检查子树再处理当前节点，便于统计子树中包含的目标节点数量，进而判断当前节点是否为LCA。

3、目标节点集合优化：将目标节点列表转换为哈希集合（如unordered\_set），使“判断节点是否为目标节点”的操作从O(n)降至O(1)，提升效率。

4、递归逻辑：

- 若当前节点是目标节点，初始化“当前子树包含的目标节点数”为1；否则为0。

- 递归计算左、右子树包含的目标节点数，累加到当前计数。

- 若当前计数等于目标节点总数，且尚未记录LCA（确保“最深”），则当前节点即为LCA。

- 返回当前子树包含的目标节点数，供父节点判断。

代码：

/\*\*

\* Definition for a binary tree node.

\* struct TreeNode {

\* int val;

\* TreeNode \*left;

\* TreeNode \*right;

\* TreeNode() : val(0), left(nullptr), right(nullptr) {}

\* TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}

\* TreeNode(int x, TreeNode \*left, TreeNode \*right) : val(x), left(left), right(right) {}

\* };

\*/

class Solution {

private:

TreeNode\* lca = nullptr; // 存储最终找到的LCA

unordered\_set<TreeNode\*> targetSet; // 目标节点集合（快速判断节点是否为目标）

int targetCount; // 目标节点总数（用于判断是否包含所有目标）

// 递归函数：后序遍历，返回当前子树包含的目标节点数

int dfs(TreeNode\* root) {

if (root == nullptr) return 0;

// 1. 递归处理左子树，获取左子树包含的目标节点数

int leftCount = dfs(root->left);

// 2. 递归处理右子树，获取右子树包含的目标节点数

int rightCount = dfs(root->right);

// 3. 计算当前子树包含的目标节点数（当前节点是否为目标 + 左右子树的数量）

int currCount = leftCount + rightCount;

if (targetSet.count(root)) {

currCount++;

}

// 4. 若当前子树包含所有目标节点，且尚未记录LCA（确保是最深的节点），则更新LCA

if (currCount == targetCount && lca == nullptr) {

lca = root;

}

// 5. 返回当前子树包含的目标节点数，供父节点使用

return currCount;

}

public:

TreeNode\* lowestCommonAncestor(TreeNode\* root, vector<TreeNode\*> &nodes) {

// 初始化目标节点集合和总数

targetCount = nodes.size();

for (TreeNode\* node : nodes) {

targetSet.insert(node);

}

// 后序遍历树，寻找LCA

dfs(root);

return lca;

}

};

代码解释：

1、初始化目标集合：

- 将输入的nodes列表转换为unordered\_set（targetSet），使“判断节点是否为目标”的操作时间从O(n)（遍历列表）降至O(1)（哈希查找）。

- 记录目标节点总数（targetCount），用于后续判断“当前子树是否包含所有目标节点”。

2、后序遍历（dfs函数）：

- 终止条件：若当前节点为nullptr，返回0（空树不包含任何目标节点）。

- 左/右子树处理：递归计算左、右子树包含的目标节点数（leftCount和rightCount）。

- 当前节点计数：累加左、右子树的计数，若当前节点是目标节点则加1，得到currCount（当前子树包含的目标总数）。

- LCA判断：若currCount == targetCount（当前子树包含所有目标），且lca尚未赋值（确保是“最深”的节点，因为后序遍历先处理子树再处理父节点，子树满足条件时会先赋值lca），则当前节点即为LCA。

3、返回结果：遍历结束后，lca存储的就是所有目标节点的最近公共祖先，直接返回即可。

复杂度分析

- 时间复杂度：O(n)，其中n是二叉树的节点数。

后序遍历会访问每个节点恰好一次，时间O(n)。

目标集合的初始化（遍历nodes列表）时间为O(k)（k是目标节点数，k ≤ n），可忽略。

哈希集合的插入和查找操作均为O(1)，不额外增加时间复杂度。

- 空间复杂度：O(n)。

递归调用栈的深度取决于树的高度，最坏情况下（链状树）为O(n)，平均情况下（平衡树）为O(log n)。

目标集合targetSet存储k个节点，空间O(k) ≤ O(n)。

关键优势

1、通用性强：完美覆盖“单个节点”“两个节点”“多个节点”的所有场景（如示例2中单个节点的LCA是自身，示例3中四个节点的LCA是5）。

2、效率最优：通过哈希集合优化目标节点判断，后序遍历确保每个节点仅访问一次，时间复杂度为O(n)，适配题目中`n≤1e4`的约束。

3、逻辑清晰：通过“子树目标节点计数”的方式，自然扩展两节点LCA的逻辑，易于理解和维护。

该方法是解决多节点LCA问题的经典且高效的方案，充分利用了二叉树的遍历特性和哈希表的高效查找能力。