```java
import java.util.Comparator;
import java.util.Iterator;
import java.util.NoSuchElementException;

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

/**
 * {@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
```

```
 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 David P & Zach
 64 *
 65 */
 66public 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         // set temp value to indices i inside array. Then exchange value with i
120         // then exchange j with temp.
121         T temp = array[i];
122         array[i] = array[j];
123         array[j] = temp;
124     }
125
126     /**
127      * Given an array that represents a complete binary tree and an index
128      * referring to the root of a subtree that would be a heap except for its
129      * root, sifts the root down to turn that whole subtree into a heap.
130      *
131      * @param <T>
132      *            type of array entries
133      * @param array
134      *            the complete binary tree
135      * @param top
136      *            the index of the root of the "subtree"
137      * @param last
138      *            the index of the last entry in the heap
139      * @param order
140      *            total preorder for sorting
141      * @updates array
142      * @requires <pre>
143      * 0 <= top  and  last < |array|  and
144      * for all i: integer
145      *     where (0 <= i  and  i < |array|)
146      *   ([entry at position i in array is not null])  and
147      * [subtree rooted at {@code top} is a complete binary tree]  and
148      * SUBTREE_IS_HEAP(array, 2 * top + 1, last,
149      *     [relation computed by order.compare method])  and
150      * SUBTREE_IS_HEAP(array, 2 * top + 2, last,
151      *     [relation computed by order.compare method])  and
152      * IS_TOTAL_PREORDER([relation computed by order.compare method])
153      * </pre>
154      * @ensures <pre>
155      * SUBTREE_IS_HEAP(array, top, last,
156      *     [relation computed by order.compare method])  and
157      * perms(array, #array)  and
158      * SUBTREE_ARRAY_ENTRIES(array, top, last) =
159      *  SUBTREE_ARRAY_ENTRIES(#array, top, last)  and
160      * [the other entries in array are the same as in #array]
161      * </pre>
162      */
163     private static <T> void siftDown(T[] array, int top, int last,
164             Comparator<T> order) {
165         assert array != null : "Violation of: array is not null";
166         assert order != null : "Violation of: order is not null";
167         assert 0 <= top : "Violation of: 0 <= top";
168         assert last < array.length : "Violation of: last < |array|";
169         for (int i = 0; i < array.length; i++) {
170             assert array[i] != null : ""
171                     + "Violation of: all entries in array are not null";
```

```
172         }
173         assert isHeap(array, 2 * top + 1, last, order) : ""
174                 + "Violation of: SUBTREE_IS_HEAP(array, 2 * top + 1, last,"
175                 + " [relation computed by order.compare method])";
176         assert isHeap(array, 2 * top + 2, last, order) : ""
177                 + "Violation of: SUBTREE_IS_HEAP(array, 2 * top + 2, last,"
178                 + " [relation computed by order.compare method])";
179         /*
180          * Impractical to check last requires clause; no need to check the other
181          * requires clause, because it must be true when using the array
182          * representation for a complete binary tree.
183          */
184
185         int left = 2 * top + 1;
186         int right = 2 * top + 2;
187
188         // Check if the right child exists (which means the left child also exists)
189         if (right <= last) {
190             // Get the values at the root and child nodes
191             T root = array[top];
192             T nodeOne = array[left];
193             T nodeTwo = array[right];
194
195             // If left child is less than or equal to right child
196             // and root is greater than right child
197             if (order.compare(nodeOne, nodeTwo) <= 0
198                     && order.compare(root, nodeTwo) > 0) {
199                 exchangeEntries(array, top, left);
200                 siftDown(array, left, last, order);
201             }
202             // If right child is less than or equal to left child
203             // and root is greater than left child
204             else if (order.compare(nodeTwo, nodeOne) <= 0
205                     && order.compare(root, nodeTwo) > 0) {
206                 exchangeEntries(array, top, right);
207                 siftDown(array, right, last, order);
208             }
209         }
210         // If only the left child exists
211         else if (left <= last) {
212             T root = array[top];
213             T nodeOne = array[left];
214
215             // If left child is less than the root
216             if (order.compare(nodeOne, root) < 0) {
217                 exchangeEntries(array, top, left);
218                 siftDown(array, left, last, order);
219             }
220         }
221
222     }
223
224     /**
225      * Heapifies the subtree of the given array rooted at the given {@code top}.
226      *
227      * @param <T>
228      *            type of array entries
```

```
229      * @param array
230      *            the complete binary tree
231      * @param top
232      *            the index of the root of the "subtree" to heapify
233      * @param order
234      *            the total preorder for sorting
235      * @updates array
236      * @requires <pre>
237      * 0 <= top  and
238      * for all i: integer
239      *     where (0 <= i  and  i < |array|)
240      *   ([entry at position i in array is not null])  and
241      * [subtree rooted at {@code top} is a complete binary tree]  and
242      * IS_TOTAL_PREORDER([relation computed by order.compare method])
243      * </pre>
244      * @ensures <pre>
245      * SUBTREE_IS_HEAP(array, top, |array| - 1,
246      *     [relation computed by order.compare method])  and
247      * perms(array, #array)
248      * </pre>
249      */
250     private static <T> void heapify(T[] array, int top, Comparator<T> order) {
251         assert array != null : "Violation of: array is not null";
252         assert order != null : "Violation of: order is not null";
253         assert 0 <= top : "Violation of: 0 <= top";
254         for (int i = 0; i < array.length; i++) {
255             assert array[i] != null : ""
256                     + "Violation of: all entries in array are not null";
257         }
258         /*
259          * Impractical to check last requires clause; no need to check the other
260          * requires clause, because it must be true when using the array
261          * representation for a complete binary tree.
262          */
263
264         // Start from the last non-leaf node and go upwards
265         int n = array.length;
266         for (int i = n / 2 - 1; i >= top; i--) {
267             siftDown(array, i, n - 1, order);
268         }
269     }
270
271     /**
272      * Constructs and returns an array representing a heap with the entries from
273      * the given {@code Queue}.
274      *
275      * @param <T>
276      *            type of {@code Queue} and array entries
277      * @param q
278      *            the {@code Queue} with the entries for the heap
279      * @param order
280      *            the total preorder for sorting
281      * @return the array representation of a heap
282      * @clears q
283      * @requires IS_TOTAL_PREORDER([relation computed by order.compare method])
284      * @ensures <pre>
285      * SUBTREE_IS_HEAP(buildHeap, 0, |buildHeap| - 1)  and
```

```
286      * perms(buildHeap, #q)  and
287      * for all i: integer
288      *     where (0 <= i  and  i < |buildHeap|)
289      *   ([entry at position i in buildHeap is not null])  and
290      * </pre>
291      */
292     @SuppressWarnings("unchecked")
293     private static <T> T[] buildHeap(Queue<T> q, Comparator<T> order) {
294         assert q != null : "Violation of: q is not null";
295         assert order != null : "Violation of: order is not null";
296         /*
297          * Impractical to check the requires clause.
298          */
299         /*
300          * With "new T[...]" in place of "new Object[...]" it does not compile;
301          * as shown, it results in a warning about an unchecked cast, though it
302          * cannot fail.
303          */
304         T[] heap = (T[]) new Object[q.length()];
305         // Create a new array with the size of the queue
306         int i = 0;
307         while (q.length() > 0) {
308             heap[i++] = q.dequeue(); // Dequeue elements from the queue and fill the array
309         }
310         heapify(heap, 0, order); // Heapify the array
311         return heap;
312     }
313
314     /**
315      * Checks if the subtree of the given {@code array} rooted at the given
316      * {@code top} is a heap.
317      *
318      * @param <T>
319      *            type of array entries
320      * @param array
321      *            the complete binary tree
322      * @param top
323      *            the index of the root of the "subtree"
324      * @param last
325      *            the index of the last entry in the heap
326      * @param order
327      *            total preorder for sorting
328      * @return true if the subtree of the given {@code array} rooted at the
329      *         given {@code top} is a heap; false otherwise
330      * @requires <pre>
331      * 0 <= top  and  last < |array|  and
332      * for all i: integer
333      *     where (0 <= i  and  i < |array|)
334      *   ([entry at position i in array is not null])  and
335      * [subtree rooted at {@code top} is a complete binary tree]
336      * </pre>
337      * @ensures <pre>
338      * isHeap = SUBTREE_IS_HEAP(array, top, last,
339      *     [relation computed by order.compare method])
340      * </pre>
341      */
342     private static <T> boolean isHeap(T[] array, int top, int last,
```

```
343             Comparator<T> order) {
344         assert array != null : "Violation of: array is not null";
345         assert 0 <= top : "Violation of: 0 <= top";
346         assert last < array.length : "Violation of: last < |array|";
347         for (int i = 0; i < array.length; i++) {
348             assert array[i] != null : ""
349                     + "Violation of: all entries in array are not null";
350         }
351         /*
352          * No need to check the other requires clause, because it must be true
353          * when using the Array representation for a complete binary tree.
354          */
355         int left = 2 * top + 1;
356         boolean isHeap = true;
357         // If left child exists
358         if (left <= last) {
359             // Check if the current node is less than or equal to the left child
360             // and recursively check the left subtree
361             isHeap = (order.compare(array[top], array[left]) <= 0)
362                     && isHeap(array, left, last, order);
363             // Calculate the right child index
364             int right = left + 1;
365             // If the left subtree satisfies the heap property, check the right subtree
366             if (isHeap && (right <= last)) {
367                 isHeap = (order.compare(array[top], array[right]) <= 0)
368                         && isHeap(array, right, last, order);
369             }
370         }
371         return isHeap;
372     }
373
374     /**
375      * Checks that the part of the convention repeated below holds for the
376      * current representation.
377      *
378      * @return true if the convention holds (or if assertion checking is off);
379      *         otherwise reports a violated assertion
380      * @convention <pre>
381      * if $this.insertionMode then
382      *   $this.heapSize = 0
383      * else
384      *   $this.entries = <>  and
385      *   for all i: integer
386      *       where (0 <= i  and  i < |$this.heap|)
387      *     ([entry at position i in $this.heap is not null])  and
388      *   SUBTREE_IS_HEAP($this.heap, 0, $this.heapSize - 1,
389      *     [relation computed by $this.machineOrder.compare method])  and
390      *   0 <= $this.heapSize <= |$this.heap|
391      * </pre>
392      */
393     private boolean conventionHolds() {
394         if (this.insertionMode) {
395             assert this.heapSize == 0 : ""
396                     + "Violation of: if $this.insertionMode then $this.heapSize = 0";
397         } else {
398             assert this.entries.length() == 0 : ""
399                     + "Violation of: if not $this.insertionMode then $this.entries = <>";
```

```
400             assert 0 <= this.heapSize : ""
401                     + "Violation of: if not $this.insertionMode then 0 <= $this.heapSize";
402             assert this.heapSize <= this.heap.length : ""
403                     + "Violation of: if not $this.insertionMode then"
404                     + " $this.heapSize <= |$this.heap|";
405             for (int i = 0; i < this.heap.length; i++) {
406                 assert this.heap[i] != null : ""
407                         + "Violation of: if not $this.insertionMode then"
408                         + " all entries in $this.heap are not null";
409             }
410             assert isHeap(this.heap, 0, this.heapSize - 1,
411                     this.machineOrder) : ""
412                             + "Violation of: if not $this.insertionMode then"
413                             + " SUBTREE_IS_HEAP($this.heap, 0, $this.heapSize - 1,"
414                             + " [relation computed by $this.machineOrder.compare"
415                             + " method])";
416         }
417         return true;
418     }
419
420     /**
421      * Creator of initial representation.
422      *
423      * @param order
424      *            total preorder for sorting
425      * @requires IS_TOTAL_PREORDER([relation computed by order.compare method]
426      * @ensures <pre>
427      * $this.insertionMode = true  and
428      * $this.machineOrder = order  and
429      * $this.entries = <>  and
430      * $this.heapSize = 0
431      * </pre>
432      */
433     private void createNewRep(Comparator<T> order) {
434         this.machineOrder = order;
435         this.insertionMode = true;
436         this.entries = new Queue1L<T>();
437         this.heapSize = 0;
438     }
439
440     /*
441      * Constructors ------------------------------------------------------------
442      */
443
444     /**
445      * Constructor from order.
446      *
447      * @param order
448      *            total preorder for sorting
449      */
450     public SortingMachine5a(Comparator<T> order) {
451         this.createNewRep(order);
452         assert this.conventionHolds();
453     }
454
455     /*
456      * Standard methods --------------------------------------------------------
```

```
457      */
458
459     @SuppressWarnings("unchecked")
460     @Override
461     public final SortingMachine<T> newInstance() {
462         try {
463             return this.getClass().getConstructor(Comparator.class)
464                     .newInstance(this.machineOrder);
465         } catch (ReflectiveOperationException e) {
466             throw new AssertionError(
467                     "Cannot construct object of type " + this.getClass());
468         }
469     }
470
471     @Override
472     public final void clear() {
473         this.createNewRep(this.machineOrder);
474         assert this.conventionHolds();
475     }
476
477     @Override
478     public final void transferFrom(SortingMachine<T> source) {
479         assert source != null : "Violation of: source is not null";
480         assert source != this : "Violation of: source is not this";
481         assert source instanceof SortingMachine5a<?> : ""
482                 + "Violation of: source is of dynamic type SortingMachine5a<?>";
483         /*
484          * This cast cannot fail since the assert above would have stopped
485          * execution in that case: source must be of dynamic type
486          * SortingMachine5a<?>, and the ? must be T or the call would not have
487          * compiled.
488          */
489         SortingMachine5a<T> localSource = (SortingMachine5a<T>) source;
490         this.insertionMode = localSource.insertionMode;
491         this.machineOrder = localSource.machineOrder;
492         this.entries = localSource.entries;
493         this.heap = localSource.heap;
494         this.heapSize = localSource.heapSize;
495         localSource.createNewRep(localSource.machineOrder);
496         assert this.conventionHolds();
497         assert localSource.conventionHolds();
498     }
499
500     /*
501      * Kernel methods ----------------------------------------------------------
502      */
503
504     @Override
505     public final void add(T x) {
506         assert x != null : "Violation of: x is not null";
507         assert this.isInInsertionMode() : "Violation of: this.insertion_mode";
508
509         this.entries.enqueue(x); // Add the element to the queue
510         assert this.conventionHolds();
511     }
512
513     @Override
```

```java
    public final void changeToExtractionMode() {
        assert this.isInInsertionMode() : "Violation of: this.insertion_mode";

        this.heap = buildHeap(this.entries, this.machineOrder);
        // Convert the queue to a heap
        this.heapSize = this.heap.length; // Set the heap size
        this.insertionMode = false; // Change the mode to extraction

        assert this.conventionHolds();
    }

    @Override
    public final T removeFirst() {
        assert !this
                .isInInsertionMode() : "Violation of: not this.insertion_mode";
        assert this.size() > 0 : "Violation of: this.contents /= {}";

        T root = this.heap[0]; // Get the root element
        if (this.heapSize > 1) {
            exchangeEntries(this.heap, 0, this.heapSize - 1);
            // Swap root with the last element
            this.heapSize--; // Reduce the heap size
            siftDown(this.heap, 0, this.heapSize - 1, this.machineOrder);
            // Restore the heap property
        } else {
            this.heapSize--; // If only one element, just reduce the heap size
        }
        assert this.conventionHolds();
        return root; // Return the root element
    }

    @Override
    public final boolean isInInsertionMode() {
        assert this.conventionHolds();
        return this.insertionMode;
    }

    @Override
    public final Comparator<T> order() {
        assert this.conventionHolds();
        return this.machineOrder;
    }

    @Override
    public final int size() {
        int size = 0;

        // if insertionmode, then get length of entries.
        if (this.insertionMode) {
            size = this.entries.length();
        } else {
            // else, get the heap size.
            size = this.heapSize;
        }
        assert this.conventionHolds();
        return size;
    }
```

```
571
572     @Override
573     public final Iterator<T> iterator() {
574         return new SortingMachine5aIterator();
575     }
576
577     /**
578      * Implementation of {@code Iterator} interface for
579      * {@code SortingMachine5a}.
580      */
581     private final class SortingMachine5aIterator implements Iterator<T> {
582
583         /**
584          * Representation iterator when in insertion mode.
585          */
586         private Iterator<T> queueIterator;
587
588         /**
589          * Representation iterator count when in extraction mode.
590          */
591         private int arrayCurrentIndex;
592
593         /**
594          * No-argument constructor.
595          */
596         private SortingMachine5aIterator() {
597             if (SortingMachine5a.this.insertionMode) {
598                 this.queueIterator = SortingMachine5a.this.entries.iterator();
599             } else {
600                 this.arrayCurrentIndex = 0;
601             }
602             assert SortingMachine5a.this.conventionHolds();
603         }
604
605         @Override
606         public boolean hasNext() {
607             boolean hasNext;
608             if (SortingMachine5a.this.insertionMode) {
609                 hasNext = this.queueIterator.hasNext();
610             } else {
611                 hasNext = this.arrayCurrentIndex < SortingMachine5a.this.heapSize;
612             }
613             assert SortingMachine5a.this.conventionHolds();
614             return hasNext;
615         }
616
617         @Override
618         public T next() {
619             assert this.hasNext() : "Violation of: ~this.unseen /= <>";
620             if (!this.hasNext()) {
621                 /*
622                  * Exception is supposed to be thrown in this case, but with
623                  * assertion-checking enabled it cannot happen because of assert
624                  * above.
625                  */
626                 throw new NoSuchElementException();
627             }
```

```
628             T next;
629             if (SortingMachine5a.this.insertionMode) {
630                 next = this.queueIterator.next();
631             } else {
632                 next = SortingMachine5a.this.heap[this.arrayCurrentIndex];
633                 this.arrayCurrentIndex++;
634             }
635             assert SortingMachine5a.this.conventionHolds();
636             return next;
637         }
638 
639         @Override
640         public void remove() {
641             throw new UnsupportedOperationException(
642                     "remove operation not supported");
643         }
644 
645     }
646 
647 }
648 
```