```java
import java.util.Comparator;

/**
 * {@code SortingMachine} represented as a {@code Queue} and an array (using an
 * embedding of heap sort), 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)))
 *
 * SUBTREE_IS_HEAP (
 *   a: string of T,
 *   start: integer,
 *   stop: integer,
 *   r: binary relation on T
 *  ) : boolean is
 *  [the subtree of a (when a is interpreted as a complete binary tree) rooted
 *   at index start and only through entry stop of a satisfies the heap
 *   ordering property according to the relation r]
 *
 * SUBTREE_ARRAY_ENTRIES (
 *   a: string of T,
 *   start: integer,
 *   stop: integer
 *  ) : finite multiset of T is
 *  [the multiset of entries in a that belong to the subtree of a
 *   (when a is interpreted as a complete binary tree) rooted at
 *   index start and only through entry stop]
 * </pre>
 * @convention <pre>
 * IS_TOTAL_PREORDER([relation computed by $this.machineOrder.compare method]  and
 * if $this.insertionMode then
 *   $this.heapSize = 0
```

```
 * else
 *   $this.entries = <>  and
 *   for all i: integer
 *       where (0 <= i  and  i < |$this.heap|)
 *     ([entry at position i in $this.heap is not null])  and
 *   SUBTREE_IS_HEAP($this.heap, 0, $this.heapSize - 1,
 *     [relation computed by $this.machineOrder.compare method])
and
 *   0 <= $this.heapSize <= |$this.heap|
 * </pre>
 * @correspondence <pre>
 * if $this.insertionMode then
 *   this = (true, $this.machineOrder,
multiset_entries($this.entries))
 * else
 *   this = (false, $this.machineOrder,
multiset_entries($this.heap[0, $this.heapSize)))
 * </pre>
 *
 * @author Alex Honigford and Jonny Pater
 *
 */
public class SortingMachine5a<T> extends
SortingMachineSecondary<T> {

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

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

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

    /**
     * Entries.
     */
    private Queue<T> entries;
```

```
 86
 87     /**
 88      * Heap.
 89      */
 90     private T[] heap;
 91
 92     /**
 93      * Heap size.
 94      */
 95     private int heapSize;
 96
 97     /**
 98      * Exchanges entries at indices {@code i} and {@code j} of
    {@code array}.
 99      *
100      * @param <T>
101      *            type of array entries
102      * @param array
103      *            the array whose entries are to be exchanged
104      * @param i
105      *            one index
106      * @param j
107      *            the other index
108      * @updates array
109      * @requires 0 <= i < |array| and 0 <= j < |array|
110      * @ensures array = [#array with entries at indices i and j
    exchanged]
111      */
112     private static <T> void exchangeEntries(T[] array, int i, int
    j) {
113         assert array != null : "Violation of: array is not null";
114         assert 0 <= i : "Violation of: 0 <= i";
115         assert i < array.length : "Violation of: i < |array|";
116         assert 0 <= j : "Violation of: 0 <= j";
117         assert j < array.length : "Violation of: j < |array|";
118
119         // checks if the two given indexes aren't equal before
    swapping them
120         if (i != j) {
121             T tmp = array[i];
122             array[i] = array[j];
123             array[j] = tmp;
124         }
125
```

```
126     }
127
128     /**
129      * Given an array that represents a complete binary tree and
    an index
130      * referring to the root of a subtree that would be a heap
    except for its
131      * root, sifts the root down to turn that whole subtree into a
    heap.
132      *
133      * @param <T>
134      *            type of array entries
135      * @param array
136      *            the complete binary tree
137      * @param top
138      *            the index of the root of the "subtree"
139      * @param last
140      *            the index of the last entry in the heap
141      * @param order
142      *            total preorder for sorting
143      * @updates array
144      * @requires <pre>
145      * 0 <= top  and  last < |array|  and
146      * for all i: integer
147      *     where (0 <= i  and  i < |array|)
148      *   ([entry at position i in array is not null])  and
149      * [subtree rooted at {@code top} is a complete binary tree]
    and
150      * SUBTREE_IS_HEAP(array, 2 * top + 1, last,
151      *     [relation computed by order.compare method])  and
152      * SUBTREE_IS_HEAP(array, 2 * top + 2, last,
153      *     [relation computed by order.compare method])  and
154      * IS_TOTAL_PREORDER([relation computed by order.compare
    method])
155      * </pre>
156      * @ensures <pre>
157      * SUBTREE_IS_HEAP(array, top, last,
158      *     [relation computed by order.compare method])  and
159      * perms(array, #array)  and
160      * SUBTREE_ARRAY_ENTRIES(array, top, last) =
161      *  SUBTREE_ARRAY_ENTRIES(#array, top, last)  and
162      * [the other entries in array are the same as in #array]
163      * </pre>
164      */
```

```
165     private static <T> void siftDown(T[] array, int top, int last,
166             Comparator<T> order) {
167         assert array != null : "Violation of: array is not null";
168         assert order != null : "Violation of: order is not null";
169         assert 0 <= top : "Violation of: 0 <= top";
170         assert last < array.length : "Violation of: last < |
    array|";
171         for (int i = 0; i < array.length; i++) {
172             assert array[i] != null : ""
173                     + "Violation of: all entries in array are not
    null";
174         }
175         assert isHeap(array, 2 * top + 1, last, order) : ""
176                 + "Violation of: SUBTREE_IS_HEAP(array, 2 * top +
    1, last,"
177                 + " [relation computed by order.compare method])";
178         assert isHeap(array, 2 * top + 2, last, order) : ""
179                 + "Violation of: SUBTREE_IS_HEAP(array, 2 * top +
    2, last,"
180                 + " [relation computed by order.compare method])";
181         /*
182          * Impractical to check last requires clause; no need to
    check the other
183          * requires clause, because it must be true when using the
    array
184          * representation for a complete binary tree.
185          */
186         //checks to make sure there are nodes to sift down in the
    array
187         //before trying to access those indexes of the array
188         if (top * 2 + 1 <= last) {
189             T smaller = array[top * 2 + 1];
190             int smallerIndex = top * 2 + 1;
191             if (top * 2 + 2 <= last) {
192                 if (order.compare(smaller, array[top * 2 + 2]) >
    0) {
193                     smaller = array[top * 2 + 2];
194                     smallerIndex = top * 2 + 2;
195                 }
196             }
197             if (order.compare(array[top], smaller) > 0) {
198                 //sifts down if a value a the top index is out of
    order
199                 exchangeEntries(array, top, smallerIndex);
```

```
200                 siftDown(array, smallerIndex, last, order);
201             }
202
203         }
204
205     }
206
207     /**
208      * Heapifies the subtree of the given array rooted at the
    given {@code top}.
209      *
210      * @param <T>
211      *            type of array entries
212      * @param array
213      *            the complete binary tree
214      * @param top
215      *            the index of the root of the "subtree" to
    heapify
216      * @param order
217      *            the total preorder for sorting
218      * @updates array
219      * @requires <pre>
220      * 0 <= top  and
221      * for all i: integer
222      *     where (0 <= i  and  i < |array|)
223      *   ([entry at position i in array is not null])  and
224      * [subtree rooted at {@code top} is a complete binary tree]
    and
225      * IS_TOTAL_PREORDER([relation computed by order.compare
    method])
226      * </pre>
227      * @ensures <pre>
228      * SUBTREE_IS_HEAP(array, top, |array| - 1,
229      *     [relation computed by order.compare method])  and
230      * perms(array, #array)
231      * </pre>
232      */
233     private static <T> void heapify(T[] array, int top,
    Comparator<T> order) {
234         assert array != null : "Violation of: array is not null";
235         assert order != null : "Violation of: order is not null";
236         assert 0 <= top : "Violation of: 0 <= top";
237         for (int i = 0; i < array.length; i++) {
238             assert array[i] != null : ""
```

```
239                     + "Violation of: all entries in array are not
    null";
240         }
241         /*
242          * Impractical to check last requires clause; no need to
    check the other
243          * requires clause, because it must be true when using the
    array
244          * representation for a complete binary tree.
245          */
246         // Check to make sure the left and right indexes wouldn't
247         //be out of the array bounds before trying to access them
248         int left = 2 * top + 1;
249         int right = 2 * top + 2;
250         int last = array.length - 1;
251         if (left <= last) {
252             heapify(array, left, order);
253             if (right <= last) {
254                 heapify(array, right, order);
255             }
256             //heapifies the left and right subtrees before sifting
257             //the top index down if needed to make a complete heap
258             siftDown(array, top, last, order);
259         }
260     }
261
262     /**
263      * Constructs and returns an array representing a heap with
    the entries from
264      * the given {@code Queue}.
265      *
266      * @param <T>
267      *            type of {@code Queue} and array entries
268      * @param q
269      *            the {@code Queue} with the entries for the heap
270      * @param order
271      *            the total preorder for sorting
272      * @return the array representation of a heap
273      * @clears q
274      * @requires IS_TOTAL_PREORDER([relation computed by
    order.compare method])
275      * @ensures <pre>
276      * SUBTREE_IS_HEAP(buildHeap, 0, |buildHeap| - 1)  and
277      * perms(buildHeap, #q)  and
```

```
278      * for all i: integer
279      *     where (0 <= i  and  i < |buildHeap|)
280      *   ([entry at position i in buildHeap is not null])  and
281      * </pre>
282      */
283     @SuppressWarnings("unchecked")
284     private static <T> T[] buildHeap(Queue<T> q, Comparator<T>
    order) {
285         assert q != null : "Violation of: q is not null";
286         assert order != null : "Violation of: order is not null";
287         /*
288          * Impractical to check the requires clause.
289          */
290         /*
291          * With "new T[...]" in place of "new Object[...]" it does
    not compile;
292          * as shown, it results in a warning about an unchecked
    cast, though it
293          * cannot fail.
294          */
295         T[] heap = (T[]) (new Object[q.length()]);
296 
297         int length = q.length();
298         for (int i = 0; i < length; i++) {
299             heap[i] = q.dequeue();
300         }
301         //Empties the queue into the heap array then heapifies
302         //the array to make it into a heap
303         heapify(heap, 0, order);
304         return heap;
305     }
306 
307     /**
308      * Checks if the subtree of the given {@code array} rooted at
    the given
309      * {@code top} is a heap.
310      *
311      * @param <T>
312      *            type of array entries
313      * @param array
314      *            the complete binary tree
315      * @param top
316      *            the index of the root of the "subtree"
317      * @param last
```

```
318      *            the index of the last entry in the heap
319      * @param order
320      *            total preorder for sorting
321      * @return true if the subtree of the given {@code array}
    rooted at the
322      *         given {@code top} is a heap; false otherwise
323      * @requires <pre>
324      * 0 <= top  and  last < |array|  and
325      * for all i: integer
326      *     where (0 <= i  and  i < |array|)
327      *   ([entry at position i in array is not null])  and
328      * [subtree rooted at {@code top} is a complete binary tree]
329      * </pre>
330      * @ensures <pre>
331      * isHeap = SUBTREE_IS_HEAP(array, top, last,
332      *     [relation computed by order.compare method])
333      * </pre>
334      */
335     private static <T> boolean isHeap(T[] array, int top, int
    last,
336             Comparator<T> order) {
337         assert array != null : "Violation of: array is not null";
338         assert 0 <= top : "Violation of: 0 <= top";
339         assert last < array.length : "Violation of: last < |
    array|";
340         for (int i = 0; i < array.length; i++) {
341             assert array[i] != null : ""
342                     + "Violation of: all entries in array are not
    null";
343         }
344         /*
345          * No need to check the other requires clause, because it
    must be true
346          * when using the Array representation for a complete
    binary tree.
347          */
348         int left = 2 * top + 1;
349         boolean isHeap = true;
350         if (left <= last) {
351             isHeap = (order.compare(array[top], array[left]) <= 0)
352                     && isHeap(array, left, last, order);
353             int right = left + 1;
354             if (isHeap && (right <= last)) {
355                 isHeap = (order.compare(array[top], array[right])
```

```
    <= 0)
356                         && isHeap(array, right, last, order);
357             }
358         }
359         return isHeap;
360     }
361
362     /**
363      * Checks that the part of the convention repeated below holds
    for the
364      * current representation.
365      *
366      * @return true if the convention holds (or if assertion
    checking is off);
367      *         otherwise reports a violated assertion
368      * @convention <pre>
369      * if $this.insertionMode then
370      *   $this.heapSize = 0
371      * else
372      *   $this.entries = <>  and
373      *   for all i: integer
374      *       where (0 <= i  and  i < |$this.heap|)
375      *     ([entry at position i in $this.heap is not null])  and
376      *   SUBTREE_IS_HEAP($this.heap, 0, $this.heapSize - 1,
377      *     [relation computed by $this.machineOrder.compare
    method])  and
378      *   0 <= $this.heapSize <= |$this.heap|
379      * </pre>
380      */
381     private boolean conventionHolds() {
382         if (this.insertionMode) {
383             assert this.heapSize == 0 : ""
384                     + "Violation of: if $this.insertionMode then
    $this.heapSize = 0";
385         } else {
386             assert this.entries.length() == 0 : ""
387                     + "Violation of: if not $this.insertionMode
    then $this.entries = <>";
388             assert 0 <= this.heapSize : ""
389                     + "Violation of: if not $this.insertionMode
    then 0 <= $this.heapSize";
390             assert this.heapSize <= this.heap.length : ""
391                     + "Violation of: if not $this.insertionMode
    then"
```

```
392                     + " $this.heapSize <= |$this.heap|";
393             for (int i = 0; i < this.heap.length; i++) {
394                 assert this.heap[i] != null : ""
395                         + "Violation of: if not
    $this.insertionMode then"
396                         + " all entries in $this.heap are not
    null";
397             }
398             assert isHeap(this.heap, 0, this.heapSize - 1,
399                     this.machineOrder) : ""
400                             + "Violation of: if not
    $this.insertionMode then"
401                             + " SUBTREE_IS_HEAP($this.heap, 0,
    $this.heapSize - 1,"
402                             + " [relation computed by
    $this.machineOrder.compare"
403                             + " method])";
404         }
405         return true;
406     }
407
408     /**
409      * Creator of initial representation.
410      *
411      * @param order
412      *            total preorder for sorting
413      * @requires IS_TOTAL_PREORDER([relation computed by
    order.compare method]
414      * @ensures <pre>
415      * $this.insertionMode = true  and
416      * $this.machineOrder = order  and
417      * $this.entries = <>  and
418      * $this.heapSize = 0
419      * </pre>
420      */
421     private void createNewRep(Comparator<T> order) {
422         // created according to the representation invariant
423         this.machineOrder = order;
424         this.insertionMode = true;
425         this.entries = new Queue1L<T>();
426         this.heapSize = 0;
427
428     }
429
```

```
430     /*
431      * Constructors
    ------------------------------------------------------------
432      */
433
434     /**
435      * Constructor from order.
436      *
437      * @param order
438      *            total preorder for sorting
439      */
440     public SortingMachine5a(Comparator<T> order) {
441         this.createNewRep(order);
442         assert this.conventionHolds();
443     }
444
445     /*
446      * Standard methods
    ----------------------------------------------------------
447      */
448
449     @SuppressWarnings("unchecked")
450     @Override
451     public final SortingMachine<T> newInstance() {
452         try {
453             return
    this.getClass().getConstructor(Comparator.class)
454                     .newInstance(this.machineOrder);
455         } catch (ReflectiveOperationException e) {
456             throw new AssertionError(
457                     "Cannot construct object of type " +
    this.getClass());
458         }
459     }
460
461     @Override
462     public final void clear() {
463         this.createNewRep(this.machineOrder);
464         assert this.conventionHolds();
465     }
466
467     @Override
468     public final void transferFrom(SortingMachine<T> source) {
469         assert source != null : "Violation of: source is not
```

```
    null";
470         assert source != this : "Violation of: source is not
    this";
471         assert source instanceof SortingMachine5a<?> : ""
472                 + "Violation of: source is of dynamic type
    SortingMachine5a<?>";
473         /*
474          * This cast cannot fail since the assert above would have
    stopped
475          * execution in that case: source must be of dynamic type
476          * SortingMachine5a<?>, and the ? must be T or the call
    would not have
477          * compiled.
478          */
479         SortingMachine5a<T> localSource = (SortingMachine5a<T>)
    source;
480         this.insertionMode = localSource.insertionMode;
481         this.machineOrder = localSource.machineOrder;
482         this.entries = localSource.entries;
483         this.heap = localSource.heap;
484         this.heapSize = localSource.heapSize;
485         localSource.createNewRep(localSource.machineOrder);
486         assert this.conventionHolds();
487         assert localSource.conventionHolds();
488     }
489
490     /*
491      * Kernel methods
    ------------------------------------------------------------
492      */
493
494     @Override
495     public final void add(T x) {
496         assert x != null : "Violation of: x is not null";
497         assert this.isInInsertionMode() : "Violation of:
    this.insertion_mode";
498
499         this.entries.enqueue(x);
500
501         assert this.conventionHolds();
502     }
503
504     @Override
505     public final void changeToExtractionMode() {
```

```
506         assert this.isInInsertionMode() : "Violation of:
    this.insertion_mode";
507
508         this.insertionMode = false;
509         //once extraction mode is activated, must take all
510         //entries out of the queue and sort them into a heap
511         this.heap = buildHeap(this.entries, this.machineOrder);
512         this.heapSize = this.heap.length;
513
514         assert this.conventionHolds();
515     }
516
517     @Override
518     public final T removeFirst() {
519         assert !this
520                 .isInInsertionMode() : "Violation of: not
    this.insertion_mode";
521         assert this.size() > 0 : "Violation of: this.contents /=
    {}";
522
523         T first = this.heap[0]; //follows siftDown algorithm from
    class
524         exchangeEntries(this.heap, 0, this.heapSize - 1);
525         this.heapSize--;
526         siftDown(this.heap, 0, this.heapSize - 1,
    this.machineOrder);
527
528         assert this.conventionHolds();
529         return first;
530     }
531
532     @Override
533     public final boolean isInInsertionMode() {
534         assert this.conventionHolds();
535         return this.insertionMode;
536     }
537
538     @Override
539     public final Comparator<T> order() {
540         assert this.conventionHolds();
541         return this.machineOrder;
542     }
543
544     @Override
```

```
545     public final int size() {
546
547         assert this.conventionHolds();
548         int size = this.entries.length();
549         //returns the length of the queue or the size of the array
550         //depending on if the machine is in insertion or
    extraction mode
551         //since when the machine is in insertionMode, heap size is
    0
552         if (!this.insertionMode) {
553             size = this.heapSize;
554         }
555         return size;
556     }
557
558     @Override
559     public final Iterator<T> iterator() {
560         return new SortingMachine5aIterator();
561     }
562
563     /**
564      * Implementation of {@code Iterator} interface for
565      * {@code SortingMachine5a}.
566      */
567     private final class SortingMachine5aIterator implements
    Iterator<T> {
568
569         /**
570          * Representation iterator when in insertion mode.
571          */
572         private Iterator<T> queueIterator;
573
574         /**
575          * Representation iterator count when in extraction mode.
576          */
577         private int arrayCurrentIndex;
578
579         /**
580          * No-argument constructor.
581          */
582         private SortingMachine5aIterator() {
583             if (SortingMachine5a.this.insertionMode) {
584                 this.queueIterator =
    SortingMachine5a.this.entries.iterator();
```

```
585             } else {
586                 this.arrayCurrentIndex = 0;
587             }
588             assert SortingMachine5a.this.conventionHolds();
589         }
590 
591         @Override
592         public boolean hasNext() {
593             boolean hasNext;
594             if (SortingMachine5a.this.insertionMode) {
595                 hasNext = this.queueIterator.hasNext();
596             } else {
597                 hasNext = this.arrayCurrentIndex <
    SortingMachine5a.this.heapSize;
598             }
599             assert SortingMachine5a.this.conventionHolds();
600             return hasNext;
601         }
602 
603         @Override
604         public T next() {
605             assert this.hasNext() : "Violation of: ~this.unseen /=
    <>";
606             if (!this.hasNext()) {
607                 /*
608                  * Exception is supposed to be thrown in this
    case, but with
609                  * assertion-checking enabled it cannot happen
    because of assert
610                  * above.
611                  */
612                 throw new NoSuchElementException();
613             }
614             T next;
615             if (SortingMachine5a.this.insertionMode) {
616                 next = this.queueIterator.next();
617             } else {
618                 next =
    SortingMachine5a.this.heap[this.arrayCurrentIndex];
619                 this.arrayCurrentIndex++;
620             }
621             assert SortingMachine5a.this.conventionHolds();
622             return next;
623         }
```

```
624
625         @Override
626         public void remove() {
627             throw new UnsupportedOperationException(
628                     "remove operation not supported");
629         }
630
631     }
632
633 }
634
```