```java
import java.lang.reflect.Constructor;
import java.util.Comparator;
import java.util.Iterator;
import java.util.NoSuchElementException;

import components.queue.Queue;
import components.queue.Queue2;
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
```

```
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 Shyam Sai Bethian and Yihone Chu
65  *
66  */
67 public 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         /*
121          * If i and j are not equal, then we switch them in the
    array.
122          */
123         if (i != j) {
124             T tmp = array[i];
125             array[i] = array[j];
126             array[j] = tmp;
127         }
128
129     }
130
131     /**
132      * Given an array that represents a complete binary tree and
    an index
133      * referring to the root of a subtree that would be a heap
    except for its
134      * root, sifts the root down to turn that whole subtree into a
    heap.
135      *
136      * @param <T>
137      *            type of array entries
138      * @param array
139      *            the complete binary tree
140      * @param top
141      *            the index of the root of the "subtree"
142      * @param last
143      *            the index of the last entry in the heap
144      * @param order
145      *            total preorder for sorting
146      * @updates array
147      * @requires <pre>
148      * 0 <= top  and  last < |array|  and
149      * for all i: integer
150      *     where (0 <= i  and  i < |array|)
151      *   ([entry at position i in array is not null])  and
152      * [subtree rooted at {@code top} is a complete binary tree]
    and
153      * SUBTREE_IS_HEAP(array, 2 * top + 1, last,
154      *     [relation computed by order.compare method])  and
155      * SUBTREE_IS_HEAP(array, 2 * top + 2, last,
156      *     [relation computed by order.compare method])  and
157      * IS_TOTAL_PREORDER([relation computed by order.compare
    method])
```

```
158      * </pre>
159      * @ensures <pre>
160      * SUBTREE_IS_HEAP(array, top, last,
161      *     [relation computed by order.compare method])  and
162      * perms(array, #array)  and
163      * SUBTREE_ARRAY_ENTRIES(array, top, last) =
164      *  SUBTREE_ARRAY_ENTRIES(#array, top, last)  and
165      * [the other entries in array are the same as in #array]
166      * </pre>
167      */
168     private static <T> void siftDown(T[] array, int top, int last,
169             Comparator<T> order) {
170         assert array != null : "Violation of: array is not null";
171         assert order != null : "Violation of: order is not null";
172         assert 0 <= top : "Violation of: 0 <= top";
173         assert last < array.length : "Violation of: last < |
    array|";
174         for (int i = 0; i < array.length; i++) {
175             assert array[i] != null : ""
176                     + "Violation of: all entries in array are not
    null";
177         }
178         assert isHeap(array, 2 * top + 1, last, order) : ""
179                 + "Violation of: SUBTREE_IS_HEAP(array, 2 * top +
    1, last,"
180                 + " [relation computed by order.compare method])";
181         assert isHeap(array, 2 * top + 2, last, order) : ""
182                 + "Violation of: SUBTREE_IS_HEAP(array, 2 * top +
    2, last,"
183                 + " [relation computed by order.compare method])";
184         /*
185          * Impractical to check last requires clause; no need to
    check the other
186          * requires clause, because it must be true when using the
    array
187          * representation for a complete binary tree.
188          */
189
190         /*
191          * First gets the index of both left and right.
192          */
193         int left = 2 * top + 1;
194         int right = 2 * top + 2;
195
```

```
196         /*
197          * If right is less than last, then that means a left and
    right node
198          * must exist.
199          */
200         if (right <= last) {
201 
202             /*
203              * If the value at the left node is smaller than the
    value at the
204              * top node and the right node, then the top node and
    left node
205              * switch. Then we recursively call swiftDown to turn
    the rest of
206              * the tree into a heap.
207              */
208             if (order.compare(array[left], array[top]) < 0
209                     && order.compare(array[left], array[right]) <
    0) {
210                 exchangeEntries(array, left, top);
211                 siftDown(array, left, last, order);
212             } else if (order.compare(array[right], array[top]) <
    0) {
213                 /*
214                  * If the value at the left node is bigger than
    the value at the
215                  * top node, but the right node is smaller than
    the top node,
216                  * then the top node and right node switch. Then
    we recursively
217                  * call swiftDown to turn the rest of the tree
    into a heap.
218                  */
219                 exchangeEntries(array, right, top);
220                 siftDown(array, right, last, order);
221             }
222         } else if (left <= last && right > last) {
223             /*
224              * If only the left node exists and there is no right
    node, then we
225              * compare the left and top node. If the left node is
    smaller, then
226              * the top and left node switch, and we call swiftDown
    to
```

```
227             * recursively turn the rest of the tree into the
    heap.
228             */
229            if (order.compare(array[left], array[top]) < 0) {
230                exchangeEntries(array, left, top);
231                siftDown(array, left, last, order);
232            }
233        }
234    }
235
236    /**
237     * Heapifies the subtree of the given array rooted at the
    given {@code top}.
238     *
239     * @param <T>
240     *            type of array entries
241     * @param array
242     *            the complete binary tree
243     * @param top
244     *            the index of the root of the "subtree" to
    heapify
245     * @param order
246     *            the total preorder for sorting
247     * @updates array
248     * @requires <pre>
249     * 0 <= top  and
250     * for all i: integer
251     *     where (0 <= i  and  i < |array|)
252     *   ([entry at position i in array is not null])  and
253     * [subtree rooted at {@code top} is a complete binary tree]
    and
254     * IS_TOTAL_PREORDER([relation computed by order.compare
    method])
255     * </pre>
256     * @ensures <pre>
257     * SUBTREE_IS_HEAP(array, top, |array| - 1,
258     *     [relation computed by order.compare method])  and
259     * perms(array, #array)
260     * </pre>
261     */
262    private static <T> void heapify(T[] array, int top,
    Comparator<T> order) {
263        assert array != null : "Violation of: array is not null";
264        assert order != null : "Violation of: order is not null";
```

```
265         assert 0 <= top : "Violation of: 0 <= top";
266         for (int i = 0; i < array.length; i++) {
267             assert array[i] != null : ""
268                     + "Violation of: all entries in array are not
    null";
269         }
270         /*
271          * Impractical to check last requires clause; no need to
    check the other
272          * requires clause, because it must be true when using the
    array
273          * representation for a complete binary tree.
274          */
275
276         /*
277          * First gets the index of both left and right.
278          */
279         int left = 2 * top + 1;
280         int right = 2 * top + 2;
281
282         /*
283          * If right is less than array length, then that means a
    left and right
284          * node must exist.
285          */
286         if (right < array.length) {
287             /*
288              * We recursively call heapify on the left node and
    right node to
289              * heapify the rest of the tree.
290              */
291             heapify(array, left, order);
292             heapify(array, right, order);
293         } else if (left < array.length && right >= array.length) {
294             /*
295              * If only the left node is smaller than the array
    length, then we
296              * recursivly call heapify only on the left node to
    heapify the rest
297              * of the tree.
298              */
299             heapify(array, left, order);
300         }
301
```

```
302         /*
303          * Calls siftDown to create the heap using the array.
304          */
305         siftDown(array, top, array.length - 1, order);
306 
307         // TODO - fill in body
308         // *** you must use the recursive algorithm discussed in
    class ***
309 
310     }
311 
312     /**
313      * Constructs and returns an array representing a heap with
    the entries from
314      * the given {@code Queue}.
315      *
316      * @param <T>
317      *            type of {@code Queue} and array entries
318      * @param q
319      *            the {@code Queue} with the entries for the heap
320      * @param order
321      *            the total preorder for sorting
322      * @return the array representation of a heap
323      * @clears q
324      * @requires IS_TOTAL_PREORDER([relation computed by
    order.compare method])
325      * @ensures <pre>
326      * SUBTREE_IS_HEAP(buildHeap, 0, |buildHeap| - 1)  and
327      * perms(buildHeap, #q)  and
328      * for all i: integer
329      *     where (0 <= i  and  i < |buildHeap|)
330      *   ([entry at position i in buildHeap is not null])  and
331      * </pre>
332      */
333     @SuppressWarnings("unchecked")
334     private static <T> T[] buildHeap(Queue<T> q, Comparator<T>
    order) {
335         assert q != null : "Violation of: q is not null";
336         assert order != null : "Violation of: order is not null";
337         /*
338          * Impractical to check the requires clause.
339          */
340         /*
341          * With "new T[...]" in place of "new Object[...]" it does
```

```
    not compile;
342          * as shown, it results in a warning about an unchecked
    cast, though it
343          * cannot fail.
344          */
345         T[] heap = (T[]) (new Object[q.length()]);
346
347         /*
348          * This while loop fills in each index of the array of
    elements from the
349          * queue q.
350          */
351         int counter = 0;
352         while (q.length() > 0) {
353             heap[counter] = q.dequeue();
354             counter++;
355         }
356         /*
357          * The resulting array is heapified when calling heapify
    on it.
358          */
359         heapify(heap, 0, order);
360
361         return heap;
362     }
363
364     /**
365      * Checks if the subtree of the given {@code array} rooted at
    the given
366      * {@code top} is a heap.
367      *
368      * @param <T>
369      *            type of array entries
370      * @param array
371      *            the complete binary tree
372      * @param top
373      *            the index of the root of the "subtree"
374      * @param last
375      *            the index of the last entry in the heap
376      * @param order
377      *            total preorder for sorting
378      * @return true if the subtree of the given {@code array}
    rooted at the
379      *         given {@code top} is a heap; false otherwise
```

```
380      * @requires <pre>
381      * 0 <= top  and  last < |array|  and
382      * for all i: integer
383      *     where (0 <= i  and  i < |array|)
384      *   ([entry at position i in array is not null])  and
385      * [subtree rooted at {@code top} is a complete binary tree]
386      * </pre>
387      * @ensures <pre>
388      * isHeap = SUBTREE_IS_HEAP(array, top, last,
389      *     [relation computed by order.compare method])
390      * </pre>
391      */
392     private static <T> boolean isHeap(T[] array, int top, int
    last,
393             Comparator<T> order) {
394         assert array != null : "Violation of: array is not null";
395         assert 0 <= top : "Violation of: 0 <= top";
396         assert last < array.length : "Violation of: last < |
    array|";
397         for (int i = 0; i < array.length; i++) {
398             assert array[i] != null : ""
399                     + "Violation of: all entries in array are not
    null";
400         }
401         /*
402          * No need to check the other requires clause, because it
    must be true
403          * when using the Array representation for a complete
    binary tree.
404          */
405
406         int left = 2 * top + 1;
407         boolean isHeap = true;
408         if (left <= last) {
409             isHeap = (order.compare(array[top], array[left]) <= 0)
410                     && isHeap(array, left, last, order);
411             int right = left + 1;
412             if (isHeap && (right <= last)) {
413                 isHeap = (order.compare(array[top], array[right])
    <= 0)
414                         && isHeap(array, right, last, order);
415             }
416         }
417         return isHeap;
```

```
418     }
419
420     /**
421      * Checks that the part of the convention repeated below holds
    for the
422      * current representation.
423      *
424      * @return true if the convention holds (or if assertion
    checking is off);
425      *         otherwise reports a violated assertion
426      * @convention <pre>
427      * if $this.insertionMode then
428      *   $this.heapSize = 0
429      * else
430      *   $this.entries = <>  and
431      *   for all i: integer
432      *       where (0 <= i  and  i < |$this.heap|)
433      *     ([entry at position i in $this.heap is not null])  and
434      *   SUBTREE_IS_HEAP($this.heap, 0, $this.heapSize - 1,
435      *     [relation computed by $this.machineOrder.compare
    method])  and
436      *   0 <= $this.heapSize <= |$this.heap|
437      * </pre>
438      */
439     private boolean conventionHolds() {
440         if (this.insertionMode) {
441             assert this.heapSize == 0 : ""
442                     + "Violation of: if $this.insertionMode then
    $this.heapSize = 0";
443         } else {
444             assert this.entries.length() == 0 : ""
445                     + "Violation of: if not $this.insertionMode
    then $this.entries = <>";
446             assert 0 <= this.heapSize : ""
447                     + "Violation of: if not $this.insertionMode
    then 0 <= $this.heapSize";
448             assert this.heapSize <= this.heap.length : ""
449                     + "Violation of: if not $this.insertionMode
    then"
450                     + " $this.heapSize <= |$this.heap|";
451             for (int i = 0; i < this.heap.length; i++) {
452                 assert this.heap[i] != null : ""
453                         + "Violation of: if not
    $this.insertionMode then"
```

```
454                         + " all entries in $this.heap are not
    null";
455             }
456             assert isHeap(this.heap, 0, this.heapSize - 1,
457                     this.machineOrder) : ""
458                             + "Violation of: if not
    $this.insertionMode then"
459                             + " SUBTREE_IS_HEAP($this.heap, 0,
    $this.heapSize - 1,"
460                             + " [relation computed by
    $this.machineOrder.compare"
461                             + " method])";
462         }
463         return true;
464     }
465
466     /**
467      * Creator of initial representation.
468      *
469      * @param order
470      *            total preorder for sorting
471      * @requires IS_TOTAL_PREORDER([relation computed by
    order.compare method]
472      * @ensures <pre>
473      * $this.insertionMode = true  and
474      * $this.machineOrder = order  and
475      * $this.entries = <>  and
476      * $this.heapSize = 0
477      * </pre>
478      */
479     private void createNewRep(Comparator<T> order) {
480
481         /*
482          * A new representation starts with insertionMode equal to
    true, 0
483          * entries, and a heapSize of 0.
484          */
485         this.insertionMode = true;
486
487         this.machineOrder = order;
488
489         this.entries = new Queue2<T>();
490
491         this.heapSize = 0;
```

```
492     }
493
494     /*
495      * Constructors
    ------------------------------------------------------------
496      */
497
498     /**
499      * Constructor from order.
500      *
501      * @param order
502      *            total preorder for sorting
503      */
504     public SortingMachine5a(Comparator<T> order) {
505         this.createNewRep(order);
506         assert this.conventionHolds();
507     }
508
509     /*
510      * Standard methods
    ------------------------------------------------------------
511      */
512
513     @SuppressWarnings("unchecked")
514     @Override
515     public final SortingMachine<T> newInstance() {
516         try {
517             Constructor<?> c =
    this.getClass().getConstructor(Comparator.class);
518             return (SortingMachine<T>)
    c.newInstance(this.machineOrder);
519         } catch (ReflectiveOperationException e) {
520             throw new AssertionError(
521                     "Cannot construct object of type " +
    this.getClass());
522         }
523     }
524
525     @Override
526     public final void clear() {
527         this.createNewRep(this.machineOrder);
528         assert this.conventionHolds();
529     }
530
```

```
531     @Override
532     public final void transferFrom(SortingMachine<T> source) {
533         assert source != null : "Violation of: source is not
    null";
534         assert source != this : "Violation of: source is not
    this";
535         assert source instanceof SortingMachine5a<?> : ""
536                 + "Violation of: source is of dynamic type
    SortingMachine5a<?>";
537         /*
538          * This cast cannot fail since the assert above would have
    stopped
539          * execution in that case: source must be of dynamic type
540          * SortingMachine5a<?>, and the ? must be T or the call
    would not have
541          * compiled.
542          */
543         SortingMachine5a<T> localSource = (SortingMachine5a<T>)
    source;
544         this.insertionMode = localSource.insertionMode;
545         this.machineOrder = localSource.machineOrder;
546         this.entries = localSource.entries;
547         this.heap = localSource.heap;
548         this.heapSize = localSource.heapSize;
549         localSource.createNewRep(localSource.machineOrder);
550         assert this.conventionHolds();
551         assert localSource.conventionHolds();
552     }
553
554     /*
555      * Kernel methods
    ------------------------------------------------------------
556      */
557
558     @Override
559     public final void add(T x) {
560         assert x != null : "Violation of: x is not null";
561         assert this.isInInsertionMode() : "Violation of:
    this.insertion_mode";
562
563         /*
564          * Enqueues x to the this SortingMachines' entries queue.
565          */
566         this.entries.enqueue(x);
```

```
567
568         assert this.conventionHolds();
569     }
570
571     @Override
572     public final void changeToExtractionMode() {
573         assert this.isInInsertionMode() : "Violation of:
    this.insertion_mode";
574
575         /*
576          * Changes insertionMode to false, and the heapSize of
    this is equal to
577          * the entries the length.
578          */
579         this.insertionMode = false;
580         this.heapSize = this.entries.length();
581
582         /*
583          * this.heap becomes the heap built using buildHeap with
    the entries and
584          * the comparator machineOrder.
585          */
586         this.heap = buildHeap(this.entries, this.machineOrder);
587
588         assert this.conventionHolds();
589     }
590
591     @Override
592     public final T removeFirst() {
593         assert !this
594                 .isInInsertionMode() : "Violation of: not
    this.insertion_mode";
595         assert this.size() > 0 : "Violation of: this.contents /=
    {}";
596
597         /*
598          * The answer is the first entry of the array this.heap.
599          */
600         T answer = this.heap[0];
601
602         /*
603          * Then we exchange the top node with the last entry in
    this.heap
604          */
```

```
605         exchangeEntries(this.heap, 0, this.heapSize - 1);
606         /*
607          * After decreasing the value of this.heapSize, we use
    siftDown to
608          * create a heap without the last entry of this.heap,
    which is the value
609          * which we "removed".
610          */
611         this.heapSize--;
612         siftDown(this.heap, 0, this.heapSize - 1,
    this.machineOrder);
613 
614         assert this.conventionHolds();
615         // Fix this line to return the result after checking the
    convention.
616         return answer;
617     }
618 
619     @Override
620     public final boolean isInInsertionMode() {
621         assert this.conventionHolds();
622         /*
623          * Returns if this is in Insertion Mode.
624          */
625         return this.insertionMode;
626     }
627 
628     @Override
629     public final Comparator<T> order() {
630         assert this.conventionHolds();
631         /*
632          * Returns the comparator used.
633          */
634         return this.machineOrder;
635     }
636 
637     @Override
638     public final int size() {
639 
640         /*
641          * If the machine is in insertionMode, then the answer is
    the length of
642          * the entries. If not, then the answer is the size of the
    heap.
```

```
643          */
644         int answer = 0;
645         if (this.insertionMode) {
646             answer = this.entries.length();
647         } else {
648             answer = this.heapSize;
649         }
650 
651         assert this.conventionHolds();
652         // Fix this line to return the result after checking the
    convention.
653         return answer;
654     }
655 
656     @Override
657     public final Iterator<T> iterator() {
658         return new SortingMachine5aIterator();
659     }
660 
661     /**
662      * Implementation of {@code Iterator} interface for
663      * {@code SortingMachine5a}.
664      */
665     private final class SortingMachine5aIterator implements
    Iterator<T> {
666 
667         /**
668          * Representation iterator when in insertion mode.
669          */
670         private Iterator<T> queueIterator;
671 
672         /**
673          * Representation iterator count when in extraction mode.
674          */
675         private int arrayCurrentIndex;
676 
677         /**
678          * No-argument constructor.
679          */
680         private SortingMachine5aIterator() {
681             if (SortingMachine5a.this.insertionMode) {
682                 this.queueIterator =
    SortingMachine5a.this.entries.iterator();
683             } else {
```

```
684                 this.arrayCurrentIndex = 0;
685             }
686             assert SortingMachine5a.this.conventionHolds();
687         }
688 
689         @Override
690         public boolean hasNext() {
691             boolean hasNext;
692             if (SortingMachine5a.this.insertionMode) {
693                 hasNext = this.queueIterator.hasNext();
694             } else {
695                 hasNext = this.arrayCurrentIndex <
    SortingMachine5a.this.heapSize;
696             }
697             assert SortingMachine5a.this.conventionHolds();
698             return hasNext;
699         }
700 
701         @Override
702         public T next() {
703             assert this.hasNext() : "Violation of: ~this.unseen /=
    <>";
704             if (!this.hasNext()) {
705                 /*
706                  * Exception is supposed to be thrown in this
    case, but with
707                  * assertion-checking enabled it cannot happen
    because of assert
708                  * above.
709                  */
710                 throw new NoSuchElementException();
711             }
712             T next;
713             if (SortingMachine5a.this.insertionMode) {
714                 next = this.queueIterator.next();
715             } else {
716                 next =
    SortingMachine5a.this.heap[this.arrayCurrentIndex];
717                 this.arrayCurrentIndex++;
718             }
719             assert SortingMachine5a.this.conventionHolds();
720             return next;
721         }
722 
```

```
723         @Override
724         public void remove() {
725             throw new UnsupportedOperationException(
726                     "remove operation not supported");
727         }
728 
729     }
730 
731 }
732 
```