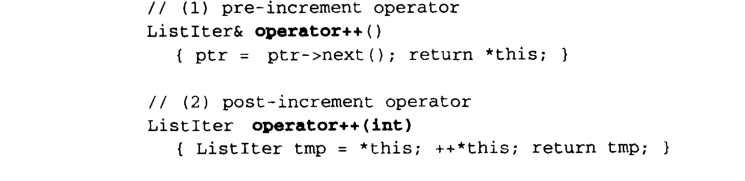
迭代器失效的总结：

Vector，如果发生空间容量变化，会经过重新适配，数据拷贝，释放原空间等步骤，之前的全部迭代器都会失效。

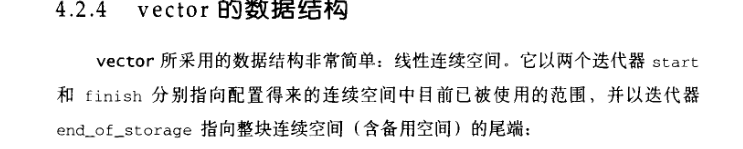
如果没有发生扩容，增，删等操作都会让作用节点之后的迭代器失效，因为操作之后后面节点会往前移动。

List ，因为是双向链表结构，增和拼接都不会造成其他节点迭代器的失效，删除只造成删除那个节点失效。

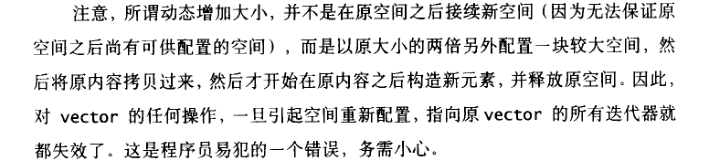
Deque由于是由多段连续的缓冲区连接而成，空间不够时分配新的缓冲区，所以迭代器不会失效，但是在缓冲区的首位时，会跳到其他缓冲区，实现较复杂。

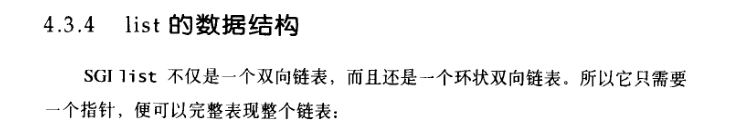
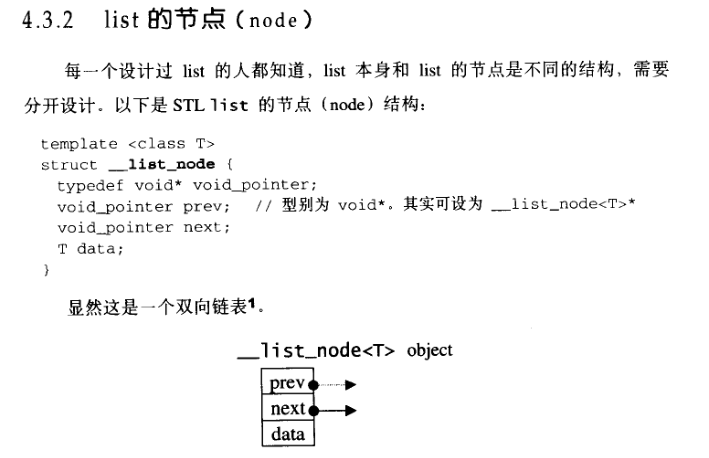
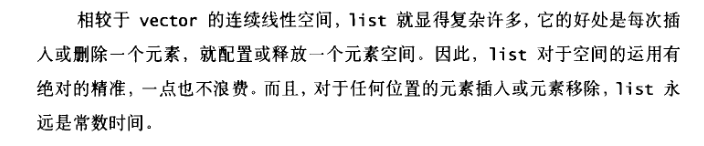


重载++ ，前置不要参数，后置要参数。前置返回自身++；后置生成临时对象，自身++，返回临时对象。

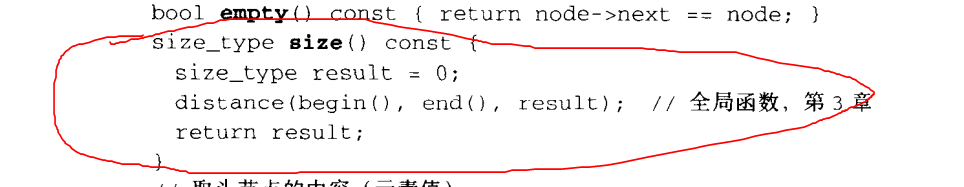








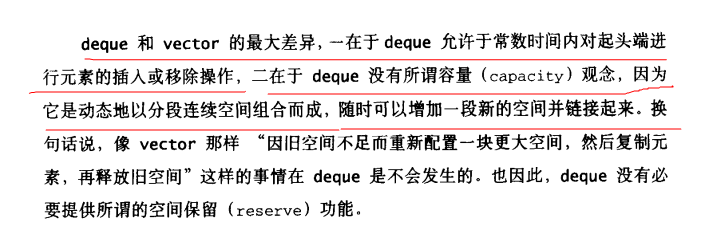
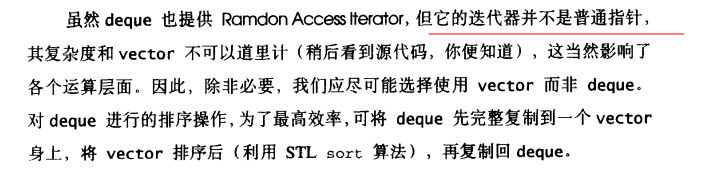
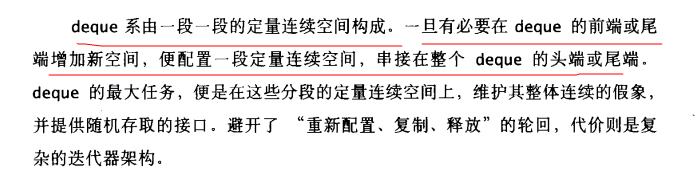
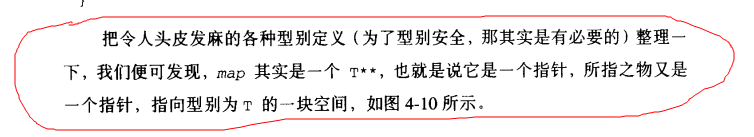
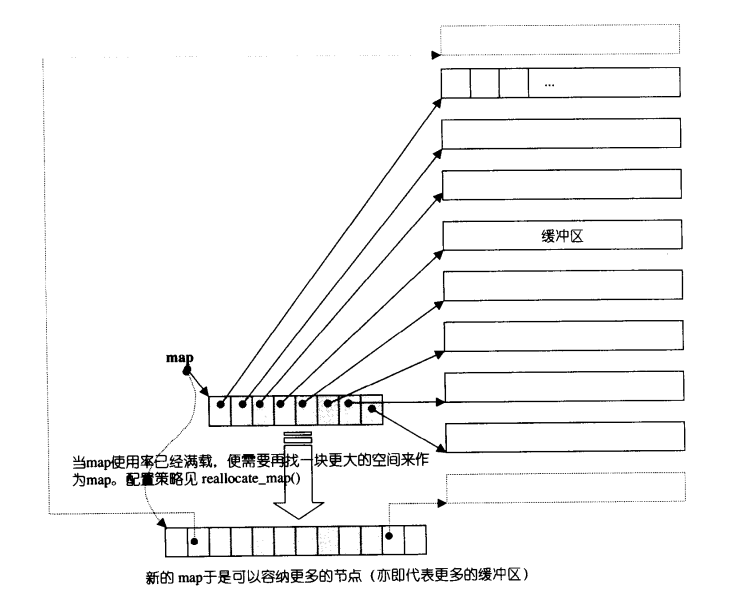
环状双向链表，因此内部需要一个指针指示头



List 的 size是线性时间，不是常数时间。

因为size和splice只能有一个满足常数时间，而另一个必须是线性时间

插入操作，是插入在选定节点之前

Deque由于是由多段连续的缓冲区连接而成，空间不够时分配新的缓冲区，所以迭代器不会失效，但是在缓冲区的首位时，会跳到其他缓冲区，实现较复杂。还有是它的中控成员map，map里面放了指向不同缓冲区的指针，但是map大小有限制，如果map已经放满了指针还要增加缓冲区，那么会重新分配更大的map区域，然后拷贝信息，添加，释放原空间。

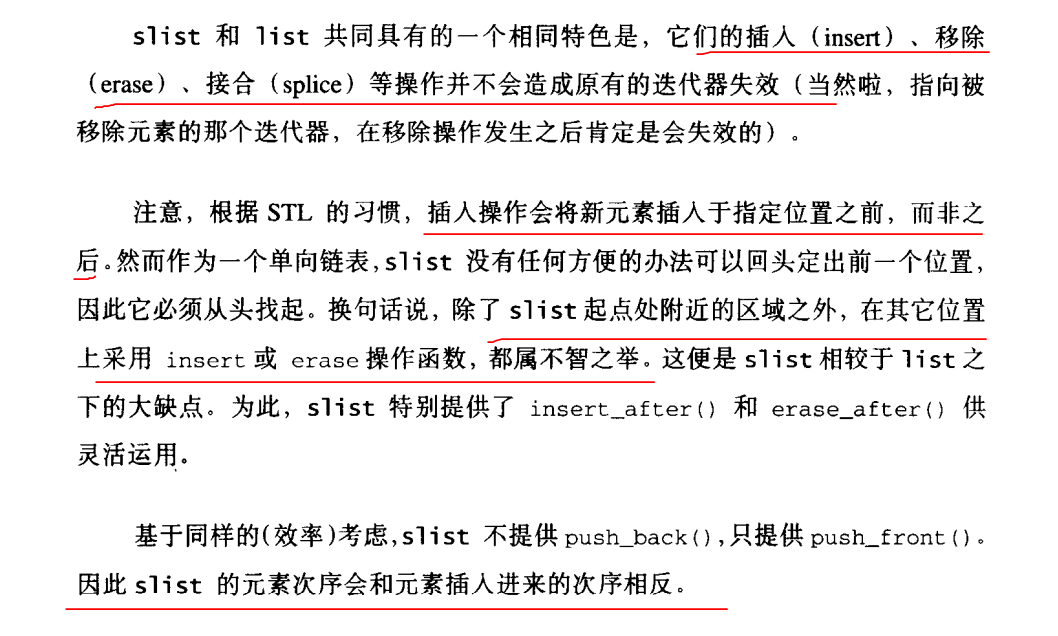
Stack可以通过deque和list作为底层实现

queue可以通过deque和list作为底层实现

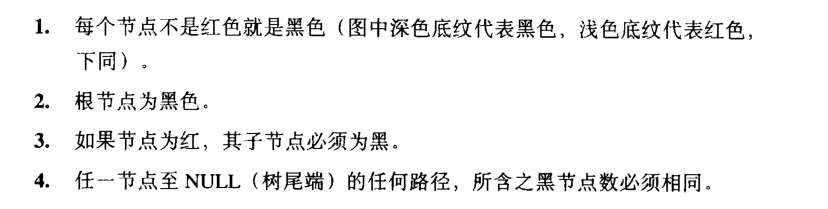
heap是完全二叉树，底层由vector实现。

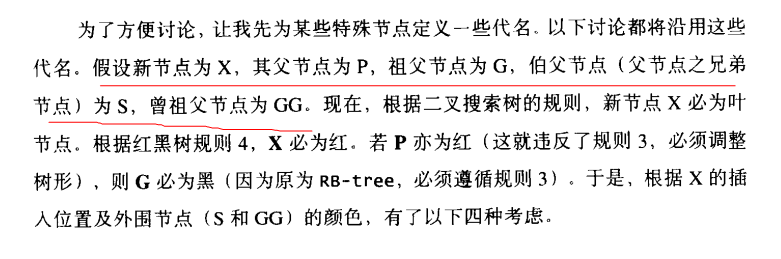
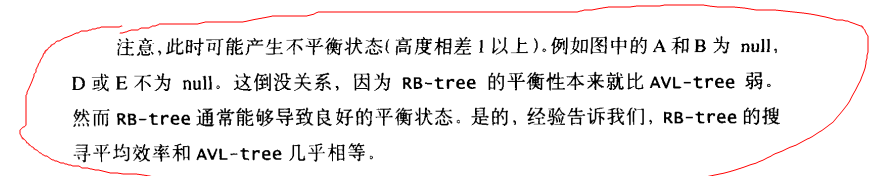
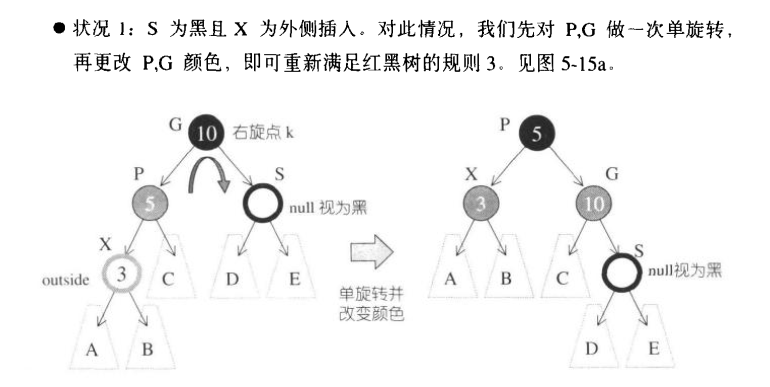
Priority\_queue 底层由vector作为容器，max\_heap作为规则，在定义对象时需要在第二三个参数提供 容器类型，比较规则。

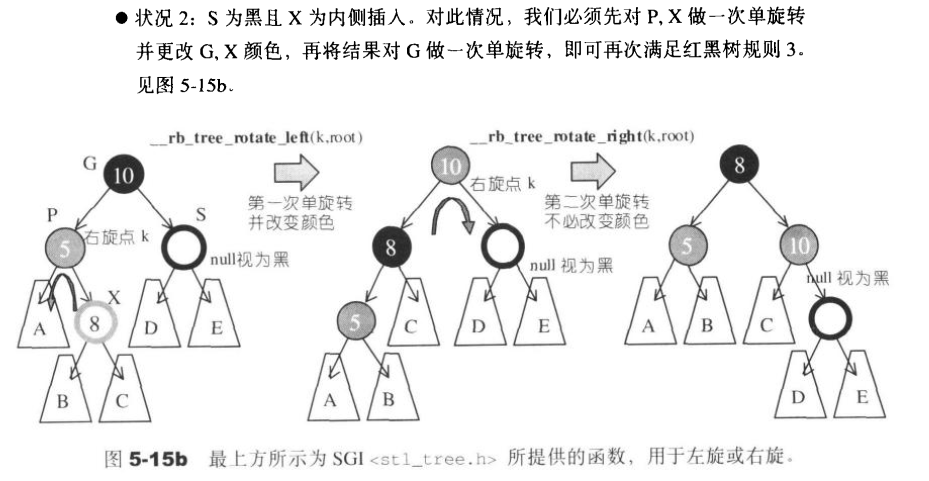
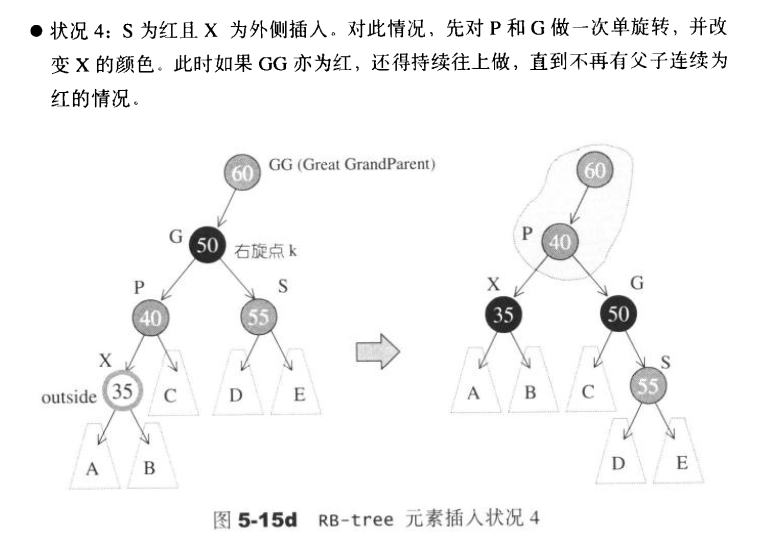
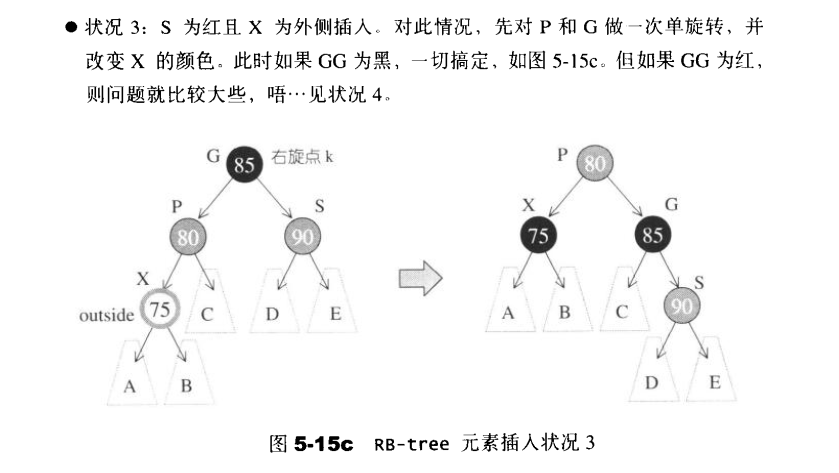
这四者都没有迭代器，因为必须满足其进出顺序。这种底层依靠其他容器，只改变接口来实现功能的一般叫做适配器。

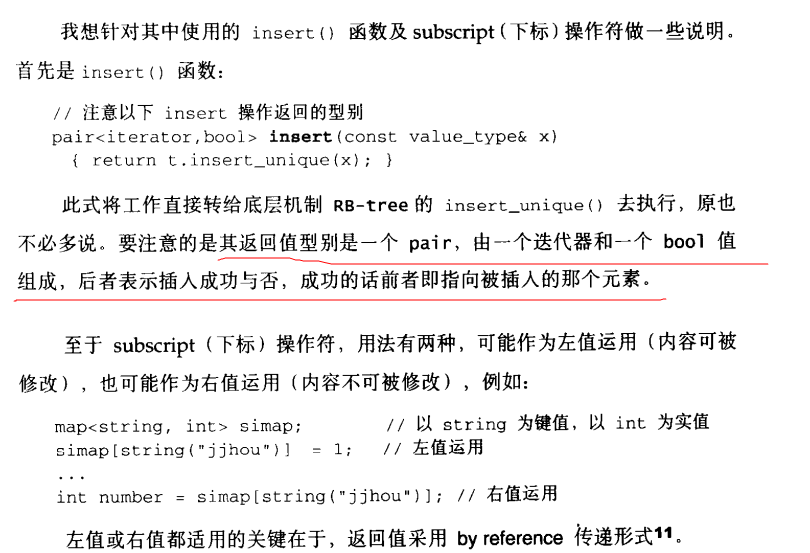
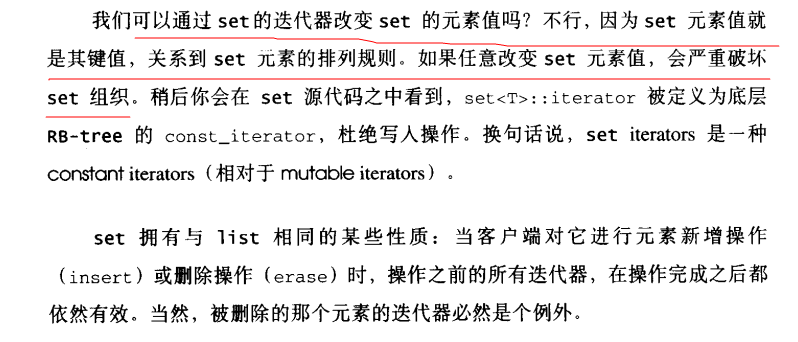


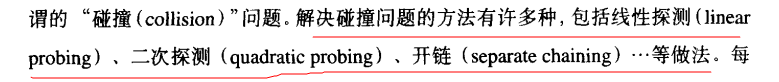
红黑树





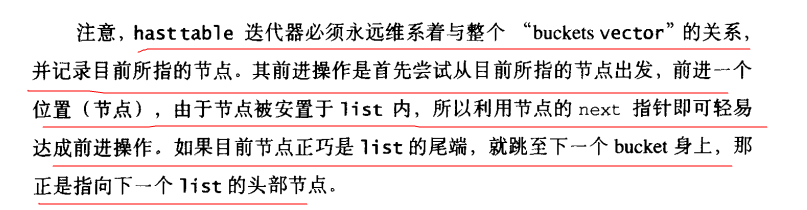


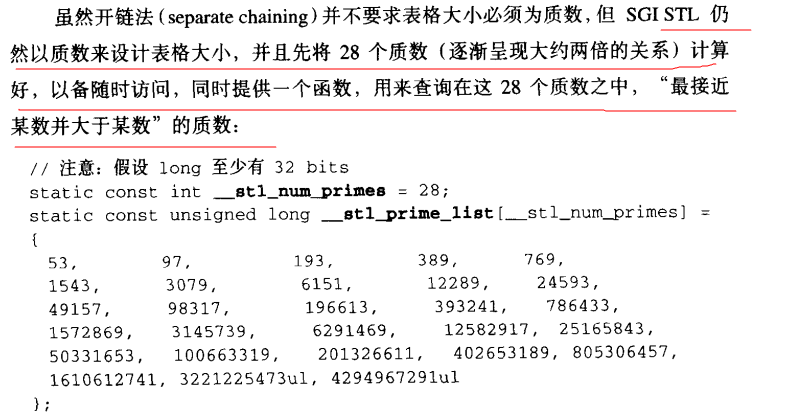
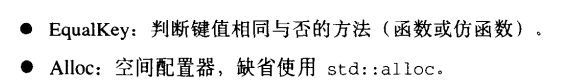
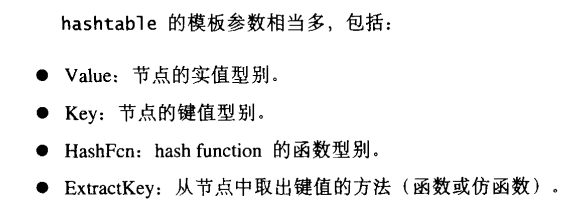
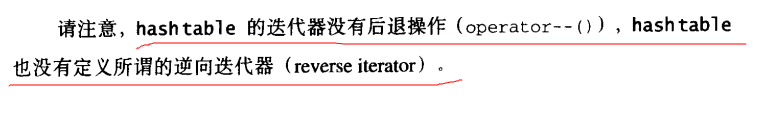
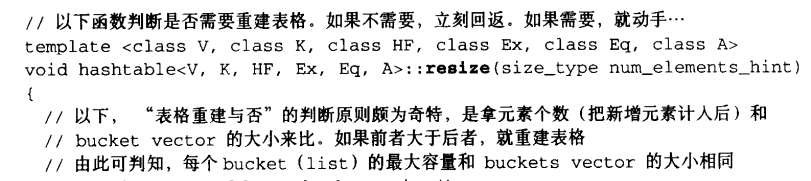
线性探测法：如果哈希之后的位置已有元素，则顺序找下去，直到有空位。最差情况会遍历一遍。

二次探测法： 如果冲突，则 H+ i\*2

再哈希：

拉链法：



’ 

C++中的hashtable为底层的hash\_set,hash\_map等的动态扩容过程：

Hashtable的BUCK以vector作为底层，实现动态扩容，但是hashtable里面有20多个质数，以53开头，大体成2倍关系。如果通过resize函数检测到需要扩容，则调用vector的扩容方式，扩充距离最近的需要数值的质数那么多个。

当hashmap中的元素越来越多的时候，碰撞的几率也就越来越高（因为数组的长度是固定的），所以为了提高查询的效率，就要对hashmap的数组进行扩容，数组扩容这个操作也会出现在ArrayList中，所以这是一个通用的操作，很多人对它的性能表示过怀疑，不过想想我们的“均摊”原理，就释然了，而在hashmap数组扩容之后，最消耗性能的点就出现了：原数组中的数据必须重新计算其在新数组中的位置，并放进去，这就是resize。

用一个 m\_max\_load\_factor 的因子来判断目前的容量需要多少个哈希桶，如果需要 rehash，那么使用素数表来算出新的桶需要多大。