98.验证二叉搜索树（中等）

1. 题目描述

   给定一个二叉树，判断其是否是一个有效的二叉搜索树。

   假设一个二叉搜索树具有如下特征：

   - 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:
       2
      / \
     1   3
   输出: true
   ```

   示例 2:

   ```
   输入:
       5
      / \
     1   4
        / \
       3   6
   输出: false
   解释: 输入为: [5,1,4,null,null,3,6]。
        根节点的值为 5 ，但是其右子节点值为 4 。
   ```

2. 简单实现

   中序遍历看是否升序

   ```
   class Solution {
   public:
       bool judge(TreeNode* root, long& pre){
           if(root->left && !judge(root->left, pre)) return false;
           if(pre >= root->val) return false;
           pre = root->val;
           if(root->right && !judge(root->right, pre)) return false;
           return true;
       }
       bool isValidBST(TreeNode* root) {
           if(!root) return true;
           long pre = long(INT_MIN) - 1;
           return judge(root, pre);
       }
   ```

```
};
```

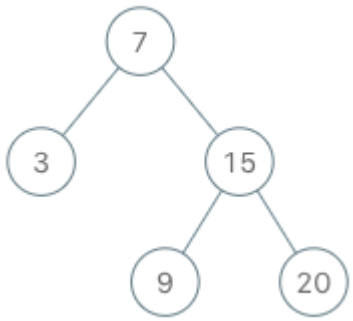
173.二叉搜索树迭代器（中等）

1. 题目描述

   实现一个二叉搜索树迭代器。你将使用二叉搜索树的根节点初始化迭代器。

   调用 `next()` 将返回二叉搜索树中的下一个最小的数。

   示例：


```
BSTIterator iterator = new BSTIterator(root);
iterator.next();    // 返回 3
iterator.next();    // 返回 7
iterator.hasNext(); // 返回 true
iterator.next();    // 返回 9
iterator.hasNext(); // 返回 true
iterator.next();    // 返回 15
iterator.hasNext(); // 返回 true
iterator.next();    // 返回 20
iterator.hasNext(); // 返回 false
```

   提示：

   - `next()` 和 `hasNext()` 操作的时间复杂度是 O(1)，并使用 O(*h*) 内存，其中 *h* 是树的高度。
   - 你可以假设 `next()` 调用总是有效的，也就是说，当调用 `next()` 时，BST 中至少存在一个下一个最小的数。

2. 简单实现

```
class BSTIterator {
public:
    vector<int> data;//中序遍历结果
    int idx;//当前迭代到的下标
    void build(TreeNode* root, vector<int>& data){//构建data数组
        if(root->left) build(root->left, data);
        data.push_back(root->val);
        if(root->right) build(root->right, data);
    }
    BSTIterator(TreeNode* root) {
        data.clear();
        if(root) build(root, data);
```

```
        idx = 0;
    }
    /** @return the next smallest number */
    int next() {
        return data[idx++];
    }
    /** @return whether we have a next smallest number */
    bool hasNext() {
        return idx < data.size();
    }
};
```

3. 最优解法

不必保存所有的data，可以用在用栈模拟递归的过程中完成next()

```
class BSTIterator {
public:
    stack<TreeNode*> s;
    BSTIterator(TreeNode* root) {
        while(root){
            s.push(root);
            root = root->left;
        }
    }
    /** @return the next smallest number */
    int next() {
        TreeNode* cur = s.top();
        s.pop();
        int ans = cur->val;
        if(cur->right){
            cur = cur->right;
            while(cur){
                s.push(cur);
                cur = cur->left;
            }
        }
        return ans;
    }
    /** @return whether we have a next smallest number */
    bool hasNext() {
        return !s.empty();
    }
};
```

700. 二叉搜索树中的搜索（简单）

1. 题目描述

给定二叉搜索树（BST）的根节点和一个值。 你需要在BST中找到节点值等于给定值的节点。 返回以该节点为根的子树。 如果节点不存在，则返回 NULL。

例如，

给定二叉搜索树：

```
        4
       / \
      2   7
     / \
    1   3

和值: 2
```

你应该返回如下子树:

```
      2
     / \
    1   3
```

在上述示例中，如果要找的值是 `5`，但因为没有节点值为 `5`，我们应该返回 `NULL`。

2. 简单实现

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

701.二叉搜索树中的插入操作（中等）

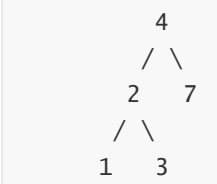
1. 题目描述

给定二叉搜索树（BST）的根节点和要插入树中的值，将值插入二叉搜索树。 返回插入后二叉搜索树的根节点。 保证原始二叉搜索树中不存在新值。

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。

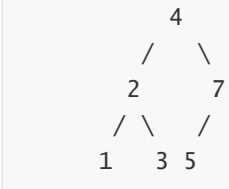
例如,

给定二叉搜索树：

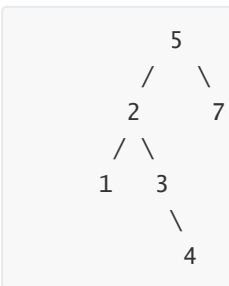


和 插入的值：5

你可以返回这个二叉搜索树：



或者这个树也是有效的：



2. 简单实现

```
class Solution {
public:
    TreeNode* insertIntoBST(TreeNode* root, int val) {
        if(!root) return new TreeNode(val);
        TreeNode* cur = root;
        while(1){
            if(cur->val < val){
                if((cur->right)) cur = cur->right;
                else{
                    cur->right = new TreeNode(val);
                    return root;
                }
            }
            else{
                if(cur->left) cur = cur->left;
                else{
                    cur->left = new TreeNode(val);
                    return root;
                }
            }
        }
```

```
            return NULL;
        }
    };
```

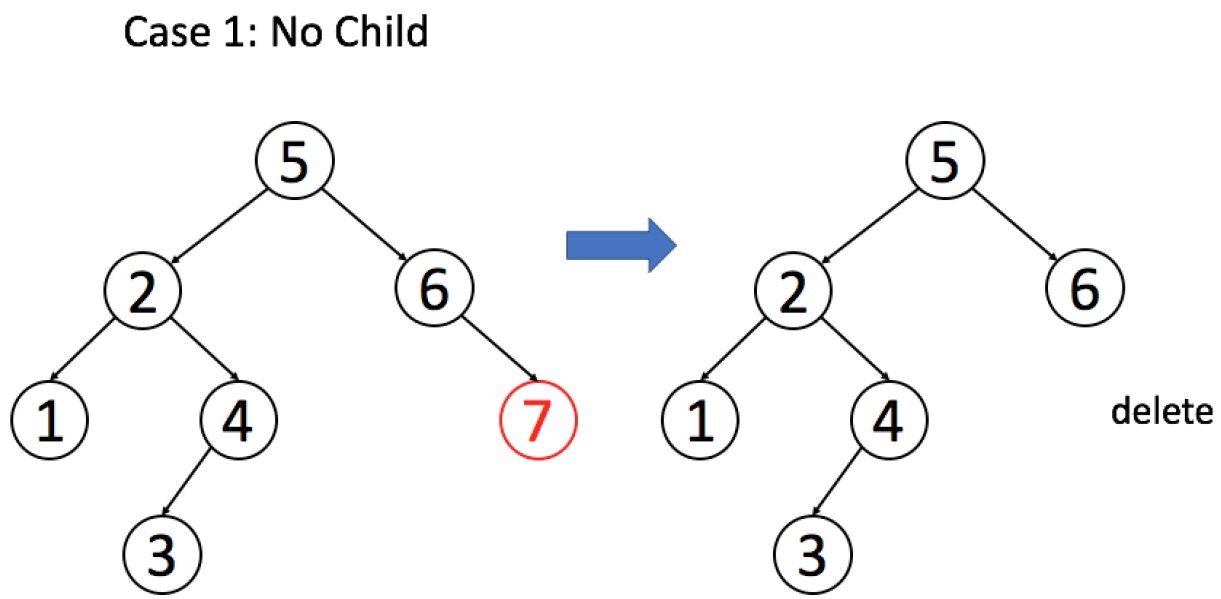
在二叉搜索树中实现删除操作

删除要比我们前面提到过的两种操作复杂许多。有许多不同的删除节点的方法，这篇文章中，我们只讨论一种使整体操作变化最小的方法。我们的方案是用一个合适的子节点来替换要删除的目标节点。根据其子节点的个数，我们需考虑以下三种情况：

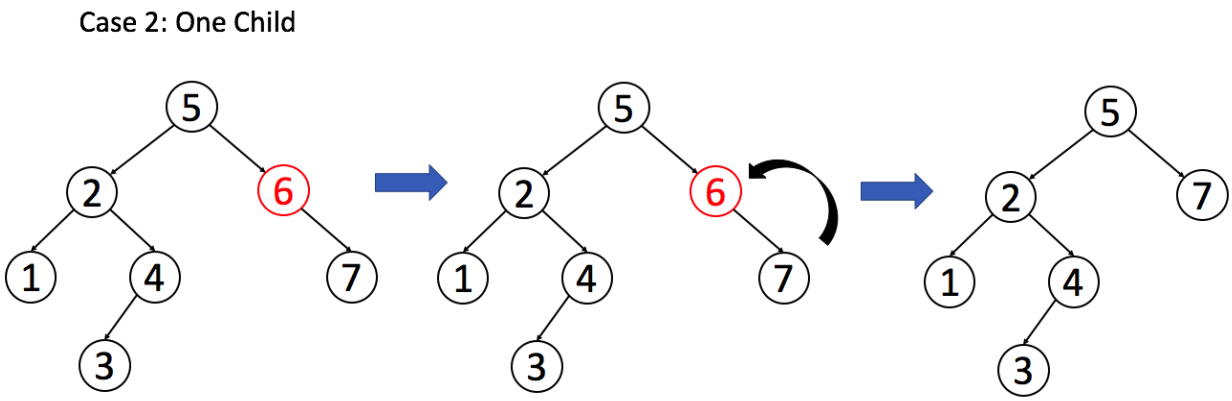
1. 如果目标节点***没有子节点***，我们可以直接移除该目标节点。
2. 如果目标节***只有一个子节点***，我们可以用其子节点作为替换。
3. 如果目标节点***有两个子节点***，我们需要用其中序后继节点或者前驱节点来替换，再删除该目标节点。

我们来看下面这几个例子，以帮助你理解删除操作的中心思想：

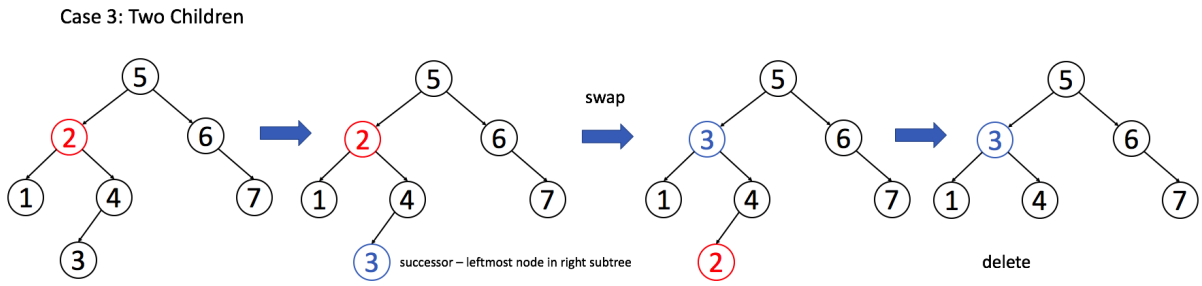
例 1：目标节点没有子节点



例 2：目标节只有一个子节点



例 3：目标节点有两个子节点



450.删除二叉搜索树中的节点（中等）

1. 题目描述

   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。

   说明： 要求算法时间复杂度为 O(h)，h 为树的高度。

   示例:

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

    5
   / \
  3   6
 / \   \
2   4   7

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。

    5
   / \
  4   6
 /     \
2       7

另一个正确答案是 [5,2,6,null,4,null,7]。

    5
   / \
  2   6
   \   \
    4   7
```

2. 简单实现

```
class Solution {
public:
    //搜索val所在节点，返回该节点的父节点和该节点
    vector<TreeNode*> searchBST(TreeNode* root, int val, TreeNode* pre) {
        if(!root) return {NULL, NULL};
        TreeNode* cur = root;
        while(cur){
            if(cur->val == val) return {pre,cur};
            pre = cur;
            if(cur->val < val) cur = cur->right;
            else cur = cur->left;

        }
        return  {NULL, NULL};
    }
    TreeNode* deleteNode(TreeNode* root, int key) {
        TreeNode* pre = NULL;
        TreeNode* node = NULL;
        vector<TreeNode*> ans = searchBST(root, key, pre);
        pre = ans[0];
        node = ans[1];
        if(!node) return root;
        if(!pre){//删除根节点
            if(!node->left && !node->right) return NULL;
            if(!node->left) return root->right;
            if(!node->right) return root->left;
        }
        else{
            if(!node->left && !node->right){

                if(pre->left == node) pre->left = NULL;
                else pre->right = NULL;
                return root;
            }
            if(!node->left){
                if(pre->left == node) pre->left = node->right;
                else pre->right = node->right;
                return root;
            }
            if(!node->right){
                if(pre->left == node) pre->left = node->left;
                else pre->right = node->left;
                return root;
            }
        }
        //左右子都存在，找到直接后继
        pre = node;
        TreeNode* cur = node->right;
        while(cur->left){
            pre = cur;
            cur = cur->left;
```

```
        }
        node->val = cur->val;
        //删除直接后继
        if(pre->left == cur) pre->left = deleteNode(cur, cur->val);
        else pre->right = deleteNode(cur, cur->val);
        return root;
    }
};
```

703. 数据流中的第K大元素]（简单）

1. 题目描述

   设计一个找到数据流中第K大元素的类（class）。注意是排序后的第K大元素，不是第K个不同的元素。

   你的 `KthLargest` 类需要一个同时接收整数 `k` 和整数数组 `nums` 的构造器，它包含数据流中的初始元素。每次调用 `KthLargest.add`，返回当前数据流中第K大的元素。

   示例:

```
int k = 3;
int[] arr = [4,5,8,2];
KthLargest kthLargest = new KthLargest(3, arr);
kthLargest.add(3);   // returns 4
kthLargest.add(5);   // returns 5
kthLargest.add(10);  // returns 5
kthLargest.add(9);   // returns 8
kthLargest.add(4);   // returns 8
```

   说明: 你可以假设 `nums` 的长度≥ `k-1` 且 `k` ≥ 1。

2. 简单实现

```
struct Node{
    int val;
    int cnt;
    Node* left;
    Node* right;
    Node(int x): val(x), cnt(1), left(NULL), right(NULL){}
};
class KthLargest {
public:
    int k;
    Node* root;//二叉搜索树
    int kth;//目前第k大元素
    KthLargest(int k, vector<int>& nums) {
        this->k = k;
        root = NULL;
        for(int i = 0; i < nums.size(); i++) kth = add(nums[i]);
    }
    int add(int val) {
        if(!root){//空树
            root = new Node(val);
```

```
            if(k == 1) kth = val;
            return kth;
        }
        else{
            //插入
            Node* cur= root;
            while(1){
                cur->cnt++;
                if(cur->val < val){
                    if(cur->right) cur = cur->right;
                    else{
                        cur->right = new Node(val);
                        break;
                    }
                }
                else{
                    if(cur->left) cur = cur->left;
                    else{
                        cur->left = new Node(val);
                        break;
                    }
                }
            }
            //找到kth
            if(root->cnt >= k){//只在节点总数大于等于k时寻找
                if(root->cnt > k && val <= kth) return kth;//无需更新
                else{
                    int curk = k;
                    cur = root;
                    while(1){
                        if(!cur->left){
                            if(curk == cur->cnt){//根节点就是所求
                                kth = cur->val;
                                return kth;
                            }
                            else//向右找
                                cur = cur->right;
                        }
                        else{
                            if(curk == (cur->cnt - cur->left->cnt)){//根节点就是所求
                                kth = cur->val;
                                return kth;
                            }
                            if(curk > (cur->cnt - cur->left->cnt)){//在左边找
                                if(cur->right) curk -= cur->right->cnt + 1;
                                else curk -= 1;
                                cur = cur->left;
                            }
                            else//在右边找
                                cur = cur->right;
                        }
                    }
                }
```

```
            }
            return kth;
        }
    }
};
```

3. 最优解法

   优先队列

```
struct Node{
    int val;
    int cnt;
    Node* left;
    Node* right;
    Node(int x): val(x), cnt(1), left(NULL), right(NULL){}
};

class KthLargest {
public:
    priority_queue<int, vector<int>, greater<int>> q;
    int k;
    KthLargest(int k, vector<int>& nums) {
        this->k = k;
        for(int i = 0; i < nums.size(); i++) add(nums[i]);
    }

    int add(int val) {
        cout << q.size() << endl;
        if(q.size() < k)
            q.push(val);
        else{
            if(val <=q.top()) return q.top();
            else{
                q.push(val);
                q.pop();
            }
        }
        return q.top();
    }
};
```

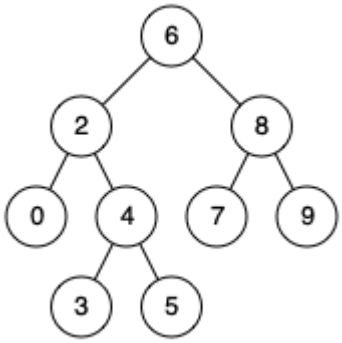
236.二叉搜索树的最近公共祖先（中等）

1. 题目描述

   给定一个二叉搜索树, 找到该树中两个指定节点的最近公共祖先。

   百度百科中最近公共祖先的定义为：“对于有根树 T 的两个结点 p、q，最近公共祖先表示为一个结点 x，满足 x 是 p、q 的祖先且 x 的深度尽可能大（**一个节点也可以是它自己的祖先**）。”

   例如，给定如下二叉搜索树: root = [6,2,8,0,4,7,9,null,null,3,5]



示例 1:

```
输入: root = [6,2,8,0,4,7,9,null,null,3,5], p = 2, q = 8
输出: 6
解释: 节点 2 和节点 8 的最近公共祖先是 6。
```

示例 2:

```
输入: root = [6,2,8,0,4,7,9,null,null,3,5], p = 2, q = 4
输出: 2
解释: 节点 2 和节点 4 的最近公共祖先是 2, 因为根据定义最近公共祖先节点可以为节点本身。
```

说明:

- 所有节点的值都是唯一的。
- p、q 为不同节点且均存在于给定的二叉搜索树中。

2. 简单实现

利用二叉搜索树的性质可以很容易判断出p,q节点在某一节点的同侧还是异侧

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

220. 存在重复元素 III（中等）

1. 题目描述

给定一个整数数组，判断数组中是否有两个不同的索引 *i* 和 *j*，使得 **nums [i]** 和 **nums [j]** 的差的绝对值最大为 *t*，并且 *i* 和 *j* 之间的差的绝对值最大为 *k*。

示例 1:

```
输入: nums = [1,2,3,1], k = 3, t = 0
输出: true
```

示例 2:

```
输入: nums = [1,0,1,1], k = 1, t = 2
输出: true
```

示例 3:

```
输入: nums = [1,5,9,1,5,9], k = 2, t = 3
输出: false
```

2. 简单实现
   - 维持一个大小不大于k的动态滑窗有序的数据结构即可，对任一nums[i]，只要在该数据结构里能找到与其差值小于等于t的元素即可
   - 懒得用二叉搜索树实现了，set也是有序的，在c++ stl里用平衡二叉搜索树实现的

```
class Solution {
public:
    bool containsNearbyAlmostDuplicate(vector<int>& nums, int k, int t) {
        if (k < 1 || t < 0 || nums.empty()) return false;
        int N = nums.size();
        set<long> m;//最大为k
        m.insert(nums[0]);
        for (int i = 1; i < N; ++i) {
            auto it = m.lower_bound(nums[i]);
            if (it != m.end() && abs(*it - nums[i]) <= t)//大于等于nums[i]的最小数
                return true;
            if (it != m.begin() && abs(*(--it) - nums[i]) <= t)//小于nums[i]的最大数
                return true;
            m.insert(nums[i]);
            if (i - k >= 0) m.erase(nums[i - k]);//去掉滑出滑窗的元素
        }
        return false;
    }
};
```

高度平衡的二叉搜索树简介

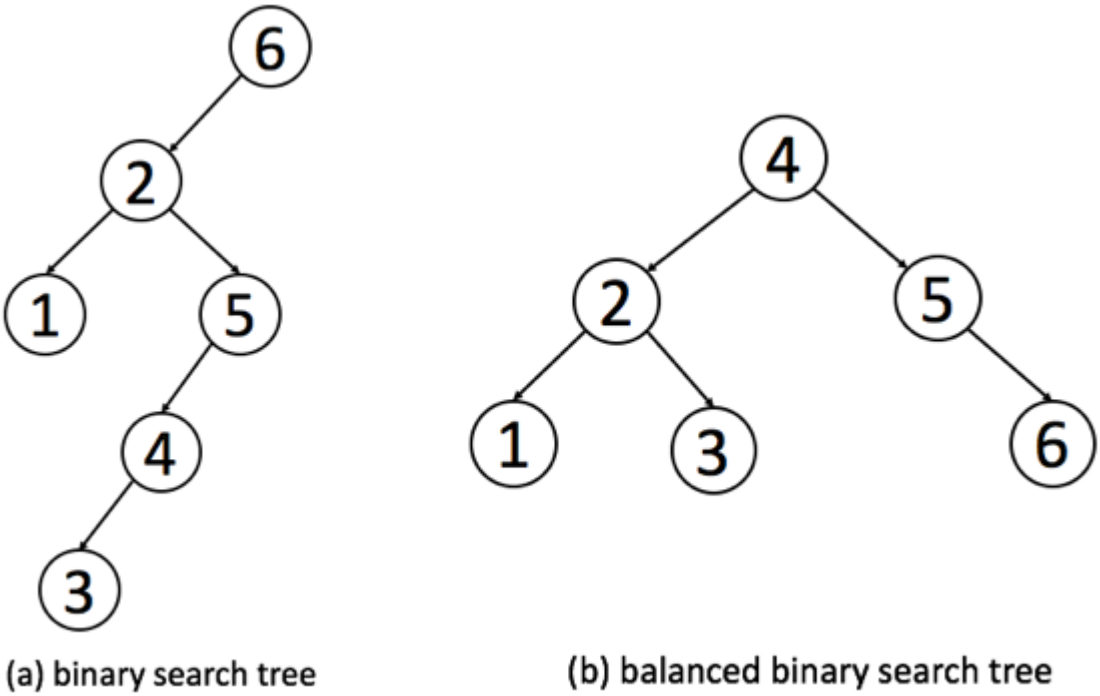
什么是一个高度平衡的二叉搜索树?

> 树结构中的常见用语:

> - 节点的深度 - 从树的根节点到该节点的边数
> - 节点的高度 - 该节点和叶子之间最长路径上的边数
> - 树的高度 - 其根节点的高度

一个`高度平衡的二叉搜索树`（`平衡二叉搜索树`）是在插入和删除任何节点之后，可以自动保持其高度最小。也就是说，有*N*个节点的平衡二叉搜索树，它的高度是*logN*。并且，每个节点的两个子树的高度不会相差超过1。

下面是一个普通二叉搜索树和一个高度平衡的二叉搜索树的例子:



根据定义, 我们可以判断出一个二叉搜索树是否是高度平衡的 (平衡二叉树)。

正如我们之前提到的, 一个有N个节点的平衡二搜索叉树的高度总是*logN*。因此，我们可以计算节点总数和树的高度，以确定这个二叉搜索树是否为高度平衡的。

同样，在定义中， 我们提到了高度平衡的二叉树一个特性: 每个节点的两个子树的深度不会相差超过1。我们也可以根据这个性质，递归地验证树。

为什么需要用到高度平衡的二叉搜索树?

我们已经介绍过了二叉树及其相关操作, 包括搜索、插入、删除。 当分析这些操作的时间复杂度时，我们需要注意的是树的高度是十分重要的考量因素。以搜索操作为例，如果二叉搜索树的高度为`*h*`，则时间复杂度为`*O(h)*`。二叉搜索树的高度的确很重要。

所以，我们来讨论一下树的节点总数*N*和高度`*h*`之间的关系。 对于一个平衡二叉搜索树, 我们已经在前文中提过, $h \geq \lfloor \log_2 N \rfloor$。但对于一个普通的二叉搜索树， 在最坏的情况下, 它可以退化成一个链。

因此，具有*N*个节点的二叉搜索树的高度在*logN*到*N*区间变化。也就是说，搜索操作的时间复杂度可以从`*logN*`变化到`*N*`。这是一个巨大的性能差异。

所以说，高度平衡的二叉搜索树对提高性能起着重要作用。

如何实现一个高度平衡的二叉搜索树?

有许多不同的方法可以实现。尽管这些实现方法的细节有所不同，但他们有相同的目标:

1. 采用的数据结构应该满足二分查找属性和高度平衡属性。
2. 采用的数据结构应该支持二叉搜索树的基本操作，包括在 `*O(logN)*` 时间内的搜索、插入和删除，即使在最坏的情况下也是如此。

我们提供了一个常见的的高度平衡二叉树列表供您参考：

- 红黑树
- AVL树
- 伸展树
- 树堆

我们不打算在本文中展开讨论这些数据结构实现的细节。

高度平衡的二叉搜索树的实际应用

高度平衡的二叉搜索树在实际中被广泛使用，因为它可以在 `*O(logN)*` 时间复杂度内执行所有搜索、插入和删除操作。

平衡二叉搜索树的概念经常运用在Set和Map中。 Set和Map的原理相似。 我们将在下文中重点讨论Set这个数据结构。

> Set(集合)是另一种数据结构，它可以存储大量key(键)而不需要任何特定的顺序或任何重复的元素。 它应该支持的基本操作是将新元素插入到Set中，并检查元素是否存在于其中。

通常，有两种最广泛使用的集合： `散列集合` (Hash Set)和 `树集合` (Tree Set)。

`树集合`, Java中的 `Treeset` 或者C++中的 `set`，是由高度平衡的二叉搜索树实现的。因此，搜索、插入和删除的时间复杂度都是 `O(logN)`。

`散列集合`, Java中的 `HashSet` 或者C++中的 `unordered_set`，是由哈希实现的, 但是平衡二叉搜索树也起到了至关重要的作用。当存在具有相同哈希键的元素过多时，将花费 `O(N)` 时间复杂度来查找特定元素，其中N是具有相同哈希键的元素的数量。 通常情况下，使用高度平衡的二叉搜索树将把时间复杂度从 `O(N)` 改善到 `O(logN)`。

哈希集和树集之间的本质区别在于树集中的键是 `有序` 的。

总结

高度平衡的二叉搜索树是二叉搜索树的特殊表示形式，旨在提高二叉搜索树的性能。但这个数据结构具体的实现方式，超出了我们这章的内容，也很少会在面试中被考察。但是了解高度平衡二叉搜索树的的基本概念，以及如何运用它帮助你进行算法设计是非常有用的。

110. 平衡二叉树（简单）

---

1. 题目描述

给定一个二叉树，判断它是否是高度平衡的二叉树。

本题中，一棵高度平衡二叉树定义为：

> 一个二叉树*每个节点*的左右两个子树的高度差的绝对值不超过1。

示例 1:

给定二叉树 `[3,9,20,null,null,15,7]`

```
    3
   / \
  9  20
    /  \
   15   7
```

返回 `true` 。

示例 2:

给定二叉树 `[1,2,2,3,3,null,null,4,4]`

```
       1
      / \
     2   2
    / \
   3   3
  / \
 4   4
```

返回 `false` 。

2. 简单实现

```
class Solution {
public:
    int balancedHeight(TreeNode* root){
        if(!root) return 0;//空树高度为0
        int lefth = balancedHeight(root->left);
        int righth = balancedHeight(root->right);
        if(lefth < 0 || righth < 0) return -1;//不是高度平衡
        if(abs(lefth - righth) > 1) return -1;//不是高度平衡
        return max(lefth, righth) + 1;//是高度平衡，返回树高
    }
    bool isBalanced(TreeNode* root) {
        if(balancedHeight(root) >= 0)
            return true;
        else
            return false;
    }
};
```

108.将有序数组转换为二叉搜索树（简单）

1. 题目描述

将一个按照升序排列的有序数组，转换为一棵高度平衡二叉搜索树。本题中，一个高度平衡二叉树是指一个二叉树*每个节点*的左右两个子树的高度差的绝对值不超过 1。

示例:

```
给定有序数组: [-10,-3,0,5,9],

一个可能的答案是：[0,-3,9,-10,null,5]，它可以表示下面这个高度平衡二叉搜索树：

      0
     / \
   -3   9
   /   /
 -10  5
```

2. 简单实现

```
class Solution {
public:
    TreeNode* sortedArrayToBST(vector<int>& nums, int l, int r){
        if(l > r) return NULL;
        if(l == r) return new TreeNode(nums[l]);
        int mid = l + (r - l) / 2;
        TreeNode* root = new TreeNode(nums[mid]);
        root->left = sortedArrayToBST(nums, l, mid - 1);
        root->right = sortedArrayToBST(nums, mid + 1, r);
        return root;
    }
    TreeNode* sortedArrayToBST(vector<int>& nums) {
        return sortedArrayToBST(nums, 0, nums.size()-1);
    }
};
```