```java
import java.util.Comparator;

import components.queue.Queue;
import components.queue.Queue1L;

/**
 * Layered implementations of secondary method {@code sort} for
 * {@code Queue<String>}.
 *
 * @param <T>
 *            type of {@code Queue} entries
 * @mathdefinitions <pre>
 * IS_TOTAL_PREORDER (
 *   r: binary relation on T
 *  ) : boolean is
 *  for all x, y, z: T
 *   ((r(x, y) or r(y, x))  and
 *    (if (r(x, y) and r(y, z)) then r(x, z)))
 *
 * IS_SORTED (
 *   s: string of T,
 *   r: binary relation on T
 *  ) : boolean is
 *  for all x, y: T where (<x, y> is substring of s) (r(x, y))
 * </pre>
 */
public final class Queue1LSort4<T> extends Queue1L<T> {

    /**
     * No-argument constructor.
     */
    public Queue1LSort4() {
        super();
    }

    /**
     * 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 void partition(Queue<T> q, T partitioner, Queue<T> front,
            Queue<T> back, Comparator<T> order) {
        assert q != null : "Violation of: q is not null";
        assert partitioner != null : "Violation of: partitioner is not null";
        assert front != null : "Violation of: front is not null";
        assert back != null : "Violation of: back is not null";
        assert order != null : "Violation of: order is not null";

        while (q.length() > 0) {

            T temp = q.dequeue();

            if (order.compare(temp, partitioner) >= 0) {
                back.enqueue(temp);
            } else {
                front.enqueue(temp);
            }

        }

    }

    @Override
    public void sort(Comparator<T> order) {
        assert order != null : "Violation of: order is not null";
        if (this.length() > 2) {
            Queue<T> temp = new Queue1L<T>();

            /*
             * Dequeue the partitioning entry from this
             */
            while (this.length() > temp.length()) {

                temp.enqueue(this.dequeue());

            }

            T partitioner = this.dequeue();
            temp.enqueue(partitioner);

            while (this.length() > 0) {

                temp.enqueue(this.dequeue());

            }

            /*
             * Partition this into two queues as discussed above
             * (you will need to declare and initialize two new queues)
             */

            Queue<T> lower = new Queue1L<T>();
            Queue<T> higher = new Queue1L<T>();


            // partition using partitioner
            partition(temp, partitioner, lower, higher, order);

            /*
             * Recursively sort the two queues
             */
            lower.sort(order);
            higher.sort(order);

            /*
             * Reconstruct this by combining the two sorted queues and the
             * partitioning entry in the proper order
             */
            this.append(lower);
            this.append(higher);


        }
    }

}

```