# 题目

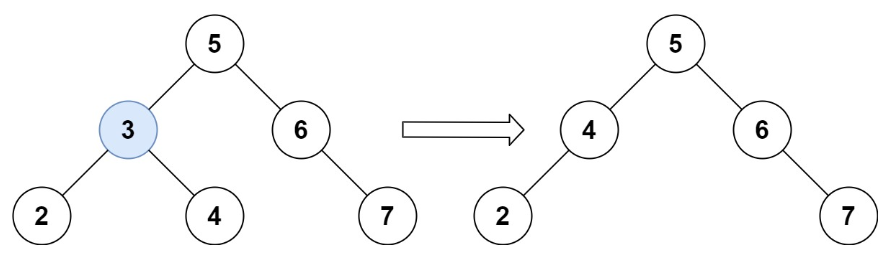
给定一个二叉搜索树的根节点 root 和一个值 key，删除二叉搜索树中的 key 对应的节点，并保证二叉搜索树的性质不变。返回二叉搜索树（有可能被更新）的根节点的引用。

一般来说，删除节点可分为两个步骤：

1、首先找到需要删除的节点；

2、如果找到了，删除它。

示例 1:



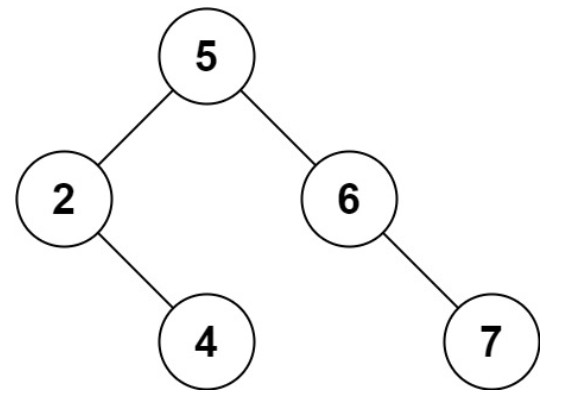
输入：root = [5,3,6,2,4,null,7], key = 3

输出：[5,4,6,2,null,null,7]

解释：给定需要删除的节点值是 3，所以我们首先找到 3 这个节点，然后删除它。

一个正确的答案是 [5,4,6,2,null,null,7], 如下图所示。

另一个正确答案是 [5,2,6,null,4,null,7]。



示例 2:

输入: root = [5,3,6,2,4,null,7], key = 0

输出: [5,3,6,2,4,null,7]

解释: 二叉树不包含值为 0 的节点

示例 3:

输入: root = [], key = 0

输出: []

提示:

节点数的范围 [0, 10^4].

-10^5 <= Node.val <= 10^5

节点值唯一

root 是合法的二叉搜索树

-10^5 <= key <= 10^5

进阶：要求算法时间复杂度为O(h)，h为树的高度。

# 分析

## 方法一：递归法、

思路：

删除二叉搜索树中的节点涉及到几种情况：

1、待删除节点没有子节点：直接删除该节点即可。

2、待删除节点有一个子节点：将该节点的父节点指向该节点的子节点。

3、待删除节点有两个子节点：找到待删除节点的后继节点（右子树的最小节点），将其值赋给待删除节点，然后递归删除后继节点。

基于这些情况，我们可以实现以下算法：

1、如果根节点为空，直接返回。

2、如果当前节点的值等于待删除节点的值：

- 如果当前节点没有左子节点，则直接返回右子节点，即删除当前节点。

- 如果当前节点没有右子节点，则直接返回左子节点，即删除当前节点。

- 如果当前节点有左右子节点，则找到其右子树的最小节点作为后继节点，将其值赋给当前节点，然后递归删除后继节点。

3、如果当前节点的值大于待删除节点的值，则在左子树中递归删除。

4、如果当前节点的值小于待删除节点的值，则在右子树中递归删除。

以下是一个可能的实现：

class Solution {

public:

TreeNode\* deleteNode(TreeNode\* root, int key) {

if (!root) return nullptr; // 根节点为空，直接返回

if (root->val == key) { // 当前节点是待删除节点

if (!root->left) return root->right; // 当前节点没有左子节点，返回右子节点

if (!root->right) return root->left; // 当前节点没有右子节点，返回左子节点

// 当前节点有左右子节点，找到右子树的最小节点作为后继节点

TreeNode\* successor = root->right;

while (successor->left) {

successor = successor->left;

}

root->val = successor->val; // 将后继节点的值赋给当前节点

root->right = deleteNode(root->right, successor->val); // 递归删除后继节点

} else if (root->val > key) { // 当前节点值大于待删除节点值，递归在左子树中删除

root->left = deleteNode(root->left, key);

} else { // 当前节点值小于待删除节点值，递归在右子树中删除

root->right = deleteNode(root->right, key);

}

return root;

}

};

这个算法中，我们递归地在左右子树中查找并删除目标节点，然后根据删除情况更新当前节点的左右子节点。

## 方法二：递归+模拟

思路：

删除二叉搜索树中的一个节点分为几种情况：

1、如果待删除节点没有子节点，直接删除即可。

2、如果待删除节点只有一个子节点，将其子节点替换为该节点即可。

3、如果待删除节点有两个子节点，可以选择将其左子树中的最大节点或右子树中的最小节点替换为该节点，然后再删除这个最大或最小节点。

具体实现如下：

class Solution {

public:

TreeNode\* deleteNode(TreeNode\* root, int key) {

if (!root) {

return nullptr;

}

if (key < root->val) {

root->left = deleteNode(root->left, key);

} else if (key > root->val) {

root->right = deleteNode(root->right, key);

} else {

// 当前节点就是要删除的节点

if (!root->left) {

// 没有左子节点，直接返回右子节点

return root->right;

} else if (!root->right) {

// 没有右子节点，直接返回左子节点

return root->left;

} else {

// 有左右子节点，选择右子树中的最小节点替换当前节点

TreeNode\* minNode = findMin(root->right);

root->val = minNode->val;

root->right = deleteNode(root->right, minNode->val);

}

}

return root;

}

TreeNode\* findMin(TreeNode\* node) {

while (node->left) {

node = node->left;

}

return node;

}

};

在这个实现中，我们首先判断待删除节点与当前节点的大小关系，然后递归地处理左子树或右子树。如果当前节点就是要删除的节点，我们分三种情况处理：没有左子节点、没有右子节点和有左右子节点。