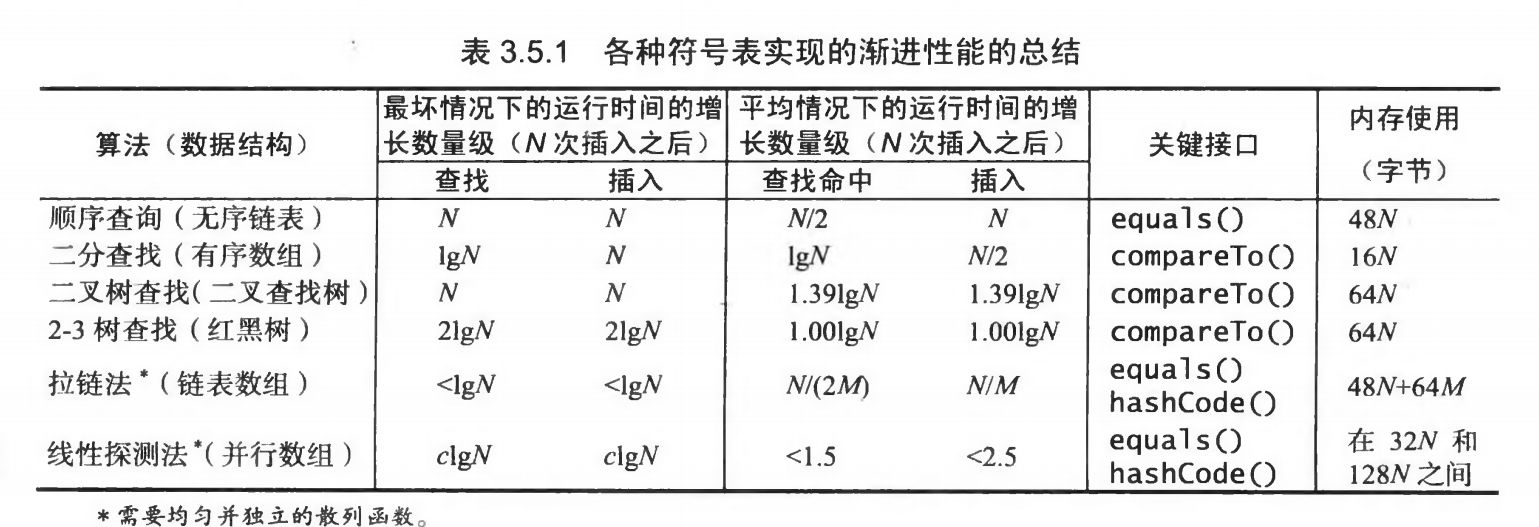
**查找算法对比**



FrequencyCounter （大量频繁使用查找算法的代名词，庞大的输入）

符号表中不允许存在相同的键，但如果插入的是相同的东西，可以在value上加一解决。

**无序链表**

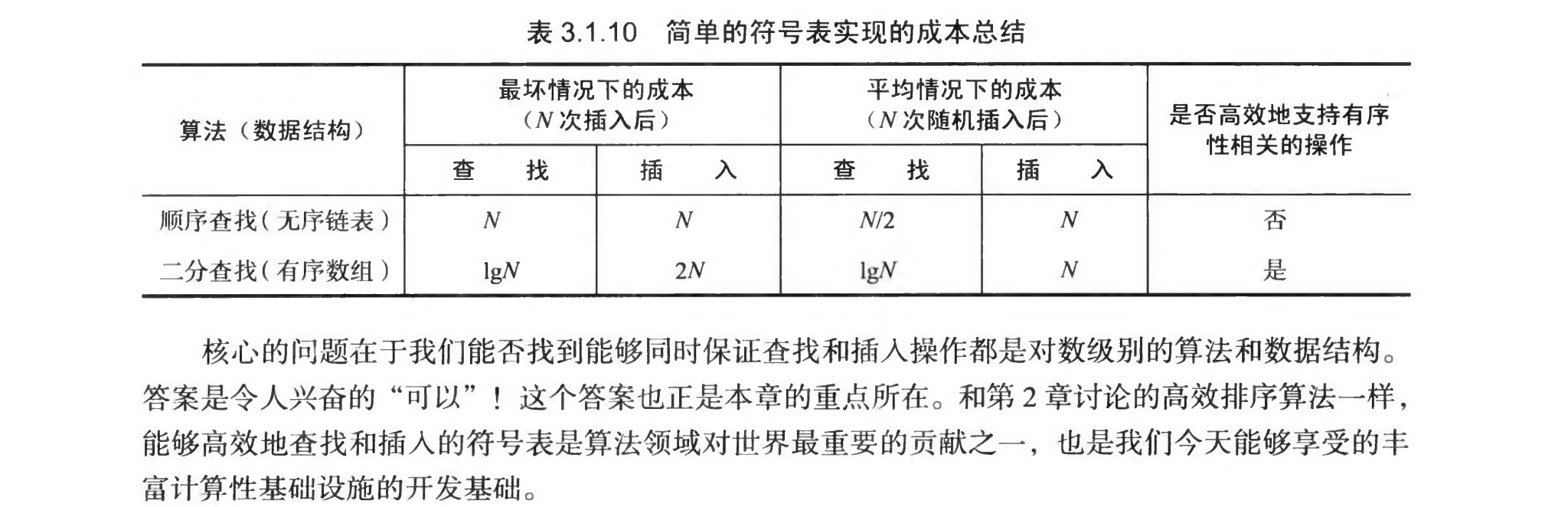
顺序查找（低效）

n个不同的值插入到空表中（每插入一个需要比较一次所以需要N (2) /2）

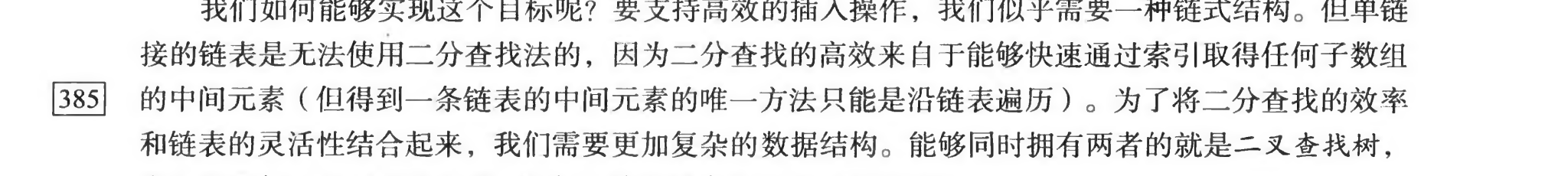
**有序数组**

二分查找（log（n+1））

依旧不能支持（已经达到了对数级） FrequencyCounter



**二叉查找树**



**特性**

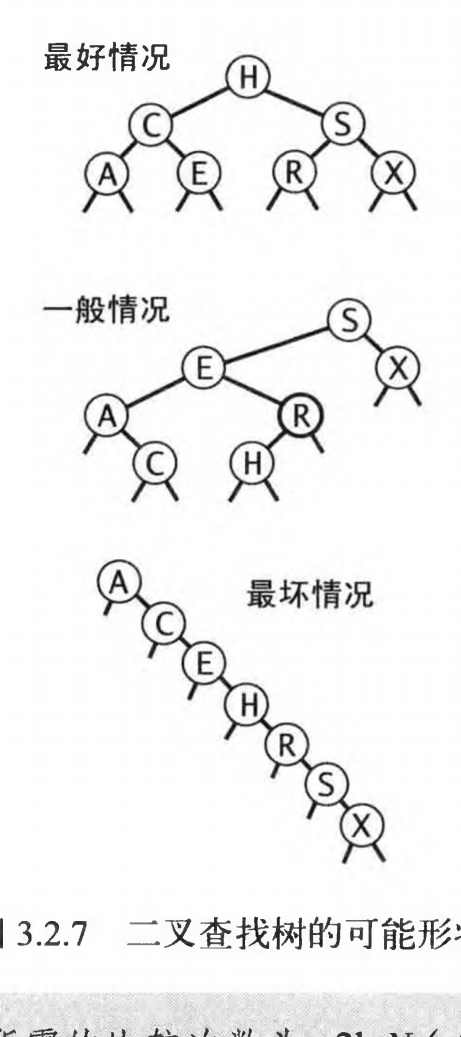
左节点小于右节点的一个二叉树

每个节点变量 N都记录这有多少子节点（所以我们添加一个节点需要更新这棵树上每个子节点N）

**性能**

算法运行时间取决于树的形状

如果输入的是无序随机的数，性能可以保证在小于3logn，有序输入，效率较低。因为这样会形成如图片中的数（最坏情况）



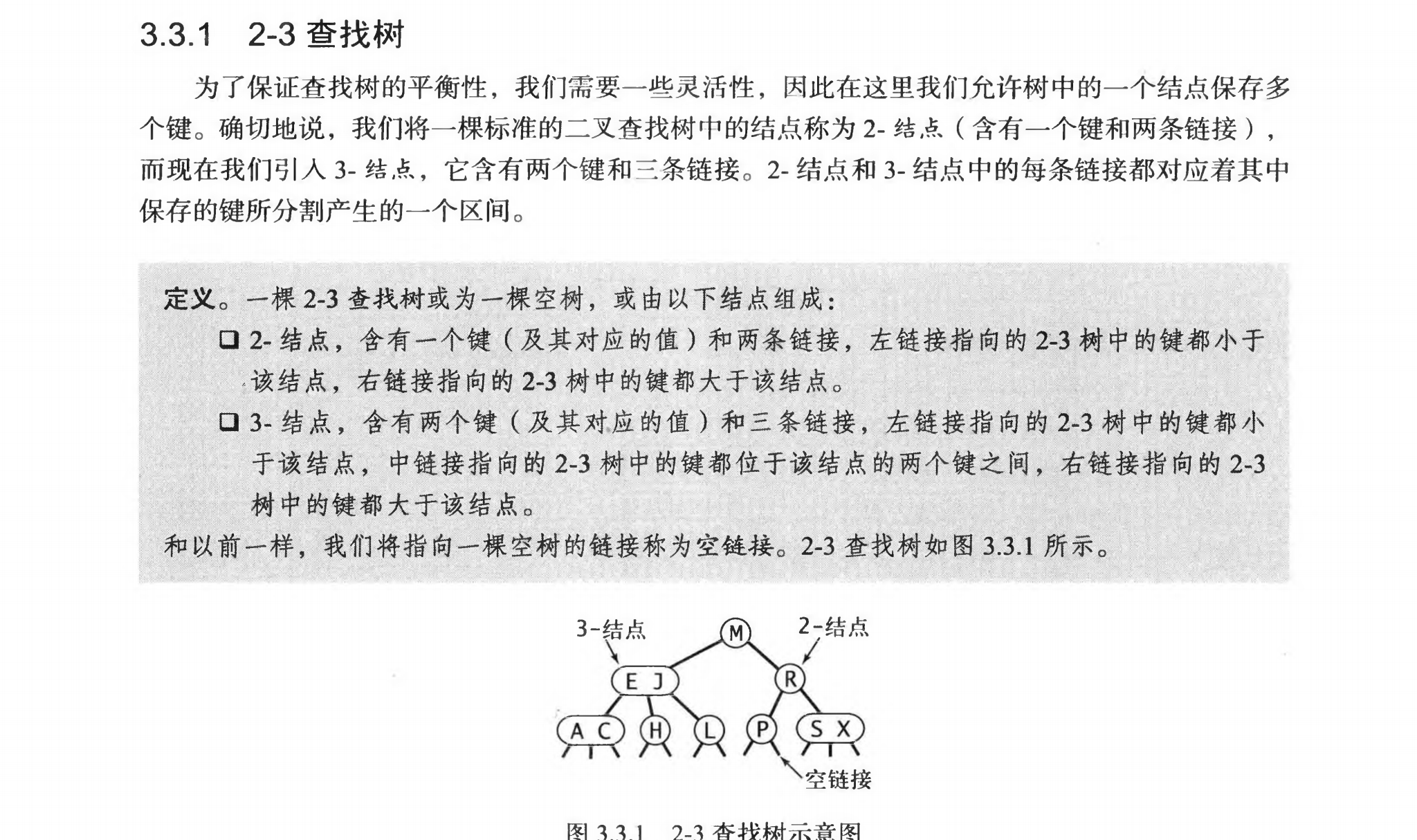
**难点 在于如何删除一个节点**

如何该节点有两个子节点 把该节点的后继节点替代他，（这个后继节点是按中序遍历来的）



**平衡查找树（无视输入顺序，对数级别）**

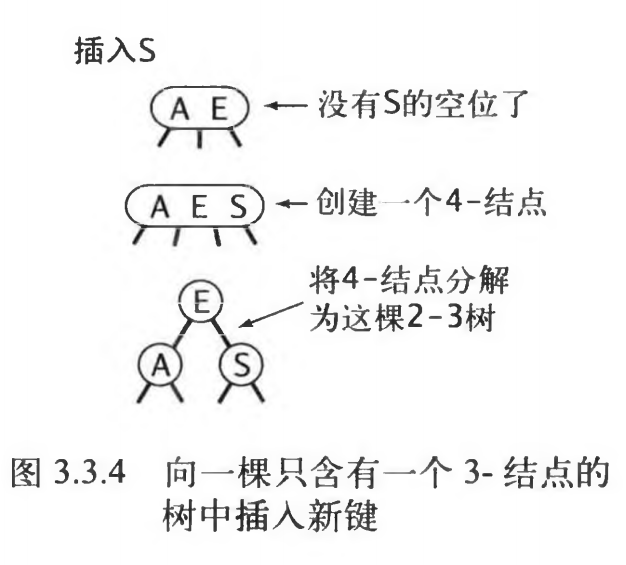
前面有点算法好的时候十分优秀，但是最坏情况无法接受



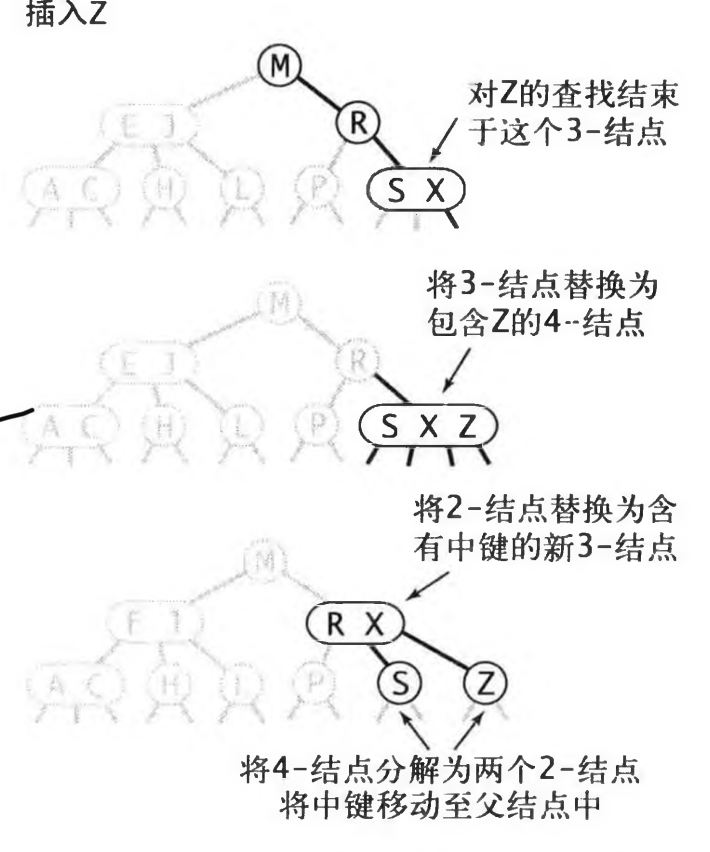
**插入（先变成4—节点然后慢慢搞）**

1.一颗只含有3节点的树插入新键

把他先这个3—节点变成4—节点，在转换成3个2—节点



2.向一个父节点为2—节点的3—节点的树中插入新键

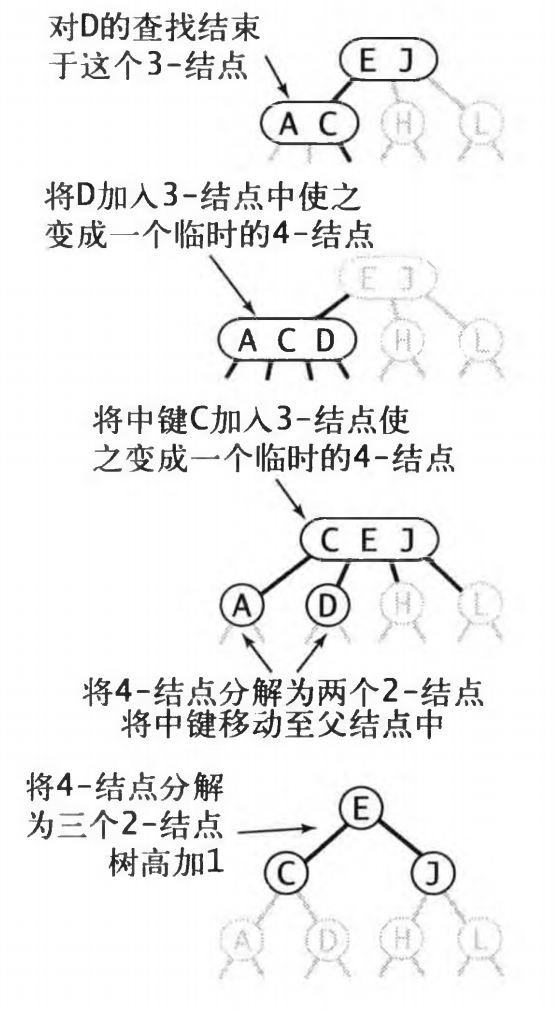


3.父节点为3—结点的3—结点中插入新键

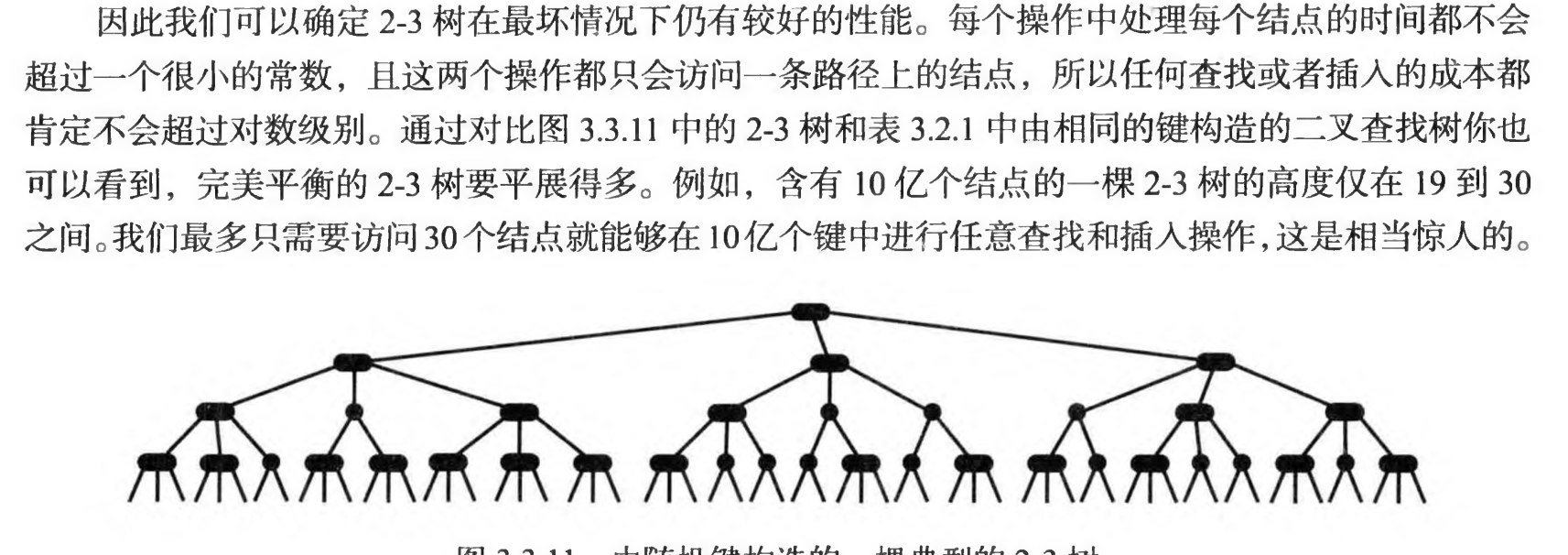
04

**4.分解根节点**

如果从插入节点到根节点全是3—节点，根节点会变成4—节点，我们就需要分解3个2—节点，树高加一。



**性能**



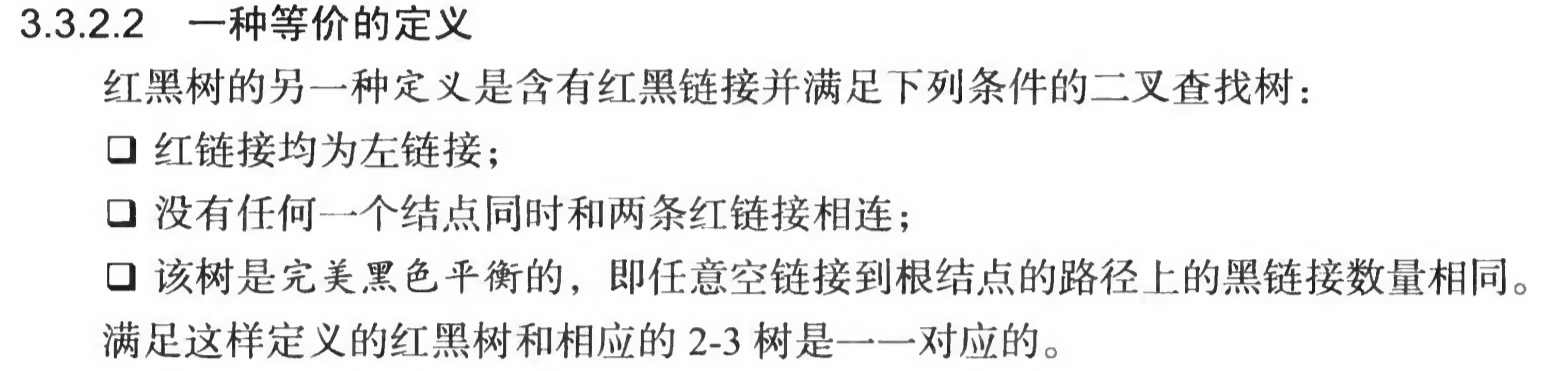
**缺点**

我们上面分析了很多种情况，所以处理极其不方便，同时要维护两个不同的结点，但是他的性能毋庸置疑。

他的数据结构名字就叫做红黑二叉查找树。

红链接表示的是一个2-3树，而不是代表那个比那个小，这个红黑二叉查找树，我们注意这个名字中的2，表示它只用了2节点查找树实现了3节点，厉害啊！

**特性**



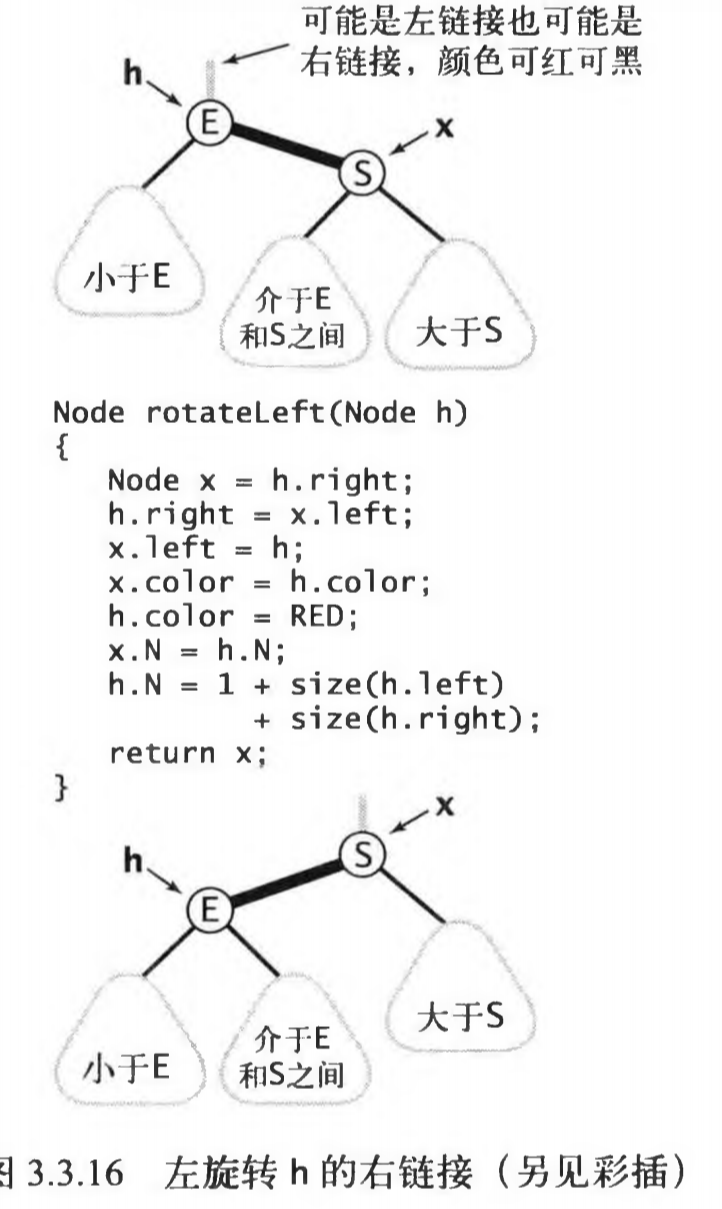
**调整（所以总的操作只有三个**

左旋转对应第一种情况

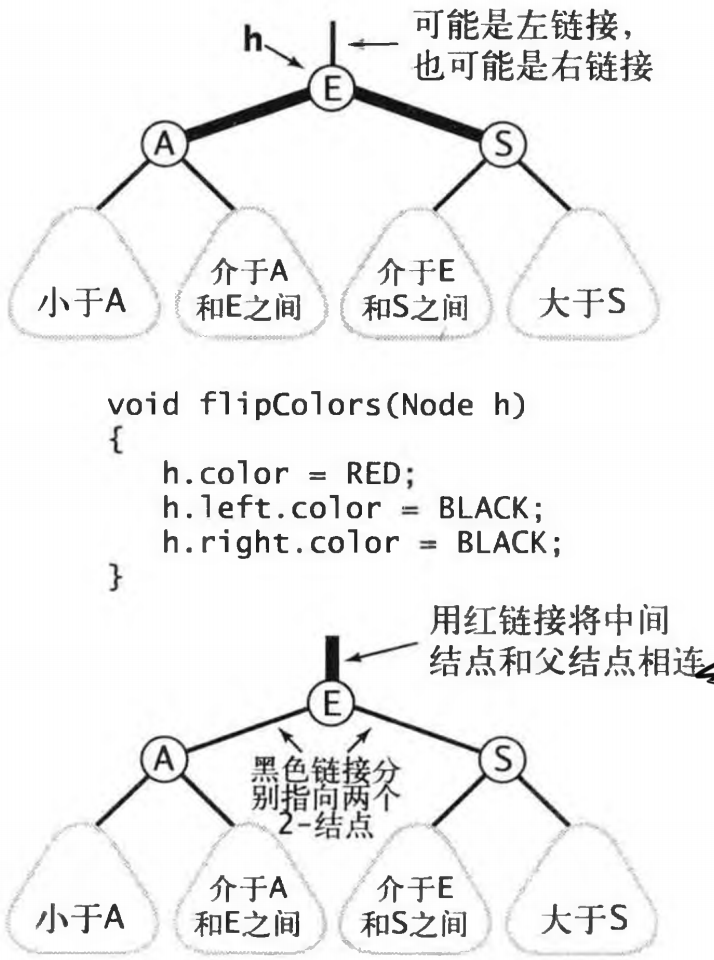
右旋转对应 左边连续两个红色

颜色转换 对应第二种情况）看p280

第一种情况 如果右子数为红色，我们就需要调整



第二种情况 如果左右子节点都是红色，所以我们把左右子节点变黑色，然后根节点变成红色。



如何保证红黑二叉树 有序性 看（p435）

**删除（很复杂）**

**散列表**

将我们想要保存的键转换为我们数组索引（关于如何转换 这个不要我们去考虑，由计算机专家来就行要相信他们）

散列表是时间与空间之间的妥协，我们可以直接将键作为（可能是一个超大的）数组的索引，那么所有查找操作只需要访问内存一次即可完成。但这种理想情况不会经常出现，因为当键很多时需要的内存太大。调整散列算法的参数就可以在空间和时间之间作出取舍。

**要为一个数据类型实现一个优秀的散列方法需要满足三个条件：**

一致性——

等价的键必然产生相等的散列值；

高效性——

计算简便；

均勾性——

均匀地散列所有的键。

**第二步是碰撞处理**

**基于拉链法的散列表**

**拉链法**的办法是将大小为M 的数组中的每个元素指向一条链表，链表中的每个结点都存储了散列值为该元素的索引的键值对。这种方法被称为拉链 法

在基于拉链法的散列表中使用大小为M 的数组能够将查找和插入操作的效率提高M 倍

散列表存入键后，将其均匀分开所以键与键之间没有顺序可言，找最大和最小键都不适合

基于线性探测法的散列表

//使用线性探测解决键冲突问题，核心思想就是利用数组空位，当使用

//哈希函数计算出数组索引后，开始检测该索引位置是否已经被使用，如

//果已经使用，则索引向后递增（如果超出数组索引，则需要进行回绕到

//数组起始索引），直到找到空位为止，或者数组空间已全部被使用，无

//法再扩冲，由于这里利用的是数组空间

性能

**线性探测的平均成本取决于**

和拉链法一样，开放地址类的散列表的性能也依 赖 于 的a称为散列表的使用率

元素在插入数组后聚集成的一组连续的条目，也就是如果这个数组插入过多的键以后，那么我们就如果重新插入一个键时，发现冲突，然后向后寻找空的位置，发现在这个插入的键的位置后面，有一段很长连续的位置都不是空的，那某这个要插入的键就需要超越这段连续位置，往后找，如果这段连续位置越长，他就需要更多的查找，消耗跟多的性能，

而这个数组容纳键的饱和度越高，越容易产生这种连续片段。为了保证性能，我们会动态调整数组的大小来保证用率在1/8到1/2之间

**但当使用率a 小于 1/2时探测的预次数只在1.5到 2.5之间。**

方法

put( ) 方法中的第条语句会调用resize（） 来保证散列表最多为半满状态。这段代码构造的散列表比原来大一倍