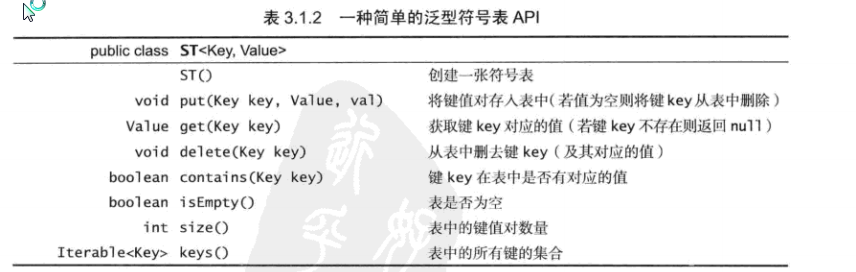
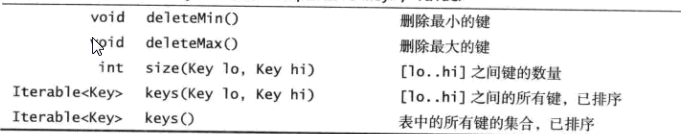
符号表  


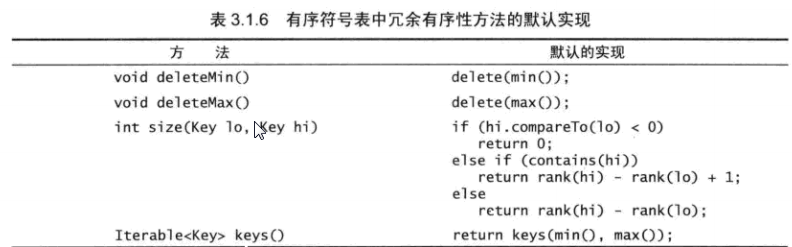
使用链表或者数组实现

有序符号表:即存储的key是支持排序的key

增加了API





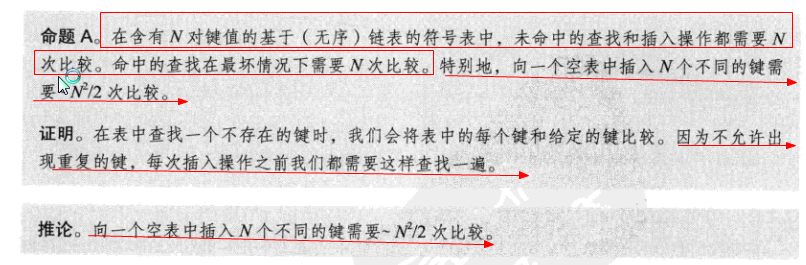


第二部分:基于链表而实现的无序链表

1.get方法会遍历链表,直到找到要查找的元素为止。

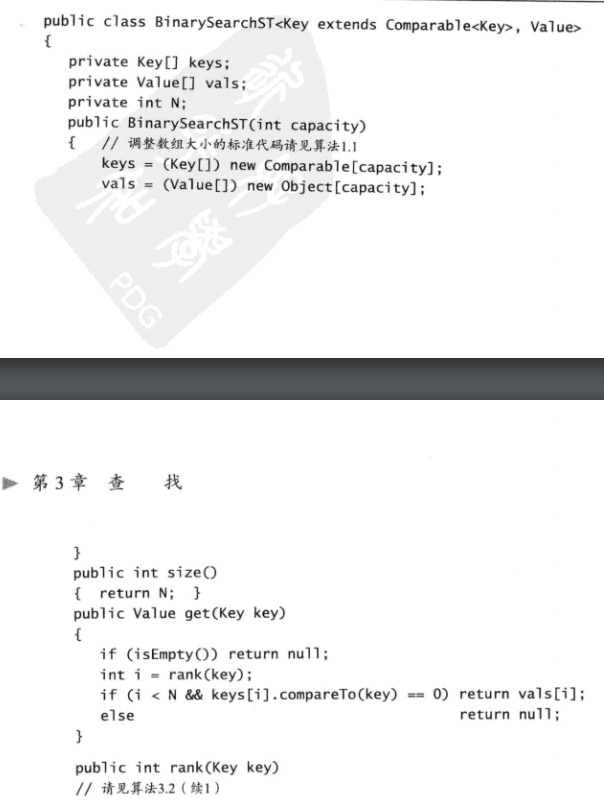
2.put方法也是要遍历链表,找到相同的key,则更新对应的value,否则在链表的头部插入新的key-value。

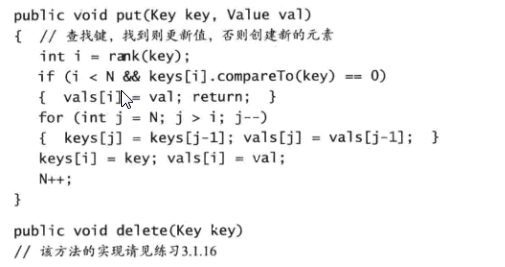
3.耗时总结---无序的是非常耗时的

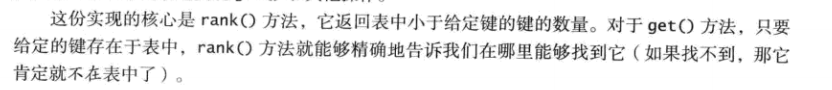




1.实现:key和value分别两个数组存储,key是有顺序的,因此找到key的下标后,直接读取value下标即可。







说明排序规则是从小到大排序的,因此rank返回的就是找到后的下标

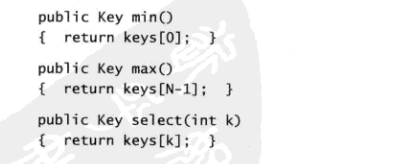
Put方法在找到后直接更新值。如果找不到,则每一个值都移动一位,然后在i位置插入该key-value,增加N的元素数量。

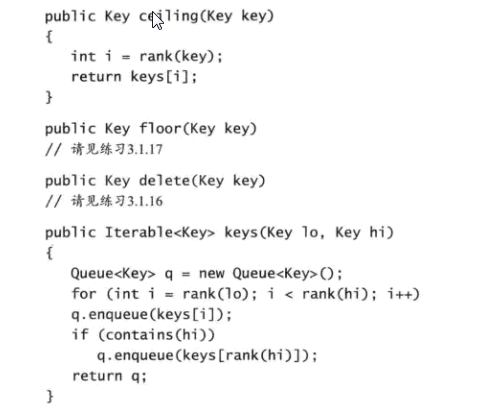
因此可以看到压力在rank查找的速度上,因此get取决于rank方法。而put不仅取决于rank,还有移动的问题,如果移动的元素很多,也会影响性能。

Rank方法实现使用二分查找方式,因此可以性能很好的找到位置.二分查找针对的是有序的集合。

其他API的实现性能上也很高。

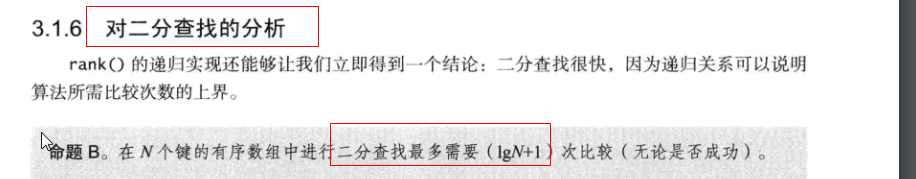
因为已经排序好了。

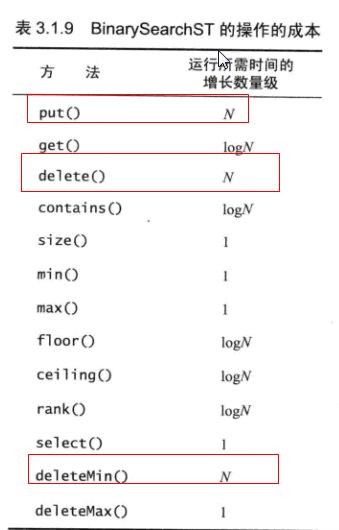




注意删除方法也会有性能消耗,取决于rank以及元素位置的移动。

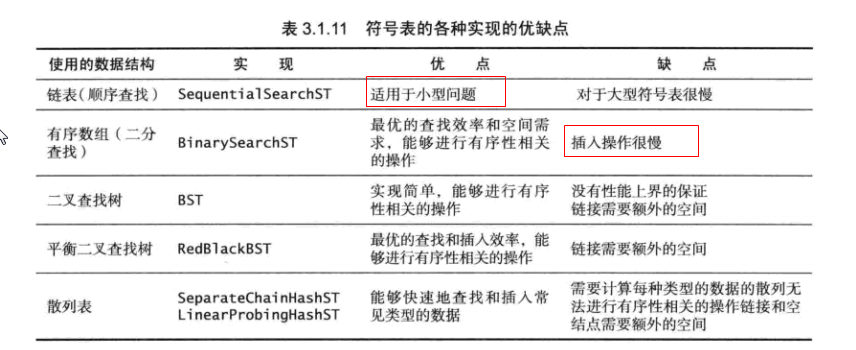
Rank很快,但是位置移动在数据量很大的时候是很慢的。





可以看到新增和删除的时候都是因为移动位置,导致很耗时.在频繁新增和删除的时候使用是有风险的。

总结



可以看到顺序的链表因为在数据量很大的时候是会很慢的,因此适用于小型数据量时候。

而有序的数组虽然查找很快,但是插入速度又很慢。

我们接下来讲解的树.可以解决查找和插入都是对数级别的。因此树才是最理想的结构

Ps:有序的时候为什么不是链表,链表可以保证插入和删除很快.,只跟相邻的节点有关系,不需要移动所有数据。但是查询的时候虽然有序,但是没有办法使用二分法,因此要一个一个查找,这个是非常耗时。而二叉树其实就是一个链表,对有序的链表进行处理的方式,这种链表是可以查找很快的。



1.含义:又称BST Binary Search Tree

a.顾名思义，就是两个叉,即一个节点有两个分支。而与链表的区别是,真正的链表是一个节点对应一个连接.而二叉树是一个节点对应两个连接。

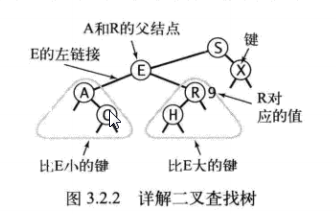
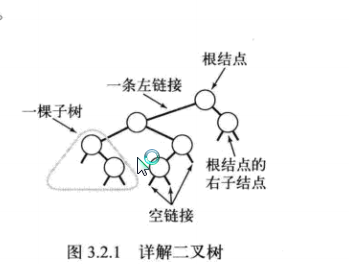
b.每一个节点的两个连接可以指向null或者其他节点,因此串联起来成了一种链表。

但是虽然每一个节点可以有两个子节点，但是他们只能有一个父节点

c.每一个节点存储的是一个key和对应的value

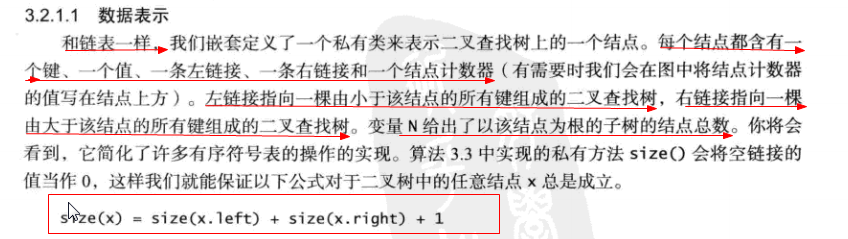
d.每一个节点都可以单独看成是一个二叉树,因此一个大的二叉树其实可以看成是若干个小的二叉树的组合。

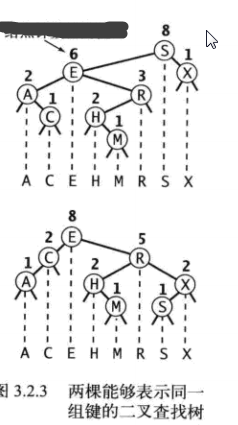
e.每一个节点的key都是可以比较的key.并且该节点的左子树<该节点<该节点的右子树



二.实现

1.size实现





同一数据可以有多种不同的二叉树。

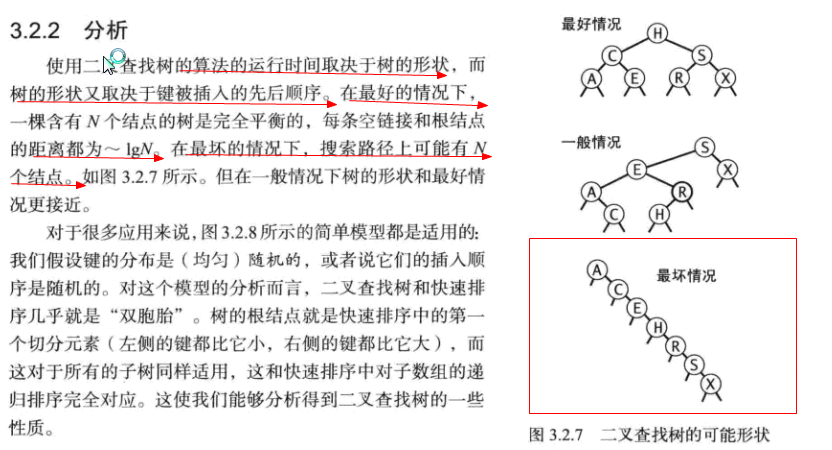
2.get实现 ----只要树是足够均衡,即高度不高,那么查找速度采用二分法一样,很快可以查询到结果。每次都从根节点开始查找,最多查找树的高度为止。要么找到,要么null找不到。

3.put 也是需要二分法查找.只要找到则更新该值.如果找不到,则创建新的节点,插入到子树中。插入速度是很快的,因为是链表,只要加一个连接即可,速度可以忽略,因此二分查找也很快,因此结果put效率很高。

注意:插入新的节点后.还要从下往上不断的递归,增加每一个父节点对应的计数器+1.因为该速度也是和get一样,因此该put方法相当于执行了两次get操作的时间。

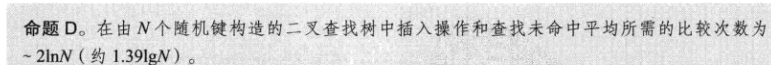
三、性能分析

1.二叉树的性能取决于插入数据的顺序。

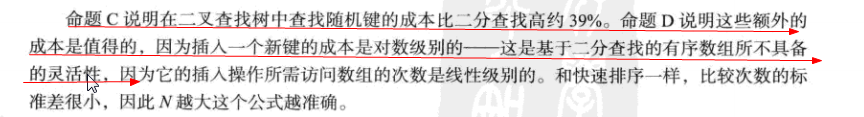


即构建树的时候一定元素是随机打乱的,如果本身就是有顺序的时候一定会出问题。









以上说明C 查找时间上比 有序数组的二分查找要高39%的损耗,但是D说明这损耗是值得的,因为他使得插入的成本也是对数级别了,而有序数组的插入是线性级别的,N越大,性能越不好。因此二叉树是有很大优势的。

而到底查询多少次取决于树的高度。树不均衡,导致树的高度越高。因此性能越不好。

四、其他方法的实现

1．找到最大值和最小值

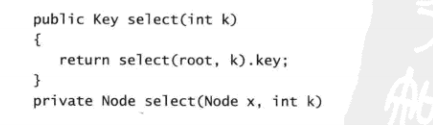
找到最左边的节点,即最小值。

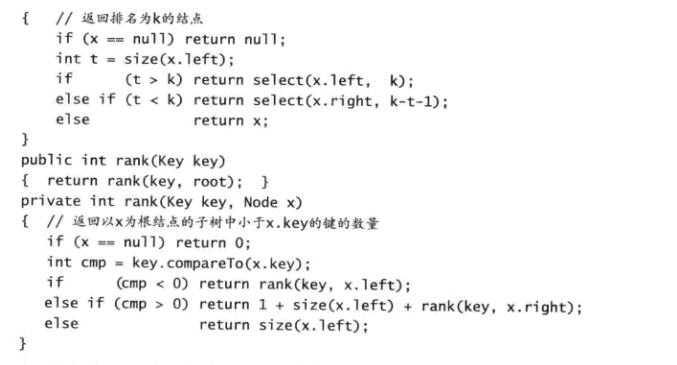
2．获取比key大的值,或者小的值,即向上取整和向下取整

也是查找方式,时间都是跟查找相关的,速度很快



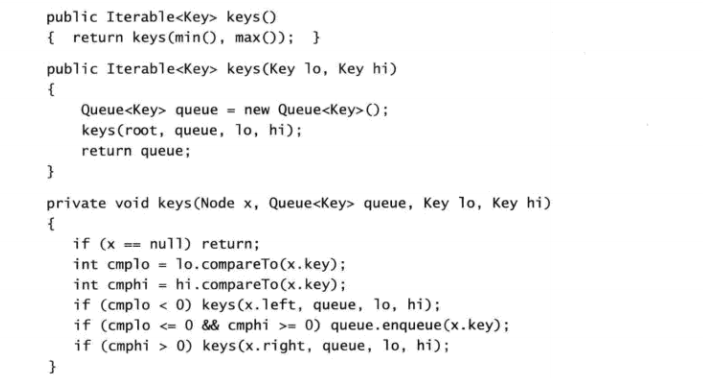
3．选择操作



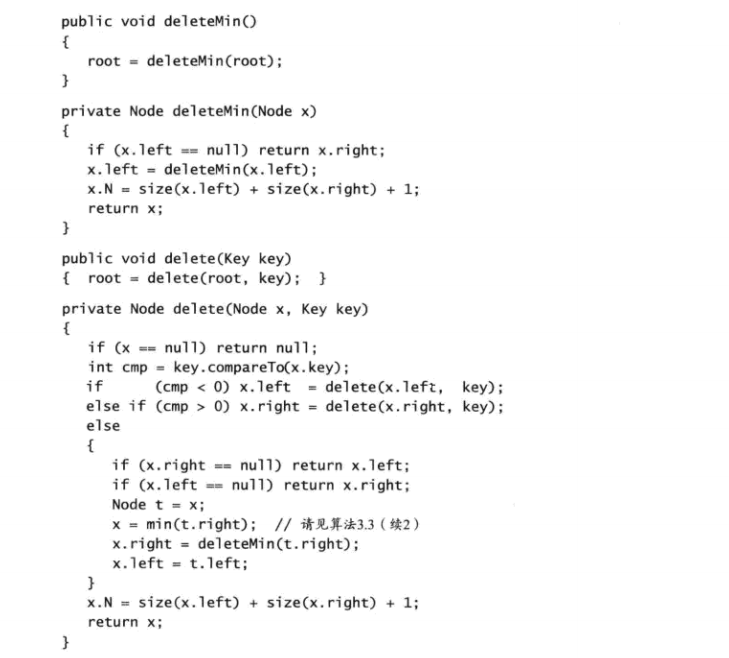


其实每一个节点有一个N,因此查找第几个元素,根据N就可以找到,也是一个二分法。

4．范围查找,使用一个队列把满足条件的都添加到该队列即可,时间也很快



5．删除操作---这个比较麻烦



1．简单的先删除最小值和最大值

就是递归找到最小值和最大值,然后删除掉链表即可.然后与查找路径一样,更新计数器。因此是2倍的查找次数。

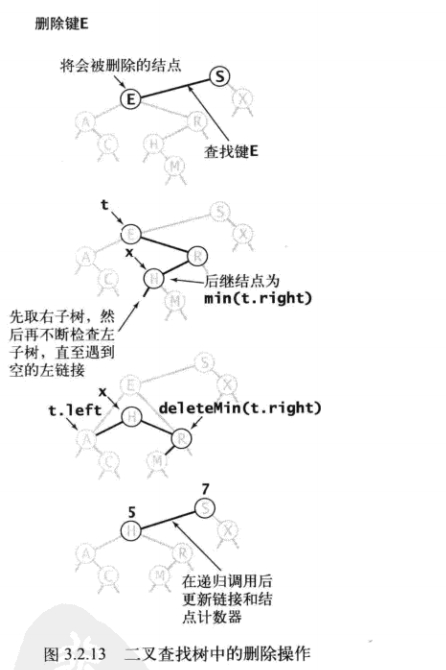
2．删除任意节点

如果删除的节点是没有子节点或者只有一个子节点,虽然不是最大值也不是最小值,但是删除即可,然后更新计数器,没有什么额外的影响。

但是删除的节点拥有两个子节点,那就比较麻烦了

以下例子:因为要删除E,但是他有两个节点,左节点肯定都比E小,右节点肯定都比E大,因此把E删除后,从右节点获取一个最小值,将其替换E即可。树依然没什么变化。

但是右边最小的节点可能也有右节点,该右节点怎么处理呢?不需要关注怎么处理,只要将最小节点删除,因此删除后自然就等于处理了。即按照删除逻辑走。



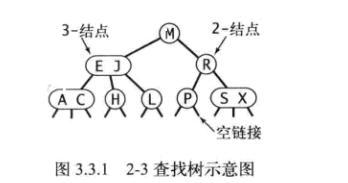
删除操作性能没什么问题,只是2倍多一些的查找时间,但是理解起来挺麻烦

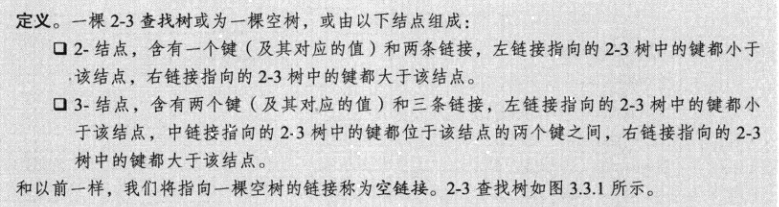
五、备注总结

Ps:现在是二叉树的元素添加顺序影响了树的形状,但是他不是均衡的二叉树,如果二叉树均衡的话,那么插入性能就不会受到添加顺序影响,那么就完美了.因此这个就是红黑树解决的问题。本质上红黑树也是二叉树,他只是解决了无论数据什么样子,插入的结果都一定还是均衡的二叉树。



一、为了平衡性,允许一个节点保存多个key,因此就不是二叉树,而是3叉或者4叉树。





注意:图上最多一个节点上有两个key,一旦有3个key的时候是内部会进行转换的,拆分成多个树。

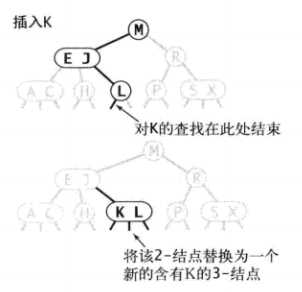
二、查找

这个很简单,就是跟二叉树一样,只是不是二分,是根据大小比较后,选择左边还是右边还是中间。

三、插入

1.向2节点插入新值

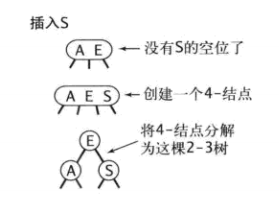
这个很容易,因为一个节点上可以有两个key,因此直接插入到该节点上即可



2. 

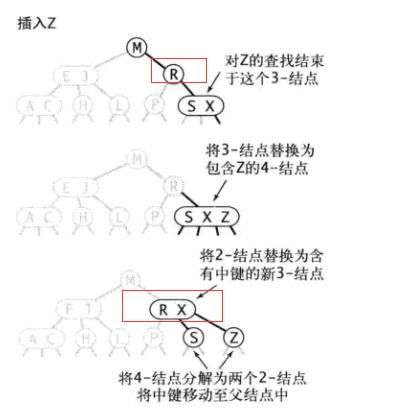
因为一个节点已经有2个key了,插入一个key后,变成3个key,然后将其转换成一个平衡树。

这样插入后高度增加1。三个key中间的key作为根。



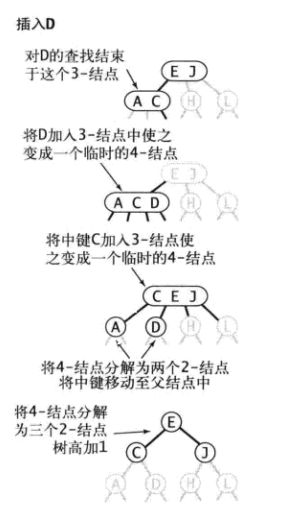
3. 

即父节点是一个key,子节点是2个key,此时向子节点插入新值。因此子节点跟上面2讲的一样,会将中间的key作为根节点,此时会插入父节点中,因为父节点有一个key,因此此时父节点有2个key,树的高度没变化。



4. 

即父节点已经存在2个key,子节点也是2个key,因此子节点插入后,会向父节点提供一个key,此时父节点也会有3个key,他也会向上进位,直到存储成功,否则到根节点后,根节点依然是3个key,那么就会产生一个新的根节点,因此高度+1。



性质

1．局部变换

我们可以看到上面的变化中,都是在局部变换的,并不影响其他节点树。而且因为树足够平衡,因此变化的次数也是在合理的常数范围内的。

2.全局性质

这些局部变换,不会影响树的全局有序性以及平衡性。

3.标准的二叉树是从上向下生长.而平衡树是从下向上增长。



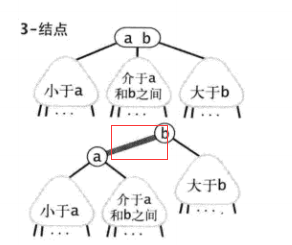




1. 定义

是平衡二叉树的一种实现,不是使用三节点、四节点的，而是用红和黑一个boolean变量即可完成。但是树不是非常平衡，但是也基本上很平衡，不是非常平衡的目的是要最终根二叉树一样,因此二叉树的API都可以用在这里,不需要关注红黑变量。

二、粗体的先表示红线

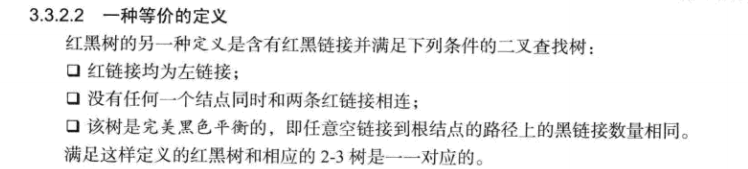


红线主要替代的是一个节点有2个key的节点。因此我们看到图,如果把红色线拉平，其实他就是ab两个key组成的一个节点.但是用红线标识后,就表示成一个二叉树了。

三、规定

规定红色线都是左连接

看高度是看黑色的高度，红色的不关注。即所谓的平衡是黑色线平衡了,因此二叉树来看因为有红色线,所以不是非常平衡的树。但是已经很平衡了



四、具体实现

主要是左旋 右旋以及换颜色来插入时候进行转换的.具体看的不是特别懂，网上找了一些资料到是看懂了，这段文字忽略，有时间仔细看一下。

忽略部分:287-296页

五、红黑树性质



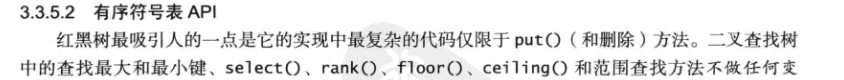
为什么是2呢,原因是他并不是绝对的平衡树,因为红色节点的存在,而红色节点又不计入在平衡树里面,因此二叉树会比lgN多一些,大概是2倍,虽然是2倍,但是是对数的2倍,因此性能也会很高。



这个是经验值,可以看到经验值就是lgN,基本上不受2倍的影响。所以性能会很高。

红黑树就是一种对插入和查找都是对数级别时间的算法,因此其他方法都是在put和get基础上操作的,因此也都能被控制在对数时间范围内。

范围查找 红黑树非对数范围,具体原因暂时不清楚,文章可能没详细看到。但是结论可以记住。



因为红黑树最后表现形式就是一个二叉树,因此二叉树的其他api实现可以不变,仅仅实现get和put即可.因此其他方法也都是在对数时间范围内实现的。



1．一个key可以转换成int,并且int很小,因此可以考虑用数组表示,数组的下标就是key对应的int,数组的元素值就是value.但是事实上并不是那么理想,有时候数据很大,导致数组很大,有时候不同的key会转换成相同的int,产生碰撞冲突

2.解决碰撞的方法



3．散列函数---将key转换成int



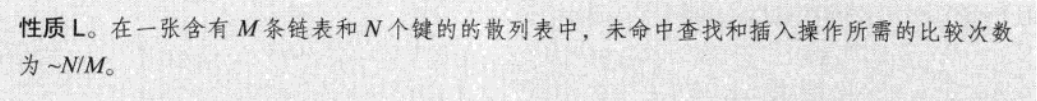
4．解决碰撞

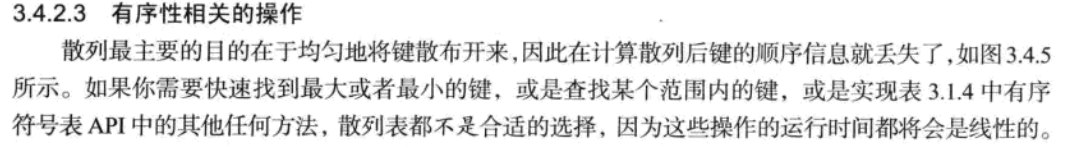


每一个key相同的数组,对应一组链表,

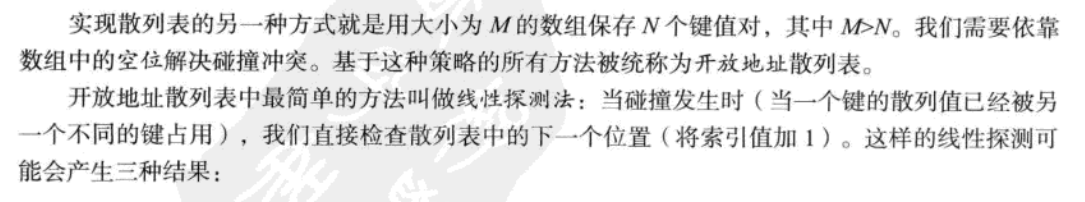
因为链表查找是顺序查,速度慢,因此此方法要首先保证散列函数对应的数组M的size足够大,以至于每一个数组的元素相对来说比较少,因此就会更加有效果。

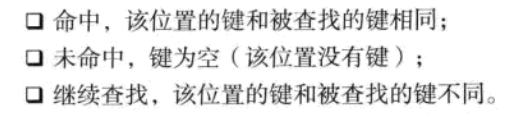
性能:











------------------------------------------------

先是无序的链表----查询和插入都慢。

改进成有序的数组---查询二分法很快,但是插入依然很慢。因为有移动数据的问题。

改进成二叉树---因为是链表形式,所以插入很快,查询又可以满足二分法,因此查询也很快.但是数据添加的顺序,会影响树的形状,导致不是平衡的二叉树,影响查询的速度。

改进成红黑树,使得无论数据添加顺序如何,都会最终是平衡二叉树。因此get和put的性能都能达到对数级别。

散列表:get查找和put会很快,但是在顺序方面查找最大值、最小值、区间等方面会很慢.因为被打算了,顺序操作相当于顺序一个一个找的过程,所以很慢。