

二叉搜索树

2022年8月17日 8:53

二叉搜索树(BST, Binary Search Tree), 也称为二叉排序树或二叉查找树。

二叉搜索树：一颗二叉树，可以为空，若不为空，满足以下性质：

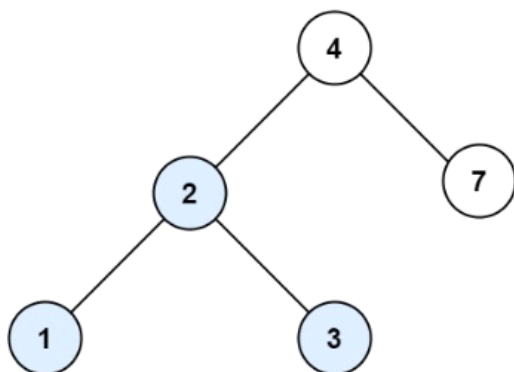
1. 非空左子树的所有键值小于其根节点的键值。
2. 非空右子树的所有键值大于其根节点的键值。
3. 左、右子树都是二叉搜索树。

一. 二叉搜索树中的搜索

给定二叉搜索树 (BST) 的根节点 `root` 和一个整数值 `val`。

你需要在 BST 中找到节点值等于 `val` 的节点。返回以该节点为根的子树。如果节点不存在，则返回 `null`。

示例 1:



输入: `root = [4,2,7,1,3]`, `val = 2`

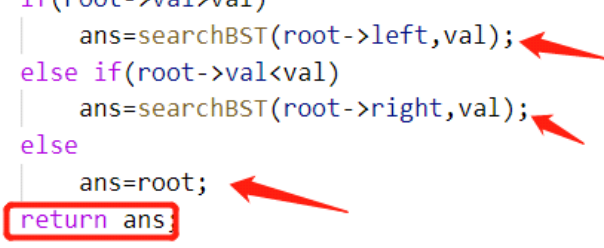
输出: `[2,1,3]`

递归

```

class Solution {
public:
    TreeNode* searchBST(TreeNode* root, int val) {
        if(!root)
            return nullptr;
        TreeNode* ans;
        if(root->val>val)
            ans=searchBST(root->left,val);
        else if(root->val<val)
            ans=searchBST(root->right,val);
        else
            ans=root;
        return ans;
    }
};

```




(救命这样写好蠢...，增加了存储 虽然也是常量空间，但是费代码 值得肯定的是这样比较清晰明了，嗯对！)

改进后：

```

class Solution {
public:
    TreeNode* searchBST(TreeNode* root, int val) {
        if(!root)
            return nullptr;
        if(root->val==val)
            return root;
        else
            return searchBST(root->val>val ? root->left : root->right,val);
    }
};

```



迭代

```

class Solution {
public:
    TreeNode* searchBST(TreeNode* root, int val) {
        while(root){
            if(root->val==val)
                return root;
            root=(root->val>val) ? root->left:root->right;
        }
        return nullptr;
    }
};

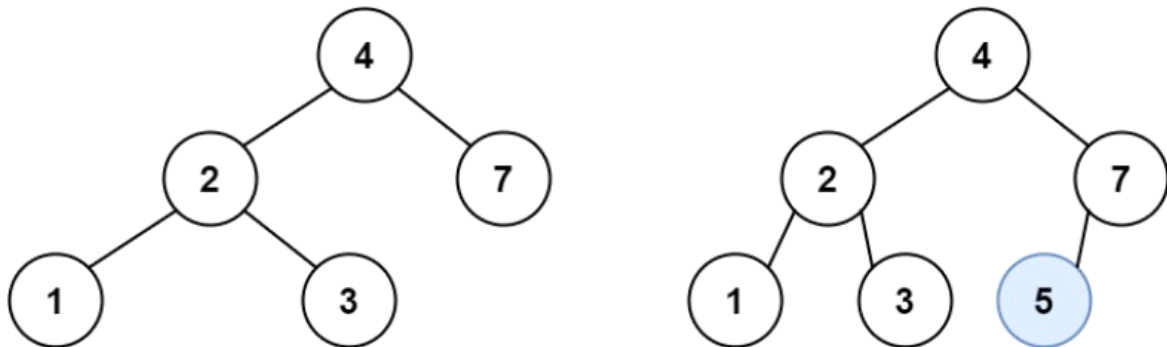
```

二. 二叉搜索树中的插入

给定二叉搜索树（BST）的根节点 `root` 和要插入树中的值 `value`，将值插入二叉搜索树。返回插入后二叉搜索树的根节点。输入数据 **保证**，新值和原始二叉搜索树中的任意节点值都不同。

注意，可能存在多种有效的插入方式，只要树在插入后仍保持为二叉搜索树即可。你可以返回 **任意有效的结果**。

示例 1:



输入: `root = [4,2,7,1,3]`, `val = 5`

输出: `[4,2,7,1,3,5]`

解释: 另一个满足题目要求可以通过的树是:

自己写的，代码较为复杂，看起来思路并不清晰

```
class Solution {
public:
    TreeNode* insertIntoBST(TreeNode* root, int val) {
        if(!root)
            return new TreeNode(val);
        TreeNode* now=root;
        TreeNode* next=now->val>val?now->left:now->right;
        while(next){
            now=next;
            next=now->val>val?now->left:now->right;
        }
        if(now->val>val)
            now->left=new TreeNode(val);
        else
            now->right=new TreeNode(val);
        return root;
    }
};
```

主要掌握迭代法

```

class Solution {
public:
    TreeNode* insertIntoBST(TreeNode* root, int val) {
        if(!root)
            return new TreeNode(val);
        if(root->val>val)
            root->left=insertIntoBST(root->left,val);
        else
            root->right=insertIntoBST(root->right,val);
        return root;
    }
};

```

思路极其清晰，在递归中进行建树

噢！实际上我的方法属于迭代，有更优化的写法

```

class Solution {
public:
    TreeNode* insertIntoBST(TreeNode* root, int val) {
        if(!root)
            return new TreeNode(val);
        TreeNode* now=root;
        TreeNode* par=nullptr;
        while(now){
            par=now;
            now=now->val>val ? now->left:now->right;
        }
        TreeNode* node=new TreeNode(val);
        if(par->val>val)
            par->left=node;
        else
            par->right=node;
        return root;
    }
};

```

三. 二叉搜索树的最近公共祖先

我们需要区分几种会出现的情况

1. p, q均在当前根节点的左子树中
2. p, q均在当前根节点的右子树中
3. p, q分别在当前根节点的左、右子树中
4. p为当前根节点, q在左子树中

5. p为当前根节点, q在右子树中

那么除了第一、二种情况, 对应的答案是: 最近祖先在左子树中、最近祖先在右子树中

其他三种情况, 均返回根节点

因此可以将代码写成如下

```
class Solution {
public:
    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
        if(root->val>p->val&&root->val>q->val)
            return lowestCommonAncestor(root->left,p,q);
        if(root->val<p->val&&root->val<q->val)
            return lowestCommonAncestor(root->right,p,q);
        return root;
    }
};
```

写成如下形式更节省空间

```
class Solution {
public:
    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
        TreeNode* ancestor = root;
        while (true) {
            if (p->val < ancestor->val && q->val < ancestor->val) {
                ancestor = ancestor->left;
            }
            else if (p->val > ancestor->val && q->val > ancestor->val) {
                ancestor = ancestor->right;
            }
            else {
                break;
            }
        }
        return ancestor;
    }
};
```