二叉搜索树的最近公共祖先

给定一个二叉搜索树, 找到该树中两个指定节点的最近公共祖先。

百度百科中最近公共祖先的定义为：“对于有根树 T 的两个结点 p、q，最近公共祖先表示为一个结点 x，满足 x 是 p、q 的祖先且 x 的深度尽可能大（一个节点也可以是它自己的祖先）。”

例如，给定如下二叉搜索树: root = [6,2,8,0,4,7,9,null,null,3,5]

示例 1:

输入： `root = [6,2,8,0,4,7,9,null,null,3,5], p = 2, q = 8`

输出： `6`

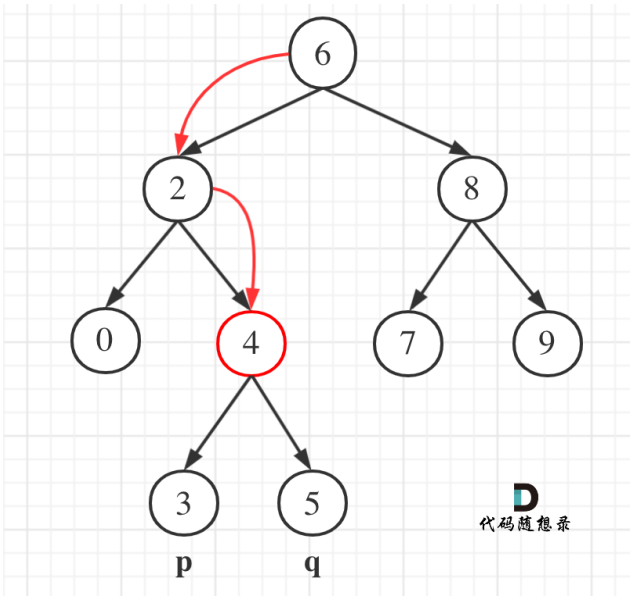
解释： 节点 2 和节点 8 的最近公共祖先是 6。

示例 2:

输入： `root = [6,2,8,0,4,7,9,null,null,3,5], p = 2, q = 4`

输出： `2`

解释： 节点 2 和节点 4 的最近公共祖先是 2，因为根据定义最近公共祖先节点可以为节点本身。



因为二叉树搜索树节点有序，所以问题变得简单。如果节点 p 比当前节点小，则说明节点 p 在当前节点的左子树；如果节点 p 比当前节点大，则说明节点 p 在当前节点的右子树。
代码思路为：
如果 pq 的值都小于当前节点的值，则递归进入当前节点的左子树；
如果 pq 的值都大于当前节点的值，则递归进入当前节点的右子树；
如果当前节点的值在 pq 两个节点的值的中间，那么这两个节点的最近公共祖先则为当前

的节点。
递归终止条件：
1 当 p 和 q 在当前的 root 的一左一右， return 当前的 root
题中没有说明 p 和 q 的左右顺序，因此 p，q 谁在谁左边都有可能
2p，q 有一个或者两个节点的值和 root 相等， return 当前的 root

搜索二叉树具有特性：左子树<root<右子树
所以 p，q 小于 root 时 左子树递归查找
p，q 大于 root 时，右子树递归查找

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
class Solution {
public:
    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q
) {
        TreeNode* res;
        if(root==nullptr||p==nullptr&&q==nullptr)
        {
            return nullptr;
        }
        if(p->val<=root->val&&q->val>=root->val||p->val>=root->val&&q->val<
=root->val)
        {
            res=root;
        }
        if(p->val<root->val&&q->val<root->val)
        {
            res=lowestCommonAncestor(root->left,p,q);
        }
        if(p->val>root->val&&q->val>root->val)
        {
            res=lowestCommonAncestor(root->right,p,q);
        }
        return res;
    }
};
```

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
class Solution {
public:
    TreeNode* dfs(TreeNode* cur, TreeNode* p, TreeNode* q)
    {
        if(cur==nullptr)
            return cur;
        if(p->val<cur->val&&q->val<cur->val)
        {
            TreeNode* left=dfs(cur->left,p,q);
            if(left!=nullptr)
                return left;
        }
        if(p->val>cur->val&&q->val>cur->val)
        {
            TreeNode* right=dfs(cur->right,p,q);
            if(right!=nullptr)
                return right;
        }
        return cur;
    }
    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q
)
    {
        return dfs(root,p,q);
    }
};
```