```
 1 import java.util.Comparator;
 9
10 /**
11  * {@code SortingMachine} represented as a {@code Queue} and an array (using an
12  * embedding of heap sort), with implementations of primary methods.
13  *
14  * @param <T>
15  *            type of {@code SortingMachine} entries
16  * @mathdefinitions <pre>
17  * IS_TOTAL_PREORDER (
18  *   r: binary relation on T
19  *  ) : boolean is
20  *  for all x, y, z: T
21  *   ((r(x, y) or r(y, x))  and
22  *    (if (r(x, y) and r(y, z)) then r(x, z)))
23  *
24  * SUBTREE_IS_HEAP (
25  *   a: string of T,
26  *   start: integer,
27  *   stop: integer,
28  *   r: binary relation on T
29  *  ) : boolean is
30  *  [the subtree of a (when a is interpreted as a complete binary tree) rooted
31  *   at index start and only through entry stop of a satisfies the heap
32  *   ordering property according to the relation r]
33  *
34  * SUBTREE_ARRAY_ENTRIES (
35  *   a: string of T,
36  *   start: integer,
37  *   stop: integer
38  *  ) : finite multiset of T is
39  *  [the multiset of entries in a that belong to the subtree of a
40  *   (when a is interpreted as a complete binary tree) rooted at
41  *   index start and only through entry stop]
42  * </pre>
43  * @convention <pre>
44  * IS_TOTAL_PREORDER([relation computed by $this.machineOrder.compare method]  and
45  * if $this.insertionMode then
46  *   $this.heapSize = 0
47  * else
48  *   $this.entries = <>  and
49  *   for all i: integer
50  *       where (0 <= i  and  i < |$this.heap|)
51  *     ([entry at position i in $this.heap is not null])  and
52  *   SUBTREE_IS_HEAP($this.heap, 0, $this.heapSize - 1,
53  *     [relation computed by $this.machineOrder.compare method])  and
54  *   0 <= $this.heapSize <= |$this.heap|
55  * </pre>
56  * @correspondence <pre>
57  * if $this.insertionMode then
58  *   this = (true, $this.machineOrder, multiset_entries($this.entries))
59  * else
60  *   this = (false, $this.machineOrder, multiset_entries($this.heap[0,
  $this.heapSize)))
61  * </pre>
62  *
63  * @author Gabe Azzarita
64  * @author Ty Fredrick
65  *
```

```
 66 */
 67public class SortingMachine5a<T> extends SortingMachineSecondary<T> {
 68
 69    /*
 70     * Private members --------------------------------------------------------
 71     */
 72
 73    /**
 74     * Order.
 75     */
 76    private Comparator<T> machineOrder;
 77
 78    /**
 79     * Insertion mode.
 80     */
 81    private boolean insertionMode;
 82
 83    /**
 84     * Entries.
 85     */
 86    private Queue<T> entries;
 87
 88    /**
 89     * Heap.
 90     */
 91    private T[] heap;
 92
 93    /**
 94     * Heap size.
 95     */
 96    private int heapSize;
 97
 98    /**
 99     * Exchanges entries at indices {@code i} and {@code j} of {@code array}.
100     *
101     * @param <T>
102     *            type of array entries
103     * @param array
104     *            the array whose entries are to be exchanged
105     * @param i
106     *            one index
107     * @param j
108     *            the other index
109     * @updates array
110     * @requires 0 <= i < |array| and 0 <= j < |array|
111     * @ensures array = [#array with entries at indices i and j exchanged]
112     */
113    private static <T> void exchangeEntries(T[] array, int i, int j) {
114        assert array != null : "Violation of: array is not null";
115        assert 0 <= i : "Violation of: 0 <= i";
116        assert i < array.length : "Violation of: i < |array|";
117        assert 0 <= j : "Violation of: 0 <= j";
118        assert j < array.length : "Violation of: j < |array|";
119
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
187         if (!isHeap(array, top, last, order)) {
188             int leftIndex = 2 * top + 1;
189             int rightIndex = 2 * top + 2;
190
191             // if left doesn't exist neither will right
192             if (leftIndex <= last) {
193                 // check that right exists
194                 if (rightIndex <= last) {
195                     // swap root with smaller child and sift down
196                     if (order.compare(array[rightIndex],
197                             array[leftIndex]) > 0) {
198                         exchangeEntries(array, top, leftIndex);
199                         siftDown(array, leftIndex, last, order);
200                     } else {
201                         exchangeEntries(array, top, rightIndex);
202                         siftDown(array, rightIndex, last, order);
203                     }
204                 } else {
205                     // if left doesn't exist, exchange with left
206                     exchangeEntries(array, top, leftIndex);
207                     siftDown(array, leftIndex, last, order);
208                 }
209             }
210
211         }
212
213     }
214
215     /**
216      * Heapifies the subtree of the given array rooted at the given {@code top}.
217      *
218      * @param <T>
219      *            type of array entries
220      * @param array
221      *            the complete binary tree
222      * @param top
223      *            the index of the root of the "subtree" to heapify
224      * @param order
225      *            the total preorder for sorting
226      * @updates array
227      * @requires <pre>
228      * 0 <= top  and
229      * for all i: integer
230      *     where (0 <= i  and  i < |array|)
231      *   ([entry at position i in array is not null])  and
232      * [subtree rooted at {@code top} is a complete binary tree]  and
233      * IS_TOTAL_PREORDER([relation computed by order.compare method])
234      * </pre>
235      * @ensures <pre>
236      * SUBTREE_IS_HEAP(array, top, |array| - 1,
237      *     [relation computed by order.compare method])  and
238      * perms(array, #array)
239      * </pre>
240      */
241     private static <T> void heapify(T[] array, int top, Comparator<T> order) {
242         assert array != null : "Violation of: array is not null";
```

```
243         assert order != null : "Violation of: order is not null";
244         assert 0 <= top : "Violation of: 0 <= top";
245         for (int i = 0; i < array.length; i++) {
246             assert array[i] != null : ""
247                     + "Violation of: all entries in array are not null";
248         }
249         /*
250          * Impractical to check last requires clause; no need to check the other
251          * requires clause, because it must be true when using the array
252          * representation for a complete binary tree.
253          */
254
255         int left = 2 * top + 1;
256         int right = 2 * top + 2;
257
258         // Make sure left and right "trees" exist
259         if (right < array.length) {
260             // if right exists, so does left
261             heapify(array, left, order);
262             heapify(array, right, order);
263         } else if (left < array.length) {
264             // else check if left exists
265             heapify(array, left, order);
266         }
267         // after heapifying left and right (if exist) sift down original root
268         siftDown(array, top, array.length - 1, order);
269
270         // *** you must use the recursive algorithm discussed in class ***
271
272     }
273
274     /**
275      * Constructs and returns an array representing a heap with the entries from
276      * the given {@code Queue}.
277      *
278      * @param <T>
279      *            type of {@code Queue} and array entries
280      * @param q
281      *            the {@code Queue} with the entries for the heap
282      * @param order
283      *            the total preorder for sorting
284      * @return the array representation of a heap
285      * @clears q
286      * @requires IS_TOTAL_PREORDER([relation computed by order.compare method])
287      * @ensures <pre>
288      * SUBTREE_IS_HEAP(buildHeap, 0, |buildHeap| - 1)  and
289      * perms(buildHeap, #q)  and
290      * for all i: integer
291      *     where (0 <= i  and  i < |buildHeap|)
292      *   ([entry at position i in buildHeap is not null])  and
293      * </pre>
294      */
295     @SuppressWarnings("unchecked")
296     private static <T> T[] buildHeap(Queue<T> q, Comparator<T> order) {
297         assert q != null : "Violation of: q is not null";
298         assert order != null : "Violation of: order is not null";
299         /*
300          * Impractical to check the requires clause.
301          */
```

```
302
303         /*
304          * With "new T[...]" in place of "new Object[...]" it does not compile;
305          * as shown, it results in a warning about an unchecked cast, though it
306          * cannot fail.
307          */
308
309         T[] heap = (T[]) (new Object[q.length()]);
310         int formerLength = q.length();
311
312         for (int k = 0; k < formerLength; k++) {
313             // dequeue item from queue and add to heap
314             heap[k] = q.dequeue();
315         }
316
317         heapify(heap, 0, order);
318         return heap;
319     }
320
321     /**
322      * Checks if the subtree of the given {@code array} rooted at the given
323      * {@code top} is a heap.
324      *
325      * @param <T>
326      *            type of array entries
327      * @param array
328      *            the complete binary tree
329      * @param top
330      *            the index of the root of the "subtree"
331      * @param last
332      *            the index of the last entry in the heap
333      * @param order
334      *            total preorder for sorting
335      * @return true if the subtree of the given {@code array} rooted at the
336      *         given {@code top} is a heap; false otherwise
337      * @requires <pre>
338      * 0 <= top  and  last < |array|  and
339      * for all i: integer
340      *     where (0 <= i  and  i < |array|)
341      *   ([entry at position i in array is not null])  and
342      * [subtree rooted at {@code top} is a complete binary tree]
343      * </pre>
344      * @ensures <pre>
345      * isHeap = SUBTREE_IS_HEAP(array, top, last,
346      *     [relation computed by order.compare method])
347      * </pre>
348      */
349     private static <T> boolean isHeap(T[] array, int top, int last,
350             Comparator<T> order) {
351         assert array != null : "Violation of: array is not null";
352         assert 0 <= top : "Violation of: 0 <= top";
353         assert last < array.length : "Violation of: last < |array|";
354         for (int i = 0; i < array.length; i++) {
355             assert array[i] != null : ""
356                     + "Violation of: all entries in array are not null";
357         }
358         /*
359          * No need to check the other requires clause, because it must be true
360          * when using the Array representation for a complete binary tree.
```

```
361          */
362
363         int left = 2 * top + 1;
364         int right = 2 * top + 2;
365         boolean heapL = true;
366         boolean heapR = true;
367
368         /*
369          * We can check for heap by making sure that both the left and right
370          * children are smaller than the "root" and then checking that the left
371          * and right are heaps
372          */
373
374         // check if left exists, if it doesn't, neither will right
375         if (left <= last) {
376             heapL = (order.compare(array[top], array[left]) <= 0)
377                     && isHeap(array, left, last, order);
378             // if left is a heap, let's check the right side (if exists)
379             if (heapL && (right <= last)) {
380                 heapR = (order.compare(array[top], array[right]) <= 0)
381                         && isHeap(array, right, last, order);
382             }
383         }
384         return heapL && heapR;
385     }
386
387     /**
388      * Checks that the part of the convention repeated below holds for the
389      * current representation.
390      *
391      * @return true if the convention holds (or if assertion checking is off);
392      *         otherwise reports a violated assertion
393      * @convention <pre>
394      * if $this.insertionMode then
395      *   $this.heapSize = 0
396      * else
397      *   $this.entries = <>  and
398      *   for all i: integer
399      *       where (0 <= i  and  i < |$this.heap|)
400      *     ([entry at position i in $this.heap is not null])  and
401      *   SUBTREE_IS_HEAP($this.heap, 0, $this.heapSize - 1,
402      *     [relation computed by $this.machineOrder.compare method])  and
403      *   0 <= $this.heapSize <= |$this.heap|
404      * </pre>
405      */
406     private boolean conventionHolds() {
407         if (this.insertionMode) {
408             assert this.heapSize == 0 : ""
409                     + "Violation of: if $this.insertionMode then $this.heapSize = 0";
410         } else {
411             assert this.entries.length() == 0 : ""
412                     + "Violation of: if not $this.insertionMode then $this.entries =
    <>";
413             assert 0 <= this.heapSize : ""
414                     + "Violation of: if not $this.insertionMode then 0 <=
    $this.heapSize";
415             assert this.heapSize <= this.heap.length : ""
416                     + "Violation of: if not $this.insertionMode then"
417                     + " $this.heapSize <= |$this.heap|";
```

```
418             for (int i = 0; i < this.heap.length; i++) {
419                 assert this.heap[i] != null : ""
420                         + "Violation of: if not $this.insertionMode then"
421                         + " all entries in $this.heap are not null";
422             }
423             assert isHeap(this.heap, 0, this.heapSize - 1,
424                     this.machineOrder) : ""
425                             + "Violation of: if not $this.insertionMode then"
426                             + " SUBTREE_IS_HEAP($this.heap, 0, $this.heapSize - 1,"
427                             + " [relation computed by $this.machineOrder.compare"
428                             + " method])";
429         }
430         return true;
431     }
432
433     /**
434      * Creator of initial representation.
435      *
436      * @param order
437      *            total preorder for sorting
438      * @requires IS_TOTAL_PREORDER([relation computed by order.compare method]
439      * @ensures <pre>
440      * $this.insertionMode = true  and
441      * $this.machineOrder = order  and
442      * $this.entries = <>  and
443      * $this.heapSize = 0
444      * </pre>
445      */
446     private void createNewRep(Comparator<T> order) {
447
448         this.machineOrder = order;
449         this.insertionMode = true;
450         this.entries = new Queue1L<T>();
451         this.heapSize = 0;
452
453     }
454
455     /*
456      * Constructors ------------------------------------------------------------
457      */
458
459     /**
460      * Constructor from order.
461      *
462      * @param order
463      *            total preorder for sorting
464      */
465     public SortingMachine5a(Comparator<T> order) {
466         this.createNewRep(order);
467         assert this.conventionHolds();
468     }
469
470     /*
471      * Standard methods --------------------------------------------------------
472      */
473
474     @SuppressWarnings("unchecked")
475     @Override
476     public final SortingMachine<T> newInstance() {
```

```
477         try {
478             return this.getClass().getConstructor(Comparator.class)
479                     .newInstance(this.machineOrder);
480         } catch (ReflectiveOperationException e) {
481             throw new AssertionError(
482                     "Cannot construct object of type " + this.getClass());
483         }
484     }
485
486     @Override
487     public final void clear() {
488         this.createNewRep(this.machineOrder);
489         assert this.conventionHolds();
490     }
491
492     @Override
493     public final void transferFrom(SortingMachine<T> source) {
494         assert source != null : "Violation of: source is not null";
495         assert source != this : "Violation of: source is not this";
496         assert source instanceof SortingMachine5a<?> : ""
497                 + "Violation of: source is of dynamic type SortingMachine5a<?>";
498         /*
499          * This cast cannot fail since the assert above would have stopped
500          * execution in that case: source must be of dynamic type
501          * SortingMachine5a<?>, and the ? must be T or the call would not have
502          * compiled.
503          */
504         SortingMachine5a<T> localSource = (SortingMachine5a<T>) source;
505         this.insertionMode = localSource.insertionMode;
506         this.machineOrder = localSource.machineOrder;
507         this.entries = localSource.entries;
508         this.heap = localSource.heap;
509         this.heapSize = localSource.heapSize;
510         localSource.createNewRep(localSource.machineOrder);
511         assert this.conventionHolds();
512         assert localSource.conventionHolds();
513     }
514
515     /*
516      * Kernel methods ------------------------------------------------------
517      */
518
519     @Override
520     public final void add(T x) {
521         assert x != null : "Violation of: x is not null";
522         assert this.isInInsertionMode() : "Violation of: this.insertion_mode";
523
524         // enqueue item and increment heapSize
525         this.entries.enqueue(x);
526
527         assert this.conventionHolds();
528     }
529
530     @Override
531     public final void changeToExtractionMode() {
532         assert this.isInInsertionMode() : "Violation of: this.insertion_mode";
533
534         // change insertionMode and build heap
535         this.insertionMode = false;
```

```
536         this.heapSize = this.entries.length();
537         this.heap = buildHeap(this.entries, this.machineOrder);
538
539         assert this.conventionHolds();
540     }
541
542     @Override
543     public final T removeFirst() {
544         assert !this
545                 .isInInsertionMode() : "Violation of: not this.insertion_mode";
546         assert this.size() > 0 : "Violation of: this.contents /= {}";
547
548         // element to return
549         T first = this.heap[0];
550
551         // swap first element with last element, then reduce size by 1 to
552         // essential remove the original first element
553         exchangeEntries(this.heap, 0, this.heapSize - 1);
554         this.heapSize--;
555
556         // sift new "root" down to restore heap
557         siftDown(this.heap, 0, this.heapSize - 1, this.machineOrder);
558
559         assert this.conventionHolds();
560
561         return first;
562     }
563
564     @Override
565     public final boolean isInInsertionMode() {
566         assert this.conventionHolds();
567
568         return this.insertionMode;
569     }
570
571     @Override
572     public final Comparator<T> order() {
573         assert this.conventionHolds();
574
575         return this.machineOrder;
576     }
577
578     @Override
579     public final int size() {
580         assert this.conventionHolds();
581
582         int size = 0;
583         // if in insertionMode, heap is empty and queue contains all the items
584         if (this.insertionMode) {
585             size = this.entries.length();
586         } else {
587             // if in extractionMode, heap contains all the items
588             size = this.heapSize;
589         }
590
591         return size;
592     }
593
594     @Override
```

```
595     public final Iterator<T> iterator() {
596         return new SortingMachine5aIterator();
597     }
598
599     /**
600      * Implementation of {@code Iterator} interface for
601      * {@code SortingMachine5a}.
602      */
603     private final class SortingMachine5aIterator implements Iterator<T> {
604
605         /**
606          * Representation iterator when in insertion mode.
607          */
608         private Iterator<T> queueIterator;
609
610         /**
611          * Representation iterator count when in extraction mode.
612          */
613         private int arrayCurrentIndex;
614
615         /**
616          * No-argument constructor.
617          */
618         private SortingMachine5aIterator() {
619             if (SortingMachine5a.this.insertionMode) {
620                 this.queueIterator = SortingMachine5a.this.entries.iterator();
621             } else {
622                 this.arrayCurrentIndex = 0;
623             }
624             assert SortingMachine5a.this.conventionHolds();
625         }
626
627         @Override
628         public boolean hasNext() {
629             boolean hasNext;
630             if (SortingMachine5a.this.insertionMode) {
631                 hasNext = this.queueIterator.hasNext();
632             } else {
633                 hasNext = this.arrayCurrentIndex < SortingMachine5a.this.heapSize;
634             }
635             assert SortingMachine5a.this.conventionHolds();
636             return hasNext;
637         }
638
639         @Override
640         public T next() {
641             assert this.hasNext() : "Violation of: ~this.unseen /= <>";
642             if (!this.hasNext()) {
643                 /*
644                  * Exception is supposed to be thrown in this case, but with
645                  * assertion-checking enabled it cannot happen because of assert
646                  * above.
647                  */
648                 throw new NoSuchElementException();
649             }
650             T next;
651             if (SortingMachine5a.this.insertionMode) {
652                 next = this.queueIterator.next();
653             } else {
```

```
654                 next = SortingMachine5a.this.heap[this.arrayCurrentIndex];
655                 this.arrayCurrentIndex++;
656             }
657             assert SortingMachine5a.this.conventionHolds();
658             return next;
659         }
660
661         @Override
662         public void remove() {
663             throw new UnsupportedOperationException(
664                     "remove operation not supported");
665         }
666
667     }
668
669 }
670
```