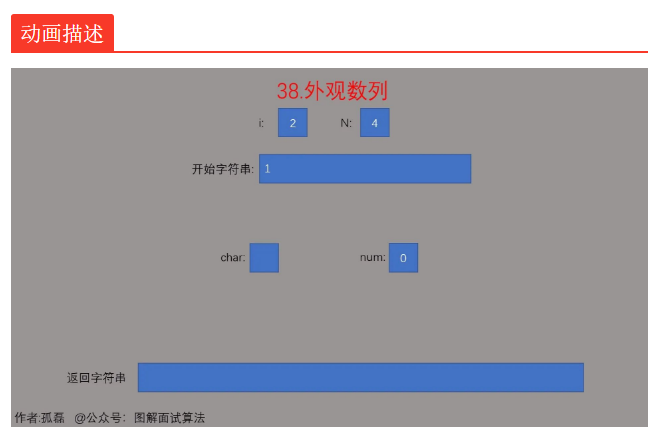
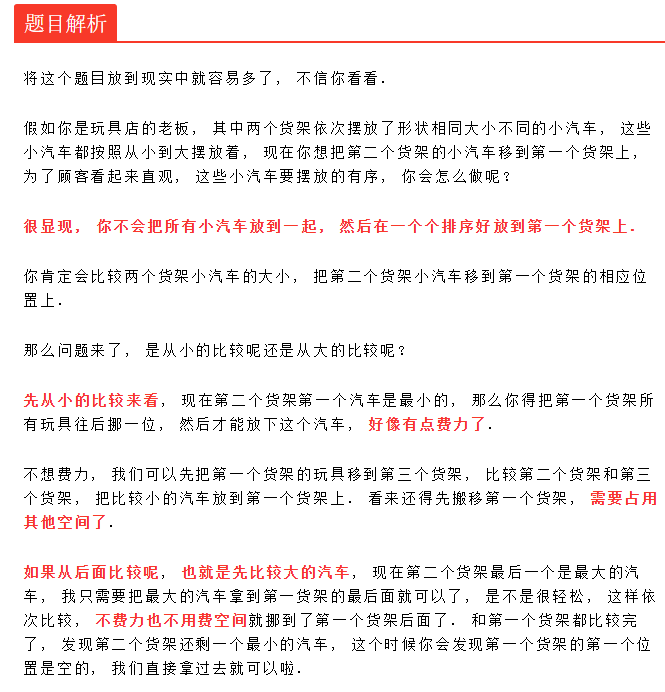
# 简单

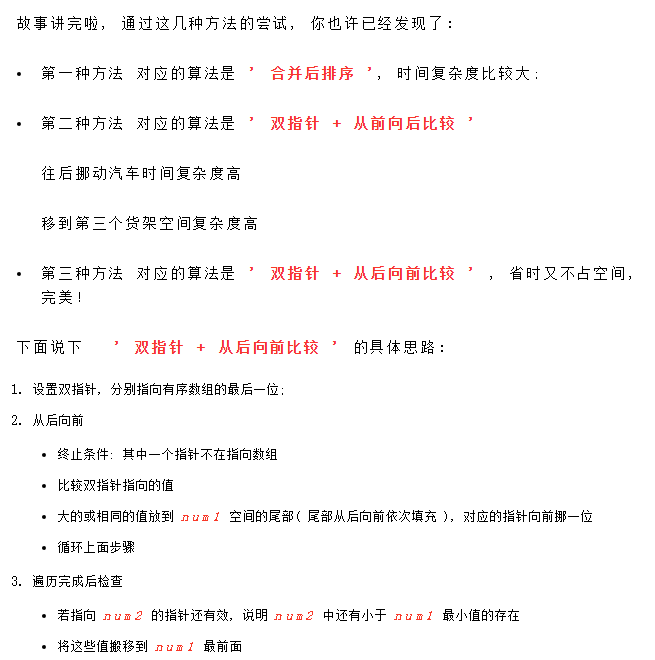
## 38\_外观数列



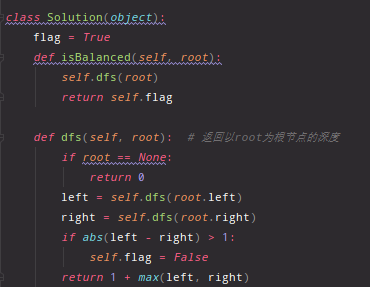


## 88\_合并两个有序数组





110\_平衡二叉树



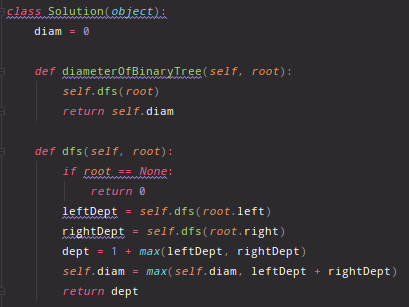
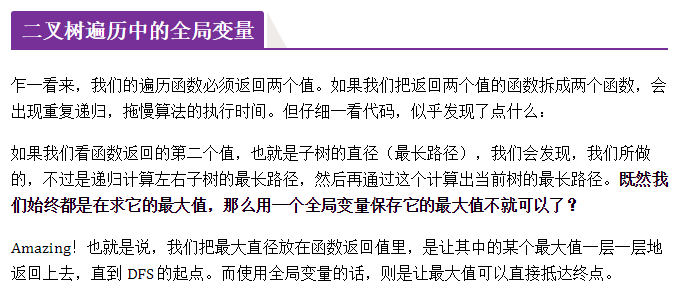
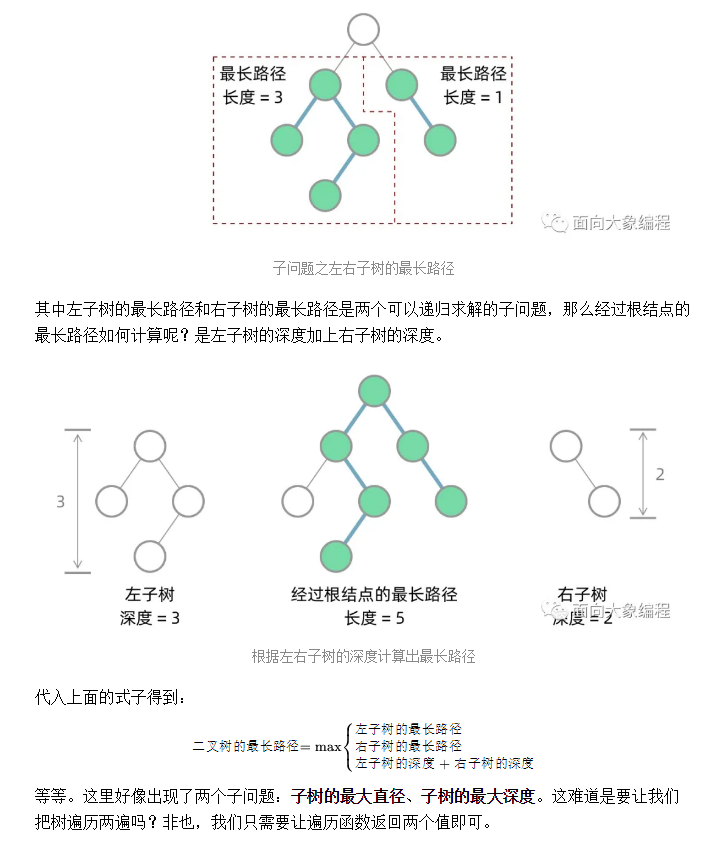
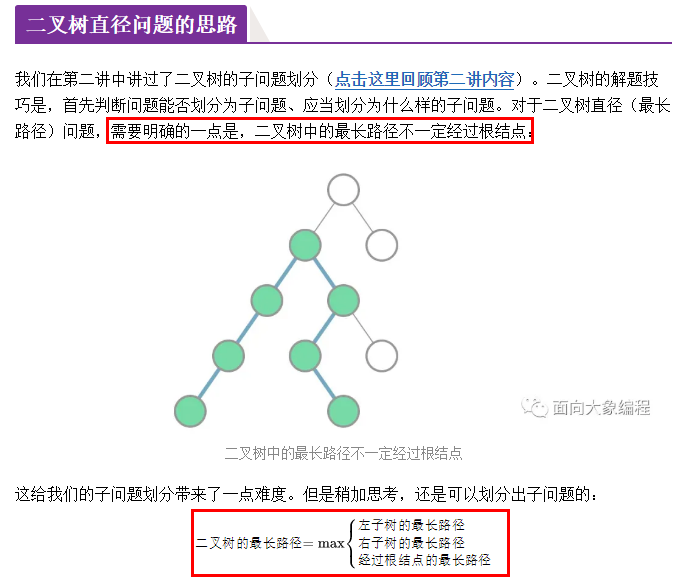
## 290\_单词规律



改进：不需要hashset，只需要讲Map中的values值取出比较即可。



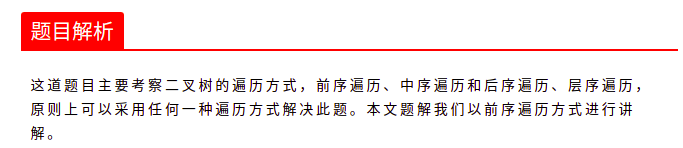
## 543\_二叉树的直径

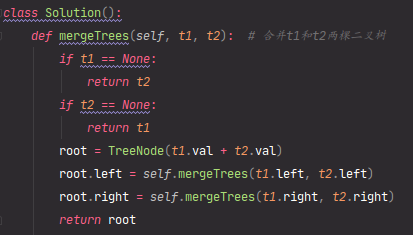


## 563\_二叉树的坡度



## 617\_合并二叉树



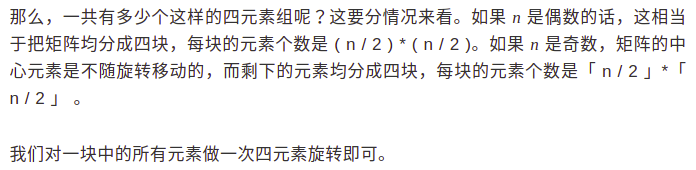


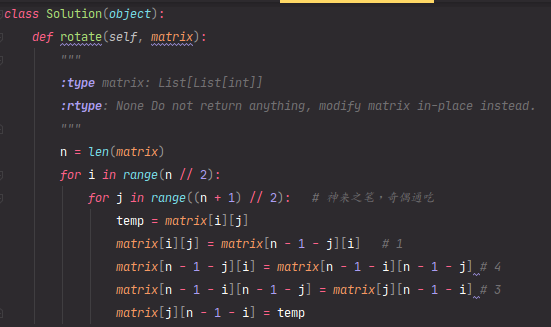
# 中等

## 48\_旋转图像







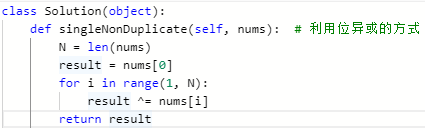


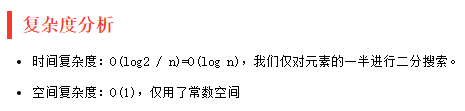
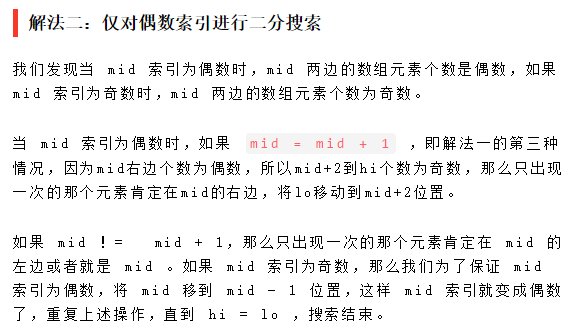
## 127\_单词接龙

将其转化为邻接表，然后利用图的BFS，因为是BFS，所以当遇到最终节点时，就会使最短路径。但是会超出时间限制，希望以后能找到解决方法

## 540\_有序数组中的单一元素

解法一：





二分了N次，直到只剩下一个元素，就是我们要找的那个数:

2 ^ N = n ----> N = logn

## 1372\_二叉树中的最长交错路径

题目不难，难得是如何只进行一次深度遍历，解决方法是：

当该往左节点时，那么就不会往右节点了，那么这个时候右节点就可以重新计算最长交错路径，达到一次遍历，两次运用的目的。

# 困难

## LCP 14\_切分数组

用gcd（a, b）> 1来判断太过耗时间，只需求出b的所有质数u[]，看a%u[i] == 0是否满足，满足就表明最大公约数大于1，这是题目设计好了的，就是要你用质数筛。

## 23\_合并k个排序链表

结合归并排序，转化为合并两个排序链表的算法。

## 72\_编辑距离

看一个例子就懂：

horse --> ros

插入s: horse --> ro dp[i][j-1]

删除e: hors --> ros dp[i-1][j]

替换e: hors --> ro dp[i-1][j-1]

注：1、当比较的字母相等时，不需要替换操作

2、当word1长度为0时，要插入word2长度，反之删除word1长度

## 124\_二叉树的最大路径和



应该划分为：（倒数第三句）

1. root.val + 左子树的最大路径和（因为右子树为负）
2. root.val + 右子树的最大路径和（因为左子树为负）
3. Root.val + 左右最大路径和（左右子树都非负）

## [1373\_二叉搜索子树的最大键值和](https://leetcode-cn.com/problems/maximum-sum-bst-in-binary-tree/)

我的思路：取一个全局变量，从上往下记录最大键值和self.sum，当不满足二叉树的性质时，self.sum清0，从不满足二叉树性质的那个节点所在层从新计算。

阶梯思路：上面的思路错误，我的想法是，从上往下，这样是大错特错的，因为加入底下不是二叉搜索树，那上面更不会是，所以应该采用的是自底向上，这里的思路很巧，返回三个值，以root为根节点的二叉搜索树的路径和，该二叉搜索树的最小值和最大值，这时应熟练处理，当root为None和当root为根节点的二叉树不是二叉搜索树时的情况，将题目条件运用的非常巧妙。



# 数据结构

## 图：

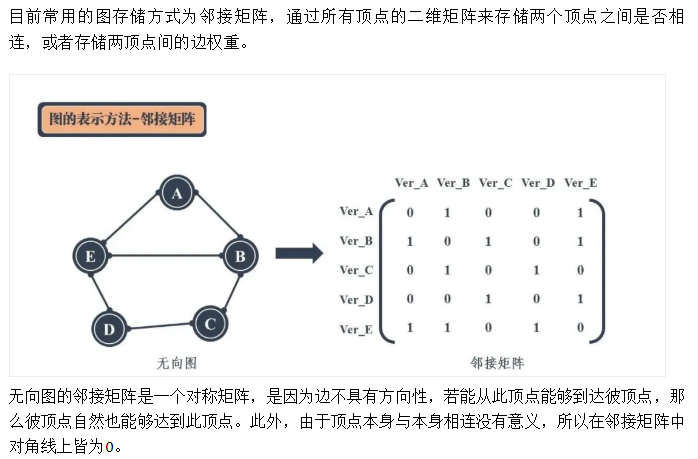
图分两种：

有向图（节点有入度和出度）、无环图（可以间接认为只有出度）

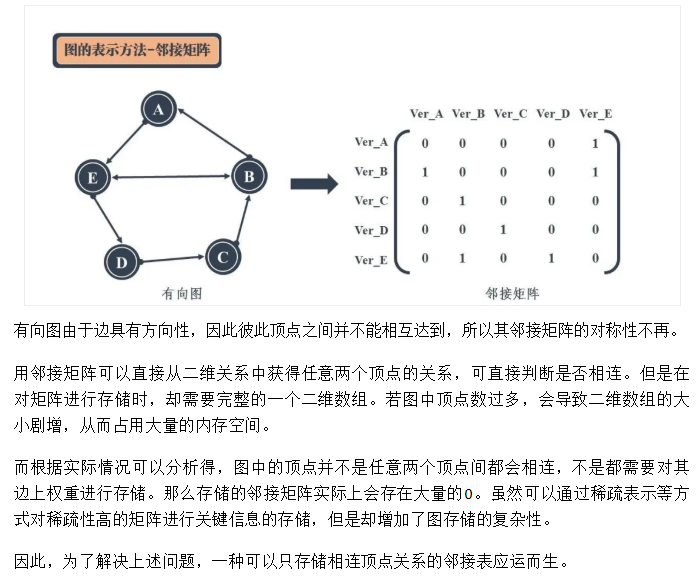
图有两种存储方式：（我的代码实现的是简单的邻接表）

A、邻接矩阵：

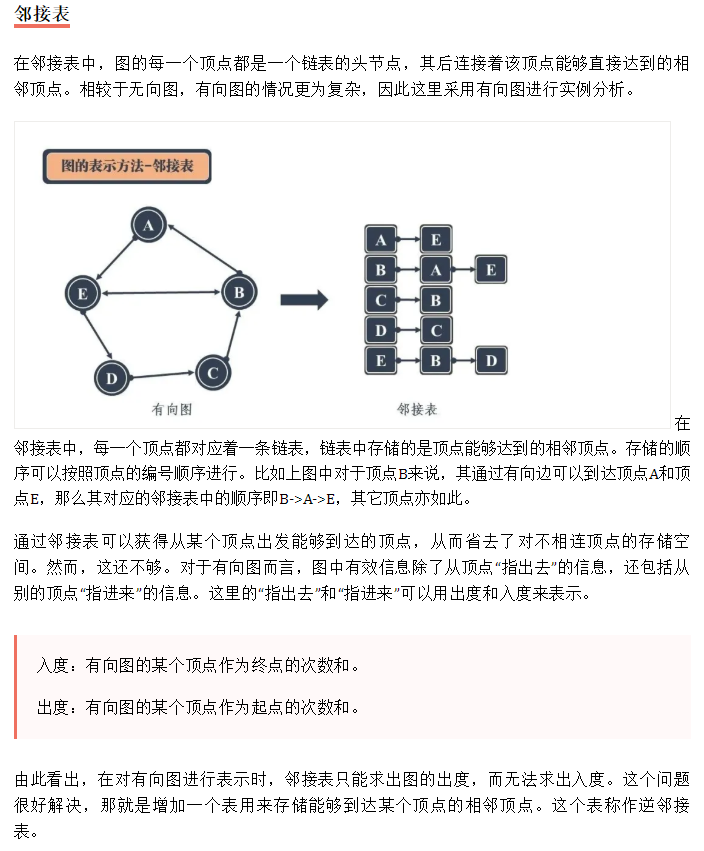
无向图：

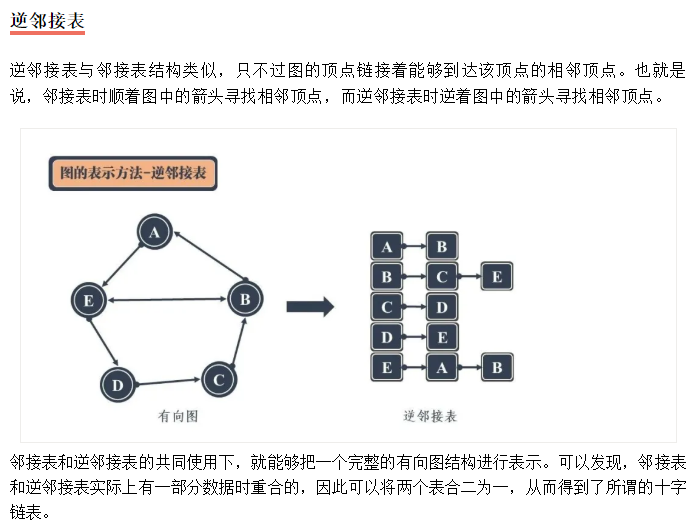


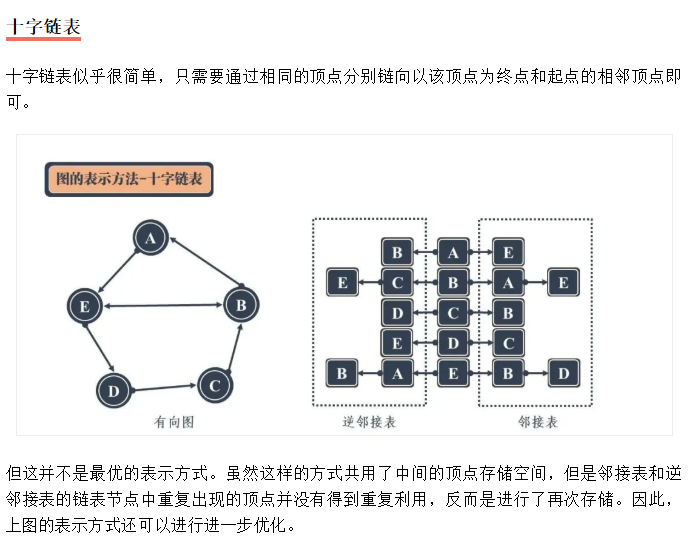
有向图：



1. 邻接表







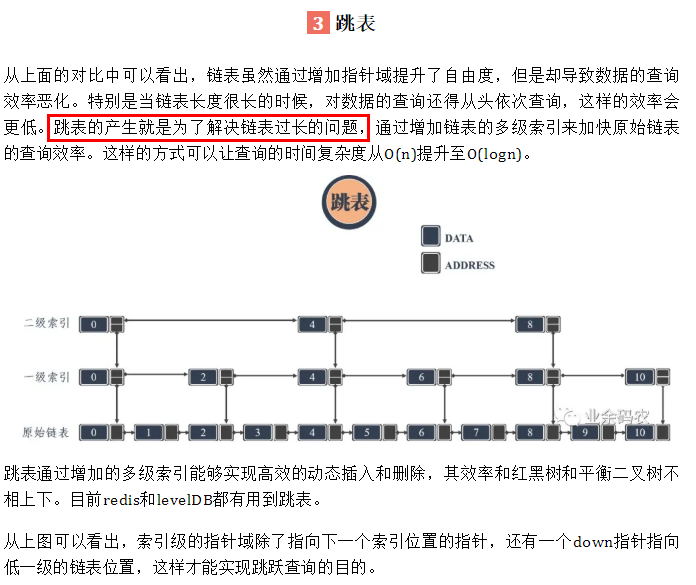


## 数组和链表



数组因为连续的存储空间所有可以实现随机访问，但这样的话增删元素会移动被修改元素之后的元素。

## 跳表



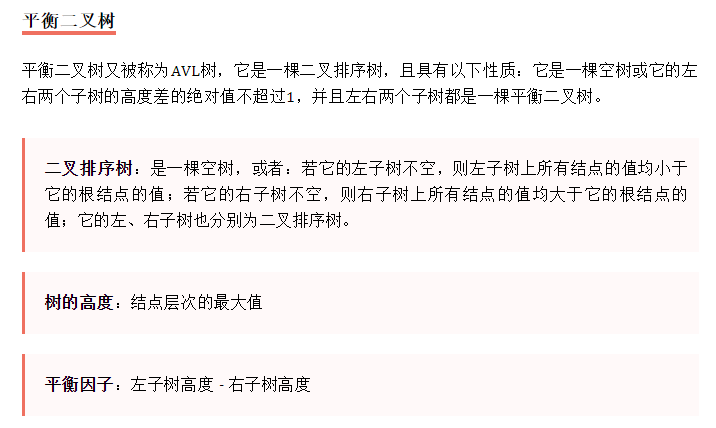
## 红黑树vs AVL

### 二者的定义

都是二叉排序树：

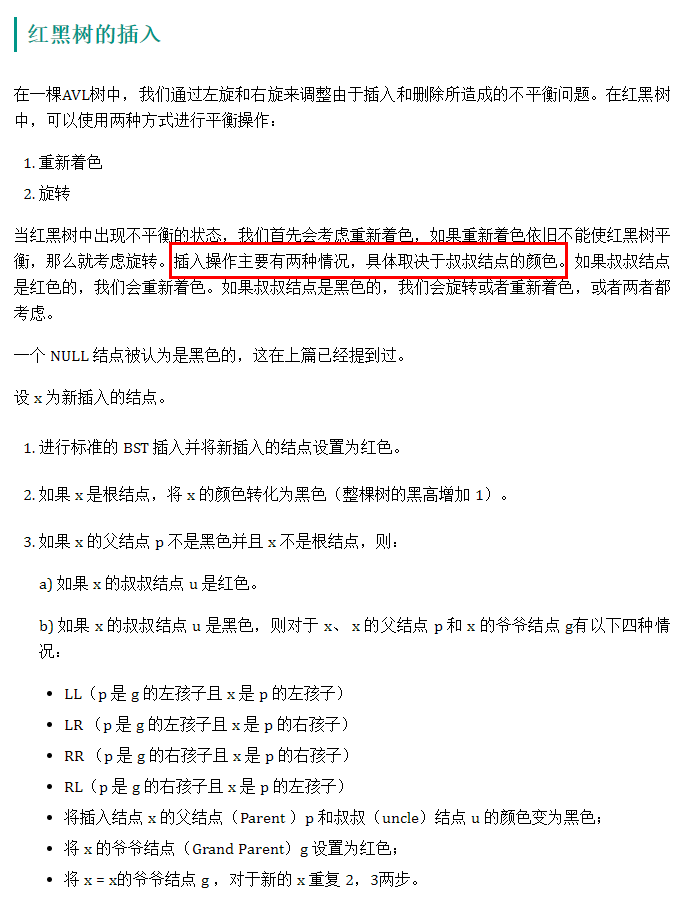
红黑树五个特性要记牢：

1. 根节点为黑
2. 节点要么为黑、要么为红
3. 不存在两个连续的红节点（红节点的子节点为黑）
4. 从任一个节点到叶子节点出发的黑高相等
5. 叶子结点为黑





### 红黑树的插入



总结：

1. 插入的节点的都是红色节点，但插入的节点作为根节点，那么将其重新着色为黑色
2. 若父节点是黑色，则不用重新着色和旋转
3. 若父节点是红色：
4. 叔叔节点是红色，那么只需将叔叔和父节点重新着色为黑色，将爷爷节点置为红色，那么视爷爷节点为新插入的节点，重复2、3步。
5. 叔叔节点是黑色，结合爷爷节点和父节点的位置关系分为：

LL（右旋爷爷节点，交换新的爷爷节点和右孩子的颜色）

LR（先左旋父节点，转化为LL）

RR（左旋爷爷节点，交换新的爷爷节点和左孩子的颜色）

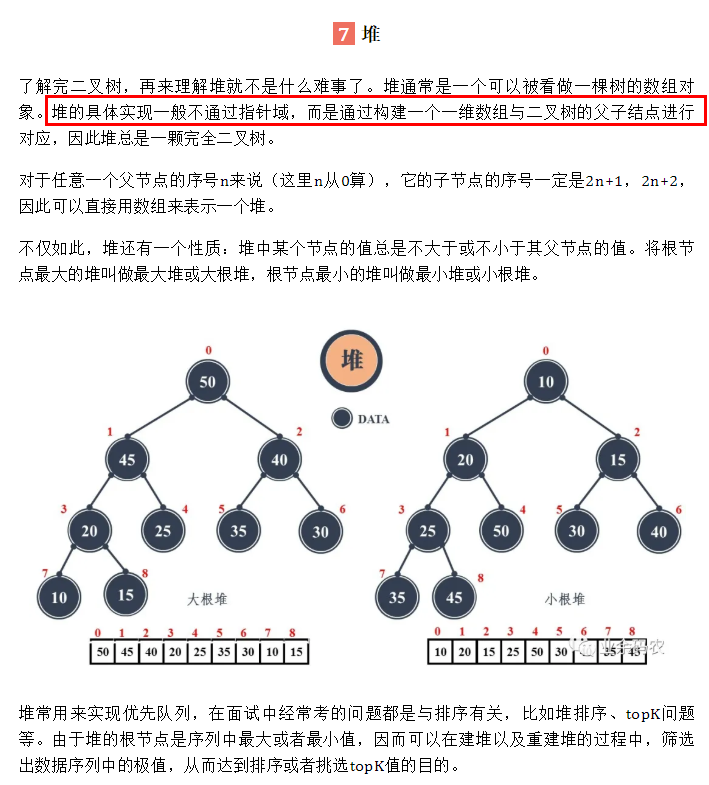
RL（右旋父节点，转化为LL）

综上：其实叔叔节点是黑色更好一点，因为不用重复2、3步。

具体可参考链接：

[https://mp.weixin.qq.com/s?\_\_biz=MzA4NDE4MzY2MA==&mid=2647521642&idx=1&sn=dda12824118e46c17374333062e37a6a&chksm=87d24669b0a5cf7f5bf8e2614deb224d15c10cd474d24c17e9354743b7a5eb72fc89469e6330&mpshare=1&scene=1&srcid=&sharer\_sharetime=1590880990893&sharer\_shareid=af8c720bd8883efc17497f6364b732e0&key=2a8d89b5c751c2413aa57fccffea18542df31efec407421ad406a849cbe0b93cbfbfb5b09634283da96025ef666499db9902fa1c86674f460f0e9c6cb41cc599987cdd2ca723d68345f24e1b8e87563b&ascene=1&uin=MTU0MDc2NTcyOQ%3D%3D&devicetype=Windows+10+x64&version=62090070&lang=zh\_CN&exportkey=A1f35tyJWazlfVY%2F5ffW9y0%3D&pass\_ticket=57ePHO%2FoAH4B7gjeo%2Fc1OHvhlixV%2BemwHldvp6%2Fij%2F%2FqHJCvi3hja8aT2wdtoe2y](https://mp.weixin.qq.com/s?__biz=MzA4NDE4MzY2MA==&mid=2647521642&idx=1&sn=dda12824118e46c17374333062e37a6a&chksm=87d24669b0a5cf7f5bf8e2614deb224d15c10cd474d24c17e9354743b7a5eb72fc89469e6330&mpshare=1&scene=1&srcid=&sharer_sharetime=1590880990893&sharer_shareid=af8c720bd8883efc17497f6364b732e0&key=2a8d89b5c751c2413aa57fccffea18542df31efec407421ad406a849cbe0b93cbfbfb5b09634283da96025ef666499db9902fa1c86674f460f0e9c6cb41cc599987cdd2ca723d68345f24e1b8e87563b&ascene=1&uin=MTU0MDc2NTcyOQ==&devicetype=Windows+10+x64&version=62090070&lang=zh_CN&exportkey=A1f35tyJWazlfVY/5ffW9y0=&pass_ticket=57ePHO/oAH4B7gjeo/c1OHvhlixV+emwHldvp6/ij//qHJCvi3hja8aT2wdtoe2y)

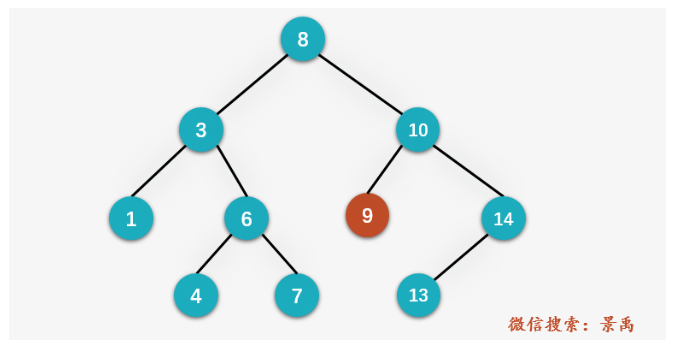
## 5、堆

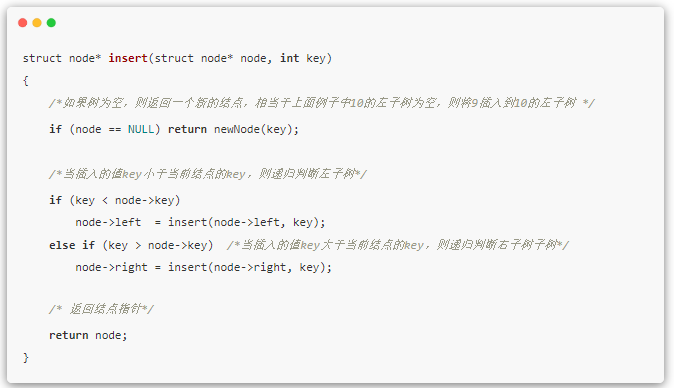


## 树

### 二叉排序树的插入

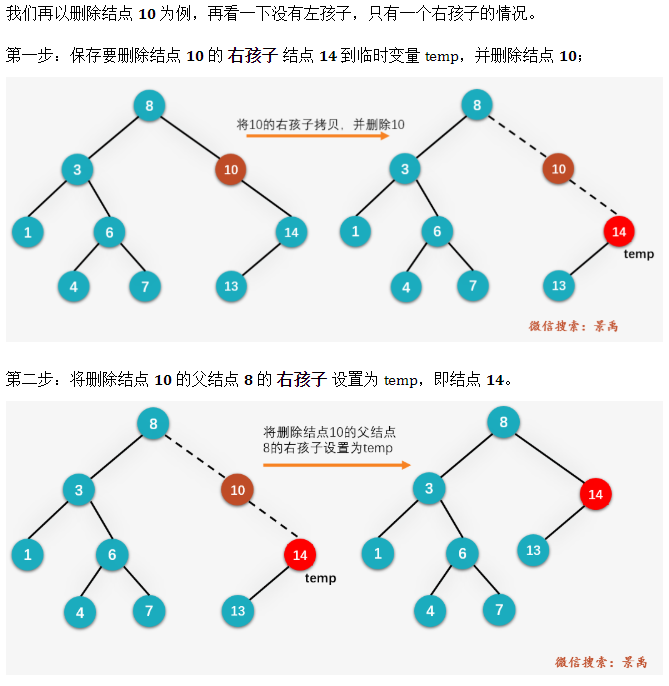
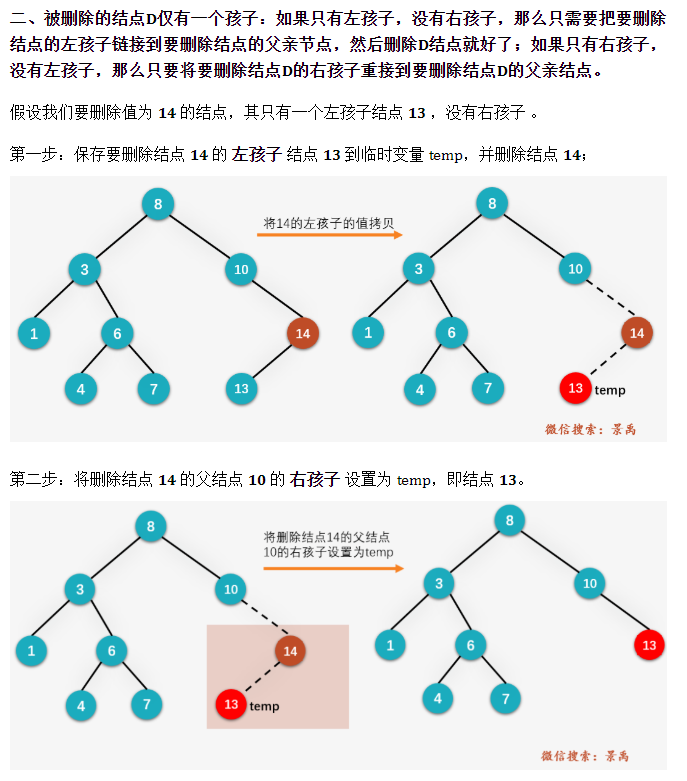
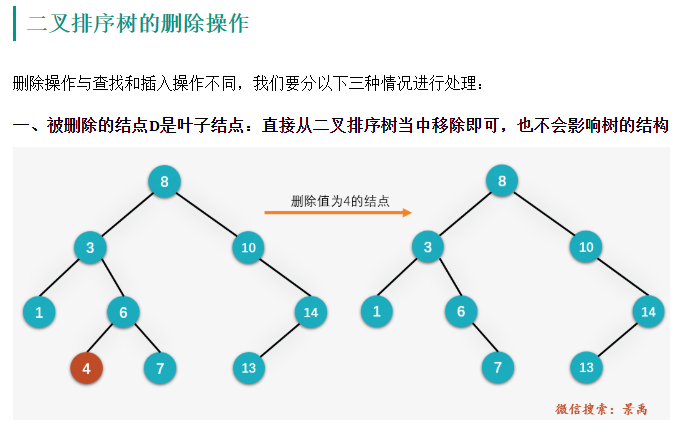
二叉树的插入其实和查找没什么区别，因为给定一个节点，进行插入时，你细品一下，一定插入的位置会是叶子节点的。





二叉树的插入其实就是构建一棵二叉树的过程，返回的应该是新的二叉树的节点。

### 二叉排序树的删除



1. **当左右孩子都存在时，根据中序遍历，那么可以将该节点的左子树的最大值或者右子树的最小值的节点替换该节点，然后删除左子树的最大值或右子树的最小值，而这两者其实都是叶子节点，所以完美解决。**

**注：1、当然你需要用到二分查找的方式去查找最小值和最大值。**

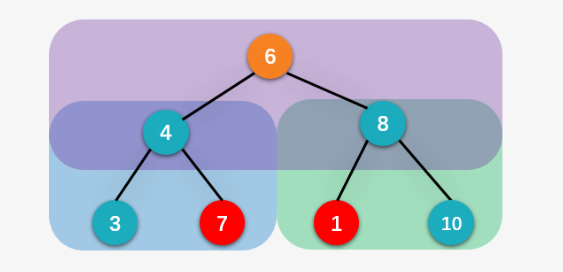
1. **删除其实也是重新构建二叉排序树的过程**

### 判断是否是二叉排序树

记住，二叉排序树的定义是，树及子树的所有左边节点小于根节点，所有右节点大于根节点。

所以下列判断方法是错的：





这样一棵树其实不是二叉排序树，但是返回True

为了只遍历一遍树，那么就要建立辅助函数，传入的参数不仅仅是一个根节点，而应该包含最大和最小节点。

