```java
import java.lang.reflect.Constructor;
import java.util.Comparator;
import java.util.Iterator;

import components.queue.Queue;
import components.queue.Queue1L;
import components.sortingmachine.SortingMachine;
import components.sortingmachine.SortingMachineSecondary;

/**
 * {@code SortingMachine} represented as a {@code Queue} (using an embedding of
 * quicksort), with implementations of primary methods.
 *
 * @param <T>
 *            type of {@code SortingMachine} 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>
 * @convention <pre>
 * IS_TOTAL_PREORDER([relation computed by $this.machineOrder.compare method])  and
 * if not $this.insertionMode then
 *  IS_SORTED($this.entries, [relation computed by $this.machineOrder.compare method])
 * </pre>
 * @correspondence <pre>
 * this =
 *   ($this.insertionMode, $this.machineOrder, multiset_entries($this.entries))
 * </pre>
 */
public class SortingMachine4<T> extends SortingMachineSecondary<T> {

    /*
     * Private members --------------------------------------------------------
     */

    /**
     * Insertion mode.
     */
    private boolean insertionMode;

    /**
     * Order.
     */
    private Comparator<T> machineOrder;

    /**
     * Entries.
```

```
 58      */
 59     private Queue<T> entries;
 60
 61     /**
 62      * Partitions {@code q} into two parts: entries no larger than
 63      * {@code partitioner} are put in {@code front}, and the rest are put in
 64      * {@code back}.
 65      *
 66      * @param <T>
 67      *            type of {@code Queue} entries
 68      * @param q
 69      *            the {@code Queue} to be partitioned
 70      * @param partitioner
 71      *            the partitioning value
 72      * @param front
 73      *            upon return, the entries no larger than {@code partitioner}
 74      * @param back
 75      *            upon return, the entries larger than {@code partitioner}
 76      * @param order
 77      *            ordering by which to separate entries
 78      * @clears q
 79      * @replaces front, back
 80      * @requires IS_TOTAL_PREORDER([relation computed by order.compare method])
 81      * @ensures <pre>
 82      * perms(#q, front * back)  and
 83      * for all x: T where (<x> is substring of front)
 84      *  ([relation computed by order.compare method](x, partitioner))  and
 85      * for all x: T where (<x> is substring of back)
 86      *  (not [relation computed by order.compare method](x, partitioner))
 87      * </pre>
 88      */
 89     private static <T> void partition(Queue<T> q, T partitioner, Queue<T> front,
 90             Queue<T> back, Comparator<T> order) {
 91         assert q != null : "Violation of: q is not null";
 92         assert partitioner != null : "Violation of: partitioner is not null";
 93         assert front != null : "Violation of: front is not null";
 94         assert back != null : "Violation of: back is not null";
 95         assert order != null : "Violation of: order is not null";
 96
 97         while (q.length() > 0) {
 98
 99             T temp = q.dequeue();
100
101             if (order.compare(temp, partitioner) >= 0) {
102                 back.enqueue(temp);
103             } else {
104                 front.enqueue(temp);
105             }
106
107         }
108
109     }
110
111     /**
112      * Sorts {@code q} according to the ordering provided by the {@code compare}
113      * method from {@code order}.
114      *
```

```
115      * @param <T>
116      *            type of {@code Queue} entries
117      * @param q
118      *            the {@code Queue} to be sorted
119      * @param order
120      *            ordering by which to sort
121      * @updates q
122      * @requires IS_TOTAL_PREORDER([relation computed by order.compare method])
123      * @ensures IS_SORTED(q, [relation computed by order.compare method])
124      */
125     private static <T> void sort(Queue<T> q, Comparator<T> order) {
126         assert order != null : "Violation of: order is not null";
127
128         if (q.length() > 2) {
129             Queue<T> temp = new Queue1L<T>();
130
131             /*
132              * Dequeue the partitioning entry from this
133              */
134             while (q.length() > temp.length()) {
135
136                 temp.enqueue(q.dequeue());
137
138             }
139
140             T partitioner = q.dequeue();
141             temp.enqueue(partitioner);
142
143             while (q.length() > 0) {
144
145                 temp.enqueue(q.dequeue());
146
147             }
148
149             /*
150              * Partition this into two queues as discussed above
151              * (you will need to declare and initialize two new queues)
152              */
153
154             Queue<T> lower = new Queue1L<T>();
155             Queue<T> higher = new Queue1L<T>();
156
157
158             // partition using partitioner
159             partition(temp, partitioner, lower, higher, order);
160
161             /*
162              * Recursively sort the two queues
163              */
164             lower.sort(order);
165             higher.sort(order);
166
167             /*
168              * Reconstruct this by combining the two sorted queues and the
169              * partitioning entry in the proper order
170              */
171             q.append(lower);
```

```
172             q.append(higher);
173
174
175         }
176
177     }
178
179     /**
180      * Creator of initial representation.
181      *
182      * @param order
183      *            total preorder for sorting
184      */
185     private void createNewRep(Comparator<T> order) {
186         this.insertionMode = true;
187         this.machineOrder = order;
188         this.entries = new Queue1L<T>();
189     }
190
191     /*
192      * Constructors ------------------------------------------------------------
193      */
194
195     /**
196      * Constructor from order.
197      *
198      * @param order
199      *            total preorder for sorting
200      */
201     public SortingMachine4(Comparator<T> order) {
202         this.createNewRep(order);
203     }
204
205     /*
206      * Standard methods --------------------------------------------------------
207      */
208
209     @SuppressWarnings("unchecked")
210     @Override
211     public final SortingMachine<T> newInstance() {
212         try {
213             Constructor<?> c = this.getClass().getConstructor(Comparator.class);
214             return (SortingMachine<T>) c.newInstance(this.machineOrder);
215         } catch (ReflectiveOperationException e) {
216             throw new AssertionError(
217                     "Cannot construct object of type " + this.getClass());
218         }
219     }
220
221     @Override
222     public final void clear() {
223         this.createNewRep(this.machineOrder);
224     }
225
226     @Override
227     public final void transferFrom(SortingMachine<T> source) {
228         assert source != null : "Violation of: source is not null";
```

```
229         assert source != this : "Violation of: source is not this";
230         assert source instanceof SortingMachine4<?> : ""
231                 + "Violation of: source is of dynamic type SortingMachine4<?>";
232         /*
233          * This cast cannot fail since the assert above would have stopped
234          * execution in that case: source must be of dynamic type
235          * SortingMachine4<?>, and the ? must be T or the call would not have
236          * compiled.
237          */
238         SortingMachine4<T> localSource = (SortingMachine4<T>) source;
239         this.insertionMode = localSource.insertionMode;
240         this.machineOrder = localSource.machineOrder;
241         this.entries = localSource.entries;
242         localSource.createNewRep(localSource.machineOrder);
243     }
244
245     /*
246      * Kernel methods ----------------------------------------------------------
247      */
248
249     @Override
250     public final void add(T x) {
251         assert x != null : "Violation of: x is not null";
252         assert this.isInInsertionMode() : "Violation of: this.insertion_mode";
253
254         this.insertionMode = true;
255         this.entries.enqueue(x);
256
257     }
258
259     @Override
260     public final void changeToExtractionMode() {
261         assert this.isInInsertionMode() : "Violation of: this.insertion_mode";
262
263         this.insertionMode = false;
264     }
265
266     @Override
267     public final T removeFirst() {
268         assert !this.isInInsertionMode() : "Violation of: not this.insertion_mode";
269         assert this.size() > 0 : "Violation of: this.contents /= {}";
270
271         this.insertionMode = false;
272         return this.entries.dequeue();
273     }
274
275     @Override
276     public final boolean isInInsertionMode() {
277
278         return this.insertionMode;
279     }
280
281     @Override
282     public final Comparator<T> order() {
283
284         return this.machineOrder;
285     }
```

```
286
287     @Override
288     public final int size() {
289
290         return this.entries.length();
291     }
292
293     @Override
294     public final Iterator<T> iterator() {
295         return this.entries.iterator();
296     }
297
298 }
299
```