```java
import java.lang.reflect.Constructor;

/**
 * {@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 Qinuo Shi & Yiming Cheng
 *
```

```
 */
public class SortingMachine5a<T> extends SortingMachineSecondary<T> {

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

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

    /**
     * Insertion mode.
     */
    private boolean insertionMode;

    /**
     * Entries.
     */
    private Queue<T> entries;

    /**
     * Heap.
     */
    private T[] heap;

    /**
     * Heap size.
     */
    private int heapSize;

    /**
     * Exchanges entries at indices {@code i} and {@code j} of {@code array}.
     *
     * @param <T>
     *            type of array entries
     * @param array
     *            the array whose entries are to be exchanged
     * @param i
     *            one index
     * @param j
     *            the other index
     * @updates array
     * @requires 0 <= i < |array| and 0 <= j < |array|
     * @ensures array = [#array with entries at indices i and j exchanged]
     */
    private static <T> void exchangeEntries(T[] array, int i, int j) {
        assert array != null : "Violation of: array is not null";
        assert 0 <= i : "Violation of: 0 <= i";
        assert i < array.length : "Violation of: i < |array|";
        assert 0 <= j : "Violation of: 0 <= j";
        assert j < array.length : "Violation of: j < |array|";

        // TODO - fill in body
        if (i != j) {
            T tool = array[i];
```

```
123             array[i] = array[j];
124             array[j] = tool;
125         }
126
127     }
128
129     /**
130      * Given an array that represents a complete binary tree and an index
131      * referring to the root of a subtree that would be a heap except for its
132      * root, sifts the root down to turn that whole subtree into a heap.
133      *
134      * @param <T>
135      *            type of array entries
136      * @param array
137      *            the complete binary tree
138      * @param top
139      *            the index of the root of the "subtree"
140      * @param last
141      *            the index of the last entry in the heap
142      * @param order
143      *            total preorder for sorting
144      * @updates array
145      * @requires <pre>
146      * 0 <= top  and  last < |array|  and
147      * for all i: integer
148      *     where (0 <= i  and  i < |array|)
149      *   ([entry at position i in array is not null])  and
150      * [subtree rooted at {@code top} is a complete binary tree]  and
151      * SUBTREE_IS_HEAP(array, 2 * top + 1, last,
152      *     [relation computed by order.compare method])  and
153      * SUBTREE_IS_HEAP(array, 2 * top + 2, last,
154      *     [relation computed by order.compare method])  and
155      * IS_TOTAL_PREORDER([relation computed by order.compare method])
156      * </pre>
157      * @ensures <pre>
158      * SUBTREE_IS_HEAP(array, top, last,
159      *     [relation computed by order.compare method])  and
160      * perms(array, #array)  and
161      * SUBTREE_ARRAY_ENTRIES(array, top, last) =
162      *  SUBTREE_ARRAY_ENTRIES(#array, top, last)  and
163      * [the other entries in array are the same as in #array]
164      * </pre>
165      */
166     private static <T> void siftDown(T[] array, int top, int last,
167             Comparator<T> order) {
168         assert array != null : "Violation of: array is not null";
169         assert order != null : "Violation of: order is not null";
170         assert 0 <= top : "Violation of: 0 <= top";
171         assert last < array.length : "Violation of: last < |array|";
172         for (int i = 0; i < array.length; i++) {
173             assert array[i] != null : ""
174                     + "Violation of: all entries in array are not null";
175         }
176         assert isHeap(array, 2 * top + 1, last, order) : ""
177                 + "Violation of: SUBTREE_IS_HEAP(array, 2 * top + 1, last,"
178                 + " [relation computed by order.compare method])";
179         assert isHeap(array, 2 * top + 2, last, order) : ""
```

```
180                 + "Violation of: SUBTREE_IS_HEAP(array, 2 * top + 2, last,"
181                 + " [relation computed by order.compare method])";
182         /*
183          * Impractical to check last requires clause; no need to check the other
184          * requires clause, because it must be true when using the array
185          * representation for a complete binary tree.
186          */
187
188         // TODO - fill in body
189         // *** you must use the recursive algorithm discussed in class ***
190         /*
191          * Declaring left and right subtree indices
192          */
193         int lLeft = 2 * top + 1;
194         int rRight = lLeft + 1;
195
196         if (array.length > 1) {
197             if (rRight <= last) {
198                 /*
199                  * If right is less than left, right is swapped in
200                  */
201                 if (order.compare(array[lLeft], array[rRight]) > 0) {
202                     /*
203                      * Right being swapped if top is larger than right
204                      */
205                     if (order.compare(array[top], array[rRight]) > 0) {
206                         exchangeEntries(array, top, rRight);
207                         siftDown(array, rRight, last, order);
208                     }
209                     /*
210                      * If there is a right subtree but the left is less than the
211                      * right, then make left index the top index
212                      */
213                 } else if (lLeft <= last) {
214                     if (order.compare(array[top], array[lLeft]) > 0) {
215                         exchangeEntries(array, top, lLeft);
216                         siftDown(array, lLeft, last, order);
217                     }
218                 }
219             } else if (lLeft <= last) {
220                 /*
221                  * If left is smaller, then left is swapped in, and then
222                  * siftDown
223                  */
224                 if (order.compare(array[top], array[lLeft]) > 0) {
225                     exchangeEntries(array, top, lLeft);
226                     siftDown(array, lLeft, last, order);
227                 }
228             }
229         }
230     }
231
232     /**
233      * Heapifies the subtree of the given array rooted at the given {@code top}.
234      *
235      * @param <T>
236      *            type of array entries
```

```
237      * @param array
238      *            the complete binary tree
239      * @param top
240      *            the index of the root of the "subtree" to heapify
241      * @param order
242      *            the total preorder for sorting
243      * @updates array
244      * @requires <pre>
245      * 0 <= top  and
246      * for all i: integer
247      *     where (0 <= i  and  i < |array|)
248      *   ([entry at position i in array is not null])  and
249      * [subtree rooted at {@code top} is a complete binary tree]  and
250      * IS_TOTAL_PREORDER([relation computed by order.compare method])
251      * </pre>
252      * @ensures <pre>
253      * SUBTREE_IS_HEAP(array, top, |array| - 1,
254      *     [relation computed by order.compare method])  and
255      * perms(array, #array)
256      * </pre>
257      */
258     private static <T> void heapify(T[] array, int top, Comparator<T> order) {
259         assert array != null : "Violation of: array is not null";
260         assert order != null : "Violation of: order is not null";
261         assert 0 <= top : "Violation of: 0 <= top";
262         for (int i = 0; i < array.length; i++) {
263             assert array[i] != null : ""
264                     + "Violation of: all entries in array are not null";
265         }
266         /*
267          * Impractical to check last requires clause; no need to check the other
268          * requires clause, because it must be true when using the array
269          * representation for a complete binary tree.
270          */
271
272         // TODO - fill in body
273         // *** you must use the recursive algorithm discussed in class ***
274         int left = 2 * top + 1;
275         int right = 2 * top + 2;
276
277         /*
278          * Run the left and right parts separately
279          */
280         if (right < array.length) {
281             heapify(array, left, order);
282             heapify(array, right, order);
283         } else if (left < array.length) {
284             heapify(array, left, order);
285         }
286
287         /*
288          * Then use siftDown to order the tree
289          */
290         siftDown(array, top, array.length - 1, order);
291
292     }
293
```

```
294     /**
295      * Constructs and returns an array representing a heap with the entries from
296      * the given {@code Queue}.
297      *
298      * @param <T>
299      *            type of {@code Queue} and array entries
300      * @param q
301      *            the {@code Queue} with the entries for the heap
302      * @param order
303      *            the total preorder for sorting
304      * @return the array representation of a heap
305      * @clears q
306      * @requires IS_TOTAL_PREORDER([relation computed by order.compare method])
307      * @ensures <pre>
308      * SUBTREE_IS_HEAP(buildHeap, 0, |buildHeap| - 1)  and
309      * perms(buildHeap, #q)  and
310      * for all i: integer
311      *     where (0 <= i  and  i < |buildHeap|)
312      *   ([entry at position i in buildHeap is not null])  and
313      * </pre>
314      */
315     @SuppressWarnings("unchecked")
316     private static <T> T[] buildHeap(Queue<T> q, Comparator<T> order) {
317         assert q != null : "Violation of: q is not null";
318         assert order != null : "Violation of: order is not null";
319         /*
320          * Impractical to check the requires clause.
321          */
322         /*
323          * With "new T[...]" in place of "new Object[...]" it does not compile;
324          * as shown, it results in a warning about an unchecked cast, though it
325          * cannot fail.
326          */
327         T[] heap = (T[]) (new Object[q.length()]);
328
329         // TODO - fill in rest of body
330         int counter = 0;
331         while (q.length() > 0) {
332             heap[counter] = q.dequeue();
333             counter++;
334         }
335
336         heapify(heap, 0, order);
337
338         return heap;
339     }
340
341     /**
342      * Checks if the subtree of the given {@code array} rooted at the given
343      * {@code top} is a heap.
344      *
345      * @param <T>
346      *            type of array entries
347      * @param array
348      *            the complete binary tree
349      * @param top
350      *            the index of the root of the "subtree"
```

```
351      * @param last
352      *            the index of the last entry in the heap
353      * @param order
354      *            total preorder for sorting
355      * @return true if the subtree of the given {@code array} rooted at the
356      *         given {@code top} is a heap; false otherwise
357      * @requires <pre>
358      * 0 <= top  and  last < |array|  and
359      * for all i: integer
360      *     where (0 <= i  and  i < |array|)
361      *   ([entry at position i in array is not null])  and
362      * [subtree rooted at {@code top} is a complete binary tree]
363      * </pre>
364      * @ensures <pre>
365      * isHeap = SUBTREE_IS_HEAP(array, top, last,
366      *     [relation computed by order.compare method])
367      * </pre>
368      */
369     private static <T> boolean isHeap(T[] array, int top, int last,
370             Comparator<T> order) {
371         assert array != null : "Violation of: array is not null";
372         assert 0 <= top : "Violation of: 0 <= top";
373         assert last < array.length : "Violation of: last < |array|";
374         for (int i = 0; i < array.length; i++) {
375             assert array[i] != null : ""
376                     + "Violation of: all entries in array are not null";
377         }
378         /*
379          * No need to check the other requires clause, because it must be true
380          * when using the Array representation for a complete binary tree.
381          */
382         int left = 2 * top + 1;
383         boolean isHeap = true;
384         if (left <= last) {
385             isHeap = (order.compare(array[top], array[left]) <= 0)
386                     && isHeap(array, left, last, order);
387             int right = left + 1;
388             if (isHeap && (right <= last)) {
389                 isHeap = (order.compare(array[top], array[right]) <= 0)
390                         && isHeap(array, right, last, order);
391             }
392         }
393         return isHeap;
394     }
395
396     /**
397      * Checks that the part of the convention repeated below holds for the
398      * current representation.
399      *
400      * @return true if the convention holds (or if assertion checking is off);
401      *         otherwise reports a violated assertion
402      * @convention <pre>
403      * if $this.insertionMode then
404      *   $this.heapSize = 0
405      * else
406      *   $this.entries = <>  and
407      *   for all i: integer
```

```
408      *       where (0 <= i  and  i < |$this.heap|)
409      *     ([entry at position i in $this.heap is not null])  and
410      *   SUBTREE_IS_HEAP($this.heap, 0, $this.heapSize - 1,
411      *     [relation computed by $this.machineOrder.compare method])  and
412      *   0 <= $this.heapSize <= |$this.heap|
413      * </pre>
414      */
415     private boolean conventionHolds() {
416         if (this.insertionMode) {
417             assert this.heapSize == 0 : ""
418                     + "Violation of: if $this.insertionMode then $this.heapSize = 0";
419         } else {
420             assert this.entries.length() == 0 : ""
421                     + "Violation of: if not $this.insertionMode then $this.entries = <>";
422             assert 0 <= this.heapSize : ""
423                     + "Violation of: if not $this.insertionMode then 0 <= $this.heapSize";
424             assert this.heapSize <= this.heap.length : ""
425                     + "Violation of: if not $this.insertionMode then"
426                     + " $this.heapSize <= |$this.heap|";
427             for (int i = 0; i < this.heap.length; i++) {
428                 assert this.heap[i] != null : ""
429                         + "Violation of: if not $this.insertionMode then"
430                         + " all entries in $this.heap are not null";
431             }
432             assert isHeap(this.heap, 0, this.heapSize - 1,
433                     this.machineOrder) : ""
434                             + "Violation of: if not $this.insertionMode then"
435                             + " SUBTREE_IS_HEAP($this.heap, 0, $this.heapSize - 1,"
436                             + " [relation computed by $this.machineOrder.compare"
437                             + " method])";
438         }
439         return true;
440     }
441
442     /**
443      * Creator of initial representation.
444      *
445      * @param order
446      *            total preorder for sorting
447      * @requires IS_TOTAL_PREORDER([relation computed by order.compare method]
448      * @ensures <pre>
449      * $this.insertionMode = true  and
450      * $this.machineOrder = order  and
451      * $this.entries = <>  and
452      * $this.heapSize = 0
453      * </pre>
454      */
455     private void createNewRep(Comparator<T> order) {
456
457         // TODO - fill in body
458         this.machineOrder = order;
459         this.insertionMode = true;
460         this.heapSize = 0;
461         this.entries = new Queue2<>();
462
463     }
464
```

```java
    /*
     * Constructors ------------------------------------------------------------
     */

    /**
     * Constructor from order.
     *
     * @param order
     *            total preorder for sorting
     */
    public SortingMachine5a(Comparator<T> order) {
        this.createNewRep(order);
        assert this.conventionHolds();
    }

    /*
     * Standard methods --------------------------------------------------------
     */

    @SuppressWarnings("unchecked")
    @Override
    public final SortingMachine<T> newInstance() {
        try {
            Constructor<?> c = this.getClass().getConstructor(Comparator.class);
            return (SortingMachine<T>) c.newInstance(this.machineOrder);
        } catch (ReflectiveOperationException e) {
            throw new AssertionError(
                    "Cannot construct object of type " + this.getClass());
        }
    }

    @Override
    public final void clear() {
        this.createNewRep(this.machineOrder);
        assert this.conventionHolds();
    }

    @Override
    public final void transferFrom(SortingMachine<T> source) {
        assert source != null : "Violation of: source is not null";
        assert source != this : "Violation of: source is not this";
        assert source instanceof SortingMachine5a<?> : ""
                + "Violation of: source is of dynamic type SortingMachine5a<?>";
        /*
         * This cast cannot fail since the assert above would have stopped
         * execution in that case: source must be of dynamic type
         * SortingMachine5a<?>, and the ? must be T or the call would not have
         * compiled.
         */
        SortingMachine5a<T> localSource = (SortingMachine5a<T>) source;
        this.insertionMode = localSource.insertionMode;
        this.machineOrder = localSource.machineOrder;
        this.entries = localSource.entries;
        this.heap = localSource.heap;
        this.heapSize = localSource.heapSize;
        localSource.createNewRep(localSource.machineOrder);
        assert this.conventionHolds();
```

```
522         assert localSource.conventionHolds();
523     }
524 
525     /*
526      * Kernel methods ------------------------------------------------------
527      */
528 
529     @Override
530     public final void add(T x) {
531         assert x != null : "Violation of: x is not null";
532         assert this.isInInsertionMode() : "Violation of: this.insertion_mode";
533 
534         // TODO - fill in body
535         this.entries.enqueue(x);
536 
537         assert this.conventionHolds();
538     }
539 
540     @Override
541     public final void changeToExtractionMode() {
542         assert this.isInInsertionMode() : "Violation of: this.insertion_mode";
543 
544         // TODO - fill in body
545         this.insertionMode = false;
546         this.heap = buildHeap(this.entries, this.machineOrder);
547         this.heapSize = this.heap.length;
548 
549         assert this.conventionHolds();
550     }
551 
552     @Override
553     public final T removeFirst() {
554         assert !this
555                 .isInInsertionMode() : "Violation of: not this.insertion_mode";
556         assert this.size() > 0 : "Violation of: this.contents /= {}";
557 
558         // TODO - fill in body
559         T removeF = this.heap[0];
560         exchangeEntries(this.heap, 0, this.heapSize - 1);
561         this.heapSize--;
562         siftDown(this.heap, 0, this.heapSize - 1, this.machineOrder);
563 
564         assert this.conventionHolds();
565 
566         return removeF;
567     }
568 
569     @Override
570     public final boolean isInInsertionMode() {
571         assert this.conventionHolds();
572         return this.insertionMode;
573     }
574 
575     @Override
576     public final Comparator<T> order() {
577         assert this.conventionHolds();
578         return this.machineOrder;
```

```
579     }
580
581     @Override
582     public final int size() {
583
584         // TODO - fill in body
585         int size = 0;
586         if (this.insertionMode) {
587             size = this.entries.length();
588         } else {
589             size = this.heapSize;
590         }
591
592         assert this.conventionHolds();
593
594         return size;
595     }
596
597     @Override
598     public final Iterator<T> iterator() {
599         return new SortingMachine5aIterator();
600     }
601
602     /**
603      * Implementation of {@code Iterator} interface for
604      * {@code SortingMachine5a}.
605      */
606     private final class SortingMachine5aIterator implements Iterator<T> {
607
608         /**
609          * Representation iterator when in insertion mode.
610          */
611         private Iterator<T> queueIterator;
612
613         /**
614          * Representation iterator count when in extraction mode.
615          */
616         private int arrayCurrentIndex;
617
618         /**
619          * No-argument constructor.
620          */
621         private SortingMachine5aIterator() {
622             if (SortingMachine5a.this.insertionMode) {
623                 this.queueIterator = SortingMachine5a.this.entries.iterator();
624             } else {
625                 this.arrayCurrentIndex = 0;
626             }
627             assert SortingMachine5a.this.conventionHolds();
628         }
629
630         @Override
631         public boolean hasNext() {
632             boolean hasNext;
633             if (SortingMachine5a.this.insertionMode) {
634                 hasNext = this.queueIterator.hasNext();
635             } else {
```

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