二叉树的属性

求普通二叉树的属性，一般是后序(也有前序)，一般要通过递归函数的返回值做计算。

二叉树：是否对称(\101. Symmetric Tree)

- 递归：后序，比较的是根节点的左子树与右子树是不是相互翻转

```
class Solution:
    def isSymmetric(self, root: Optional[TreeNode]) -> bool:
        if not root:
            return False
        return self.compare(root.left,root.right)
    def compare(self,l,r):
        if l == None and r!= None:
            return False
        elif l != None and r== None:
            return False
        elif l == None and r == None:
            return True
        elif l.val == r.val:
            return self.compare(l.left,r.right) and self.compare(l.right,r.left)
```

- 迭代：使用队列/栈将两个节点顺序放入容器中进行比较

二叉树：求最大深度(Maximum Depth of Binary Tree)

- 递归：后序，求根节点最大高度就是最大深度，通过递归函数的返回值做计算树的高度

```
class Solution:
    def maxDepth(self, root: Optional[TreeNode]) -> int:
        return self.traverse(root)
    def traverse(self,root):
        if not root:
            return 0
        l = self.traverse(root.left)
        r = self.traverse(root.right)
        return max(l,r)+1
```

- 迭代：层序遍历

二叉树：求最大直径(543. Diameter of Binary Tree)

错误解答： 最大直径是左子树和右子树的最大深度之和，但是万一最大直径没有经过根节点呢？所以说对于树中的每一个结点，都要把它视为根节点，然后比较所有结点的左子树和右子树的最大深度之和，取其中的最大值。

注意的地方：

1. get_diameter函数返回的是深度！
2. 全局变量max_diameter要加上self

```
class Solution:
    def diameterOfBinaryTree(self, root: Optional[TreeNode]) -> int:
        self.max_diameter = 0
        def get_diameter(root):
            if not root:
                return 0
            left = get_diameter(root.left) #深度
            right = get_diameter(root.right) #深度
            diameter = left + right #直径
            self.max_diameter = max(self.max_diameter,diameter)
            return max(left,right)+1  #这里返回的是深度！！因为上面求的是深度
        get_diameter(root)
        return self.max_diameter
```

124. Binary Tree Maximum Path Sum

思路同二叉树的直径。

```
class Solution:
    def maxPathSum(self, root: Optional[TreeNode]) -> int:
        #思路同二叉树的直径
        self.max_value = -inf
        def get_value(root):
            if not root:
                return 0
            mid = 0
            #看看是否比0大 比0小会在mid计算上产生副作用 所以直接删掉
            left_max = max(0,get_value(root.left))
            right_max = max(0,get_value(root.right))
            mid = left_max + right_max + root.val
            self.max_value = max(mid,self.max_value)
            return max(left_max,right_max) + root.val #不再返回高度 而是返回数值
        get_value(root)
        return self.max_value
```

二叉树：求最小深度(Minimum Depth of Binary Tree)

- 递归：后序，求根节点最小高度就是最小深度，注意最小深度的定义

```python
class Solution:
    def minDepth(self, root: Optional[TreeNode]) -> int:
        height = self.traverse(root)
        return height

    def traverse(self,root):
        if not root:
            return 0
        left_height = self.traverse(root.left)
        right_height = self.traverse(root.right)
        if root.left != None and root.right == None:
            return left_height + 1
        if root.left == None and root.right != None:
            return right_height + 1

        return min(left_height, right_height)+1
```

- 迭代：层序遍历

二叉树：求有多少个节点(222. Count Complete Tree Nodes)

- 递归：后序，通过递归函数的返回值计算节点数量

```python
class Solution:
    def countNodes(self, root: Optional[TreeNode]) -> int:
        return self.cnt(root)
    def cnt(self,root):
        if not root:
            return 0
        l = self.cnt(root.left)
        r = self.cnt(root.right)
        return l+r+1
```

- 迭代：层序遍历

二叉树：是否平衡

该题复用了求最大深度的函数

- 递归：后序，注意后序求高度和前序求深度，递归过程判断高度差

```
class Solution:
    def isBalanced(self, root: Optional[TreeNode]) -> bool:
        if self.traverse(root)!=-1:
            return True
        else:
            return False

    def traverse(self,root):
        if not root:
            return 0

        l = self.traverse(root.left)
        if(l == -1): return -1

        r = self.traverse(root.right)
        if(r == -1): return -1

        if(abs(l-r)>1): return -1

        return max(l,r)+1
```

- 迭代：效率很低，不推荐

二叉树：找所有路径(257. Binary Tree Paths)

- 递归：前序，方便让父节点指向子节点，涉及回溯处理根节点到叶子的所有路径

```
class Solution:
    #前序遍历
    def binaryTreePaths(self, root: Optional[TreeNode]) -> List[str]:
        path = ''
        result = []
        if not root: return result
        self.traversal(root, path, result)
        return result

    def traversal(self, cur: TreeNode, path: str, result: List[str]) -> None:
        path += str(cur.val)
        # 若当前节点为leave，直接输出
```

```
        if not cur.left and not cur.right:
            result.append(path)

        if cur.left:
            # + '->' 是隐藏回溯
            self.traversal(cur.left, path + '->', result)

        if cur.right:
            self.traversal(cur.right, path + '->', result)
```

类似题目：求最小深度。

- 迭代：一个栈模拟递归，一个栈来存放对应的遍历路径

二叉树中求和/求值问题

二叉树：求左叶子之和(404. Sum of Left Leaves)

- 递归：后序，必须三层约束条件，才能判断是否是左叶子。

```
class Solution:
    def sumOfLeftLeaves(self, root: Optional[TreeNode]) -> int:
        return self.traverse(root)

    def traverse(self,root):
        if not root:
            return 0

        l = self.traverse(root.left)
        r = self.traverse(root.right)

        mid = 0
        if root.left != None and root.left.left == None and root.left.right == None:
            mid += root.left.val

        return mid + l + r
```

- 迭代：直接模拟后序遍历

二叉树：求左下角的值(513. Find Bottom Left Tree Value)

- 递归：顺序无所谓，优先左孩子搜索，同时找深度最大的叶子节点。
- 迭代：层序遍历找最后一行最左边

```python
class Solution:
    def findBottomLeftValue(self, root: Optional[TreeNode]) -> int:
        return self.levelOrder(root)

    def levelOrder(self, root: TreeNode) -> List[List[int]]:
        results = 0
        if not root:
            return results

        from collections import deque
        que = deque([root])

        while que:
            size = len(que) #size一定要写在这里，因为每轮都会变
            for i in range(size):
                if i==0:
                    results = que[i].val #res只记录最左边元素就行
                cur = que.popleft()
                if cur.left:
                    que.append(cur.left)
                if cur.right:
                    que.append(cur.right)
        return results
```

129.Sum Root to Leaf Numbers

细品

```java
// 具有返回值的dfs方法
class Solution {
    public int sumNumbers(TreeNode root) {
        return dfs(root, 0);
    }
    private int dfs(TreeNode node, int num){
        if(node == null) return 0;
        num = num * 10 + node.val;
        if(node.left == null && node.right == null){
            return num; // 叶子结点返回本路径数字
        }
        return dfs(node.left, num) + dfs(node.right, num); // 其他结点返回儿子代表的路径数字的和
    }
}
```

```java
// 无需返回值的dfs方法
// 只关心在叶子结点处的累加，#1，#2，#3的位置是任意的，即前序中序后序都可以
class Solution {
    int sum = 0;
    public int sumNumbers(TreeNode root) {
        dfs(root, 0);
        return sum;
    }
    private void dfs(TreeNode node, int num){
        if(node == null) return;
        num = num * 10 + node.val;
        if(node.left == null && node.right == null){ // #1
            sum += num;
        }
        dfs(node.left, num); // #2
        dfs(node.right, num); // #3
    }
}
```

```python
class Solution:
    def sumNumbers(self, root: Optional[TreeNode]) -> int:
        def traverse(root,presum):
            if not root:
                return 0

            total = presum * 10 + root.val
            if not root.left and not root.right:
                return total
            else:
                return traverse(root.left,total) + traverse(root.right,total)
        return traverse(root,0)
```

二叉树：求路径总和(112. Path Sum && 113. Path Sum II)

- 递归：顺序无所谓，递归函数返回值为bool类型是为了搜索一条边，没有返回值是搜索整棵树。

```python
## 112. Path Sum
# 输出的是bool
class Solution:
    def hasPathSum(self, root: Optional[TreeNode], targetSum: int) -> bool:
        if not root:
            return False
        if not root.left and not root.right:
            return targetSum == root.val
        l = self.hasPathSum(root.left,targetSum-root.val)
        r = self.hasPathSum(root.right,targetSum-root.val)
        return l or r

# 也可以这样写 清晰展现整个回溯过程
class solution:
    def haspathsum(self, root: treenode, targetsum: int) -> bool:
        def isornot(root, targetsum) -> bool:
            if (not root.left) and (not root.right) and targetsum == 0:
                return true  # 遇到叶子节点，并且计数为0
            if (not root.left) and (not root.right):
                return false  # 遇到叶子节点，计数不为0
            if root.left:
                targetsum -= root.left.val  # 左节点
                if isornot(root.left, targetsum): return true  # 递归，处理左节点
                targetsum += root.left.val  # 回溯
            if root.right:
                targetsum -= root.right.val  # 右节点
                if isornot(root.right, targetsum): return true  # 递归，处理右节点
```

```
                targetsum += root.right.val  # 回溯
            return false

        if root == none:
            return false  # 别忘记处理空treenode
        else:
            return isornot(root, targetsum - root.val)
```

```
## 113. Path Sum II
## 类似题目 257
class Solution:
    def pathSum(self, root: TreeNode, targetSum: int) -> List[List[int]]:
        res = list()
        path = list()

        def dfs(root: TreeNode, targetSum: int):
            if not root:
                return
            path.append(root.val)
            targetSum -= root.val
            if not root.left and not root.right and targetSum == 0:
                res.append(path[:])
            dfs(root.left, targetSum)
            dfs(root.right, targetSum)
            path.pop()

        dfs(root, targetSum)
        return res

# 也可以这样写 清晰展现整个回溯过程
class solution:
    def pathsum(self, root: treenode, targetsum: int) -> list[list[int]]:

        def traversal(cur_node, remain):
            if not cur_node.left and not cur_node.right:
                if remain == 0:
                    result.append(path[:])
            return

            if cur_node.left:
                path.append(cur_node.left.val)
                traversal(cur_node.left, remain-cur_node.left.val)
                path.pop()

            if cur_node.right:
```

```
                path.append(cur_node.right.val)
                traversal(cur_node.right, remain-cur_node.left.val)
                path.pop()

        result, path = [], []
        if not root:
            return []
        path.append(root.val)
        traversal(root, targetsum - root.val)
        return result
```

- 迭代：栈里元素不仅要记录节点指针，还要记录从头结点到该节点的路径数值总和

二叉树的构造

涉及到二叉树的构造，无论普通二叉树还是二叉搜索树一定前序，都是先构造中节点。

翻转二叉树(226. Invert Binary Tree)

- 递归：前序，交换左右孩子

```
class Solution:
    def invertTree(self, root: Optional[TreeNode]) -> Optional[TreeNode]:
        if not root:
            return
        l = self.invertTree(root.left)
        r = self.invertTree(root.right)
        root.left = r
        root.right = l
        return root
```

- 迭代：直接模拟前序遍历

构造二叉树(105,106Construct Binary Tree from Preorder and Inorder Traversal)

- 递归：前序，重点在于找分割点，分左右区间构造
- 思路：通过前序或者后续遍历找到中序的中结点（在第一个和最后一个）的值，然后在中序上定位这个点，切割中序的前半部分和后半部分，然后再通过中序前后半部分的长度切割前序和后续，最后扔进递归。

```
# 前序+中序
class Solution:
    def buildTree(self, preorder: List[int], inorder: List[int]) -> TreeNode:
        # 第一步：特殊情况讨论：树为空. 或者说是递归终止条件
        if not preorder:
```

```
            return None

        # 第二步：前序遍历的第一个就是当前的中间节点.
        #构造二叉树节点的方法！先把数值找出来，再用TreeNode
        root_val = preorder[0]
        root = TreeNode(root_val)

        # 第三步：找切割点.
        separator_idx = inorder.index(root_val)

        # 第四步：切割inorder数组. 得到inorder数组的左,右半边.
        inorder_left = inorder[:separator_idx]
        inorder_right = inorder[separator_idx + 1:]

        # 第五步：切割preorder数组. 得到preorder数组的左,右半边.
        # ⭐ 重点1：中序数组大小一定跟前序数组大小是相同的.
        preorder_left = preorder[1:1 + len(inorder_left)]
        preorder_right = preorder[1 + len(inorder_left):]

        # 第六步：递归
        root.left = self.buildTree(preorder_left, inorder_left)
        root.right = self.buildTree(preorder_right, inorder_right)

        return root
```

```
# 中序+后续
class Solution:
    def buildTree(self, inorder: List[int], postorder: List[int]) ->
Optional[TreeNode]:
        if not postorder:
            return None

        root_val = postorder[-1]
        root = TreeNode(root_val)

        separator = inorder.index(root_val)

        inorder_l = inorder[:separator]
        inorder_r = inorder[separator+1:]

        postorder_l = postorder[:len(inorder_l)]
        postorder_r = postorder[len(inorder_l):-1]

        root.left = self.buildTree(inorder_l,postorder_l)
        root.right = self.buildTree(inorder_r,postorder_r)
```

```
        return root
```

- 迭代：比较复杂，意义不大

构造最大的二叉树(654. Maximum Binary Tree)

- 递归：前序，分割点为数组最大值，分左右区间构造

```
class Solution:
    def constructMaximumBinaryTree(self, nums: List[int]) -> Optional[TreeNode]:
        #和105 106差不多 只是根节点是当前最大元素
        if not nums:
            return None

        root_val = max(nums)
        root = TreeNode(root_val)
        root_idx = nums.index(root_val)
        l = nums[:root_idx]
        r = nums[root_idx+1:]
        root.left = self.constructMaximumBinaryTree(l)
        root.right = self.constructMaximumBinaryTree(r)
        return root
```

- 迭代：比较复杂，意义不大

合并两个二叉树

- 递归：前序，同时操作两个树的节点，注意合并的规则

```
class Solution:
    def mergeTrees(self, root1: Optional[TreeNode], root2: Optional[TreeNode]) ->
Optional[TreeNode]:
        if root1 and not root2:
            return root1
        elif not root1 and root2:
            return root2
        elif root1 and root2: #中
            root1.val += root2.val
        else:
            return None

        root1.left = self.mergeTrees(root1.left,root2.left) #左
```

```
        root1.right = self.mergeTrees(root1.right,root2.right) #右

        return root1
```

- 迭代：使用队列，类似层序遍历

二叉搜索树中的插入操作

- 递归：顺序无所谓，通过递归函数返回值添加节点

```
class Solution:
    def insertIntoBST(self, root: Optional[TreeNode], val: int) -> Optional[TreeNode]:
        if not root:
            return TreeNode(val)

        if root.val < val:
            root.right = self.insertIntoBST(root.right,val)
        elif root.val > val:
            root.left = self.insertIntoBST(root.left,val)

        return root
```

- 迭代：按序遍历，需要记录插入父节点，这样才能做插入操作

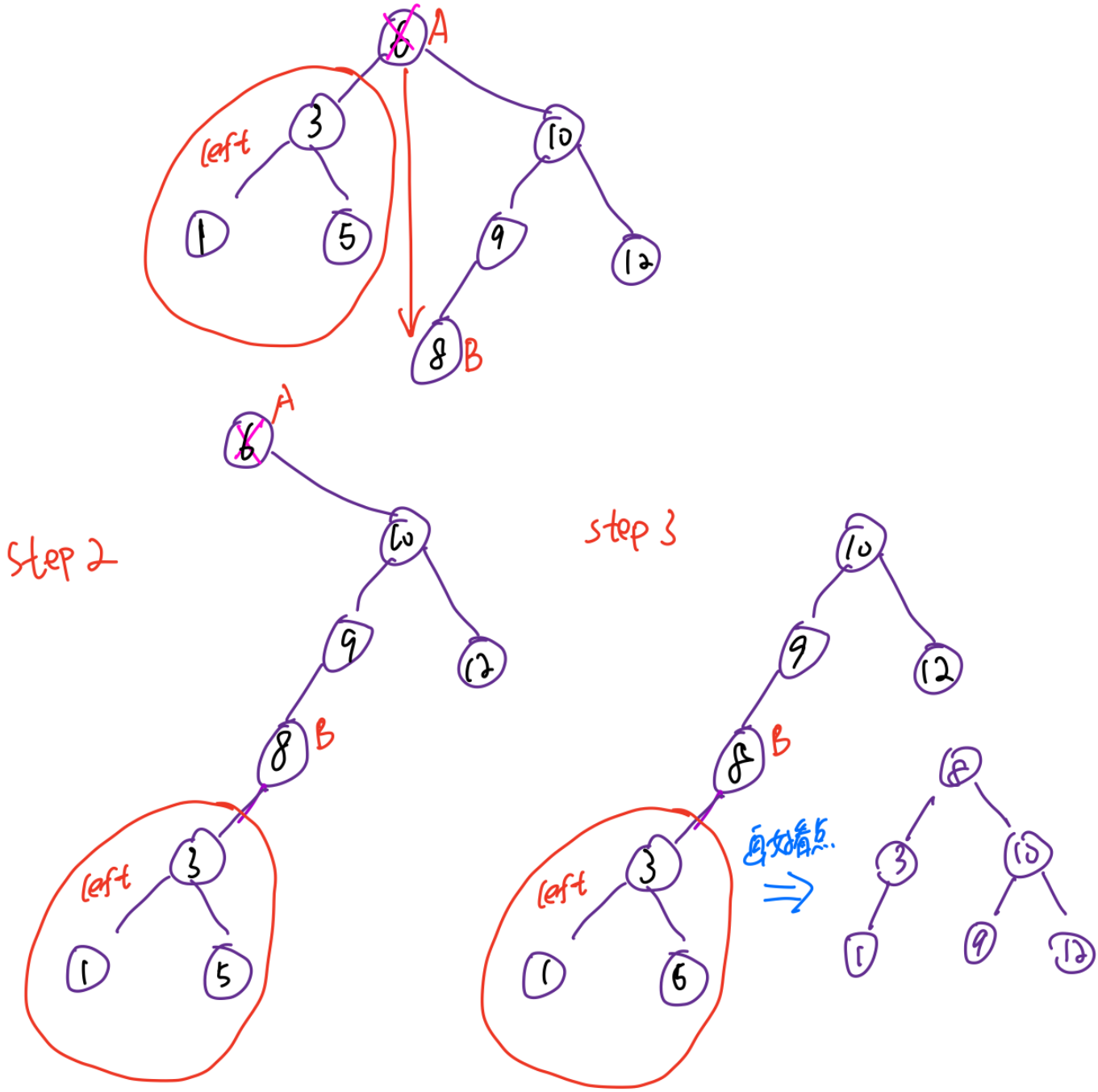
二叉搜索树中的删除操作

- 递归：前序，想清楚删除非叶子节点的情况

```
class Solution:
    def deleteNode(self, root: Optional[TreeNode], key: int) -> Optional[TreeNode]:
        if not root : return  None # 节点为空，返回
        if root.val < key :
            root.right = self.deleteNode(root.right, key)
        elif root.val > key :
            root.left = self.deleteNode(root.left, key)
        else: # root.val == key
            # 当前节点的左子树为空，返回当前的右子树
            if not root.left : return root.right
            # 当前节点的右子树为空，返回当前的左子树
            if not root.right: return root.left

            #如何更换?
            # 左右子树都不为空，找到右孩子的最左节点 记为p
```

```
            node = root.right
            while node.left :
                node = node.left
            # 将当前节点的左子树挂在p的左孩子上
            node.left = root.left
            # 当前节点的右子树替换掉当前节点，完成当前节点的删除
            root = root.right
        return root
```



- 迭代：有序遍历，较复杂

修剪二叉搜索树

- 递归：前序，通过递归函数返回值删除节点

```
class Solution:
    def trimBST(self, root: Optional[TreeNode], low: int, high: int) ->
Optional[TreeNode]:
        if not root:
            return None

        if root.val < low:
            return self.trimBST(root.right,low,high)
        if root.val > high:
            return self.trimBST(root.left,low,high)

        #根节点要是满足了 其他原封不动!
        if low <= root.val <= high:
            root.left = self.trimBST(root.left,low,high)
            root.right = self.trimBST(root.right,low,high)
            return root
```

- 迭代：有序遍历，较复杂

构造二叉搜索树

- 递归：前序，数组中间节点分割

```
class Solution:
    def sortedArrayToBST(self, nums: List[int]) -> Optional[TreeNode]:
        #中间结点就是根节点
        return self.traverse(nums,0,len(nums)-1)

    #二分搜索
    def traverse(self,nums,left,right):
        if(left > right): #二分搜索的终止条件!
            return None
        mid = left + (right - left)//2
        root = TreeNode(nums[mid])
        root.left = self.traverse(nums,left,mid-1)
        root.right = self.traverse(nums,mid+1,right)
        return root
```

- 迭代：较复杂，通过三个队列来模拟

116. Populating Next Right Pointers in Each Node

非常经典的递归思想的使用

```
class Solution:
    def connect(self, root: 'Optional[Node]') -> 'Optional[Node]':
        if not root:
            return None
        def traverse(node1,node2):
            if not node1 or not node2:
                return None
            node1.next = node2
            traverse(node1.left,node1.right)
            traverse(node2.left,node2.right)
            traverse(node1.right,node2.left)

        traverse(root.left,root.right)
        return root
```

117.Populating Next Right Pointers in Each Node

层序遍历

```
class Solution:
    def connect(self, root: 'Node') -> 'Node':
        if not root:
            return None
        q = deque([root])
        while q:
            s = len(q)
            tail = None
            for _ in range(s):
                cur = q.popleft()
                if tail:
                    tail.next = cur
                tail = cur
                if cur.left:
                    q.append(cur.left)
                if cur.right:
                    q.append(cur.right)
        return root
```

99. Recover Binary Search Tree

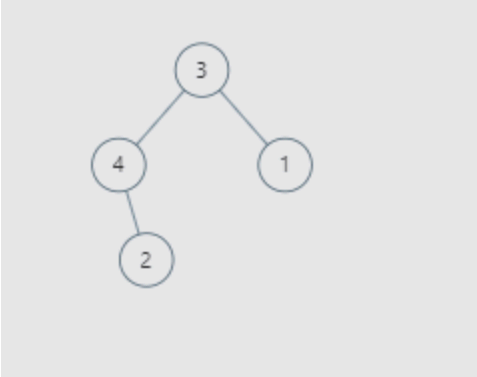
这道题难点，是找到那两个交换节点，把它交换过来就行了。

这里我们二叉树搜索树的中序遍历(中序遍历遍历元素是递增的)

如下图所示，中序遍历顺序是 `4,2,3,1`，我们只要找到节点 `4` 和节点 `1` 交换顺序即可！

这里我们有个规律发现这两个节点：
第一个节点，是第一个按照中序遍历时候前一个节点大于后一个节点，我们选取前一个节点，这里指节点 `4`；

第二个节点，是在第一个节点找到之后，后面出现前一个节点大于后一个节点，我们选择后一个节点，这里指节点 `1`；




```
class Solution:
    def recoverTree(self, root: TreeNode) -> None:
        """
        Do not return anything, modify root in-place instead.
        """
        self.firstNode = None
        self.secondNode = None
        self.preNode = TreeNode(float("-inf"))

        def in_order(root):
            if not root:
                return
            in_order(root.left)
            # 中序改造
            if self.firstNode == None and self.preNode.val >= root.val:
                self.firstNode = self.preNode
            if self.firstNode and self.preNode.val >= root.val:
                self.secondNode = root
```

```
            self.preNode = root

            in_order(root.right)

        in_order(root)
        self.firstNode.val, self.secondNode.val = self.secondNode.val, self.firstNode.val
```

二叉搜索树(BST)的属性

求二叉搜索树的属性，一定是中序了，要不白瞎了有序性了。

[二叉搜索树中的搜索]

- 递归：二叉搜索树的递归是有方向的

```
class Solution:
    def searchBST(self, root: TreeNode, val: int) -> TreeNode:
        # 为什么要有返回值：
        #   因为搜索到目标节点就要立即return，
        #   这样才是找到节点就返回（搜索某一条边），如果不加return，就是遍历整棵树了。

        if not root or root.val == val:
            return root

        if root.val > val:
            return self.searchBST(root.left, val)

        if root.val < val:
            return self.searchBST(root.right, val)
```

- 迭代：因为有方向，所以迭代法很简单

```
class Solution:
    def searchBST(self, root: TreeNode, val: int) -> TreeNode:
        while root is not None:
            if val < root.val: root = root.left
            elif val > root.val: root = root.right
            else: return root
        return None
```

是不是二叉搜索树

- 递归：中序，相当于变成了判断一个序列是不是递增的

```python
class Solution:
    # 中序遍历
    def inOrder(self, root: TreeNode, res):
        if root == None:
            return
        self.inOrder(root.left, res)
        res.append(root.val)
        self.inOrder(root.right, res)


    def isValidBST(self, root: TreeNode) -> bool:
        res = []
        self.inOrder(root, res)
        return all(x<y for x,y in zip(res,res[1:]))  #技巧
```

- 迭代：模拟中序，逻辑相同

求二叉搜索树的最小绝对差

- 递归：中序，双指针操作

```python
## 先排序 在2个对比
class Solution:
    def inorder(self,root,res):
        if not root:
            return None
        self.inorder(root.left,res)
        res.append(root.val)
        self.inorder(root.right,res)


    def getMinimumDifference(self, root: Optional[TreeNode]) -> int:
        res =[]
        self.inorder(root,res)
        return min(abs(x-y) for x,y in zip(res,res[1:]))
```

- 迭代：模拟中序，逻辑相同

求二叉搜索树的众数

- 递归：中序，清空结果集的技巧，遍历一遍便可求众数集合

二叉搜索树转成累加树

- 递归：中序，双指针操作累加
- 迭代：模拟中序，逻辑相同

96. Unique Binary Search Trees

```
class Solution:
    def numTrees(self, n: int) -> int:
        #每一个节点作为根节点形成的树都是二叉搜索树
        #dp[i]: i个结点可以形成多少颗二叉搜索树
        dp = [0] * (n+1)
        dp[0] = 1 #边界
        dp[1] = 1
        for i in range(2,n+1):
            for j in range(1,i+1):
            #有n个结点，j为根节点，左子树节点的个数为j-1，右子树节点个数为n-j
                dp[i] += dp[j-1] * dp[i-j]
        return dp[n]
```

二叉树公共祖先问题

二叉树的公共祖先问题(236)

- 递归：后序，回溯，找到左子树出现目标值，右子树节点目标值的节点。

```
class Solution:
    def lowestCommonAncestor(self, root: 'TreeNode', p: 'TreeNode', q: 'TreeNode') ->
'TreeNode':
        #base
        if not root or root == p or root == q:
            return root

        #后序遍历
        left = self.lowestCommonAncestor(root.left,p,q)
        right = self.lowestCommonAncestor(root.right,p,q)
```

```
        if not left and not right: #两边都找不到
            return None
        elif left and not right: #都在左边
            return left
        elif right and not left: #都在右边
            return right
        elif left and right: #分别在树的两边
            return root
```

递归解析：

1. **终止条件：**
    1. 当越过叶节点，则直接返回 $null$ ；
    2. 当 $root$ 等于 p, q ，则直接返回 $root$ ；
2. **递推工作：**
    1. 开启递归左子节点，返回值记为 $left$ ；
    2. 开启递归右子节点，返回值记为 $right$ ；
3. **返回值：** 根据 $left$ 和 $right$ ，可展开为四种情况；
    1. 当 $left$ 和 $right$ **同时为空** ：说明 $root$ 的左 / 右子树中都不包含 p, q ，返回 $null$ ；
    2. 当 $left$ 和 $right$ **同时不为空** ：说明 p, q 分列在 $root$ 的 **异侧** （分别在 左 / 右子树），因此 $root$ 为最近公共祖先，返回 $root$ ；
    3. 当 $left$ **为空** ，$right$ **不为空** ：p, q 都不在 $root$ 的左子树中，直接返回 $right$ 。具体可分为两种情况：
        1. p, q 其中一个在 $root$ 的 **右子树** 中，此时 $right$ 指向 p（假设为 p ）；
        2. p, q 两节点都在 $root$ 的 **右子树** 中，此时的 $right$ 指向 **最近公共祖先节点** ；
    4. 当 $left$ **不为空** ， $right$ **为空** ：与情况 `3.` 同理；

二叉搜索树的公共祖先问题(235)

- 递归：顺序无所谓，如果节点的数值在目标区间就是最近公共祖先

```
class Solution:
    def lowestCommonAncestor(self, root: 'TreeNode', p: 'TreeNode', q: 'TreeNode') ->
'TreeNode':
        #Case 1: p,q都在左子树
        if root.val > p.val and root.val > q.val:
            return self.lowestCommonAncestor(root.left, p, q)

        #Case 2: p,q都在右子树
        if root.val < p.val and root.val < q.val:
            return self.lowestCommonAncestor(root.right, p, q)

        #Case3:其余情况
        return root
```

- 迭代：按序遍历