```java
import java.util.Comparator;

/**
 * {@code Queue} represented as a {@code Sequence} of entries, with
 * implementations of primary methods.
 *
 * @param <T>
 *            type of {@code Queue} entries
 * @correspondence this = $this.entries
 */
public class HelloWorld {

    public static void main(String[] args) {
        Queue<Integer> test = new Queue1L<>();
        test.enqueue(1);
        test.enqueue(2);
        test.enqueue(3);
        test.rotate(1);
        System.out.println(test);

    }

    /**
     * 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
```

```
45      * @replaces front, back
46      * @requires IS_TOTAL_PREORDER([relation computed by
    order.compare method])
47      * @ensures <pre>
48      * perms(#q, front * back)  and
49      * for all x: T where (<x> is substring of front)
50      *  ([relation computed by order.compare method](x,
    partitioner))  and
51      * for all x: T where (<x> is substring of back)
52      *  (not [relation computed by order.compare method](x,
    partitioner))
53      * </pre>
54      */
55     private static <T> void partition(Queue<T> q, T partitioner,
    Queue<T> front,
56             Queue<T> back, Comparator<T> order) {
57         front.clear();
58         back.clear();
59         while (q.length() > 0) {
60             T temp = q.dequeue();
61             if (order.compare(temp, partitioner) < 0) {
62                 front.enqueue(temp);
63             } else {
64                 back.enqueue(temp);
65             }
66         }
67     }
68
69     /**
70      * Sorts {@code this} according to the ordering provided by
    the
71      * {@code compare} method from {@code order}.
72      *
73      * @param order
74      *            ordering by which to sort
75      * @updates this
76      * @requires IS_TOTAL_PREORDER([relation computed by
    order.compare method])
77      * @ensures <pre>
78      * perms(this, #this)  and
79      * IS_SORTED(this, [relation computed by order.compare
    method])
80      * </pre>
81      */
```

```
 82     public void sort(Comparator<T> order) {
 83         if (this.length() > 1) {
 84             /*
 85              * Dequeue the partitioning entry from this
 86              */
 87             T temp = this.deque();
 88 
 89             /*
 90              * Partition this into two queues as discussed above
    (you will need
 91              * to declare and initialize two new queues)
 92              */
 93             Queue<T> front = new Queue2<T>();
 94             Queue<T> back = front.newInstance();
 95             partition(this, temp, front, back, order);
 96 
 97             /*
 98              * Recursively sort the two queues
 99              */
100             front.sort(order);
101             back.sort(order);
102 
103             /*
104              * Reconstruct this by combining the two sorted queues
    and the
105              * partitioning entry in the proper order
106              */
107             this.enque(temp);
108             this.append(front);
109             this.append(back);
110 
111         }
112     }
113 
114 }
```