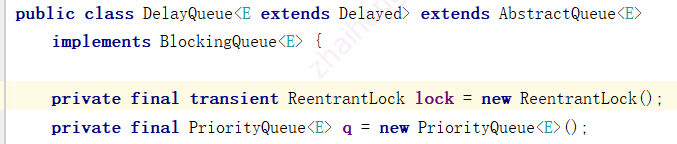
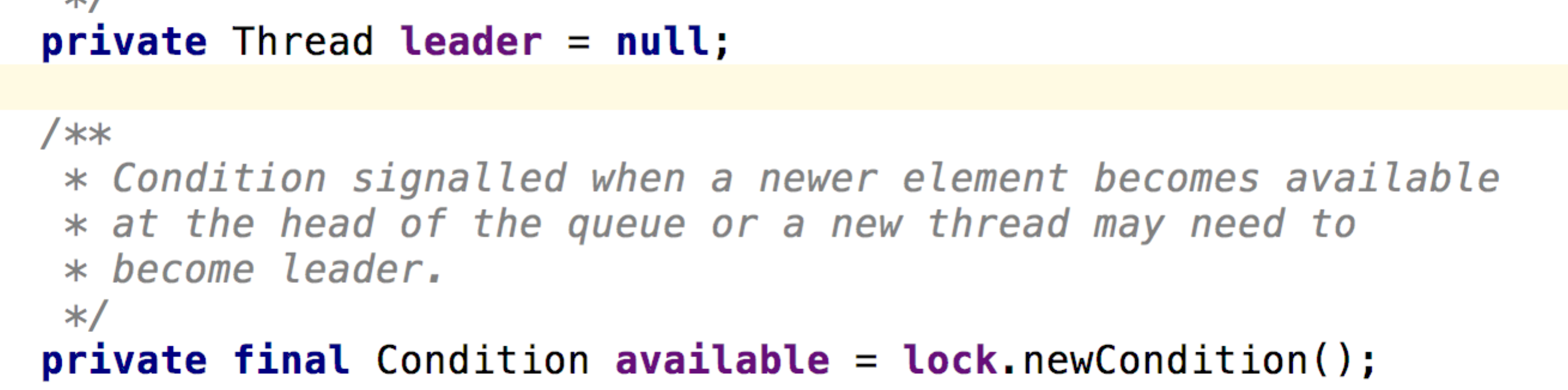
# DelayQueue



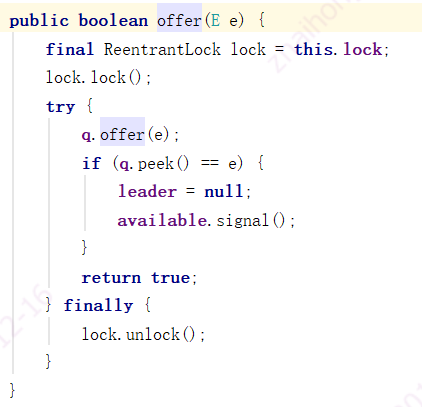


DelayQueue，JDK提供的内存级延迟队列。队列中的元素必须继承Delayd接口，实现getDelay()和compareTo()方法。

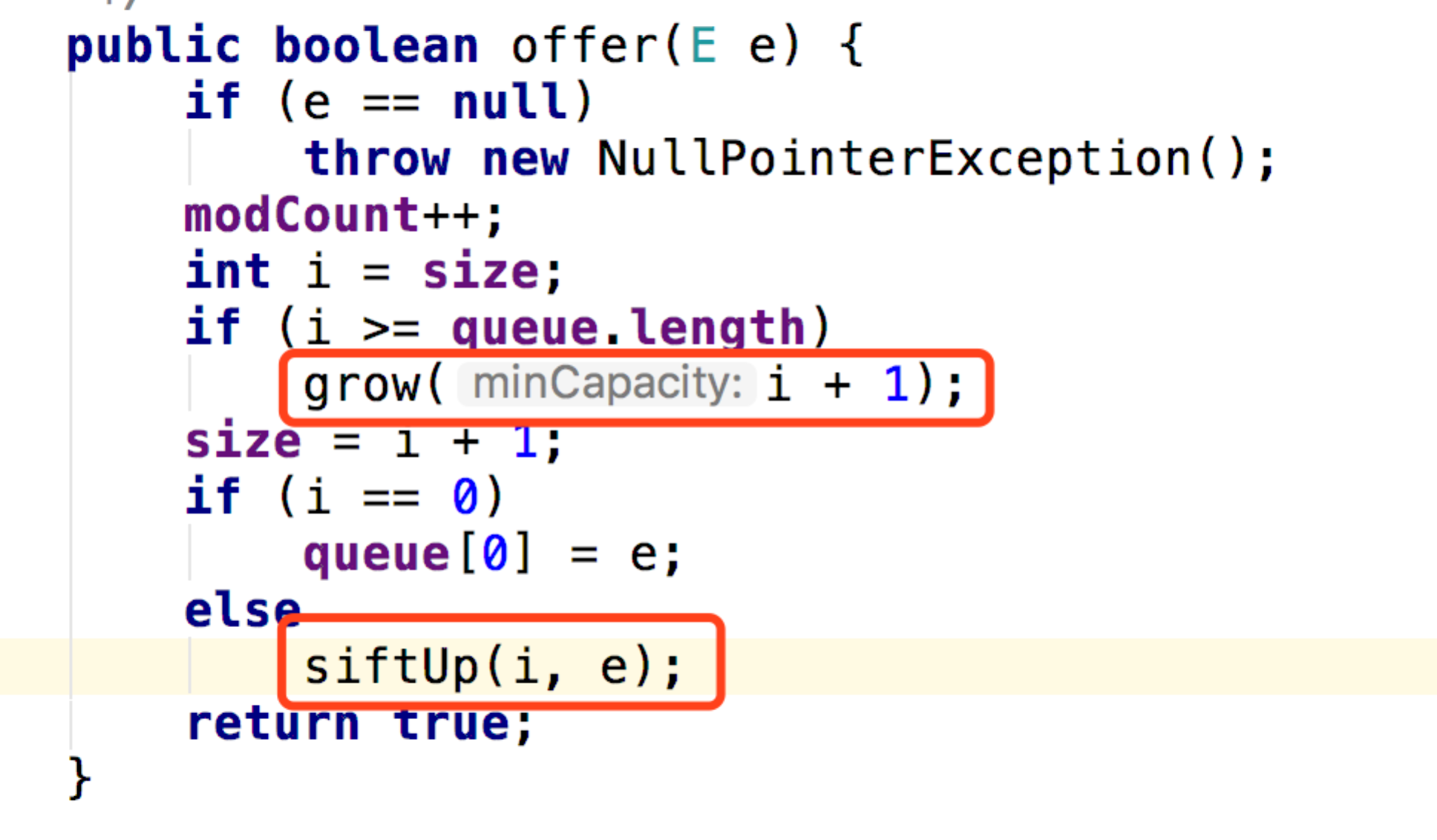
延迟队列的实现依赖了优先级队列和重入锁。优先级队列并不是线程安全的，在日常的使用中一定要注意这一点。延迟队列的工作原理是当元素入队，此时优先级队列是小顶堆，通过compareTo方法调整堆的排序，将最小（最快开始执行）的元素放入堆首。消费时，使用getDelay方法判断是否符合延时条件，满足则出队，否则不返回。

一、offer()：

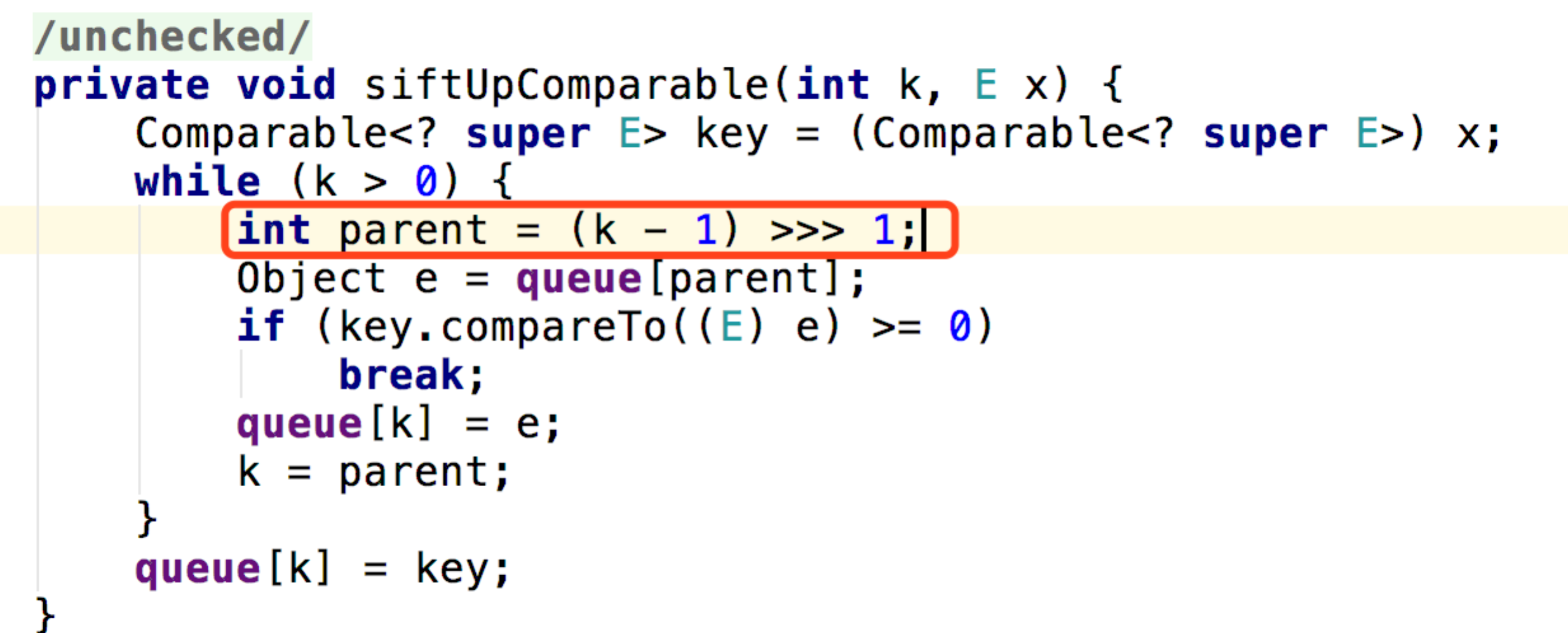
首先看看offer(e)方法，容易理解，当堆顶（队首）元素改变时需要通知获取线程，leader和available先设为【分支一】，后面继续讲。优先级队列就是堆，优先级队列不是线性数据结构，但它的动作类似队列，不要搞混。



延迟队列默认使用小顶堆。接着进入优先级队列。详细阅读发现优先级队列使用数组存储，是一颗完全二叉树。这里需要关注grow和siftUp方法。grow即扩容，当队列长度较小（小于64）时，队列成倍扩容，当超过64时，每次增加50%。节约内存，但扩容次数会增多。



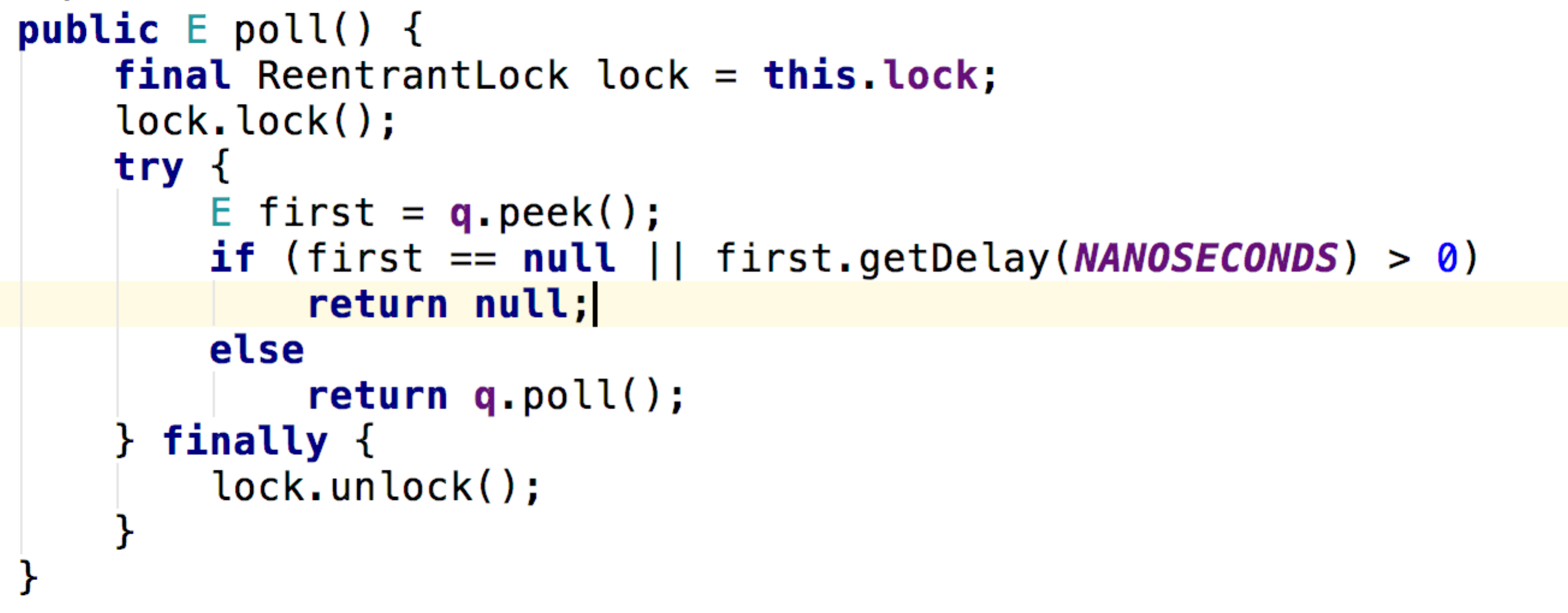
如果队列不空在siftUp中才放入堆中。sift是shift的缩写，即将元素放置在完全二叉树下一个位点，并逐层上升。由于默认是小顶堆则没有传入comparator对象，使用延迟对象的compareTo方法，看下实现：



完全二叉树的特性，节点序号按层次遍历（由上到下、由左到右）排列，则左孩子 = 2 \* parent，右孩子 = 2 \* parent + 1。可以用逆运算找到parent。如果parent > 孩子，则将parent下降，继续向上对比更高的parent；否则找到插入位置。

二、poll()，take():

非阻塞的从延迟队列中取元素，根据getDelay方法返回值是否大于0判断延迟时间是否达到。



阻塞的取元素，用take完成，当condition.await()或await(long)时，会释放它对应的锁。如果队列中没有元素，如第一个红框所示，该线程会一直等待下去，直到队首元素更新，注意这里只是队首而不是插入元素。接着如果队首元素延时时间到了，则poll出去。如果还在延时，则需要判断是否已经有其他线程已经等待下一个元素了。这时用leader标记第一个请求下一个元素的线程。

如果leader不为空，则其他线程都要一直等待。如果是第一个线程，会修改leader为当前线程，并等待delay时长后再次判断延时。leader睡眠期间，其他线程进入take方法会看到leader不为空。leader线程睡眠delay后会释放leader引用。当leader线程拿到元素后会唤醒其他await的线程，执行完lock.unlock()后，后续线程继续竞争锁（await醒来时尝试获取锁）。

在leader等待过程中可能发生两种情况：  
 第一，队首元素没有更新。过程就像上面讲的

第二，队首元素更新，offer方法释放leader，并通知所有正在等待的线程（包括leader和follower线程）。这时这些线程去竞争锁，竞争到的成为新的leader。

leader的作用，注释中写的是最小化不必要的等待时间。我倒认为能极大的减少竞争。否则多个线程大概都在一个时刻唤醒又要在await结束时争抢锁，除了抢到线程以外，其他线程没必要醒来。有了leader，只有leader会在delay后醒来。

到此【分支一】结束。



接着是带有超时时间的阻塞方法poll(long)。有了对阻塞方法的理解，带有超时时间的更好理解了。只要理解了一点，超时时间方法就很简单。就是当传入的超时时间<delay时，不管有多少线程都不构成竞争关系，因为超时时间内不会有元素到期，所以第二个红框先判断超时时间是否<delay。当超时时间>=delay，构成竞争关系，后续关于leader的设置和释放跟take相同。另一个点是timeLeft，队首值改变，唤醒，此时leader线程等待时间不满delay，还剩timeLeft，所以等待了delay – timeLeft，在超时时间中应减去delay – timeLeft。-的优先级高于-=。

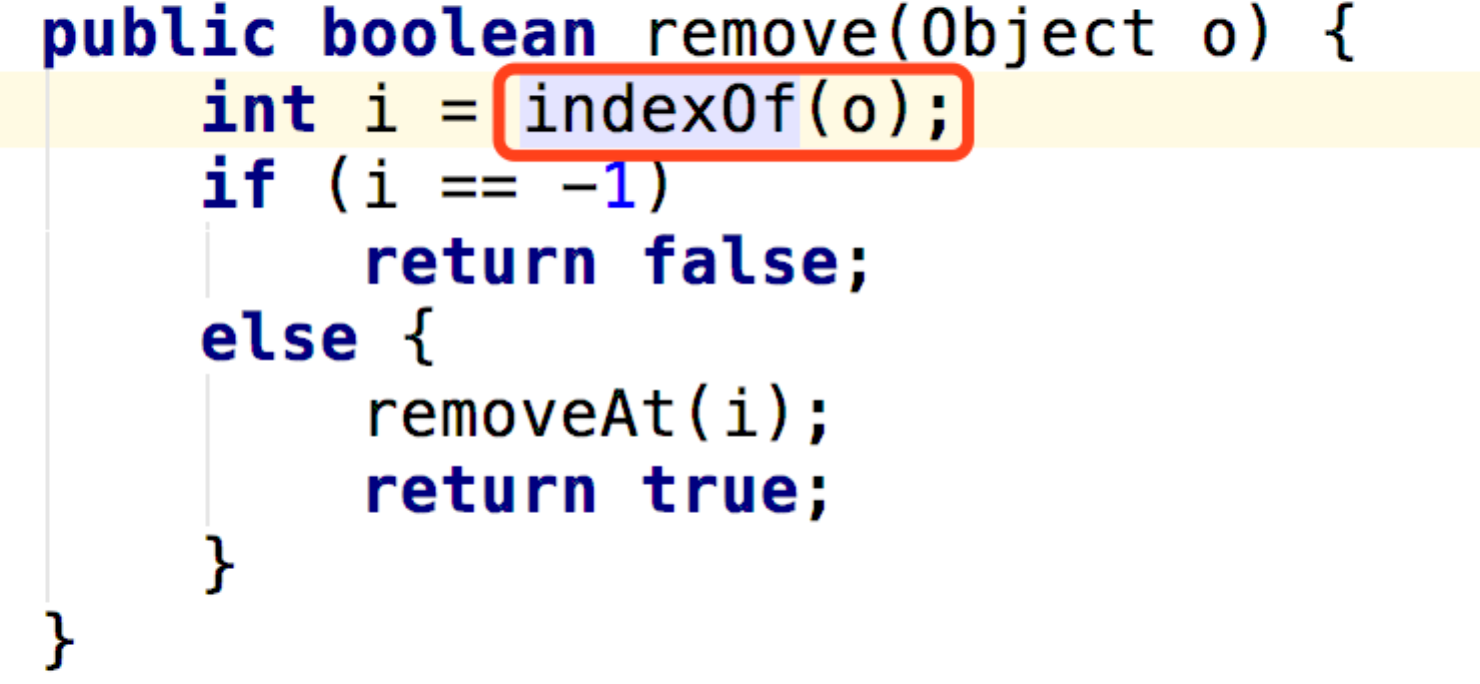


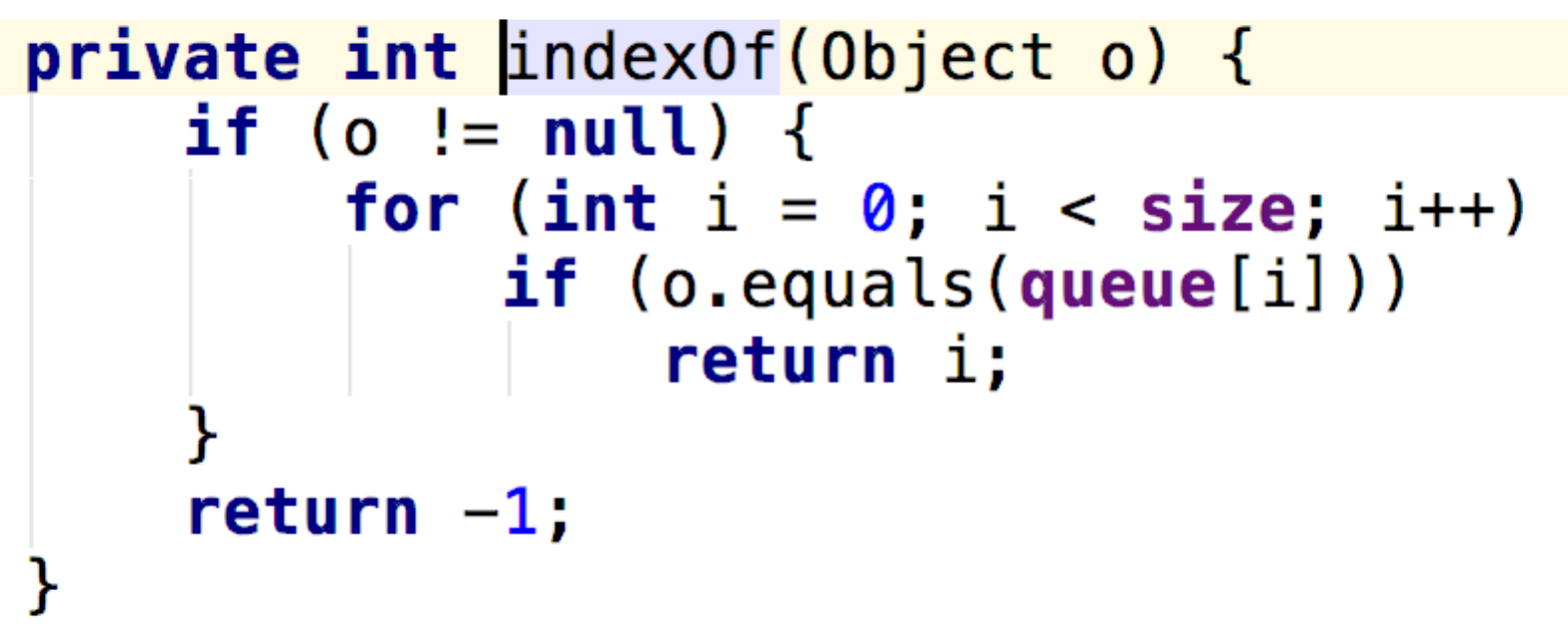
三、remove()

延迟队列的remove方法完全依赖优先级队列的删除方法。

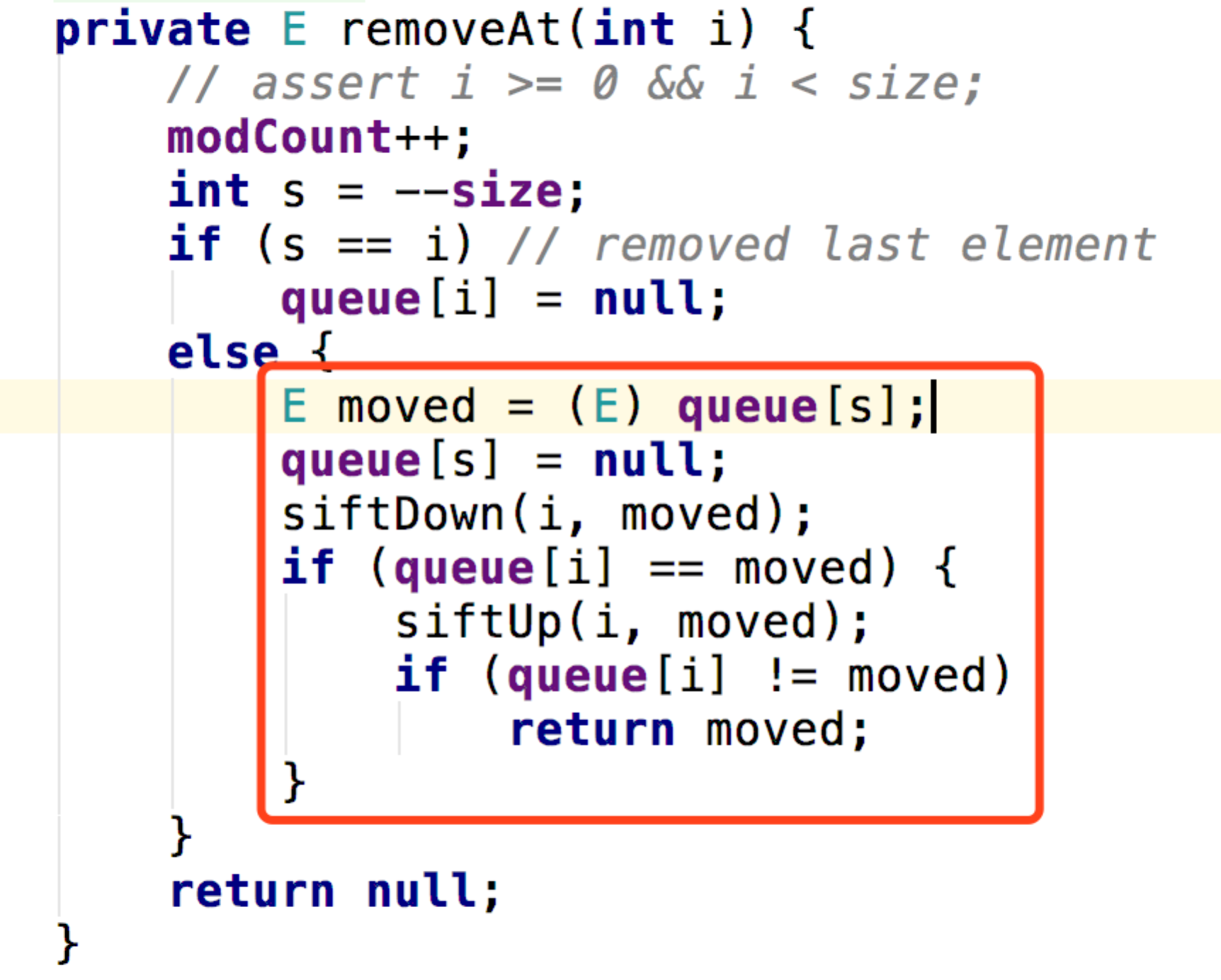
还记得插入元素的siftUp方法吗，先插入完全二叉树的下一个节点，再依次向上跟parent比较，如果parent更大则将parent下降，像冒泡一样。

接着看删除，过程比较类似。首先遍历数组找到要删除的位点，由此可知remove的时间复杂度是O(n)。接着执行removeAt(i)。

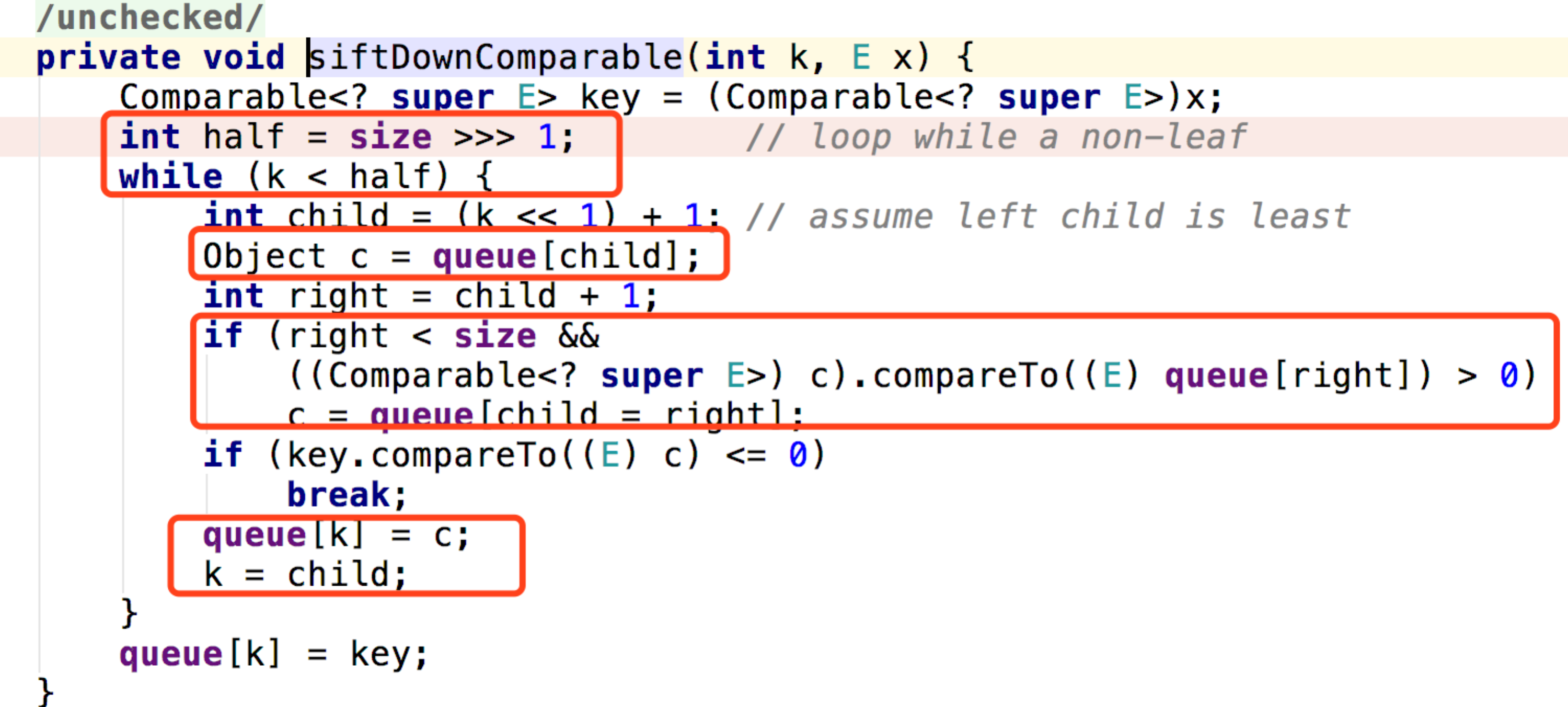




s为二叉树最后一个元素下标，i为要删除的元素下标。首先将最后一个元素有moved指针保存，原位则置为null。接着进行siftDown操作。siftDown的会将i元素删掉，并从i元素的左右孩子中找到较小的上升到i位置。同理如果仍有孩子，会继续上升，直到堆有序。将moved插入到合适的位置。



让我们再详细看看siftDown的逻辑。第一个红框，是要判断k位置（最初就是i位置），是否包含孩子。由完全二叉树的特性，每一层节点数翻倍，当节点序号小于一半则一定有孩子。第二个红框表示左孩子值。第三个红框，现在要找左右孩子其中之一代替parent，需要找较小的那个，这样才符合小顶堆的规律，如果右孩子更小，则child指针变更，同时c指针也变为右孩子。接着当moved元素（原二叉树最后一个元素）已经小于孩子节点，则moved应该放在此处，不应继续循环。第四个红框，当moved元素仍然大于选中的孩子节点，需要将孩子上升，继续向下寻找位置。



下面两幅图是删除元素的实例图，分别表示删除带孩子节点、删除叶子节点和key < c，中途break的情况。

小顶堆只有parent和child有大小关系，如果不在同一个链路上，不能保证大小关系。因此如果出现queue[i] == moved即直接将最后一个元素替代了要删除的元素，虽然经过siftDown向下一定是parent <= child，但向上不一定保证。所以要经过siftUp调整。

