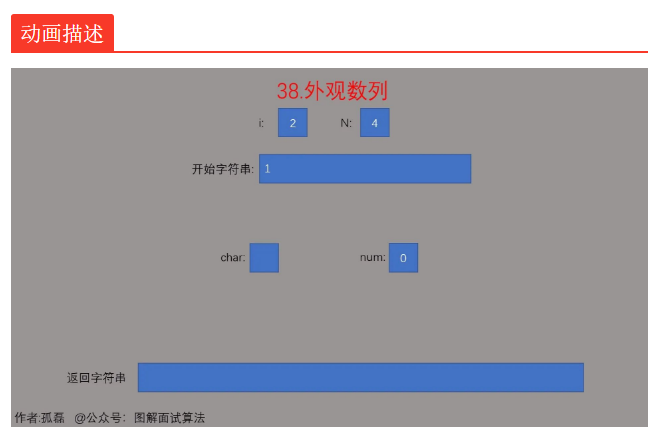
# 简单

## 1\_两数之和

暴力求解并不难，如果要达到O(n)的时间复杂度的话，那么就只能遍历一次，那么就要用到hash表了。这里建立的是map，分别存储nums[i]和索引i。

## 38\_外观数列

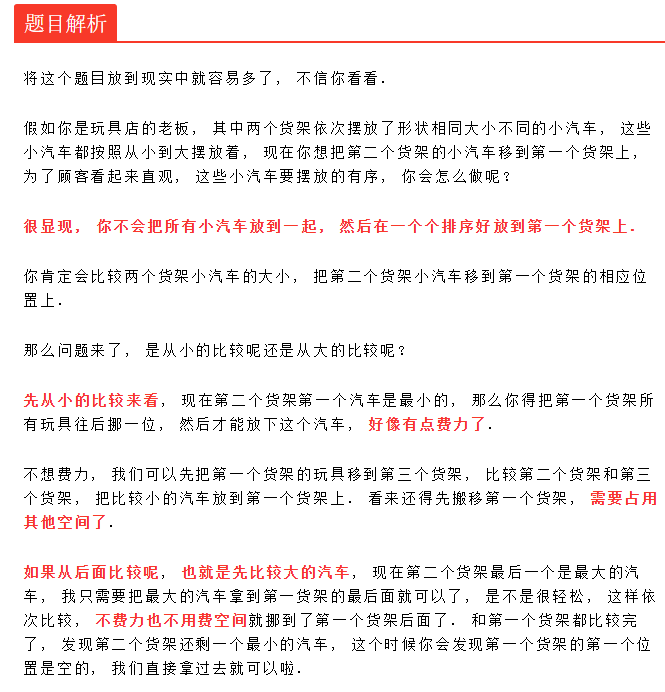


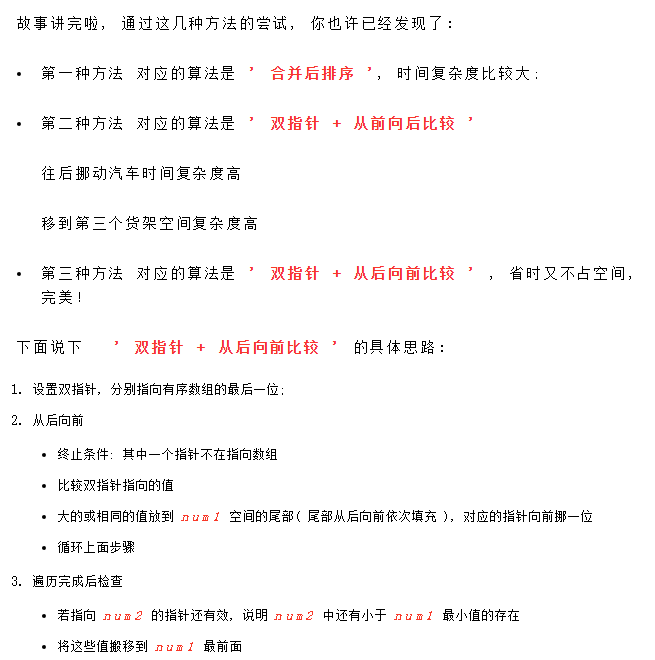


## 53\_最大子序和

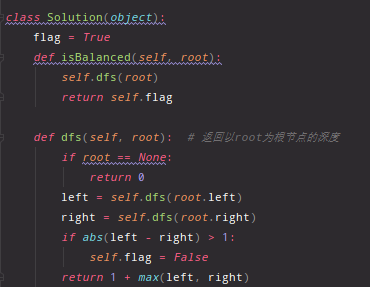
1. maxSum = -2e32
2. Dp[i]表示以nums[i]结尾的最大连续子数组和
3. 记得使用空间优化

## 88\_合并两个有序数组



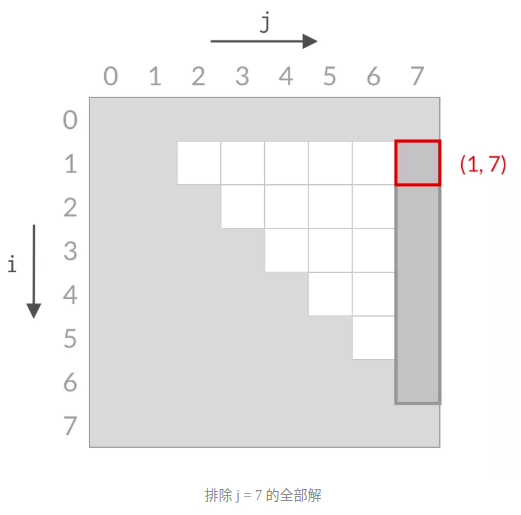


110\_平衡二叉树



## 167\_两数之和2

当然可以采用两数之和的hash表的方法去做，但是因为是有序的，如果利用双指针去缩减搜索空间，这样才是最适合这道题的解法。



## 198\_打家劫舍

Dp[i]表明打劫前i家商店的最高金额，但是第i家商店的金额是nums[i-1]

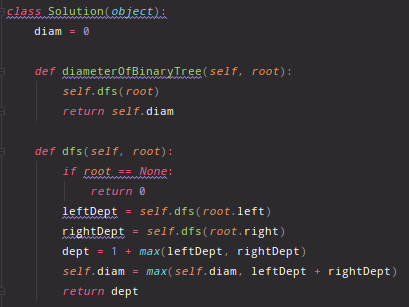
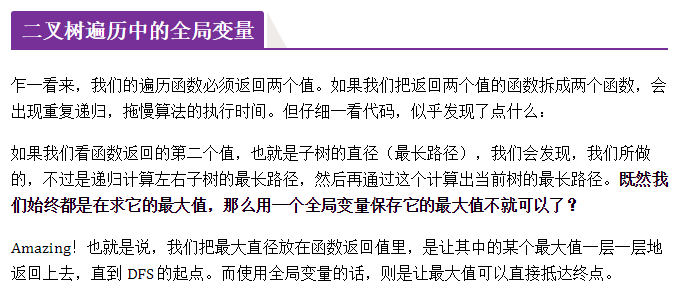
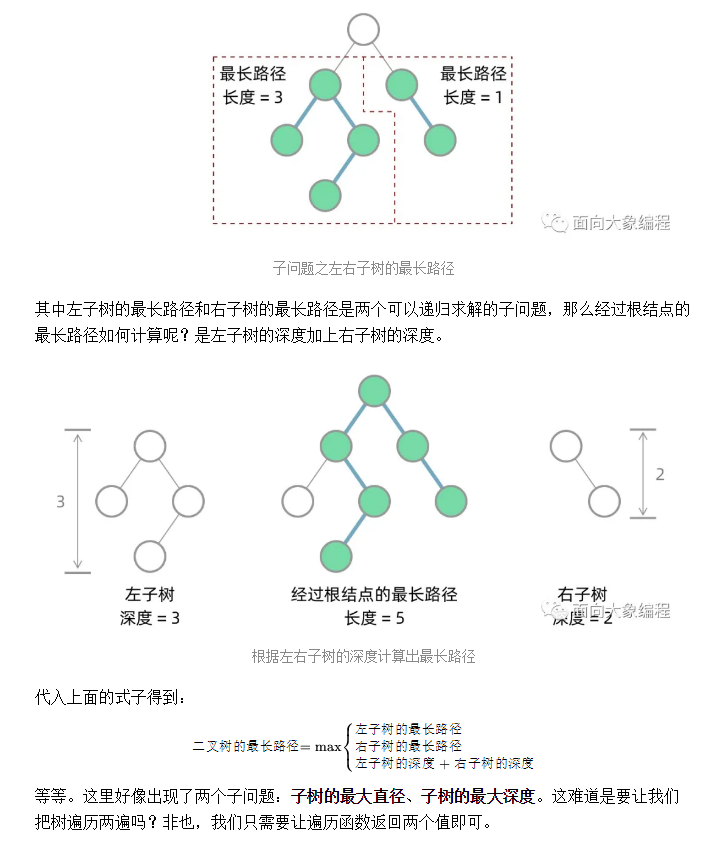
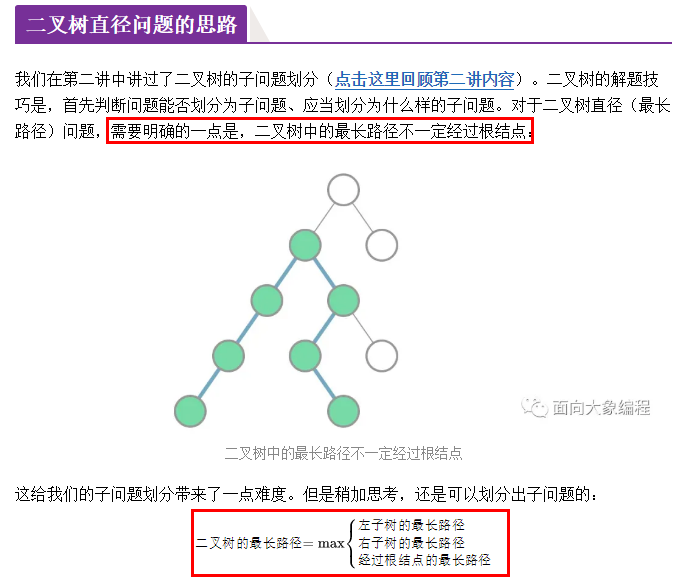
## 290\_单词规律



改进：不需要hashset，只需要讲Map中的values值取出比较即可。



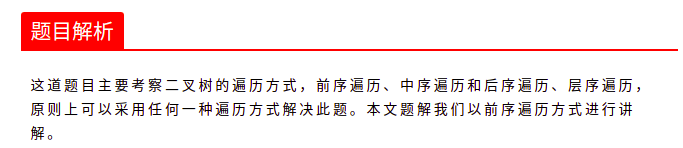
## 543\_二叉树的直径

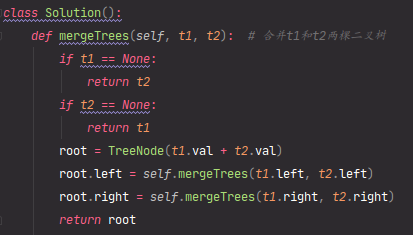


## 563\_二叉树的坡度



## 617\_合并二叉树





# 中等

## 46\_全排列

关键思想时，回溯数组，k左边为已选集合，k右边为候选集合，当k到达末尾，候选集合为空时，实现一次全排列，记录所有情况即可。

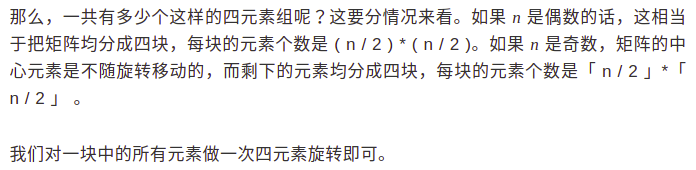
变种：1、从n个元素中选m个数全排列，那么就是已选集合达到m个时，记录一下即可。

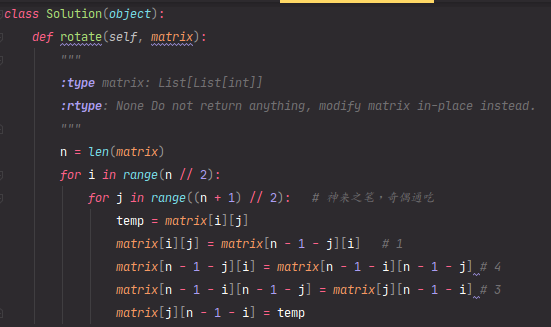
2、从n个元素中选m个数，即组合，注意两点，数组要排序，不交换。见77\_组合题。

## 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

## 152\_最大乘积子数组

1. 数组中有负数，所以maxProduct = -2e32
2. 相比于最大连续和子数组来说，这里因为是乘积，所以要维持两个数，一个最大乘积，一个最小乘积，前者一定为正，后者一定为负，当遇到负数时，交换两者。这是精髓，一定要记住。

## 213\_打家劫舍2

在打家劫舍的基础上分为两种情况：

1、# 第一间房屋必偷， 那么就要从第三间房屋开始偷起， 偷到第N - 1间房屋为止

2、# 第一间不偷，偷到最后一间房屋为止

重用代码。

## 240\_搜索二维矩阵2

从左下角或者从右上角开始，以右上角A[x,y]为例:

If target > A[x, y]:

y--

Elif target < A[x, y]:

x++

Else:

Return [x,y]

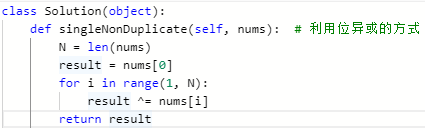
## 337\_打家劫舍3

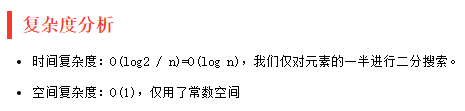
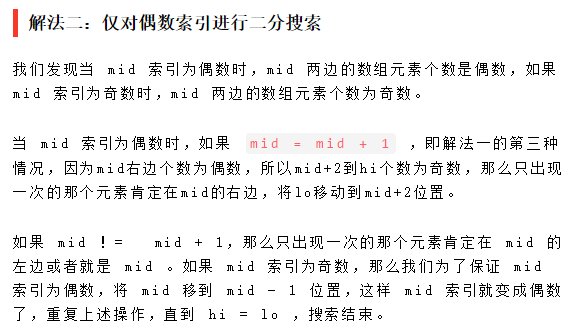
递归函数表示# 从root节点进入 偷root节点和不偷root节点返回的最大金额

注意：兄弟节点不算相邻

## 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

## 718\_最长重复子数组

# dp[i][j] 表示A以第i个字符结尾和B前j个字符结尾的最长重复子数组长度



空间优化：无法空间优化，因为递推关系式既设计上一行也涉及上一列

## 978\_最长湍流子数组

1. dp从1开始计数，因为刚开始和当两数相等时，从新开始寻找湍流子数组时，本身计为1，所以从第二个开始循环，相等之后，也相当与从相等之后的下一个数开始循环
2. 设置flag，flag为1表示前面为上升趋势，为0为下降趋势。

## 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。

## 1143\_最长公共子序列

参考718题

# 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。

## 42\_接雨水

双指针问题，left，right分别记录当前柱子的高度，根据木桶理论，与左右较矮柱子的高度差即为当前位置能够蓄水的体积，所以从较矮柱子那边开始计算雨水的体积，来回切换，直到左指针大于右指针。

如果用单指针，每次都要计算两侧的leftMax和rightMax，那么时间复杂度将是O（n2）。

## 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：

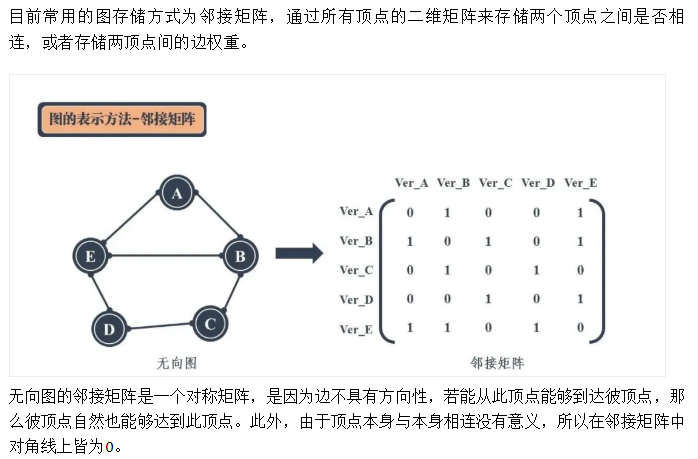
图分两种：

有向图（节点有入度和出度）、无环图（可以间接认为只有出度）

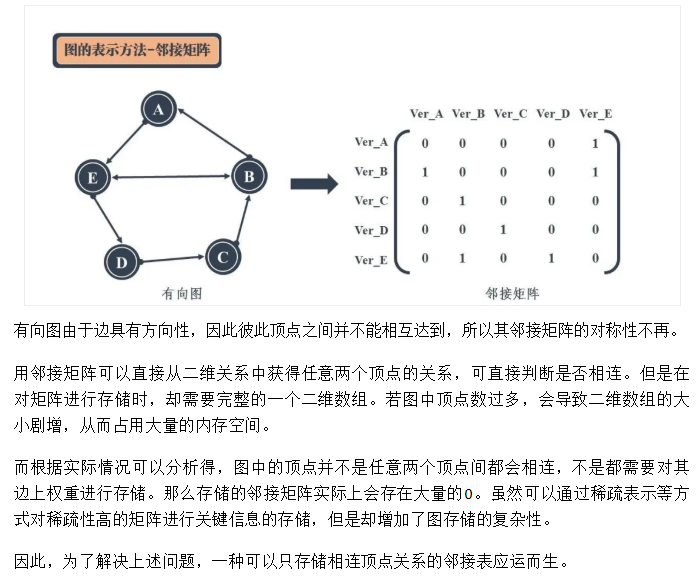
图有两种存储方式：（我的代码实现的是简单的邻接表）

A、邻接矩阵：

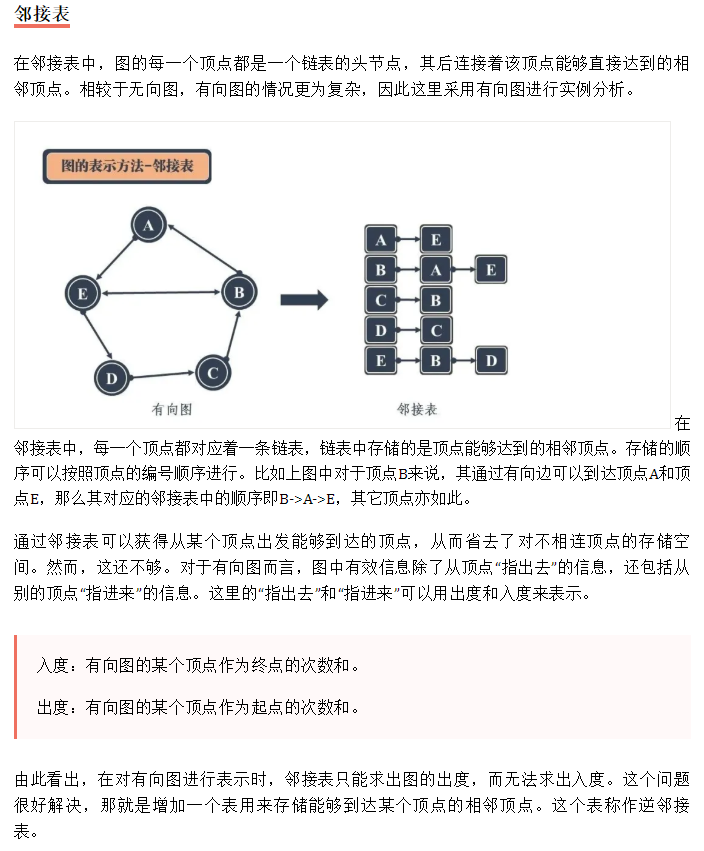
无向图：

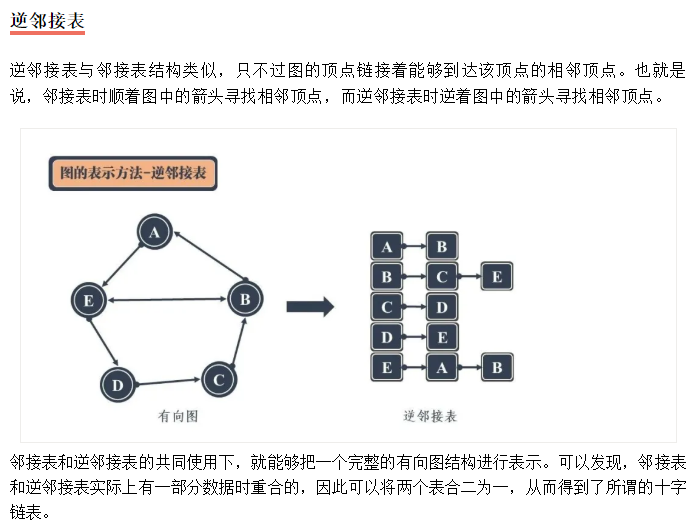


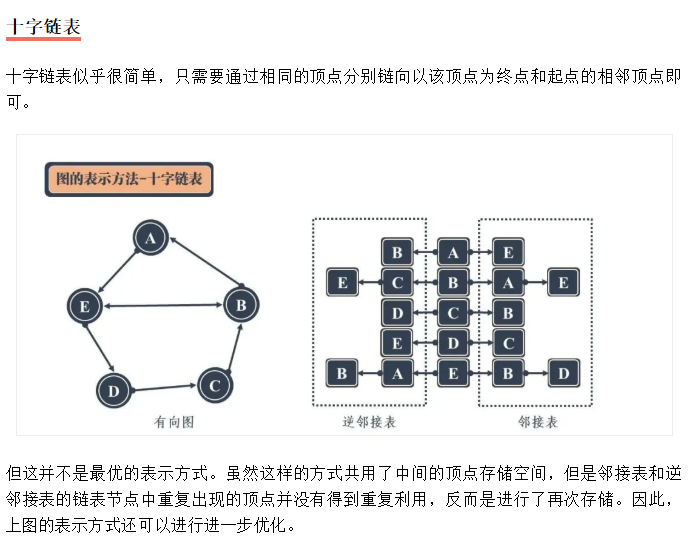
有向图：



1. 邻接表







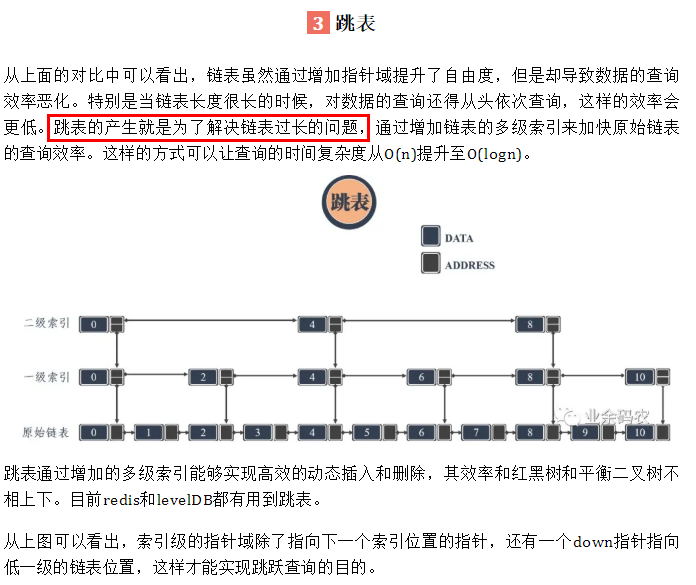


## 数组和链表



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



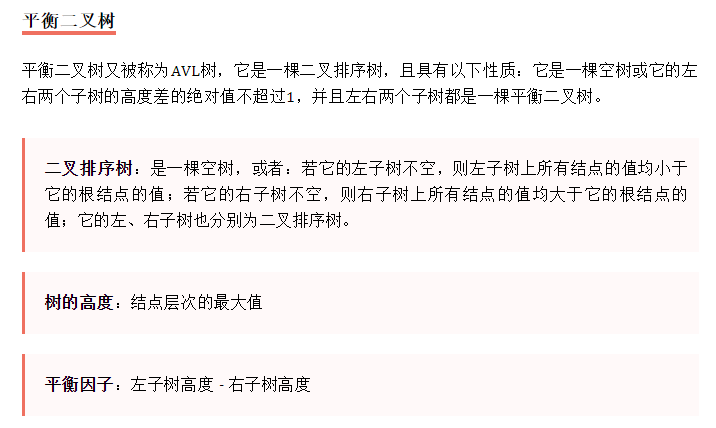
## 红黑树vs AVL

### 二者的定义

都是二叉排序树：

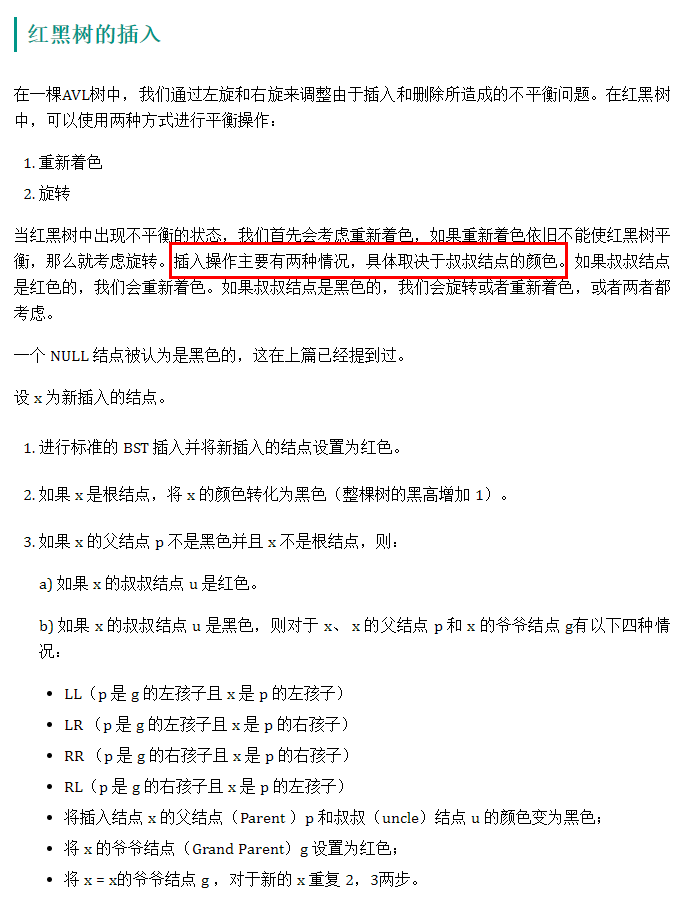
红黑树五个特性要记牢：

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。

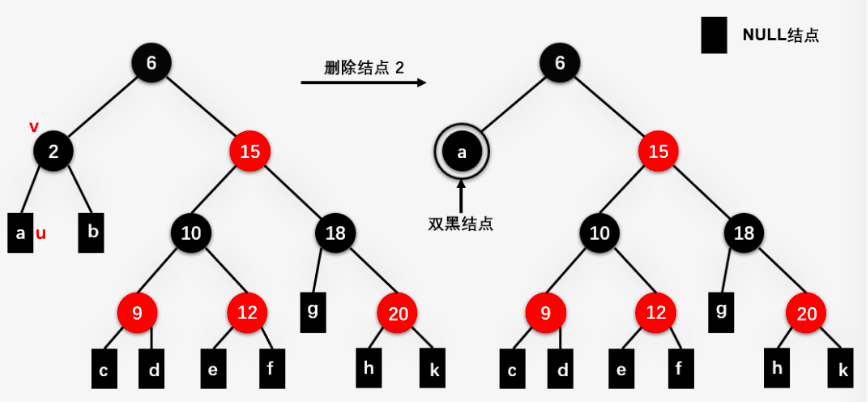
### 红黑树的删除操作

通过检查兄弟节点的颜色来决定恰当的平衡操作。

插入是为了避免红黑树中出现两个连续的红色节点，而删除操作是为了避免删除黑色节点导致黑高降低。

当删除的节点v是黑色节点，且其被黑色子节点替换时，其子节点就被标记为双黑：

删除操作的最主要任务就是将双黑节点转化为我们的普通黑色节点。

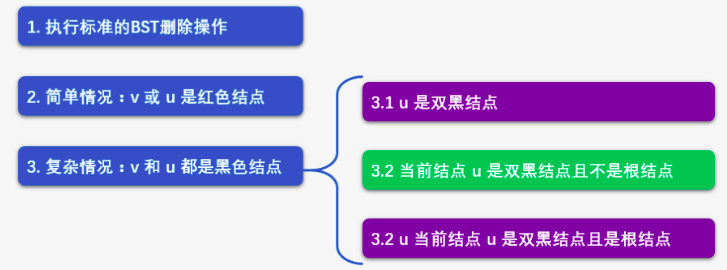
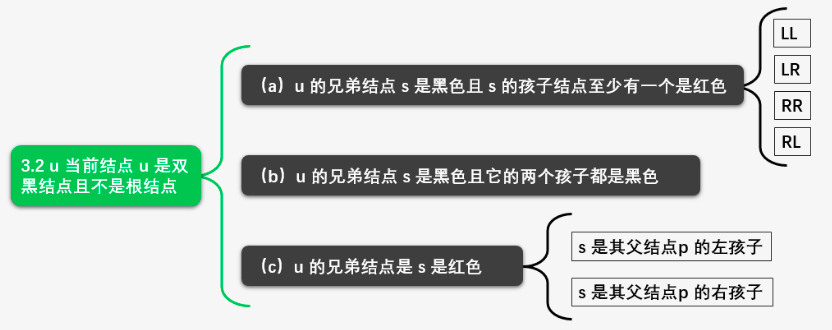


图中，为了方便，双黑节点直接变为了叶子节点，试想一下不是叶子节点的情况，你就能想明白了。

既定删除节点为v，u是用来替换v的孩子节点：

删除操作总纲：

1. 执行标准的BST的删除操作。
2. 简单情况，u或者v是红色
3. 复杂情况：u和v都是黑色节点

1. 标准的BST删除操作：

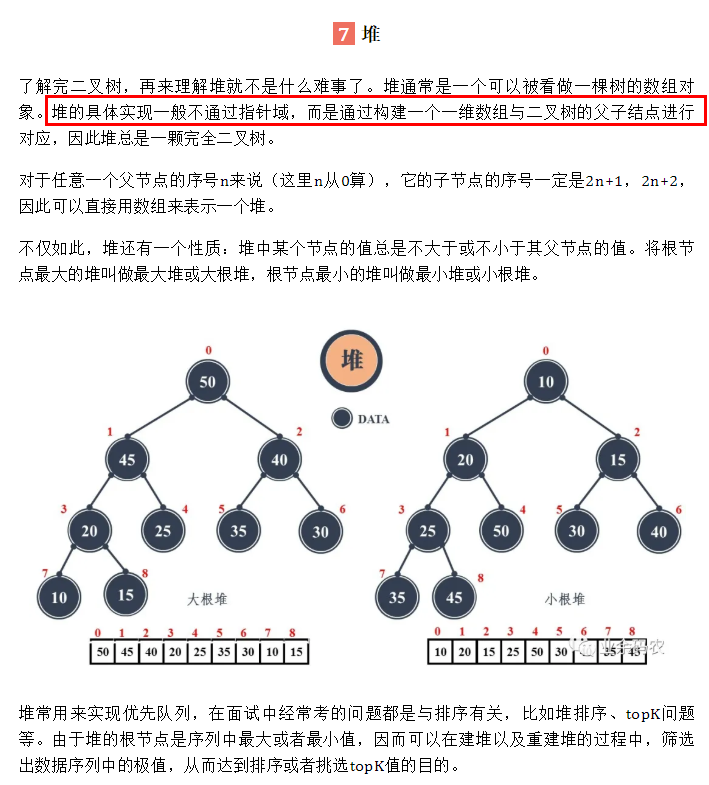
值得一提的是，最终都会以删除一个叶子节点或者只有一个孩子的节点而结束。

1. 简单情况：u和v有一个是红色节点

将u替换v，并置黑

1. 复杂情况：u和v都是黑色节点

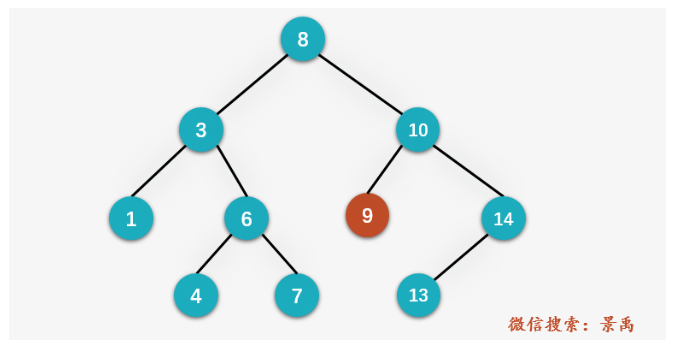
## 5、堆

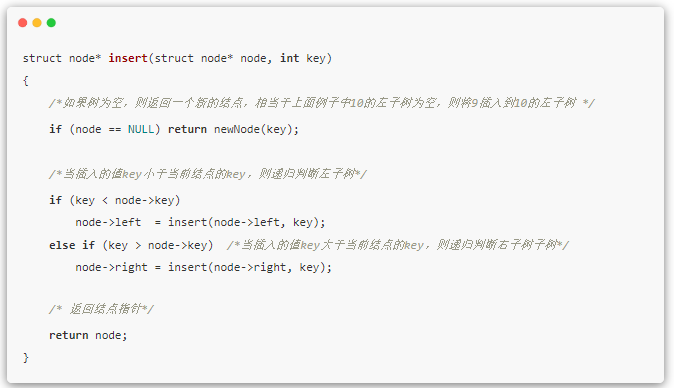


## 树

### 二叉排序树的插入

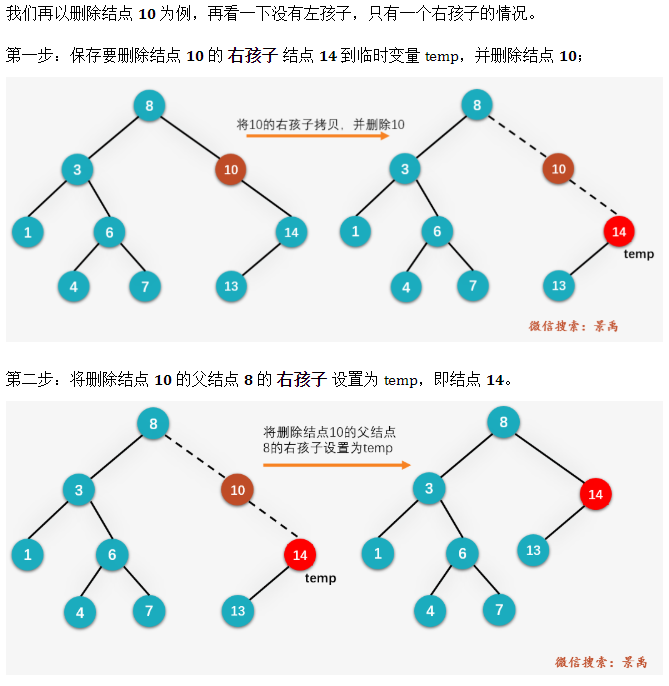
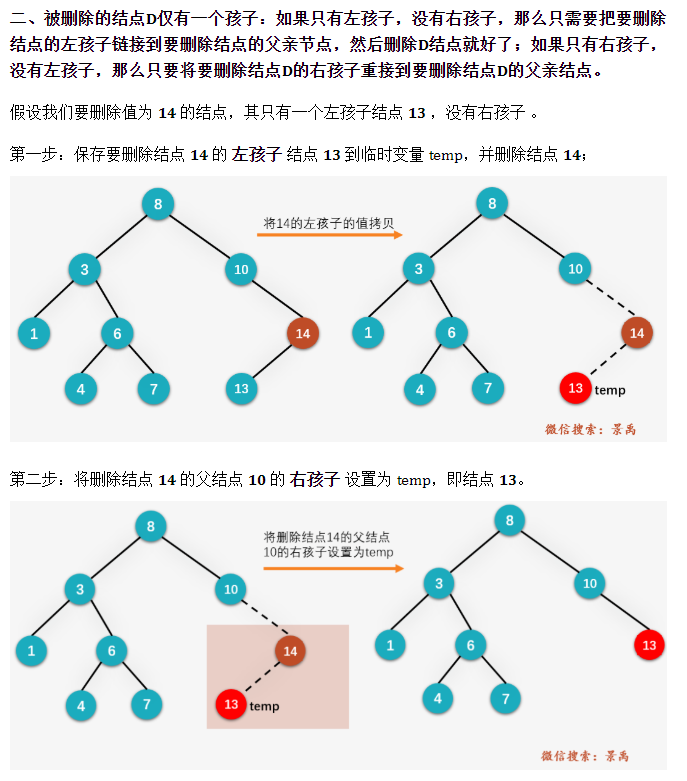
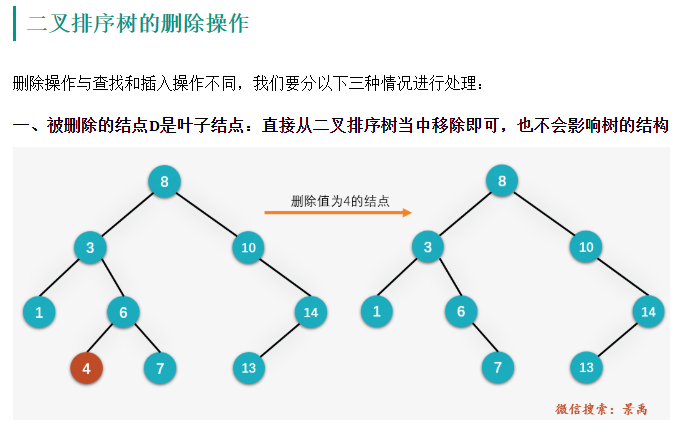
二叉树的插入其实和查找没什么区别，因为给定一个节点，进行插入时，你细品一下，一定插入的位置会是叶子节点的。





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。**

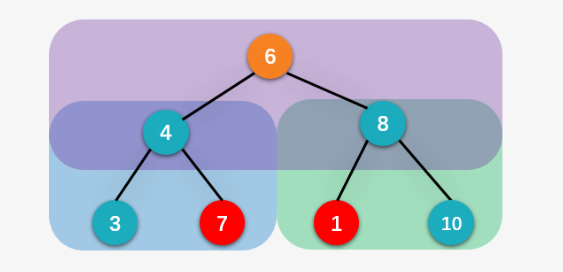
1. **删除其实也是重新构建二叉排序树的过程**

### 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：

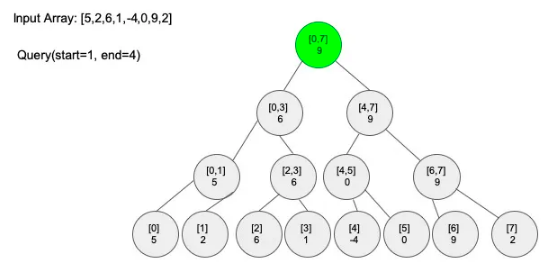




这样一棵树其实不是二叉排序树，但是返回True

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。

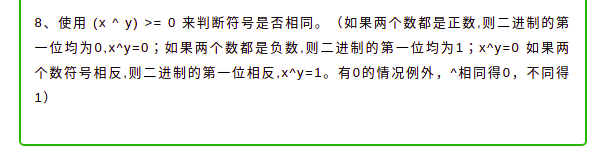
## 线段树



存的是区间和该树的最大值

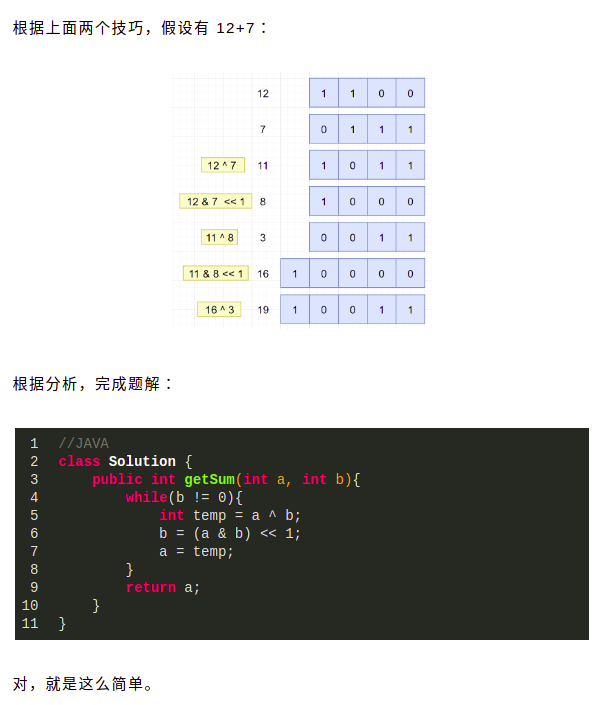
## 位





具体实现看代码： 数据结构--位

### 两数之和

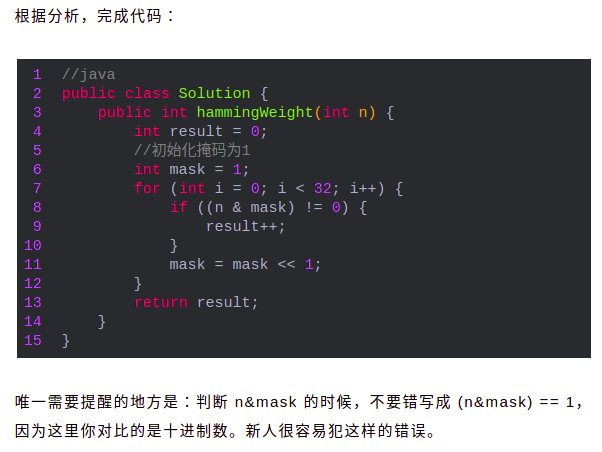


### 是否是2的幂



### 汉明距离





而且 *# 最好将1向左移动，而不是将数向右移动，因为算术右移的话，左侧补的是符号位*

### 只出现一次的数字（其余的数字出现也是奇数次）

## 链表

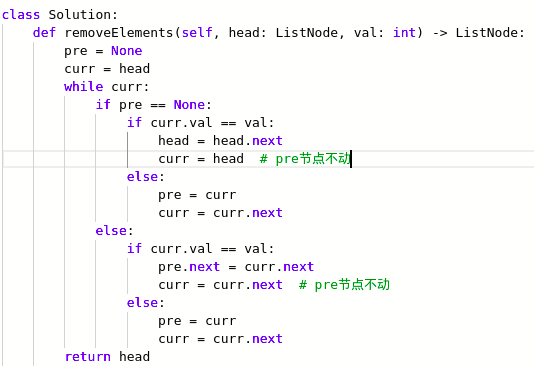
### 链表框架



链表框架最好加上newHead，newHead.next = head， 然后pre = newHead，这样就不用解决pre == Null的问题了。

### 移除链表元素

这个不能照搬链表框架，因为当删除元素时，pre会指向那个删除的元素



### 反转链表的k个元素：

这是一道运用原来的反转链表思路会超时的题目：还是运用我之前的解法解法比较好，因为要求时间复杂度为O（n）。运用改良过后的链表框架解题，值得注意的几点是：

1. 要记录开始反转时的链表的节点
2. 当前节点cur通过pre.next来获得。
3. 当index == n时，直接返回newHead.next即可