```
 1 import java.util.Comparator;
 2 import java.util.Iterator;
 3 import java.util.NoSuchElementException;
 4
 5 import components.queue.Queue;
 6 import components.queue.Queue1L;
 7 import components.sortingmachine.SortingMachine;
 8 import components.sortingmachine.SortingMachineSecondary;
 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 Chloe Feller and Krish Patel
 64 *
 65 */
 66 public class SortingMachine5a<T> extends SortingMachineSecondary<T> {
 67
 68     /*
 69      * Private members ------------------------------------------------------
 70      */
 71
 72     /**
 73      * Order.
 74      */
 75     private Comparator<T> machineOrder;
 76
 77     /**
 78      * Insertion mode.
 79      */
 80     private boolean insertionMode;
 81
 82     /**
 83      * Entries.
 84      */
 85     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         T tmp = array[i];
120         array[i] = array[j];
121         array[j] = tmp;
122
123     }
124
125     /**
126      * Given an array that represents a complete binary tree and an index
127      * referring to the root of a subtree that would be a heap except for its
128      * root, sifts the root down to turn that whole subtree into a heap.
129      *
130      * @param <T>
131      *            type of array entries
132      * @param array
133      *            the complete binary tree
134      * @param top
135      *            the index of the root of the "subtree"
136      * @param last
137      *            the index of the last entry in the heap
138      * @param order
139      *            total preorder for sorting
140      * @updates array
141      * @requires <pre>
142      * 0 <= top  and  last < |array|  and
143      * for all i: integer
144      *     where (0 <= i  and  i < |array|)
145      *   ([entry at position i in array is not null])  and
146      * [subtree rooted at {@code top} is a complete binary tree]  and
147      * SUBTREE_IS_HEAP(array, 2 * top + 1, last,
148      *     [relation computed by order.compare method])  and
149      * SUBTREE_IS_HEAP(array, 2 * top + 2, last,
150      *     [relation computed by order.compare method])  and
151      * IS_TOTAL_PREORDER([relation computed by order.compare method])
152      * </pre>
153      * @ensures <pre>
154      * SUBTREE_IS_HEAP(array, top, last,
155      *     [relation computed by order.compare method])  and
156      * perms(array, #array)  and
157      * SUBTREE_ARRAY_ENTRIES(array, top, last) =
158      *  SUBTREE_ARRAY_ENTRIES(#array, top, last)  and
159      * [the other entries in array are the same as in #array]
160      * </pre>
161      */
162     private static <T> void siftDown(T[] array, int top, int last,
163             Comparator<T> order) {
164         assert array != null : "Violation of: array is not null";
165         assert order != null : "Violation of: order is not null";
166         assert 0 <= top : "Violation of: 0 <= top";
167         assert last < array.length : "Violation of: last < |array|";
168         for (int i = 0; i < array.length; i++) {
169             assert array[i] != null : ""
170                     + "Violation of: all entries in array are not null";
171         }
172         assert isHeap(array, 2 * top + 1, last, order) : ""
173                 + "Violation of: SUBTREE_IS_HEAP(array, 2 * top + 1, last,"
174                 + " [relation computed by order.compare method])";
175         assert isHeap(array, 2 * top + 2, last, order) : ""
176                 + "Violation of: SUBTREE_IS_HEAP(array, 2 * top + 2, last,"
```

```
177                 + " [relation computed by order.compare method])";
178         /*
179          * Impractical to check last requires clause; no need to check the other
180          * requires clause, because it must be true when using the array
181          * representation for a complete binary tree.
182          */
183
184         // *** you must use the recursive algorithm discussed in class ***
185         int left = 2 * top + 1;
186         int right = left + 2;
187
188         if (last - top > 0) {
189             if (!isHeap(array, top, last, order)) {
190                 int s = top;
191
192                 if (left <= last) {
193                     s = left;
194                 } else {
195                     s = right;
196                 }
197
198                 if (order.compare(array[top], array[s]) > 0) {
199                     exchangeEntries(array, top, s);
200                     siftDown(array, s, last, order);
201                 }
202             }
203         }
204     }
205
206     /**
207      * Heapifies the subtree of the given array rooted at the given {@code top}.
208      *
209      * @param <T>
210      *            type of array entries
211      * @param array
212      *            the complete binary tree
213      * @param top
214      *            the index of the root of the "subtree" to heapify
215      * @param order
216      *            the total preorder for sorting
217      * @updates array
218      * @requires <pre>
219      * 0 <= top  and
220      * for all i: integer
221      *     where (0 <= i  and  i < |array|)
222      *   ([entry at position i in array is not null])  and
223      * [subtree rooted at {@code top} is a complete binary tree]  and
224      * IS_TOTAL_PREORDER([relation computed by order.compare method])
225      * </pre>
226      * @ensures <pre>
227      * SUBTREE_IS_HEAP(array, top, |array| - 1,
228      *     [relation computed by order.compare method])  and
229      * perms(array, #array)
230      * </pre>
231      */
232     private static <T> void heapify(T[] array, int top, Comparator<T> order) {
233         assert array != null : "Violation of: array is not null";
234         assert order != null : "Violation of: order is not null";
235         assert 0 <= top : "Violation of: 0 <= top";
```

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

```
295         int i = 0;
296
297         while (q.length() > 0) {
298             heap[i] = q.dequeue();
299             i++;
300         }
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
423         this.insertionMode = true;
424         this.machineOrder = order;
425         this.heapSize = 0;
426         this.entries = new Queue1L<>();
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         this.heapSize = this.entries.length();
510         this.heap = buildHeap(this.entries, this.machineOrder);
511
512         assert this.conventionHolds();
513     }
514
515     @Override
516     public final T removeFirst() {
517         assert !this
518                 .isInInsertionMode() : "Violation of: not this.insertion_mode";
519         assert this.size() > 0 : "Violation of: this.contents /= {}";
520
521         T root = this.heap[0];
522
523         this.heapSize--;
524
525         if (this.heapSize > 1) {
526             exchangeEntries(this.heap, 0, this.heapSize);
527             siftDown(this.heap, 0, this.heapSize - 1, this.machineOrder);
528         }
```

```
529
530         assert this.conventionHolds();
531         // Fix this line to return the result after checking the convention.
532         return root;
533     }
534
535     @Override
536     public final boolean isInInsertionMode() {
537         assert this.conventionHolds();
538         return this.insertionMode;
539     }
540
541     @Override
542     public final Comparator<T> order() {
543         assert this.conventionHolds();
544         return this.machineOrder;
545     }
546
547     @Override
548     public final int size() {
549
550         int size = this.heapSize;
551
552         if (this.insertionMode) {
553             size = this.entries.length();
554         }
555         assert this.conventionHolds();
556         // Fix this line to return the result after checking the convention.
557         return size;
558     }
559
560     @Override
561     public final Iterator<T> iterator() {
562         return new SortingMachine5aIterator();
563     }
564
565     /**
566      * Implementation of {@code Iterator} interface for
567      * {@code SortingMachine5a}.
568      */
569     private final class SortingMachine5aIterator implements Iterator<T> {
570
571         /**
572          * Representation iterator when in insertion mode.
573          */
574         private Iterator<T> queueIterator;
575
576         /**
577          * Representation iterator count when in extraction mode.
578          */
579         private int arrayCurrentIndex;
580
581         /**
582          * No-argument constructor.
583          */
584         private SortingMachine5aIterator() {
585             if (SortingMachine5a.this.insertionMode) {
586                 this.queueIterator = SortingMachine5a.this.entries.iterator();
587             } else {
```

```
588                 this.arrayCurrentIndex = 0;
589             }
590             assert SortingMachine5a.this.conventionHolds();
591         }
592 
593         @Override
594         public boolean hasNext() {
595             boolean hasNext;
596             if (SortingMachine5a.this.insertionMode) {
597                 hasNext = this.queueIterator.hasNext();
598             } else {
599                 hasNext = this.arrayCurrentIndex < SortingMachine5a.this.heapSize;
600             }
601             assert SortingMachine5a.this.conventionHolds();
602             return hasNext;
603         }
604 
605         @Override
606         public T next() {
607             assert this.hasNext() : "Violation of: ~this.unseen /= <>";
608             if (!this.hasNext()) {
609                 /*
610                  * Exception is supposed to be thrown in this case, but with
611                  * assertion-checking enabled it cannot happen because of assert
612                  * above.
613                  */
614                 throw new NoSuchElementException();
615             }
616             T next;
617             if (SortingMachine5a.this.insertionMode) {
618                 next = this.queueIterator.next();
619             } else {
620                 next = SortingMachine5a.this.heap[this.arrayCurrentIndex];
621                 this.arrayCurrentIndex++;
622             }
623             assert SortingMachine5a.this.conventionHolds();
624             return next;
625         }
626 
627         @Override
628         public void remove() {
629             throw new UnsupportedOperationException(
630                     "remove operation not supported");
631         }
632 
633     }
634 
635 }
636 
```