验证二叉搜索树

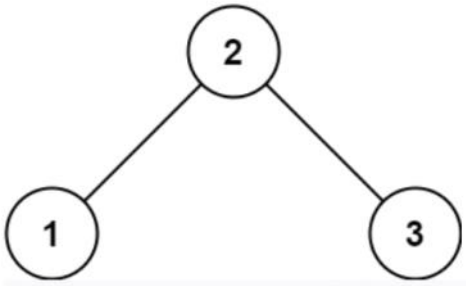
2022年8月18日 9:04

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

首先仔细回想一下关于BST的定义：

1. 左子树只包含小于当前节点的数
2. 右子树只包含大于当前节点的数
3. 所有左子树和右子树自身必须是二叉搜索树

```
class Solution {
private:
    int min;
    int max;
    void get_min(TreeNode* root){
        if(root->val<min)
            min=root->val;
        if(root->left) get_min(root->left);
        if(root->right) get_min(root->right);
    }
    void get_max(TreeNode* root){
        if(root->val>max)
            max=root->val;
        if(root->left) get_max(root->left);
        if(root->right) get_max(root->right);
    }
```

```
public:
    bool isValidBST(TreeNode* root) {
        if(!root){
            return false;
        }
        bool flag1=true,flag2=true,flag3=true,flag4=true;
        if(root->right){
            min=root->right->val;
            get_min(root->right);
            if(min<=root->val)
                flag1=false;
            flag3=isValidBST(root->right);
        }
        if(root->left){
            max=root->left->val;
            get_max(root->left);
            if(max>=root->val)
                flag2=false;
            flag4=isValidBST(root->left);
        }
        return flag1&&flag2&&flag3&&flag4;


    }
};
```

首先记录下自己写的烂代码和烂思路：

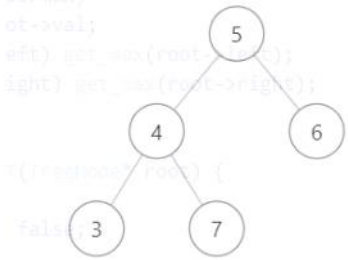
第一次提交时候的思路是通过递归

那么对问题进行拆解，每次递归中需要做什么？

满足左孩子小于根节点，右孩子大于根节点

然后继续递归即可

但实质上，这是有很大问题的！



这只是保证了左孩子小于根节点，但并为保证左子树全都小于根节点

一切问题出在这里

因此，需要更正为：左子树小于根节点，右子树大于根节点（需要怎么做会在后续给出）

第二次提交时候的思路实质上是意识到了上述问题，但没想出来递归要怎么写...

然后我就转化为，**左子树中的最大值小于根节点，右子树的最小值大于根节点**

（其实还有点小聪明...我指在走歪路这件事情上）

然后就写了一堆烂代码，我觉得边界判断（传入root为空时应该怎么办）上还可以进行优化！先去试试...（缩减变化不大，过于复杂，这里不再罗列）

题解：

解法一：

使用递归，每次递归中需满足**左子树小于根节点，右子树大于根节点**的条件

建立一个辅助函数，传入根节点与区间范围【lower,upper】

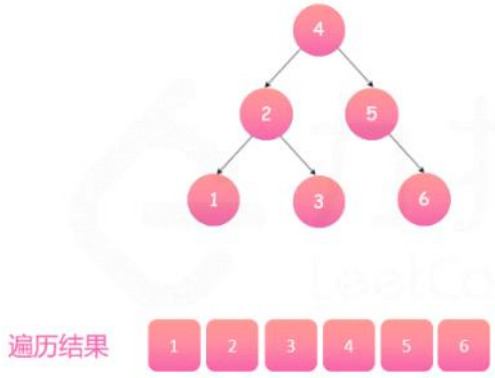
左子树需要满足区间范围为【lower，root->val】

右子树需要满足区间范围为【root->val，upper】

至于lower和upper则分别设为负无穷与正无穷即可（这儿使用LONG_MIN,LONG_MAX，因为本体卡边界值）

```
class Solution {
private:
    bool ValidFunction(TreeNode* root,long long lower,long long upper){
        if(!root){
            return true;
        }
        if( root->val <= lower || root->val>=upper ){
            return false;
        }
        return ValidFunction(root->left,lower,root->val)&&ValidFunction
(root->right,root->val,upper);
    }
public:
    bool isValidBST(TreeNode* root) {
        return ValidFunction(root,LONG_MIN,LONG_MAX);
    }
};
```

第二种思路是通过中序遍历



可以知道，若BTS进行中序遍历，那么输出结果一定满足严格递增的

我们可以通过递归或是迭代的方式完成中序遍历并进行比较

递归法：

为了避免根节点为边界值，采用prev指针指向前一个节点，避免了边界值的判断

```
class Solution {
private:
    TreeNode* prev=nullptr;
public:
    bool isValidBST(TreeNode* root) {
        if(!root)
            return true;
        bool left=isValidBST(root->left);
        if(prev&&prev->val>=root->val){
            return false;
        }
        prev=root;
        bool right=isValidBST(root->right);
        return left&&right;
    }
};
```

迭代法：

为了避免根节点为边界值，采用flag标识记录是否为第一次判断节点，避免边界值的判断

```cpp
class Solution {
private:
    bool flag=false;
    int prev;
public:
    bool isValidBST(TreeNode* root) {
        stack<TreeNode* >s;
        while(!s.empty()||root!=nullptr){
            while(root){
                s.push(root);
                root=root->left;
            }
            root=s.top();
            s.pop();
            if(flag&&prev>=root->val){
                return false;
            }
            flag=true;
            prev=root->val;
            root=root->right;
        }
        return true;
    }
};
```