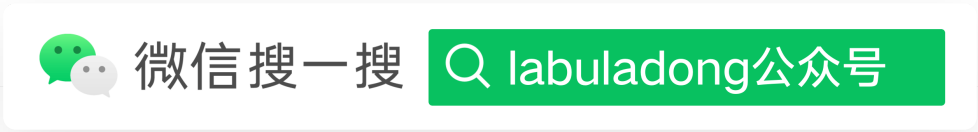
东哥带你刷二叉搜索树（基操篇）



通知：数据结构精品课持续更新中，详情见这里。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

牛客	LeetCode	力扣	难度
-	98. Validate Binary Search Tree	98. 验证二叉搜索树	🟠
-	450. Delete Node in a BST	450. 删除二叉搜索树中的节点	🟠
-	700. Search in a Binary Search Tree	700. 二叉搜索树中的搜索	🟢
-	701. Insert into a Binary Search Tree	701. 二叉搜索树中的插入操作	🟠

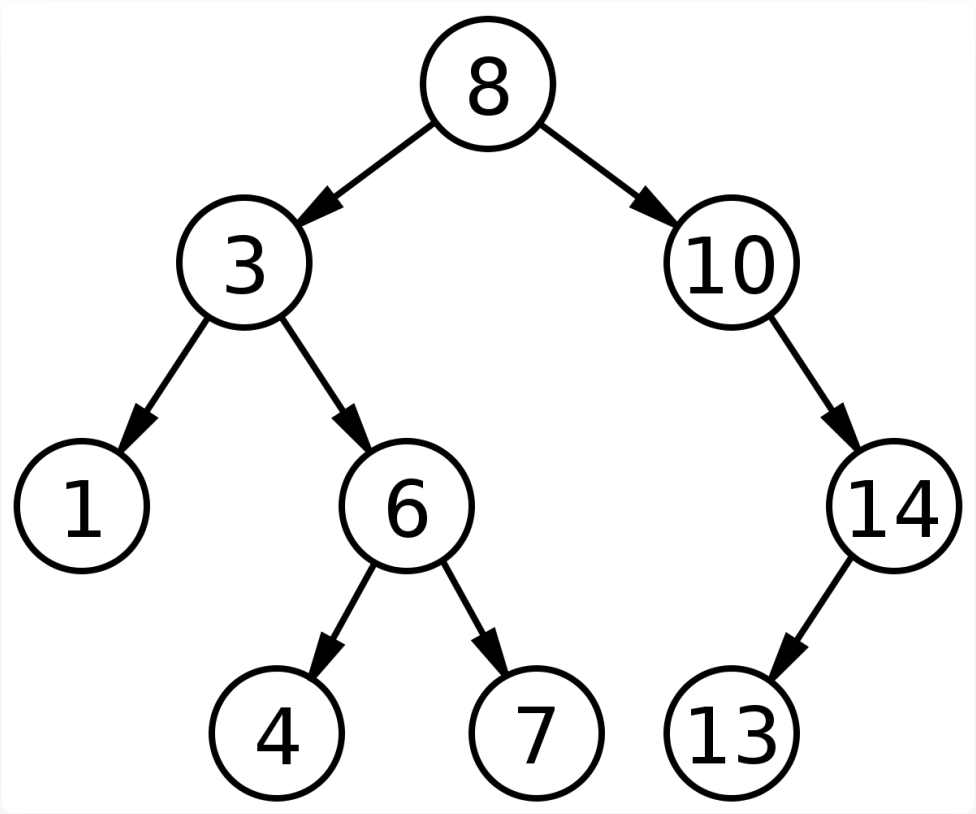
———

PS：刷题插件 集成了手把手刷二叉树功能，按照公式和套路讲解了 150 道二叉树题目，可手把手带你刷完二叉树分类的题目，迅速掌握递归思维。

我们前文 东哥带你刷二叉搜索树（特性篇） 介绍了 BST 的基本特性，还利用二叉搜索树「中序遍历有序」的特性来解决了几道题目，本文来实现 BST 的基础操作：判断 BST 的合法性、增、删、查。其中「删」和「判断合法性」略微复杂。

BST 的基础操作主要依赖「左小右大」的特性，可以在二叉树中做类似二分搜索的操作，寻找一

个元素的效率很高。比如下面这就是一棵合法的二叉树：



对于 BST 相关的问题，你可能会经常看到类似下面这样的代码逻辑：

```
void BST(TreeNode root, int target) {
    if (root.val == target)
        // 找到目标，做点什么
    if (root.val < target)
        BST(root.right, target);
    if (root.val > target)
        BST(root.left, target);
}
```

这个代码框架其实和二叉树的遍历框架差不多，无非就是利用了 BST 左小右大的特性而已。接下来看下 BST 这种结构的基础操作是如何实现的。

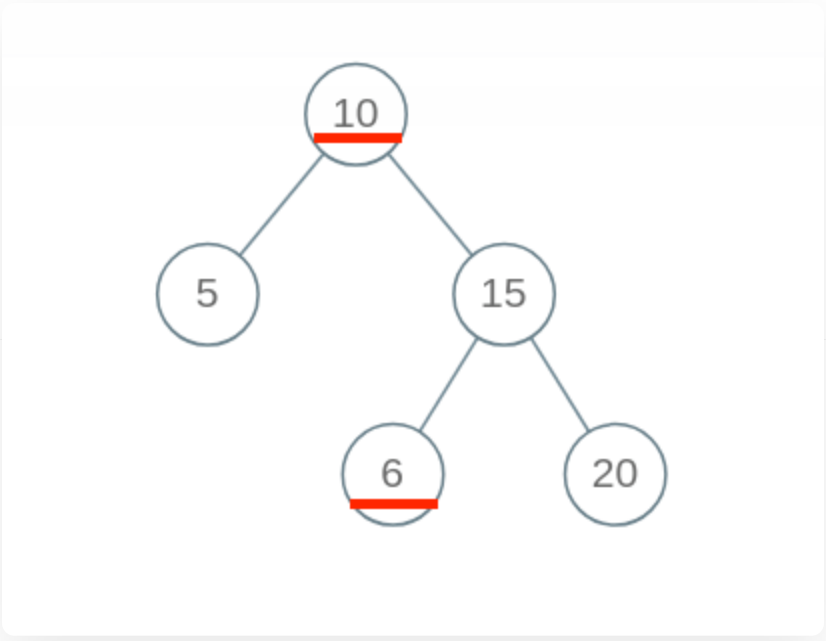
一、判断 BST 的合法性

力扣第 98 题「验证二叉搜索树」就是让你判断输入的 BST 是否合法。注意，这里是有坑的哦，按照 BST 左小右大的特性，每个节点想要判断自己是否是合法的 BST 节点，要做的事不就是比较自己和左右孩子吗？感觉应该这样写代码：

```
boolean isValidBST(TreeNode root) {
    if (root == null) return true;
    // root 的左边应该更小
    if (root.left != null && root.left.val >= root.val)
        return false;
    // root 的右边应该更大
    if (root.right != null && root.right.val <= root.val)
        return false;

    return isValidBST(root.left)
        && isValidBST(root.right);
}
```

但是这个算法出现了错误，BST 的每个节点应该要小于右边子树的**所有**节点，下面这个二叉树显然不是 BST，因为节点 10 的右子树中有一个节点 6，但是我们的算法会把它判定为合法 BST：



出现问题的原因在于，对于每一个节点 `root`，代码值检查了它的左右孩子节点是否符合左小右大的原则；但是根据 BST 的定义，`root` 的整个左子树都要小于 `root.val`，整个右子树都要大于

`root.val`。

问题是，对于某一个节点 `root`，他只能管得了自己的左右子节点，怎么把 `root` 的约束传递给左右子树呢？请看正确的代码：

```
boolean isValidBST(TreeNode root) {
    return isValidBST(root, null, null);
}

/* 限定以 root 为根的子树节点必须满足 max.val > root.val > min.val */
boolean isValidBST(TreeNode root, TreeNode min, TreeNode max) {
    // base case
    if (root == null) return true;
    // 若 root.val 不符合 max 和 min 的限制，说明不是合法 BST
    if (min != null && root.val <= min.val) return false;
    if (max != null && root.val >= max.val) return false;
    // 限定左子树的最大值是 root.val，右子树的最小值是 root.val
    return isValidBST(root.left, min, root)
        && isValidBST(root.right, root, max);
}
```

我们通过使用辅助函数，增加函数参数列表，在参数中携带额外信息，将这种约束传递给子树的所有节点，这也是二叉树算法的一个小技巧吧。

在 BST 中搜索元素

力扣第 700 题「二叉搜索树中的搜索」就是让你在 BST 中搜索值为 `target` 的节点，函数签名如下：

```
TreeNode searchBST(TreeNode root, int target);
```

如果是在一棵普通的二叉树中寻找，可以这样写代码：

```
TreeNode searchBST(TreeNode root, int target);
    if (root == null) return null;
    if (root.val == target) return root;
```

```
        // 当前节点没找到就递归地去左右子树寻找
        TreeNode left = searchBST(root.left, target);
        TreeNode right = searchBST(root.right, target);

        return left != null ? left : right;
    }
```

这样写完全正确，但这段代码相当于穷举了所有节点，适用于所有二叉树。那么应该如何充分利用 BST 的特殊性，把「左小右大」的特性用上？

很简单，其实不需要递归地搜索两边，类似二分查找思想，根据 `target` 和 `root.val` 的大小比较，就能排除一边。我们把上面的思路稍稍改动：

```
    TreeNode searchBST(TreeNode root, int target) {
        if (root == null) {
            return null;
        }
        // 去左子树搜索
        if (root.val > target) {
            return searchBST(root.left, target);
        }
        // 去右子树搜索
        if (root.val < target) {
            return searchBST(root.right, target);
        }
        return root;
    }
```

在 BST 中插入一个数

对数据结构的操作无非遍历 + 访问，遍历就是「找」，访问就是「改」。具体到这个问题，插入一个数，就是先找到插入位置，然后进行插入操作。

上一个问题，我们总结了 BST 中的遍历框架，就是「找」的问题。直接套框架，加上「改」的操作即可。**一旦涉及「改」，就类似二叉树的构造问题，函数要返回 `TreeNode` 类型，并且要对递归调用的返回值进行接收**。

```
    TreeNode insertIntoBST(TreeNode root, int val) {
```

```
    // 找到空位置插入新节点
    if (root == null) return new TreeNode(val);
    // if (root.val == val)
    //     BST 中一般不会插入已存在元素
    if (root.val < val)
        root.right = insertIntoBST(root.right, val);
    if (root.val > val)
        root.left = insertIntoBST(root.left, val);
    return root;
}
```

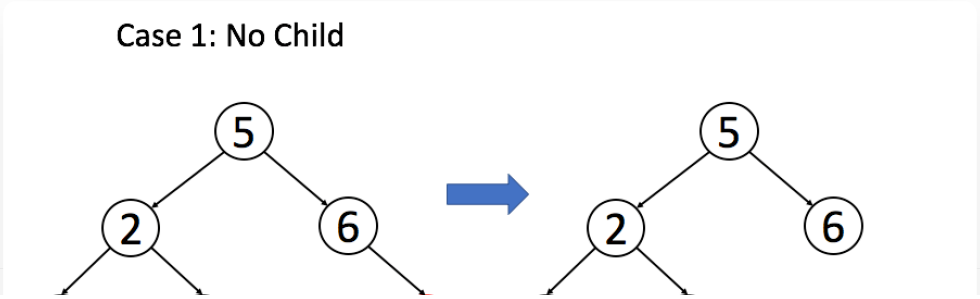
三、在 BST 中删除一个数

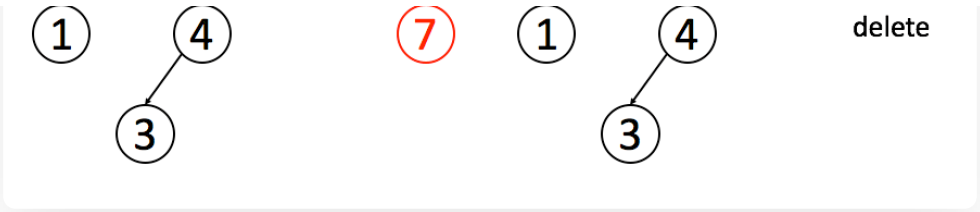
这个问题稍微复杂，跟插入操作类似，先「找」再「改」，先把框架写出来再说：

```
TreeNode deleteNode(TreeNode root, int key) {
    if (root.val == key) {
        // 找到啦，进行删除
    } else if (root.val > key) {
        // 去左子树找
        root.left = deleteNode(root.left, key);
    } else if (root.val < key) {
        // 去右子树找
        root.right = deleteNode(root.right, key);
    }
    return root;
}
```

找到目标节点了，比方说是节点 `A`，如何删除这个节点，这是难点。因为删除节点的同时不能破坏 BST 的性质。有三种情况，用图片来说明。

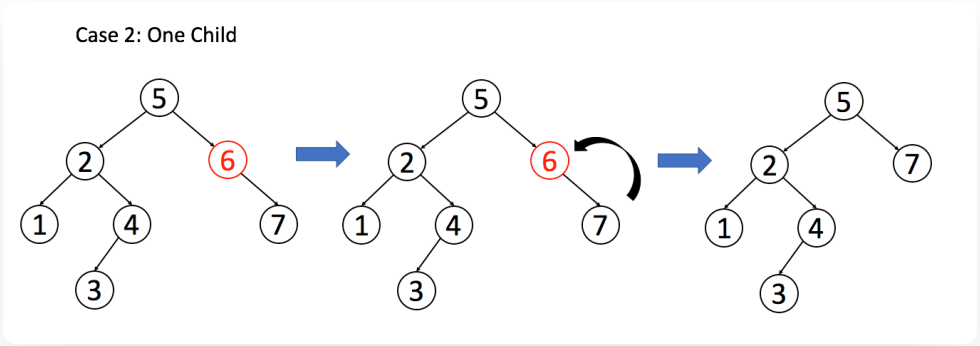
情况 1：`A` 恰好是末端节点，两个子节点都为空，那么它可以当场去世了。





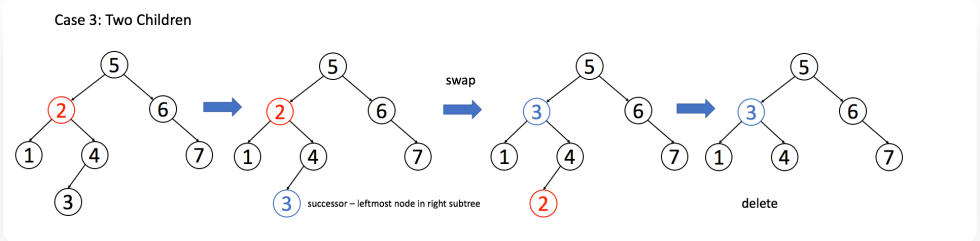

```
if (root.left == null && root.right == null)
    return null;
```

情况 2：A 只有一个非空子节点，那么它要让这个孩子接替自己的位置。




```
// 排除了情况 1 之后
if (root.left == null) return root.right;
if (root.right == null) return root.left;
```

情况 3：A 有两个子节点，麻烦了，为了不破坏 BST 的性质，A 必须找到左子树中最大的那个节点，或者右子树中最小的那个节点来接替自己。我们以第二种方式讲解。



```
if (root.left != null && root.right != null) {
    // 找到右子树的最小节点
    TreeNode minNode = getMin(root.right);
    // 把 root 改成 minNode
    root.val = minNode.val;
    // 转而去删除 minNode
    root.right = deleteNode(root.right, minNode.val);
}
```

三种情况分析完毕，填入框架，简化一下代码：

```
TreeNode deleteNode(TreeNode root, int key) {
    if (root == null) return null;
    if (root.val == key) {
        // 这两个 if 把情况 1 和 2 都正确处理了
        if (root.left == null) return root.right;
        if (root.right == null) return root.left;
        // 处理情况 3
        // 获得右子树最小的节点
        TreeNode minNode = getMin(root.right);
        // 删除右子树最小的节点
        root.right = deleteNode(root.right, minNode.val);
        // 用右子树最小的节点替换 root 节点
        minNode.left = root.left;
        minNode.right = root.right;
        root = minNode;
    } else if (root.val > key) {
        root.left = deleteNode(root.left, key);
    } else if (root.val < key) {
        root.right = deleteNode(root.right, key);
    }
    return root;
}

TreeNode getMin(TreeNode node) {
    // BST 最左边的就是最小的
    while (node.left != null) node = node.left;
    return node;
}
```

这样，删除操作就完成了。注意一下，上述代码在处理情况 3 时通过一系列略微复杂的链表操作交换 `root` 和 `minNode` 两个节点：

```
// 处理情况 3
// 获得右子树最小的节点
TreeNode minNode = getMin(root.right);
// 删除右子树最小的节点
root.right = deleteNode(root.right, minNode.val);
// 用右子树最小的节点替换 root 节点
minNode.left = root.left;
minNode.right = root.right;
root = minNode;
```

有的读者可能会疑惑，替换 `root` 节点为什么这么麻烦，直接改 `val` 字段不就行了？看起来还更简洁易懂：

```
// 处理情况 3
// 获得右子树最小的节点
TreeNode minNode = getMin(root.right);
// 删除右子树最小的节点
root.right = deleteNode(root.right, minNode.val);
// 用右子树最小的节点替换 root 节点
root.val = minNode.val;
```

仅对于这道算法题来说是可以的，但这样操作并不完美，我们一般不会通过修改节点内部的值来交换节点。因为在实际应用中，BST 节点内部的数据域是用户自定义的，可以非常复杂，而 BST 作为数据结构（一个工具人），其操作应该和内部存储的数据域解耦，所以我们更倾向于使用指针操作来交换节点，根本没必要关心内部数据。

最后总结一下吧，通过这篇文章，我们总结出了如下几个技巧：

1、如果当前节点会对下面的子节点有整体影响，可以通过辅助函数增长参数列表，借助参数传递信息。

2、在二叉树递归框架之上，扩展出一套 BST 代码框架：

```
void BST(TreeNode root, int target) {
    if (root.val == target)
        // 找到目标，做点什么
    if (root.val < target)
        BST(root.right, target);
    if (root.val > target)
        BST(root.left, target);
}
```

3、根据代码框架掌握了 BST 的增删查改操作。

▸ **引用本文的文章**

- - - - - - - - - - - - - -

《labuladong 的算法小抄》已经出版，关注公众号查看详情；后台回复关键词「进群」可加入算法群；回复「PDF」可获取精华文章 PDF：

共同维护高质量学习环境，评论礼仪见这里，违者直接拉黑不解释

29 Comments - *powered by utteranc.es*

biaoboss commented on Nov 20, 2021

> 注意一下，这个删除操作并不完美，因为我们一般不会通过 root.val = minNode.val 修改节点内部的值来交换节点，而是通过一系列略微复杂的链表操作交换 root 和 minNode 两个节点。
> 因为具体应用中，val 域可能会是一个复杂的数据结构，修改起来非常麻烦；而链表操作无非改一改指针，而不会去碰内部数据。

这里我觉得其实问题的不大， 如果是个复杂的数据结构，这里里修改 `root.val = minNode.val` 也不过是修改一个地址而已，是吧