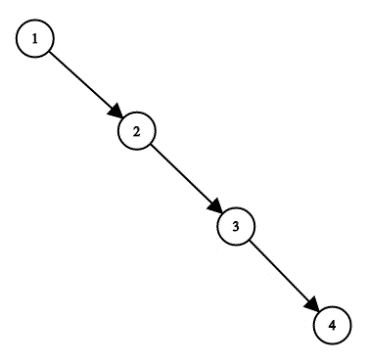
# 题目

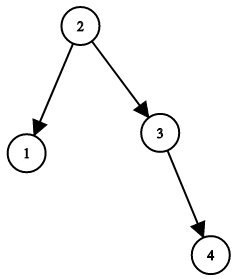
给你一棵二叉搜索树，请你返回一棵 平衡后 的二叉搜索树，新生成的树应该与原来的树有着相同的节点值。

如果一棵二叉搜索树中，每个节点的两棵子树高度差不超过1，我们就称这棵二叉搜索树是平衡的 。

如果有多种构造方法，请你返回任意一种。

**示例：**





**输入：**root = [1,null,2,null,3,null,4,null,null]

**输出：**[2,1,3,null,null,null,4]

**解释：**这不是唯一的正确答案，[3,1,4,null,2,null,null] 也是一个可行的构造方案。

**提示：**

树节点的数目在 1 到 10^4 之间。

树节点的值互不相同，且在 1 到 10^5之间。

# 分析

要解决这个问题，我们需要将一棵棵可能不平衡的二叉搜索树（BST）转换为平衡的二叉搜索树，同时保持原树的节点值不变。平衡的二叉搜索树要求每个节点的左右子树高度差不超过1。

解题思路

二叉搜索树的核心特性是其中序遍历结果为升序序列。基于这一特性，我们可以采用以下步骤实现平衡化：

1、中序遍历提取节点值：对原二叉搜索树进行中序遍历，将所有节点值按升序存储在数组中。这一步利用了BST中序遍历的有序性。

2、构建平衡BST：以升序数组为基础，采用分治思想构建平衡BST。具体来说，选择数组的中间元素作为根节点，中间元素左侧的子数组用于构建左子树，右侧的子数组用于构建右子树。通过这种方式，左右子树的节点数量尽可能均衡，从而保证树的平衡性。

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 {

public:

TreeNode\* balanceBST(TreeNode\* root) {

// 存储中序遍历的结果（升序）

vector<int> vals;

inorder(root, vals);

// 基于升序数组构建平衡BST

return build(vals, 0, vals.size() - 1);

}

private:

// 中序遍历二叉搜索树，获取升序节点值数组

void inorder(TreeNode\* root, vector<int>& vals) {

if (root == nullptr) return;

inorder(root->left, vals); // 遍历左子树

vals.push\_back(root->val); // 记录当前节点值

inorder(root->right, vals); // 遍历右子树

}

// 基于升序数组的[left, right]区间构建平衡BST

TreeNode\* build(vector<int>& vals, int left, int right) {

if (left > right) return nullptr; // 区间无效，返回空节点

// 选择中间元素作为根节点，保证左右子树平衡

int mid = left + (right - left) / 2;

TreeNode\* root = new TreeNode(vals[mid]);

// 递归构建左子树（左半部分数组）

root->left = build(vals, left, mid - 1);

// 递归构建右子树（右半部分数组）

root->right = build(vals, mid + 1, right);

return root;

}

};

解释：

1、中序遍历：inorder函数通过递归遍历原BST的左子树、记录当前节点值、遍历右子树，将节点值按升序存入vals数组。这一步的时间复杂度为O(n)（n为节点总数），空间复杂度为O(n)（存储所有节点值）。

2、构建平衡BST：build函数采用分治策略，以升序数组的中间元素为根节点，确保左右子树的节点数量差不超过1。左子树由中间元素左侧的子数组构建，右子树由中间元素右侧的子数组构建，递归执行这一过程即可得到平衡BST。这一步的时间复杂度为O(n)（每个节点被访问一次），空间复杂度为O(log n)（递归栈深度，平衡树的高度为log n）。

通过以上两步，我们高效地将原BST转换为平衡BST，且新树与原树的节点值完全一致。