```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
 *   this = (true, $this.machineOrder, multiset_entries($this.entries))
 * else
 *   this = (false, $this.machineOrder, multiset_entries($this.heap[0, $this.heapSize)))
 * </pre>
 *
 * @author Charan Nanduri and Evan Frisbie
 *
 */
public class SortingMachine5a<T> extends SortingMachineSecondary<T> {

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

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

    /**
     * Insertion mode.
     */
    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         //First we need to hold one of the values
120         T holdVar = array[i];
121
122         //Now we can swap them, since this is just an array it is
```

```
   fairly simple
123         array[i] = array[j];
124         array[j] = holdVar;
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        // Start by grabbing length of array and ledt and right
    children
188        int left = 2 * top + 1;
189        int right = left + 1;
190
191        //'sifts' through the array
192        //exchange nodes if not sorted, checking both left and
    right
193        if (left <= last) {
194            if (right <= last) {
195                if (order.compare(array[left], array[right]) < 0) {
196                    if (order.compare(array[left], array[top]) < 0)
    {
197                        exchangeEntries(array, top, left);
```

```
198                         siftDown(array, left, last, order);
199                     }
200                 } else {
201                     if (order.compare(array[right], array[top]) <
    0) {
202                         exchangeEntries(array, top, right);
203                         siftDown(array, right, last, order);
204                     }
205                 }
206             } else {
207                 if (order.compare(array[left], array[top]) < 0) {
208                     exchangeEntries(array, top, left);
209                 }
210             }
211         }
212
213         // *** you must use the recursive algorithm discussed in
    class ***
214
215     }
216
217     /**
218      * Heapifies the subtree of the given array rooted at the given
    {@code top}.
219      *
220      * @param <T>
221      *            type of array entries
222      * @param array
223      *            the complete binary tree
224      * @param top
225      *            the index of the root of the "subtree" to heapify
226      * @param order
227      *            the total preorder for sorting
228      * @updates array
229      * @requires <pre>
230      * 0 <= top  and
231      * for all i: integer
232      *     where (0 <= i  and  i < |array|)
233      *   ([entry at position i in array is not null])  and
234      * [subtree rooted at {@code top} is a complete binary tree]
    and
235      * IS_TOTAL_PREORDER([relation computed by order.compare
    method])
236      * </pre>
237      * @ensures <pre>
```

```
238     * SUBTREE_IS_HEAP(array, top, |array| - 1,
239     *     [relation computed by order.compare method])  and
240     * perms(array, #array)
241     * </pre>
242     */
243    private static <T> void heapify(T[] array, int top,
    Comparator<T> order) {
244        assert array != null : "Violation of: array is not null";
245        assert order != null : "Violation of: order is not null";
246        assert 0 <= top : "Violation of: 0 <= top";
247        for (int i = 0; i < array.length; i++) {
248            assert array[i] != null : ""
249                    + "Violation of: all entries in array are not
    null";
250        }
251        /*
252         * Impractical to check last requires clause; no need to
    check the other
253         * requires clause, because it must be true when using the
    array
254         * representation for a complete binary tree.
255         */
256
257        //Once again, let's start by getting our integer values
258        int finalInt = array.length - 1;
259        int leftInt = 2 * top + 1;
260        int rightInt = leftInt + 1;
261
262        //check if the array is sifted
263        boolean isSifted = isHeap(array, top, finalInt, order);
264
265        //If it is not sorted
266        if (!isSifted) {
267
268            //Then we can trace a path down the lowest nodes
269            heapify(array, rightInt, order);
270            heapify(array, leftInt, order);
271
272            //And call our sifting function which will sort through
    the array
273            //creating a mathematical heap for us
274            siftDown(array, top, finalInt, order);
275        }
276
277    }
```

```
278
279     /**
280      * Constructs and returns an array representing a heap with the
    entries from
281      * the given {@code Queue}.
282      *
283      * @param <T>
284      *            type of {@code Queue} and array entries
285      * @param q
286      *            the {@code Queue} with the entries for the heap
287      * @param order
288      *            the total preorder for sorting
289      * @return the array representation of a heap
290      * @clears q
291      * @requires IS_TOTAL_PREORDER([relation computed by
    order.compare method])
292      * @ensures <pre>
293      * SUBTREE_IS_HEAP(buildHeap, 0, |buildHeap| - 1)  and
294      * perms(buildHeap, #q)  and
295      * for all i: integer
296      *     where (0 <= i  and  i < |buildHeap|)
297      *   ([entry at position i in buildHeap is not null])  and
298      * </pre>
299      */
300     @SuppressWarnings("unchecked")
301     private static <T> T[] buildHeap(Queue<T> q, Comparator<T>
    order) {
302         assert q != null : "Violation of: q is not null";
303         assert order != null : "Violation of: order is not null";
304         /*
305          * Impractical to check the requires clause.
306          */
307         /*
308          * With "new T[...]" in place of "new Object[...]" it does
    not compile;
309          * as shown, it results in a warning about an unchecked
    cast, though it
310          * cannot fail.
311          */
312         T[] heap = (T[]) (new Object[q.length()]);
313
314         //first I am going to grab the length of the queue
315         int length = q.length();
316
317         //Now we loop through all the entries in the queue and
```

```
    stick them in
318         //the array
319         for (int i = 0; i < length; i++) {
320             heap[i] = q.dequeue();
321         }
322
323         //Now we can use the functions we created to turn this from
    an array
324         //into an actual heap (pretty cool)
325         heapify(heap, 0, order);
326
327         //Then return the heap
328         return heap;
329     }
330
331     /**
332      * Checks if the subtree of the given {@code array} rooted at
    the given
333      * {@code top} is a heap.
334      *
335      * @param <T>
336      *            type of array entries
337      * @param array
338      *            the complete binary tree
339      * @param top
340      *            the index of the root of the "subtree"
341      * @param last
342      *            the index of the last entry in the heap
343      * @param order
344      *            total preorder for sorting
345      * @return true if the subtree of the given {@code array}
    rooted at the
346      *         given {@code top} is a heap; false otherwise
347      * @requires <pre>
348      * 0 <= top  and  last < |array|  and
349      * for all i: integer
350      *     where (0 <= i  and  i < |array|)
351      *   ([entry at position i in array is not null])  and
352      * [subtree rooted at {@code top} is a complete binary tree]
353      * </pre>
354      * @ensures <pre>
355      * isHeap = SUBTREE_IS_HEAP(array, top, last,
356      *     [relation computed by order.compare method])
357      * </pre>
358      */
```

```
359     private static <T> boolean isHeap(T[] array, int top, int last,
360             Comparator<T> order) {
361         assert array != null : "Violation of: array is not null";
362         assert 0 <= top : "Violation of: 0 <= top";
363         assert last < array.length : "Violation of: last < |
    array|";
364         for (int i = 0; i < array.length; i++) {
365             assert array[i] != null : ""
366                     + "Violation of: all entries in array are not
    null";
367         }
368         /*
369          * No need to check the other requires clause, because it
    must be true
370          * when using the Array representation for a complete
    binary tree.
371          */
372         int left = 2 * top + 1;
373         boolean isHeap = true;
374         if (left <= last) {
375             isHeap = (order.compare(array[top], array[left]) <= 0)
376                     && isHeap(array, left, last, order);
377             int right = left + 1;
378             if (isHeap && (right <= last)) {
379                 isHeap = (order.compare(array[top], array[right])
    <= 0)
380                         && isHeap(array, right, last, order);
381             }
382         }
383         return isHeap;
384     }
385
386     /**
387      * Checks that the part of the convention repeated below holds
    for the
388      * current representation.
389      *
390      * @return true if the convention holds (or if assertion
    checking is off);
391      *         otherwise reports a violated assertion
392      * @convention <pre>
393      * if $this.insertionMode then
394      *   $this.heapSize = 0
395      * else
396      *   $this.entries = <>  and
```

```
397      *   for all i: integer
398      *       where (0 <= i  and  i < |$this.heap|)
399      *     ([entry at position i in $this.heap is not null])  and
400      *   SUBTREE_IS_HEAP($this.heap, 0, $this.heapSize - 1,
401      *     [relation computed by $this.machineOrder.compare
    method])  and
402      *   0 <= $this.heapSize <= |$this.heap|
403      * </pre>
404      */
405     private boolean conventionHolds() {
406         if (this.insertionMode) {
407             assert this.heapSize == 0 : ""
408                     + "Violation of: if $this.insertionMode then
    $this.heapSize = 0";
409         } else {
410             assert this.entries.length() == 0 : ""
411                     + "Violation of: if not $this.insertionMode
    then $this.entries = <>";
412             assert 0 <= this.heapSize : ""
413                     + "Violation of: if not $this.insertionMode
    then 0 <= $this.heapSize";
414             assert this.heapSize <= this.heap.length : ""
415                     + "Violation of: if not $this.insertionMode
    then"
416                     + " $this.heapSize <= |$this.heap|";
417             for (int i = 0; i < this.heap.length; i++) {
418                 assert this.heap[i] != null : ""
419                         + "Violation of: if not $this.insertionMode
    then"
420                         + " all entries in $this.heap are not
    null";
421             }
422             assert isHeap(this.heap, 0, this.heapSize - 1,
423                     this.machineOrder) : ""
424                             + "Violation of: if not
    $this.insertionMode then"
425                             + " SUBTREE_IS_HEAP($this.heap, 0,
    $this.heapSize - 1,"
426                             + " [relation computed by
    $this.machineOrder.compare"
427                             + " method])";
428         }
429         return true;
430     }
431
```

```
432     /**
433      * Creator of initial representation.
434      *
435      * @param order
436      *            total preorder for sorting
437      * @requires IS_TOTAL_PREORDER([relation computed by
    order.compare method]
438      * @ensures <pre>
439      * $this.insertionMode = true  and
440      * $this.machineOrder = order  and
441      * $this.entries = <>  and
442      * $this.heapSize = 0
443      * </pre>
444      */
445     private void createNewRep(Comparator<T> order) {
446
447         //In this part we have to initialize all those private
    values at the top
448         //to the correct starting values
449         this.insertionMode = true;
450         this.heapSize = 0;
451
452         this.machineOrder = order;
453         this.entries = new Queue1L<T>();
454
455         //I believe that heap is created in buildHeap so it is not
    needed here
456
457     }
458
459     /*
460      * Constructors
    -----------------------------------------------------------
461      */
462
463     /**
464      * Constructor from order.
465      *
466      * @param order
467      *            total preorder for sorting
468      */
469     public SortingMachine5a(Comparator<T> order) {
470         this.createNewRep(order);
471         assert this.conventionHolds();
472     }
```

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

```
512         this.heap = localSource.heap;
513         this.heapSize = localSource.heapSize;
514         localSource.createNewRep(localSource.machineOrder);
515         assert this.conventionHolds();
516         assert localSource.conventionHolds();
517     }
518 
519     /*
520      * Kernel methods
    ------------------------------------------------------------
521      */
522 
523     @Override
524     public final void add(T x) {
525         assert x != null : "Violation of: x is not null";
526         assert this.isInInsertionMode() : "Violation of:
    this.insertion_mode";
527 
528         //Here we are just going to add to the queue
529         this.entries.enqueue(x);
530 
531         assert this.conventionHolds();
532     }
533 
534     @Override
535     public final void changeToExtractionMode() {
536         assert this.isInInsertionMode() : "Violation of:
    this.insertion_mode";
537 
538         //I'll first set the insertion mode to false
539         this.insertionMode = false;
540 
541         //Then we need to build the heap. At the sizes we are
    working at it is
542         //going to be much quicker to just do this all at once at
    the end here.
543         //So build it and fix the new length
544         this.heapSize = this.entries.length();
545         this.heap = buildHeap(this.entries, this.machineOrder);
546 
547         assert this.conventionHolds();
548     }
549 
550     @Override
551     public final T removeFirst() {
```

```
552         assert !this
553                 .isInInsertionMode() : "Violation of: not
    this.insertion_mode";
554         assert this.size() > 0 : "Violation of: this.contents /=
    {}";
555
556         //The first value is going to be at 0 which is the root
557         T result = this.heap[0];
558         int length = this.heapSize;
559         //Since we pulled something out, we no longer have a heap
560         exchangeEntries(this.heap, 0, length - 1);
561         this.heapSize--;
562         siftDown(this.heap, 0, length - 1, this.machineOrder);
563
564         //make sure the convention still holds
565         assert this.conventionHolds();
566
567         //return the removed root
568         return result;
569     }
570
571     @Override
572     public final boolean isInInsertionMode() {
573         assert this.conventionHolds();
574         return this.insertionMode;
575     }
576
577     @Override
578     public final Comparator<T> order() {
579         assert this.conventionHolds();
580         return this.machineOrder;
581     }
582
583     @Override
584     public final int size() {
585
586         //create a result value to hold the final value
587         int result = 0;
588
589         //Note here that we technically have two sizes that we can
    pull from
590         if (!this.insertionMode) {
591             result = this.heapSize;
592         } else if (this.insertionMode) {
593             result = this.entries.length();
```

```
594         }
595
596         //Check that nothing has changed with out convention
597         assert this.conventionHolds();
598
599         //return the result
600         return result;
601     }
602
603     @Override
604     public final Iterator<T> iterator() {
605         return new SortingMachine5aIterator();
606     }
607
608     /**
609      * Implementation of {@code Iterator} interface for
610      * {@code SortingMachine5a}.
611      */
612     private final class SortingMachine5aIterator implements
    Iterator<T> {
613
614         /**
615          * Representation iterator when in insertion mode.
616          */
617         private Iterator<T> queueIterator;
618
619         /**
620          * Representation iterator count when in extraction mode.
621          */
622         private int arrayCurrentIndex;
623
624         /**
625          * No-argument constructor.
626          */
627         private SortingMachine5aIterator() {
628             if (SortingMachine5a.this.insertionMode) {
629                 this.queueIterator =
    SortingMachine5a.this.entries.iterator();
630             } else {
631                 this.arrayCurrentIndex = 0;
632             }
633             assert SortingMachine5a.this.conventionHolds();
634         }
635
636         @Override
```

```
637         public boolean hasNext() {
638             boolean hasNext;
639             if (SortingMachine5a.this.insertionMode) {
640                 hasNext = this.queueIterator.hasNext();
641             } else {
642                 hasNext = this.arrayCurrentIndex <
    SortingMachine5a.this.heapSize;
643             }
644             assert SortingMachine5a.this.conventionHolds();
645             return hasNext;
646         }
647 
648         @Override
649         public T next() {
650             assert this.hasNext() : "Violation of: ~this.unseen /=
    <>";
651             if (!this.hasNext()) {
652                 /*
653                  * Exception is supposed to be thrown in this case,
    but with
654                  * assertion-checking enabled it cannot happen
    because of assert
655                  * above.
656                  */
657                 throw new NoSuchElementException();
658             }
659             T next;
660             if (SortingMachine5a.this.insertionMode) {
661                 next = this.queueIterator.next();
662             } else {
663                 next =
    SortingMachine5a.this.heap[this.arrayCurrentIndex];
664                 this.arrayCurrentIndex++;
665             }
666             assert SortingMachine5a.this.conventionHolds();
667             return next;
668         }
669 
670         @Override
671         public void remove() {
672             throw new UnsupportedOperationException(
673                     "remove operation not supported");
674         }
675 
676     }
```

```
677
678 }
679
```