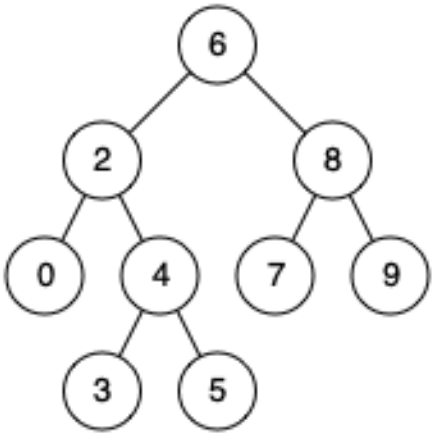
235. Lowest Common Ancestor of a Binary Search Tree

Created	@September 9, 2022 1:15 PM
Difficulty	Medium
LC Url	https://leetcode.com/problems/lowest-common-ancestor-of-a-binary-search-tree/
Importance	****
Tag	NEET Recursion Tree
Video	

Given a binary search tree (BST), find the lowest common ancestor (LCA) node of two given nodes in the BST.

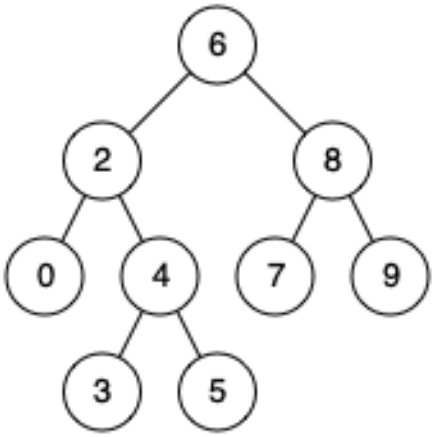
According to the definition of LCA on Wikipedia: “The lowest common ancestor is defined between two nodes `p` and `q` as the lowest node in `T` that has both `p` and `q` as descendants (where we allow **a node to be a descendant of itself**).”

Example 1:




```
Input: root = [6,2,8,0,4,7,9,null,null,3,5], p = 2, q = 8
Output: 6
Explanation: The LCA of nodes 2 and 8 is 6.
```

Example 2:




```
Input: root = [6,2,8,0,4,7,9,null,null,3,5], p = 2, q = 4
Output: 2
Explanation: The LCA of nodes 2 and 4 is 2, since a node can be a descendant of itself according to the LCA definition.
```

Example 3:

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

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 BST.

Solution

如果在 BST 中寻找最近公共祖先，反而容易很多，主要利用 BST 左小右大（左子树所有节点都比当前节点小，右子树所有节点都比当前节点大）的特点即可。

1、如果 `p` 和 `q` 都比当前节点小，那么显然 `p` 和 `q` 都在左子树，那么 LCA 在左子树。

2、如果 `p` 和 `q` 都比当前节点大，那么显然 `p` 和 `q` 都在右子树，那么 LCA 在右子树。

3、一旦发现 `p` 和 `q` 在当前节点的两侧，说明当前节点就是 LCA。

```
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution:
    def lowestCommonAncestor(self, root: 'TreeNode', p: 'TreeNode', q: 'TreeNode') -> 'TreeNode':
        if not root:
            return None

        # 保证p.val < q.val
        if p.val > q.val:
```

```
            return self.lowestCommonAncestor(root, q, p)

        # p和q分别在root的左右子树，那么root就是LCA
        if p.val <= root.val <= q.val:
            return root

        if root.val > q.val:
            return self.lowestCommonAncestor(root.left, p, q)
        else:
            return self.lowestCommonAncestor(root.right, p, q)
```

方法一：两次遍历

注意到题目中给出的是一棵「二叉搜索树」，因此我们可以快速地找出树中的某个节点以及从根节点到该节点的路径，例如我们需要找到节点 p：

- 我们从根节点开始遍历；
- 如果当前节点就是 p，那么成功地找到了节点；
- 如果当前节点的值大于 p 的值，说明 p 应该在当前节点的左子树，因此将当前节点移动到它的左子节点；
- 如果当前节点的值小于 p 的值，说明 p 应该在当前节点的右子树，因此将当前节点移动到它的右子节点。

对于节点 q 同理。在寻找节点的过程中，我们可以顺便记录经过的节点，这样就得到了从根节点到被寻找节点的路径。

当我们分别得到了从根节点到 p 和 q 的路径之后，我们就可以很方便地找到它们的最近公共祖先了。显然，p 和 q 的最近公共祖先就是从根节点到它们路径上的「分岔点」，也就是最后一个相同的节点。因此，如果我们设从根节点到 p 的路径为数组 $path\_p$，从根节点到 q 的路径为数组 $path\_q$，那么只要找出最大的编号 i，其满足

$$path\_p[i] = path\_q[i]$$

那么对应的节点就是「分岔点」，即 p 和 q 的最近公共祖先就是 $path\_p[i]$（或 $path\_q[i]$）。

代码

```
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution:
    def lowestCommonAncestor(self, root: 'TreeNode', p: 'TreeNode', q: 'TreeNode') -> 'TreeNode':
        def getPath(root, target):
            path = []
            node = root
            while node != target:
                path.append(node)
                if target.val < node.val:
                    node = node.left
                else:
                    node = node.right
            path.append(node)
            return path

        path_p = getPath(root, p)
        path_q = getPath(root, q)
```

```
        ancestor = None
        for u, v in zip(path_p, path_q):
            if u == v:
                ancestor = u
            else:
                break
        return ancestor

        # 链接：https://leetcode.cn/problems/lowest-common-ancestor-of-a-binary-search-tree/solution/er-cha-sou-suo-shu-de-zui-jin-gong-gong
```

复杂度分析

- 时间复杂度：$O(n)$，其中 n 是给定的二叉搜索树中的节点个数。上述代码需要的时间与节点 p 和 q 在树中的深度线性相关，而在最坏的情况下，树呈现链式结构，p 和 q 一个是树的唯一叶子结点，一个是该叶子结点的父节点，此时时间复杂度为 $\Theta(n)$。
- 空间复杂度：$O(n)$，我们需要存储根节点到 p 和 q 的路径。和上面的分析方法相同，在最坏的情况下，路径的长度为 $\Theta(n)$，因此需要 $\Theta(n)$ 的空间。

方法二：一次遍历

在方法一中，我们对从根节点开始，通过遍历找出到达节点 p 和 q 的路径，一共需要两次遍历。我们也可以考虑将这两个节点放在一起遍历。

整体的遍历过程与方法一中的类似：

- 我们从根节点开始遍历；
- 如果当前节点的值大于 p 和 q 的值，说明 p 和 q 应该在当前节点的左子树，因此将当前节点移动到它的左子节点；
- 如果当前节点的值小于 p 和 q 的值，说明 p 和 q 应该在当前节点的右子树，因此将当前节点移动到它的右子节点；
- 如果当前节点的值不满足上述两条要求，那么说明当前节点就是「分岔点」。此时，p 和 q 要么在当前节点的不同的子树中，要么其中一个就是当前节点。

可以发现，如果我们将这两个节点放在一起遍历，我们就省去了存储路径需要的空间。

```
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution:
    def lowestCommonAncestor(self, root: 'TreeNode', p: 'TreeNode', q: 'TreeNode') -> 'TreeNode':
        ancestor = root
        while True:
            if p.val < ancestor.val and q.val < ancestor.val:
                ancestor = ancestor.left
            elif p.val > ancestor.val and q.val > ancestor.val:
                ancestor = ancestor.right
            else:
                break
        return ancestor
```

复杂度分析

- 时间复杂度：$O(n)$，其中 n 是给定的二叉搜索树中的节点个数。分析思路与方法一相同。
- 空间复杂度：$O(1)$。