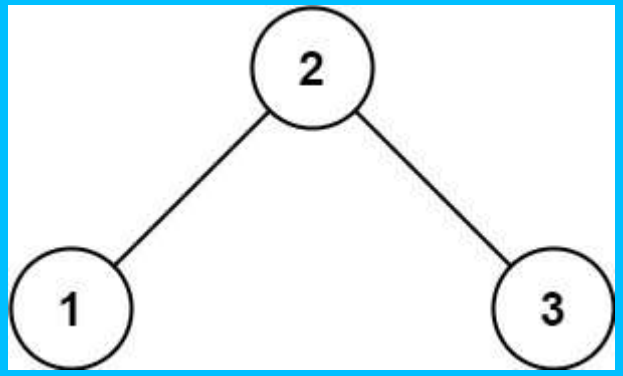
20、验证二叉搜索树（2024071，98题，中等，27min）

给你一个二叉树的根节点 root ， 判断其是否是一个有效的二叉搜索树。

有效 二叉搜索树定义如下：

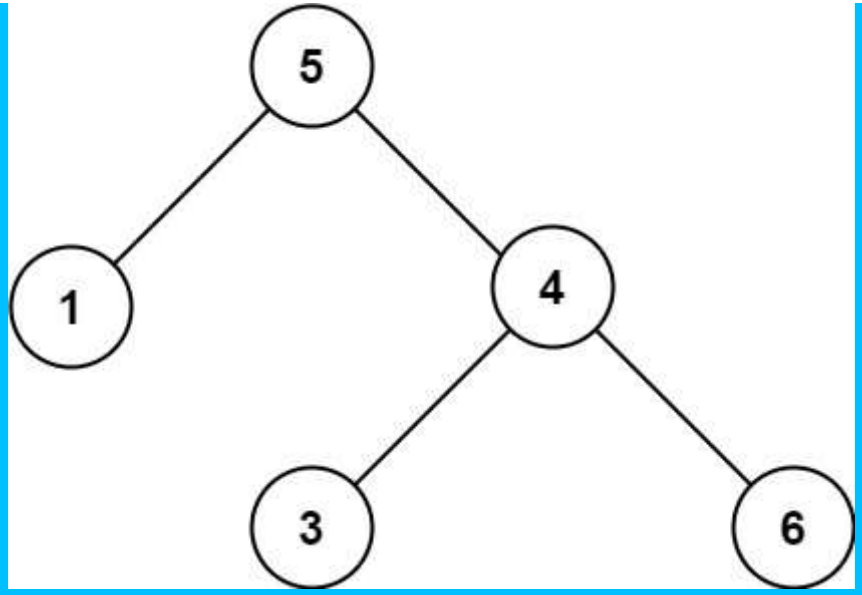
- 节点的左子树只包含 小于 当前节点的数。
- 节点的右子树只包含 大于 当前节点的数。
- 所有左子树和右子树自身必须也是二叉搜索树。

示例 1：



- 输入：root = [2,1,3]
- 输出：true

示例 2：



- 输入：root = [5,1,4,null,null,3,6]
- 输出：false
- 解释：根节点的值是 5 ，但是右子节点的值是 4 。

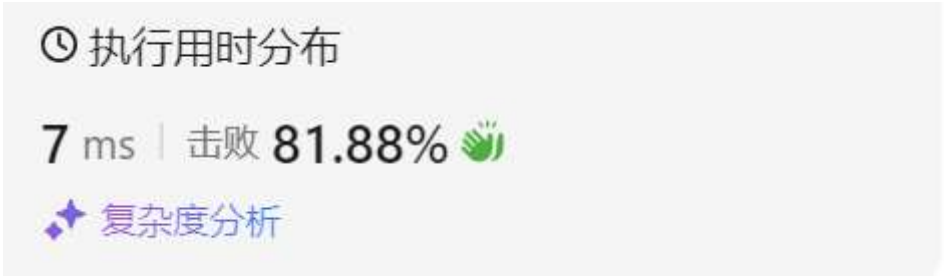
提示：

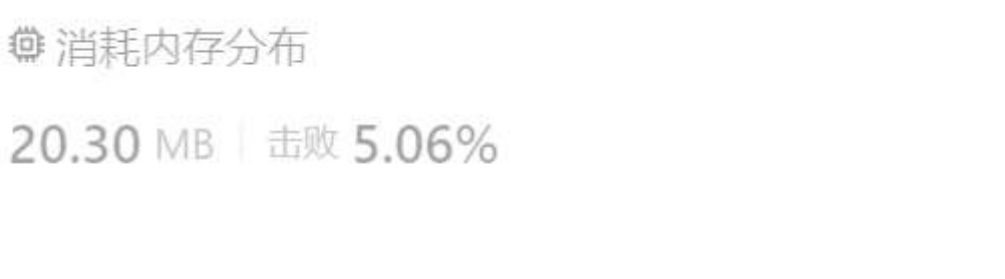
- 树中节点数目范围在[1, 104] 内
- -231 <= Node.val <= 231 - 1

自己答案（递归法，通过！27min）

```
/*
先得到中序遍历结果，再遍历判断是否为升序
*/
 class Solution {
public:
```

```
    void checkBST(TreeNode* cur_node, vector<int>& path){
        if(cur_node == nullptr) return;


        checkBST(cur_node->left, path);
        path.push_back(cur_node->val);
        checkBST(cur_node->right, path);
    }

    bool isValidBST(TreeNode* root) {
        vector<int> vec;

        checkBST(root, vec);

        for(int i = 1; i < vec.size(); i++){
            if(vec[i] <= vec[i-1]){
                return false;
            }
        }
        return true;
    }
};
```





仿答案（在递归中检查是否为二叉搜索树V1）

```cpp
/*
递归方法，在递归中检查是否为二叉搜索树
采用中序遍历，递归函数返回当前节点的左右子树的返回的bool值
额外的，需要记录当前的最大值cur_max_val，因为中序遍历二叉搜索树，
    cur_max_val应该是升序的，若遇到小于等于，说明不是有效的二叉搜索树！返回false
递归函数返回左右子树的与结果；所以有一个子树为false，最终结果为false！
*/
 class Solution {
public:
    long long cur_max_val = LONG_MIN;  // 查一下还有哪些类似的宏定义！！！
    bool isValidBST(TreeNode* root) {
        if(root == nullptr) return true;

        bool left = isValidBST(root->left);
        if(cur_max_val < root->val){
            cur_max_val = root->val;
        }else{
            return false;
        }
        bool right = isValidBST(root->right);

        return left && right;
    }
};
```

执行用时分布

14 ms | 击败 20.46%

复杂度分析

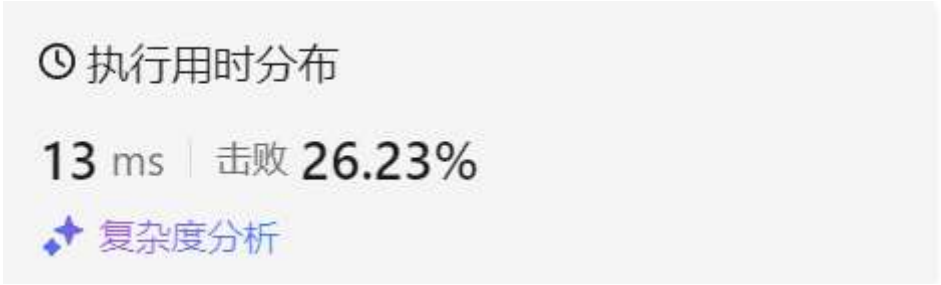
消耗内存分布

20.00 MB | 击败 10.31%

仿答案（在递归中检查是否为二叉搜索树V2）

```
/*
递归方法，在递归中检查是否为二叉搜索树
不使用最小的long long作为最初的比较值，而是用一个变量记录第一个节点！！！
采用中序遍历，递归函数返回当前节点的左右子树的返回的bool值
额外的，需要记录当前的最大值cur_max_val对应的节点TreeNode* pre，因为中序遍历二叉搜索树，
    pre的val应该是升序的，若遇到小于等于，说明不是有效的二叉搜索树！返回false
递归函数返回左右子树的与结果；所以有一个子树为false，最终结果为false！
*/
 class Solution {
public:
    // long long cur_max_val = LONG_MIN;
    TreeNode* pre = nullptr;

    bool isValidBST(TreeNode* root) {
        if(root == nullptr) return true;

        bool left = isValidBST(root->left);
        if(pre == nullptr){
            pre = root;
        }else if(pre->val < root->val){
            pre = root;
        }else{
            return false;
        }
        bool right = isValidBST(root->right);

        return left && right;
    }
};

/*随想录写法！！！
由于中序遍历，所以必然是按照从小到大的顺序遍历的
    若中间相邻的val不是递增的，则说明不是有效的二叉搜索树。
    所以可以每遍历一个节点，更新依次pre即可！！！
```

```
*/
class Solution {
public:
    TreeNode* pre = NULL; // 用来记录前一个节点
    bool isValidBST(TreeNode* root) {
        if (root == NULL) return true;
        bool left = isValidBST(root->left);

        if (pre != NULL && pre->val >= root->val) return false;
        pre = root; // 记录前一个节点

        bool right = isValidBST(root->right);
        return left && right;
    }
};
```



仿答案（迭代法）

```
/*
迭代方法
使用栈记录中序遍历入栈的节点
先入栈节点，有左节点则继续入栈，直到找到最左节点
    出栈最左节点，也是中节点；判断是否大于上个节点，不是返回false
    处理当前节点cur_node后，可能其有右节点，所以将当前节点指向右节点cur_node = cur_node->right!
        此时的右节点在下一轮被看做根节点，重复找最左节点的步骤；
```

```
        由此实现了左中右的遍历顺序！！！
*/
 class Solution {
public:
    stack<TreeNode*> stk;

    bool isValidBST(TreeNode* root) {

        TreeNode* cur_node = root;
        TreeNode* pre = nullptr;
        // stk.push(cur_node);

        while(cur_node != nullptr || !stk.empty()){
            if(cur_node != nullptr){   // 连续将节点入栈，有左节点，则继续入栈。达到连续将左节点入栈效果
                // stk.push(cur_node->left);
                stk.push(cur_node);       // 左
                cur_node = cur_node->left;
            }else{   // 找到最左节点
                cur_node = stk.top();     // 中（其实也是当前的最左节点）
                stk.pop();

                if(pre != nullptr && pre->val >= cur_node->val) return false;
                pre = cur_node;

                cur_node = cur_node->right;  // 右（前面处理了左中节点，若其有右节点，现在则处理，但其实也是现将其看做根节点，然后找其最左节点！！！）
            }
        }
        return true;
    }
};
```

执行用时分布

11 ms | 击败 43.57%

复杂度分析

消耗内存分布

20.00 MB | 击败 9.89%