```java
    /**
     * Partitions {@code q} into two parts: entries no larger than
     * {@code partitioner} are put in {@code front}, and the rest are put in
     * {@code back}.
     *
     * @param <T>
     *            type of {@code Queue} entries
     * @param q
     *            the {@code Queue} to be partitioned
     * @param partitioner
     *            the partitioning value
     * @param front
     *            upon return, the entries no larger than {@code partitioner}
     * @param back
     *            upon return, the entries larger than {@code partitioner}
     * @param order
     *            ordering by which to separate entries
     * @clears q
     * @replaces front, back
     * @requires IS_TOTAL_PREORDER([relation computed by order.compare method])
     * @ensures <pre>
     * perms(#q, front * back)  and
     * for all x: T where (<x> is substring of front)
     *  ([relation computed by order.compare method](x, partitioner))  and
     * for all x: T where (<x> is substring of back)
     *  (not [relation computed by order.compare method](x, partitioner))
     * </pre>
     */
    private static <T> void partition(Queue<T> q, T partitioner, Queue<T> front,
            Queue<T> back, Comparator<T> order) {
        while (q.length() > 0) {
            T value = q.dequeue();
            if (order.compare(partitioner, value) > 0) {
                front.enqueue(value);
            } else {
                back.enqueue(value);
            }
        }
    }

    /**
     * Sorts {@code this} according to the ordering provided by the
     * {@code compare} method from {@code order}.
     *
     * @param order
     *            ordering by which to sort
     * @updates this
     * @requires IS_TOTAL_PREORDER([relation computed by order.compare method])
     * @ensures <pre>
     * perms(this, #this)  and
     * IS_SORTED(this, [relation computed by order.compare method])
     * </pre>
     */
    public void sort(Comparator<T> order) {
        if (this.length() > 1) {
            /*
             * Dequeue the partitioning entry from this
             */
            T val = this.dequeue();
            /*
             * Partition this into two queues as discussed above (you will need
             * to declare and initialize two new queues)
             */
            Queue<T> front = new Queue1L();
            Queue<T> back = new Queue1L();
            // This needs to be changed;
            partition(this, val, front, back, order);
            /*
             * Recursively sort the two queues
```

```
70                */
71               // This needs to be changed
72               front.sort(order);
73               back.sort(order);
74               /*
75                * Reconstruct this by combining the two sorted queues and the
76                * partitioning entry in the proper order
77                */
78               this.append(front);
79               this.enqueue(val);
80               this.append(back);
81           }
82       }
```