

```
1 import java.util.Comparator;
9
10 /**
11  * {@code SortingMachine} represented as a {@code Queue} and
12  * an array (using an
13  * embedding of heap sort), with implementations of primary
14  * methods.
15  *
16  * @param <T>
17  *     type of {@code SortingMachine} entries
18  * @mathdefinitions <pre>
19  * IS_TOTAL_PREORDER (
20  *   r: binary relation on T
21  * ) : boolean is
22  *   for all x, y, z: T
23  *     ((r(x, y) or r(y, x)) and
24  *       (if (r(x, y) and r(y, z)) then r(x, z)))
25  *
26  * SUBTREE_IS_HEAP (
27  *   a: string of T,
28  *   start: integer,
29  *   stop: integer,
30  *   r: binary relation on T
31  * ) : boolean is
32  *   [the subtree of a (when a is interpreted as a complete
33  *     binary tree) rooted
34  *     at index start and only through entry stop of a
35  *     satisfies the heap
36  *     ordering property according to the relation r]
37  *
38  * SUBTREE_ARRAY_ENTRIES (
39  *   a: string of T,
40  *   start: integer,
41  *   stop: integer
42  * ) : finite multiset of T is
43  *   [the multiset of entries in a that belong to the subtree
44  *     of a
45  *     (when a is interpreted as a complete binary tree) rooted
46  *     at
47  *     index start and only through entry stop]
48  * </pre>
49  * @convention <pre>
50  * IS_TOTAL_PREORDER([relation computed by
51  *   $this.machineOrder.compare method] and
52  *   if $this.insertionMode then
53  *     $this.heapSize = 0
54  *   else
```

```
48 *   $this.entries = <> and
49 *   for all i: integer
50 *       where (0 <= i and i < |$this.heap|)
51 *       ([entry at position i in $this.heap is not null]) and
52 *       SUBTREE_IS_HEAP($this.heap, 0, $this.heapSize - 1,
53 *       [relation computed by $this.machineOrder.compare
method]) and
54 *   0 <= $this.heapSize <= |$this.heap|
55 * </pre>
56 * @correspondence <pre>
57 * if $this.insertionMode then
58 *   this = (true, $this.machineOrder,
multiset_entries($this.entries))
59 * else
60 *   this = (false, $this.machineOrder,
multiset_entries($this.heap[0, $this.heapSize]))
61 * </pre>
62 *
63 * @author Zhuoyang Li + Xinci Ma
64 *
65 */
66 public class SortingMachine5a<T> extends
SortingMachineSecondary<T> {
67
68     /*
69     * Private members
70     */
71
72     /**
73     * Order.
74     */
75     private Comparator<T> machineOrder;
76
77     /**
78     * Insertion mode.
79     */
80     private boolean insertionMode;
81
82     /**
83     * Entries.
84     */
85     private Queue<T> entries;
86
87     /**
88     * Heap.
89     */
```

```
90     private T[] heap;
91
92     /**
93      * Heap size.
94      */
95     private int heapSize;
96
97     /**
98      * Exchanges entries at indices {@code i} and {@code j}
99      * 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
112     *                  j exchanged]
113     */
114     private static <T> void exchangeEntries(T[] array, int i,
115     int j) {
116         assert array != null : "Violation of: array is not
117         null";
118         assert 0 <= i : "Violation of: 0 <= i";
119         assert i < array.length : "Violation of: i < |
120         array|";
121         assert 0 <= j : "Violation of: 0 <= j";
122         assert j < array.length : "Violation of: j < |
123         array|";
124
125         T temp = array[i];
126         array[i] = array[j];
127         array[j] = temp;
128     }
129
130     /**
131     * Given an array that represents a complete binary tree
132     * and an index
133     * * referring to the root of a subtree that would be a
134     * heap except for its
135     * * root, sifts the root down to turn that whole subtree
```

```

    into a heap.
129     *
130     * @param <T>
131     *         type of array entries
132     * @param array
133     *         the complete binary tree
134     * @param top
135     *         the index of the root of the "subtree"
136     * @param last
137     *         the index of the last entry in the heap
138     * @param order
139     *         total preorder for sorting
140     * @updates array
141     * @requires <pre>
142     * 0 <= top and last < |array| and
143     * for all i: integer
144     *   where (0 <= i and i < |array|)
145     *   ([entry at position i in array is not null]) and
146     *   [subtree rooted at {@code top} is a complete binary
147     *   tree] and
148     *   [relation computed by order.compare method]) and
149     *   [relation computed by order.compare method]) and
150     *   [relation computed by order.compare method]) and
151     *   [relation computed by order.compare
152     *   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
164     last,
165     Comparator<T> order) {
166         assert array != null : "Violation of: array is not
167         null";
168         assert order != null : "Violation of: order is not
169         null";
170         assert 0 <= top : "Violation of: 0 <= top";
171         assert last < array.length : "Violation of: last < |
172         array|";
173         for (int i = 0; i < array.length; i++) {

```

```
169         assert array[i] != null : ""
170             + "Violation of: all entries in array are
    not null";
171     }
172     assert isHeap(array, 2 * top + 1, last, order) : ""
173         + "Violation of: SUBTREE_IS_HEAP(array, 2 *
    top + 1, last,"
174         + " [relation computed by order.compare
    method])";
175     assert isHeap(array, 2 * top + 2, last, order) : ""
176         + "Violation of: SUBTREE_IS_HEAP(array, 2 *
    top + 2, last,"
177         + " [relation computed by order.compare
    method])";
178     /*
179     * Impractical to check last requires clause; no need
    to check the other
180     * requires clause, because it must be true when
    using the array
181     * representation for a complete binary tree.
182     */
183
184     int smallest = top;
185     int leftChildIndex = 2 * top + 1;
186     int rightChildIndex = 2 * top + 2;
187
188     if (leftChildIndex <= last
189         && order.compare(array[leftChildIndex],
    array[smallest]) < 0) {
190         smallest = leftChildIndex;
191     }
192
193     if (rightChildIndex <= last
194         && order.compare(array[rightChildIndex],
    array[smallest]) < 0) {
195         smallest = rightChildIndex;
196     }
197
198     if (smallest != top) {
199         exchangeEntries(array, top, smallest);
200         siftDown(array, smallest, last, order);
201     }
202     // *** you must use the recursive algorithm discussed
    in class ***
203
204 }
205
```

```

206  /**
207   * Heapifies the subtree of the given array rooted at the
   given {@code top}.
208   *
209   * @param <T>
210   *         type of array entries
211   * @param array
212   *         the complete binary tree
213   * @param top
214   *         the index of the root of the "subtree" to
   heapify
215   * @param order
216   *         the total preorder for sorting
217   * @updates array
218   * @requires <pre>
219   * 0 <= top and
220   * for all i: integer
221   *   where (0 <= i and i < |array|)
222   *   ([entry at position i in array is not null]) and
223   * [subtree rooted at {@code top} is a complete binary
   tree] and
224   * IS_TOTAL_PREORDER([relation computed by order.compare
   method])
225   * </pre>
226   * @ensures <pre>
227   * SUBTREE_IS_HEAP(array, top, |array| - 1,
228   * [relation computed by order.compare method]) and
229   * perms(array, #array)
230   * </pre>
231   */
232   private static <T> void heapify(T[] array, int top,
   Comparator<T> order) {
233       assert array != null : "Violation of: array is not
   null";
234       assert order != null : "Violation of: order is not
   null";
235       assert 0 <= top : "Violation of: 0 <= top";
236       for (int i = 0; i < array.length; i++) {
237           assert array[i] != null : ""
238               + "Violation of: all entries in array are
   not null";
239       }
240       /**
241        * Impractical to check last requires clause; no need
   to check the other
242        * requires clause, because it must be true when
   using the array

```

```
243         * representation for a complete binary tree.
244         */
245
246         int last = array.length - 1;
247         int leftChildIndex = 2 * top + 1;
248         int rightChildIndex = 2 * top + 2;
249
250         if (leftChildIndex <= last) {
251             heapify(array, leftChildIndex, order);
252         }
253
254         if (rightChildIndex <= last) {
255             heapify(array, rightChildIndex, order);
256         }
257
258         siftDown(array, top, last, order);
259         // *** you must use the recursive algorithm discussed
in class ***
260     }
261
262
263     /**
264     * Constructs and returns an array representing a heap
with the entries from
265     * the given {@code Queue}.
266     *
267     * @param <T>
268     *         type of {@code Queue} and array entries
269     * @param q
270     *         the {@code Queue} with the entries for the
heap
271     * @param order
272     *         the total preorder for sorting
273     * @return the array representation of a heap
274     * @clears q
275     * @requires IS_TOTAL_PREORDER([relation computed by
order.compare method])
276     * @ensures <pre>
277     * SUBTREE_IS_HEAP(buildHeap, 0, |buildHeap| - 1) and
278     * perms(buildHeap, #q) and
279     * for all i: integer
280     *     where (0 <= i and i < |buildHeap|)
281     *     ([entry at position i in buildHeap is not null])
and
282     * </pre>
283     */
284     @SuppressWarnings("unchecked")
```

```

285     private static <T> T[] buildHeap(Queue<T> q,
    Comparator<T> order) {
286         assert q != null : "Violation of: q is not null";
287         assert order != null : "Violation of: order is not
    null";
288         /*
289          * Impractical to check the requires clause.
290          */
291         /*
292          * With "new T[...]" in place of "new Object[...]" it
    does not compile;
293          * as shown, it results in a warning about an
    unchecked cast, though it
294          * cannot fail.
295          */
296         T[] heap = (T[]) (new Object[q.length()]);
297
298         int index = 0;
299         while (q.length() > 0) {
300             heap[index++] = q.dequeue();
301         }
302
303         for (int i = (heap.length / 2) - 1; i >= 0; i--) {
304             heapify(heap, i, order);
305         }
306
307         return heap;
308     }
309
310     /**
311      * Checks if the subtree of the given {@code array}
    rooted at the given
312      * {@code top} is a heap.
313      *
314      * @param <T>
315      *         type of array entries
316      * @param array
317      *         the complete binary tree
318      * @param top
319      *         the index of the root of the "subtree"
320      * @param last
321      *         the index of the last entry in the heap
322      * @param order
323      *         total preorder for sorting
324      * @return true if the subtree of the given {@code array}
    rooted at the
325      *         given {@code top} is a heap; false otherwise

```



```
326     * @requires <pre>
327     * 0 <= top and last < |array| and
328     * for all i: integer
329     *     where (0 <= i and i < |array|)
330     *     ([entry at position i in array is not null]) and
331     * [subtree rooted at {@code top} is a complete binary
    tree]
332     * </pre>
333     * @ensures <pre>
334     * isHeap = SUBTREE_IS_HEAP(array, top, last,
335     * [relation computed by order.compare method])
336     * </pre>
337     */
338     private static <T> boolean isHeap(T[] array, int top, int
    last,
339         Comparator<T> order) {
340         assert array != null : "Violation of: array is not
    null";
341         assert 0 <= top : "Violation of: 0 <= top";
342         assert last < array.length : "Violation of: last < |
    array|";
343         for (int i = 0; i < array.length; i++) {
344             assert array[i] != null : ""
345                 + "Violation of: all entries in array are
    not null";
346         }
347         /*
348          * No need to check the other requires clause,
    because it must be true
349          * when using the Array representation for a complete
    binary tree.
350          */
351         int left = 2 * top + 1;
352         boolean isHeap = true;
353         if (left <= last) {
354             isHeap = (order.compare(array[top], array[left])
    <= 0)
355                 && isHeap(array, left, last, order);
356             int right = left + 1;
357             if (isHeap && (right <= last)) {
358                 isHeap = (order.compare(array[top],
    array[right]) <= 0)
359                     && isHeap(array, right, last, order);
360             }
361         }
362         return isHeap;
363     }
```

```
364
365  /**
366   * Checks that the part of the convention repeated below
   holds for the
367   * current representation.
368   *
369   * @return true if the convention holds (or if assertion
   checking is off);
370   * otherwise reports a violated assertion
371   * @convention <pre>
372   * if $this.insertionMode then
373   *   $this.heapSize = 0
374   * else
375   *   $this.entries = <> and
376   *   for all i: integer
377   *     where (0 <= i and i < |$this.heap|)
378   *       ([entry at position i in $this.heap is not null])
   and
379   *   SUBTREE_IS_HEAP($this.heap, 0, $this.heapSize - 1,
380   *   [relation computed by $this.machineOrder.compare
   method]) and
381   *   0 <= $this.heapSize <= |$this.heap|
382   * </pre>
383   */
384   private boolean conventionHolds() {
385     if (this.insertionMode) {
386       assert this.heapSize == 0 : ""
387         + "Violation of: if $this.insertionMode
   then $this.heapSize = 0";
388     } else {
389       assert this.entries.length() == 0 : ""
390         + "Violation of: if not
   $this.insertionMode then $this.entries = <>";
391       assert 0 <= this.heapSize : ""
392         + "Violation of: if not
   $this.insertionMode then 0 <= $this.heapSize";
393       assert this.heapSize <= this.heap.length : ""
394         + "Violation of: if not
   $this.insertionMode then"
395         + " $this.heapSize <= |$this.heap|";
396       for (int i = 0; i < this.heap.length; i++) {
397         assert this.heap[i] != null : ""
398           + "Violation of: if not
   $this.insertionMode then"
399           + " all entries in $this.heap are not
   null";
400       }
```

```
401         assert isHeap(this.heap, 0, this.heapSize - 1,
402                        this.machineOrder) : ""
403                        + "Violation of: if not
    $this.insertionMode then"
404                        + " SUBTREE_IS_HEAP($this.heap,
    0, $this.heapSize - 1,"
405                        + " [relation computed by
    $this.machineOrder.compare"
406                        + " method])";
407     }
408     return true;
409 }
410
411 /**
412  * Creator of initial representation.
413  *
414  * @param order
415  *      total preorder for sorting
416  * @requires IS_TOTAL_PREORDER([relation computed by
    order.compare method]
417  * @ensures <pre>
418  * $this.insertionMode = true and
419  * $this.machineOrder = order and
420  * $this.entries = <> and
421  * $this.heapSize = 0
422  * </pre>
423  */
424 private void createNewRep(Comparator<T> order) {
425
426     this.insertionMode = true;
427     this.machineOrder = order;
428     this.entries = new Queue1L<T>();
429     this.heap = null;
430     this.heapSize = 0;
431 }
432
433
434 /*
435  * Constructors
436  */
437
438 /**
439  * Constructor from order.
440  *
441  * @param order
442  *      total preorder for sorting
```

```
443     */
444     public SortingMachine5a(Comparator<T> order) {
445         this.createNewRep(order);
446         assert this.conventionHolds();
447     }
448
449     /*
450     * Standard methods
451     */
452
453     @SuppressWarnings("unchecked")
454     @Override
455     public final SortingMachine<T> newInstance() {
456         try {
457             return
458                 this.getClass().getConstructor(Comparator.class)
459                     .newInstance(this.machineOrder);
460         } catch (ReflectiveOperationException e) {
461             throw new AssertionError(
462                 "Cannot construct object of type " +
463                 this.getClass());
464         }
465     }
466
467     @Override
468     public final void clear() {
469         this.createNewRep(this.machineOrder);
470         assert this.conventionHolds();
471     }
472
473     @Override
474     public final void transferFrom(SortingMachine<T> source)
475     {
476         assert source != null : "Violation of: source is not
477         null";
478         assert source != this : "Violation of: source is not
479         this";
480         assert source instanceof SortingMachine5a<?> : ""
481             + "Violation of: source is of dynamic type
482             SortingMachine5a<?>";
483         /*
484          * This cast cannot fail since the assert above would
485          have stopped
486          * execution in that case: source must be of dynamic
487          type
488          * SortingMachine5a<?>, and the ? must be T or the
```

```
call would not have
481     * compiled.
482     */
483     SortingMachine5a<T> localSource =
    (SortingMachine5a<T>) source;
484     this.insertionMode = localSource.insertionMode;
485     this.machineOrder = localSource.machineOrder;
486     this.entries = localSource.entries;
487     this.heap = localSource.heap;
488     this.heapSize = localSource.heapSize;
489     localSource.createNewRep(localSource.machineOrder);
490     assert this.conventionHolds();
491     assert localSource.conventionHolds();
492 }
493
494 /*
495  * Kernel methods
496  */
497
498 @Override
499 public final void add(T x) {
500     assert x != null : "Violation of: x is not null";
501     assert this.isInInsertionMode() : "Violation of:
this.insertion_mode";
502
503     this.entries.enqueue(x);
504
505     assert this.conventionHolds();
506 }
507
508 @Override
509 public final void changeToExtractionMode() {
510     assert this.isInInsertionMode() : "Violation of:
this.insertion_mode";
511
512     this.insertionMode = false; //change to extraction
mode
513     this.heap = buildHeap(this.entries,
this.machineOrder); //build heap
514     this.heapSize = this.heap.length;
515
516     assert this.conventionHolds();
517 }
518
519 @Override
520 public final T removeFirst() {
```

```
521         assert !this
522             .isInInsertionMode() : "Violation of: not
this.insertion_mode";
523         assert this.size() > 0 : "Violation of:
this.contents /= {}";
524
525         T removed = this.heap[0]; //remove the root
526
527         if (this.heap.length > 1) {
528             exchangeEntries(this.heap, 0, this.heapSize - 1);
529         }
530         this.heapSize--;
531         siftDown(this.heap, 0, this.heapSize - 1,
this.machineOrder);
532
533         assert this.conventionHolds();
534         return removed;
535     }
536
537     @Override
538     public final boolean isInInsertionMode() {
539         assert this.conventionHolds();
540         return this.insertionMode;
541     }
542
543     @Override
544     public final Comparator<T> order() {
545         assert this.conventionHolds();
546         return this.machineOrder;
547     }
548
549     @Override
550     public final int size() {
551
552         int currentSize;
553
554         if (this.insertionMode) {
555             currentSize = this.entries.length(); //if in
insertion mode
556         } else {
557             currentSize = this.heapSize; //if in extraction
mode
558         }
559
560         assert this.conventionHolds();
561         return currentSize;
562     }
```

```
563
564     @Override
565     public final Iterator<T> iterator() {
566         return new SortingMachine5aIterator();
567     }
568
569     /**
570      * Implementation of {@code Iterator} interface for
571      * {@code SortingMachine5a}.
572      */
573     private final class SortingMachine5aIterator implements
574         Iterator<T> {
575         /**
576          * Representation iterator when in insertion mode.
577          */
578         private Iterator<T> queueIterator;
579
580         /**
581          * Representation iterator count when in extraction
582          mode.
583          */
584         private int arrayCurrentIndex;
585
586         /**
587          * No-argument constructor.
588          */
589         private SortingMachine5aIterator() {
590             if (SortingMachine5a.this.insertionMode) {
591                 this.queueIterator =
592                     SortingMachine5a.this.entries.iterator();
593             } else {
594                 this.arrayCurrentIndex = 0;
595             }
596             assert SortingMachine5a.this.conventionHolds();
597         }
598
599         @Override
600         public boolean hasNext() {
601             boolean hasNext;
602             if (SortingMachine5a.this.insertionMode) {
603                 hasNext = this.queueIterator.hasNext();
604             } else {
605                 hasNext = this.arrayCurrentIndex <
606                     SortingMachine5a.this.heapSize;
607             }
608             assert SortingMachine5a.this.conventionHolds();
609         }
610     }
611 }
```

```
606         return hasNext;
607     }
608
609     @Override
610     public T next() {
611         assert this.hasNext() : "Violation of:
~this.unseen /= <>";
612         if (!this.hasNext()) {
613             /*
614              * Exception is supposed to be thrown in this
        case, but with
615              * assertion-checking enabled it cannot
        happen because of assert
616              * above.
617              */
618             throw new NoSuchElementException();
619         }
620         T next;
621         if (SortingMachine5a.this.insertionMode) {
622             next = this.queueIterator.next();
623         } else {
624             next =
        SortingMachine5a.this.heap[this.arrayCurrentIndex];
625             this.arrayCurrentIndex++;
626         }
627         assert SortingMachine5a.this.conventionHolds();
628         return next;
629     }
630
631     @Override
632     public void remove() {
633         throw new UnsupportedOperationException(
634             "remove operation not supported");
635     }
636
637 }
638
639 }
640
```