```
 1 import java.lang.reflect.Constructor;
10
11 /**
12  * {@code SortingMachine} represented as a {@code Queue} and an array (using an
13  * embedding of heap sort), with implementations of primary methods.
14  *
15  * @param <T>
16  *            type of {@code SortingMachine} entries
17  * @mathdefinitions <pre>
18  * IS_TOTAL_PREORDER (
19  *   r: binary relation on T
20  *  ) : boolean is
21  *  for all x, y, z: T
22  *   ((r(x, y) or r(y, x))  and
23  *    (if (r(x, y) and r(y, z)) then r(x, z)))
24  *
25  * SUBTREE_IS_HEAP (
26  *   a: string of T,
27  *   start: integer,
28  *   stop: integer,
29  *   r: binary relation on T
30  *  ) : boolean is
31  *  [the subtree of a (when a is interpreted as a complete binary tree) rooted
32  *   at index start and only through entry stop of a satisfies the heap
33  *   ordering property according to the relation r]
34  *
35  * SUBTREE_ARRAY_ENTRIES (
36  *   a: string of T,
37  *   start: integer,
38  *   stop: integer
39  *  ) : finite multiset of T is
40  *  [the multiset of entries in a that belong to the subtree of a
41  *   (when a is interpreted as a complete binary tree) rooted at
42  *   index start and only through entry stop]
43  * </pre>
44  * @convention <pre>
45  * IS_TOTAL_PREORDER([relation computed by $this.machineOrder.compare method]  and
46  * if $this.insertionMode then
47  *   $this.heapSize = 0
48  * else
49  *   $this.entries = <>  and
50  *   for all i: integer
51  *       where (0 <= i  and  i < |$this.heap|)
52  *     ([entry at position i in $this.heap is not null])  and
53  *   SUBTREE_IS_HEAP($this.heap, 0, $this.heapSize - 1,
54  *     [relation computed by $this.machineOrder.compare method])  and
55  *   0 <= $this.heapSize <= |$this.heap|
56  * </pre>
57  * @correspondence <pre>
58  * if $this.insertionMode then
59  *   this = (true, $this.machineOrder, multiset_entries($this.entries))
60  * else
61  *   this = (false, $this.machineOrder, multiset_entries($this.heap[0, $this.heapSize)))
62  * </pre>
63  *
64  * @author Put your name here
65  *
```

```
 */
public class SortingMachine5a<T> extends SortingMachineSecondary<T> {

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

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

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

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

    /**
     * Heap.
     */
    private T[] heap;

    /**
     * Heap size.
     */
    private int heapSize;

    /**
     * Exchanges entries at indices {@code i} and {@code j} of {@code array}.
     *
     * @param <T>
     *            type of array entries
     * @param array
     *            the array whose entries are to be exchanged
     * @param i
     *            one index
     * @param j
     *            the other index
     * @updates array
     * @requires 0 <= i < |array| and 0 <= j < |array|
     * @ensures array = [#array with entries at indices i and j exchanged]
     */
    private static <T> void exchangeEntries(T[] array, int i, int j) {
        assert array != null : "Violation of: array is not null";
        assert 0 <= i : "Violation of: 0 <= i";
        assert i < array.length : "Violation of: i < |array|";
        assert 0 <= j : "Violation of: 0 <= j";
        assert j < array.length : "Violation of: j < |array|";

        T temp;
        T atI = array[i];
        temp = atI;
```

```
123         array[i] = array[j];
124         array[j] = temp;
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
187         // *** you must use the recursive algorithm discussed in class ***
188
189         // if not at last child of heap
190         if (top != last) {
191             if (order.compare(array[top], array[top * 2 + 1]) <= 0
192                     && top * 2 + 1 <= last) {
193                 // if left subtree exists and
194                 // if top is <= left subtree root, -> swap top with left subtree root
195                 // maintaining; left <= middle <= right
196                 exchangeEntries(array, top, top * 2 + 1);
197                 siftDown(array, top * 2 + 1, last, order);
198
199             } else if (order.compare(array[top], array[top * 2 + 2]) >= 0
200                     && top * 2 + 2 <= last) {
201                 // if right subtree exists and
202                 // else if top is >= right subtree root, -> swap top with right root
203                 // maintaining; left <= top <= right
204                 exchangeEntries(array, top, top * 2 + 1);
205                 siftDown(array, top * 2 + 2, last, order);
206
207             // if top is neither <= left root or >= right root, it is already ordered
208         }
209
210     }
211
212     /**
213      * Heapifies the subtree of the given array rooted at the given {@code top}.
214      *
215      * @param <T>
216      *            type of array entries
217      * @param array
218      *            the complete binary tree
219      * @param top
220      *            the index of the root of the "subtree" to heapify
221      * @param order
222      *            the total preorder for sorting
223      * @updates array
224      * @requires <pre>
225      * 0 <= top  and
226      * for all i: integer
227      *     where (0 <= i  and  i < |array|)
228      *   ([entry at position i in array is not null])  and
229      * [subtree rooted at {@code top} is a complete binary tree]  and
230      * IS_TOTAL_PREORDER([relation computed by order.compare method])
231      * </pre>
232      * @ensures <pre>
233      * SUBTREE_IS_HEAP(array, top, |array| - 1,
234      *     [relation computed by order.compare method])  and
235      * perms(array, #array)
236      * </pre>
```

```
237      */
238     private static <T> void heapify(T[] array, int top, Comparator<T> order) {
239         assert array != null : "Violation of: array is not null";
240         assert order != null : "Violation of: order is not null";
241         assert 0 <= top : "Violation of: 0 <= top";
242         for (int i = 0; i < array.length; i++) {
243             assert array[i] != null : ""
244                     + "Violation of: all entries in array are not null";
245         }
246         /*
247          * Impractical to check last requires clause; no need to check the other
248          * requires clause, because it must be true when using the array
249          * representation for a complete binary tree.
250          */
251
252         // *** you must use the recursive algorithm discussed in class ***
253         if (top * 2 + 2 <= array.length) { // right subtree implies left
254             heapify(array, top * 2 + 1, order);
255             heapify(array, top * 2 + 2, order);
256             siftDown(array, top, array.length, order);
257         } else if (top * 2 + 1 <= array.length) { // only left subtree
258             heapify(array, top * 2 + 1, order);
259             siftDown(array, top, array.length, order);
260         }
261
262     }
263
264     /**
265      * Constructs and returns an array representing a heap with the entries from
266      * the given {@code Queue}.
267      *
268      * @param <T>
269      *            type of {@code Queue} and array entries
270      * @param q
271      *            the {@code Queue} with the entries for the heap
272      * @param order
273      *            the total preorder for sorting
274      * @return the array representation of a heap
275      * @clears q
276      * @requires IS_TOTAL_PREORDER([relation computed by order.compare method])
277      * @ensures <pre>
278      * SUBTREE_IS_HEAP(buildHeap, 0, |buildHeap| - 1)  and
279      * perms(buildHeap, #q)  and
280      * for all i: integer
281      *     where (0 <= i  and  i < |buildHeap|)
282      *   ([entry at position i in buildHeap is not null])  and
283      * </pre>
284      */
285     @SuppressWarnings("unchecked")
286     private static <T> T[] buildHeap(Queue<T> q, Comparator<T> order) {
287         assert q != null : "Violation of: q is not null";
288         assert order != null : "Violation of: order is not null";
289         /*
290          * Impractical to check the requires clause.
291          */
292         /*
293          * With "new T[...]" in place of "new Object[...]" it does not compile;
```

```
294          * as shown, it results in a warning about an unchecked cast, though it
295          * cannot fail.
296          */
297         T[] heap = (T[]) (new Object[q.length()]);
298 
299         for (int i = 0; i < heap.length; i++) {
300             heap[i] = q.dequeue();
301         }
302         heapify(heap, 0, order);
303 
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         this.insertionMode = true;
423         this.machineOrder = order;
424         this.entries = new Queue1L<T>();
425         this.heapSize = 0;
426
427     }
428
429     /*
430      * Constructors ------------------------------------------------------------
431      */
432
433     /**
434      * Constructor from order.
435      *
436      * @param order
437      *            total preorder for sorting
438      */
439     public SortingMachine5a(Comparator<T> order) {
440         this.createNewRep(order);
441         assert this.conventionHolds();
442     }
443
444     /*
445      * Standard methods --------------------------------------------------------
446      */
447
448     @SuppressWarnings("unchecked")
449     @Override
450     public final SortingMachine<T> newInstance() {
451         try {
452             Constructor<?> c = this.getClass().getConstructor(Comparator.class);
453             return (SortingMachine<T>) c.newInstance(this.machineOrder);
454         } catch (ReflectiveOperationException e) {
455             throw new AssertionError(
456                     "Cannot construct object of type " + this.getClass());
457         }
458     }
459
460     @Override
461     public final void clear() {
462         this.createNewRep(this.machineOrder);
463         assert this.conventionHolds();
464     }
```

```
465
466     @Override
467     public final void transferFrom(SortingMachine<T> source) {
468         assert source != null : "Violation of: source is not null";
469         assert source != this : "Violation of: source is not this";
470         assert source instanceof SortingMachine5a<?> : ""
471                 + "Violation of: source is of dynamic type SortingMachine5a<?>";
472         /*
473          * This cast cannot fail since the assert above would have stopped
474          * execution in that case: source must be of dynamic type
475          * SortingMachine5a<?>, and the ? must be T or the call would not have
476          * compiled.
477          */
478         SortingMachine5a<T> localSource = (SortingMachine5a<T>) source;
479         this.insertionMode = localSource.insertionMode;
480         this.machineOrder = localSource.machineOrder;
481         this.entries = localSource.entries;
482         this.heap = localSource.heap;
483         this.heapSize = localSource.heapSize;
484         localSource.createNewRep(localSource.machineOrder);
485         assert this.conventionHolds();
486         assert localSource.conventionHolds();
487     }
488
489     /*
490      * Kernel methods ------------------------------------------------------
491      */
492
493     @Override
494     public final void add(T x) {
495         assert x != null : "Violation of: x is not null";
496         assert this.isInInsertionMode() : "Violation of: this.insertion_mode";
497
498         this.entries.enqueue(x);
499
500         assert this.conventionHolds();
501     }
502
503     @Override
504     public final void changeToExtractionMode() {
505         assert this.isInInsertionMode() : "Violation of: this.insertion_mode";
506
507         this.insertionMode = false;
508         buildHeap(this.entries, this.machineOrder);
509
510         assert this.conventionHolds();
511     }
512
513     @Override
514     public final T removeFirst() {
515         assert !this
516                 .isInInsertionMode() : "Violation of: not this.insertion_mode";
517         assert this.size() > 0 : "Violation of: this.contents /= {}";
518
519         T first = this.heap[0];
520
521         exchangeEntries(this.heap, 0, this.heapSize - 1);
```

```
522         siftDown(this.heap, 0, this.heapSize - 1, this.machineOrder);
523
524         assert this.conventionHolds();
525         return first;
526     }
527
528     @Override
529     public final boolean isInInsertionMode() {
530         assert this.conventionHolds();
531         return this.insertionMode;
532     }
533
534     @Override
535     public final Comparator<T> order() {
536         assert this.conventionHolds();
537         return this.machineOrder;
538     }
539
540     @Override
541     public final int size() {
542
543         int size;
544         if (this.insertionMode) {
545             size = this.entries.length();
546         } else {
547             size = this.heapSize;
548         }
549
550         assert this.conventionHolds();
551         return size;
552     }
553
554     @Override
555     public final Iterator<T> iterator() {
556         return new SortingMachine5aIterator();
557     }
558
559     /**
560      * Implementation of {@code Iterator} interface for
561      * {@code SortingMachine5a}.
562      */
563     private final class SortingMachine5aIterator implements Iterator<T> {
564
565         /**
566          * Representation iterator when in insertion mode.
567          */
568         private Iterator<T> queueIterator;
569
570         /**
571          * Representation iterator count when in extraction mode.
572          */
573         private int arrayCurrentIndex;
574
575         /**
576          * No-argument constructor.
577          */
578         private SortingMachine5aIterator() {
```

```
579             if (SortingMachine5a.this.insertionMode) {
580                 this.queueIterator = SortingMachine5a.this.entries.iterator();
581             } else {
582                 this.arrayCurrentIndex = 0;
583             }
584             assert SortingMachine5a.this.conventionHolds();
585         }
586
587         @Override
588         public boolean hasNext() {
589             boolean hasNext;
590             if (SortingMachine5a.this.insertionMode) {
591                 hasNext = this.queueIterator.hasNext();
592             } else {
593                 hasNext = this.arrayCurrentIndex < SortingMachine5a.this.heapSize;
594             }
595             assert SortingMachine5a.this.conventionHolds();
596             return hasNext;
597         }
598
599         @Override
600         public T next() {
601             assert this.hasNext() : "Violation of: ~this.unseen /= <>";
602             if (!this.hasNext()) {
603                 /*
604                  * Exception is supposed to be thrown in this case, but with
605                  * assertion-checking enabled it cannot happen because of assert
606                  * above.
607                  */
608                 throw new NoSuchElementException();
609             }
610             T next;
611             if (SortingMachine5a.this.insertionMode) {
612                 next = this.queueIterator.next();
613             } else {
614                 next = SortingMachine5a.this.heap[this.arrayCurrentIndex];
615                 this.arrayCurrentIndex++;
616             }
617             assert SortingMachine5a.this.conventionHolds();
618             return next;
619         }
620
621         @Override
622         public void remove() {
623             throw new UnsupportedOperationException(
624                     "remove operation not supported");
625         }
626
627     }
628
629 }
630
```