The Ohio State University

PROJECT 5: SORTINGMACHINE WITH HEAPSORT

Daniil Gofman

Ansh Pachauri

SW 2: Dev & Dsgn

Paolo Bucci

Yiyang Chen

Shivam Gupta

October 10, 2023

```
 1 import java.util.Comparator;
 2 import java.util.Iterator;
 3 import java.util.NoSuchElementException;
 4
 5 import components.queue.Queue;
 6 import components.queue.Queue1L;
 7 import components.sortingmachine.SortingMachine;
 8 import components.sortingmachine.SortingMachineSecondary;
 9
10 /**
11  * {@code SortingMachine} represented as a {@code Queue} and an
   array (using an
12  * embedding of heap sort), with implementations of primary
   methods.
13  *
14  * @param <T>
15  *            type of {@code SortingMachine} entries
16  * @mathdefinitions <pre>
17  * IS_TOTAL_PREORDER (
18  *   r: binary relation on T
19  *  ) : boolean is
20  *  for all x, y, z: T
21  *   ((r(x, y) or r(y, x))  and
22  *    (if (r(x, y) and r(y, z)) then r(x, z)))
23  *
24  * SUBTREE_IS_HEAP (
25  *   a: string of T,
26  *   start: integer,
27  *   stop: integer,
28  *   r: binary relation on T
29  *  ) : boolean is
30  *  [the subtree of a (when a is interpreted as a complete
   binary tree) rooted
31  *   at index start and only through entry stop of a satisfies
   the heap
32  *   ordering property according to the relation r]
33  *
34  * SUBTREE_ARRAY_ENTRIES (
35  *   a: string of T,
```

```
 36 *   start: integer,
 37 *   stop: integer
 38 *  ) : finite multiset of T is
 39 *  [the multiset of entries in a that belong to the subtree of
    a
 40 *   (when a is interpreted as a complete binary tree) rooted
    at
 41 *   index start and only through entry stop]
 42 * </pre>
 43 * @convention <pre>
 44 * IS_TOTAL_PREORDER([relation computed by
    $this.machineOrder.compare method]  and
 45 * if $this.insertionMode then
 46 *   $this.heapSize = 0
 47 * 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 Daniil Gofman, Ansh Pachauri
 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} 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        if (i != j) {
120            T tmp = array[i];
121            array[i] = array[j];
122            array[j] = tmp;
123        }
124    }
125
126    /**
127     * Given an array that represents a complete binary tree
    and an index
128     * referring to the root of a subtree that would be a heap
    except for its
129     * root, sifts the root down to turn that whole subtree
    into a heap.
130     *
131     * @param <T>
132     *            type of array entries
133     * @param array
134     *            the complete binary tree
135     * @param top
136     *            the index of the root of the "subtree"
137     * @param last
```

```
138     *            the index of the last entry in the heap
139     * @param order
140     *            total preorder for sorting
141     * @updates array
142     * @requires <pre>
143     * 0 <= top  and  last < |array|  and
144     * for all i: integer
145     *     where (0 <= i  and  i < |array|)
146     *   ([entry at position i in array is not null])  and
147     * [subtree rooted at {@code top} is a complete binary
    tree]  and
148     * SUBTREE_IS_HEAP(array, 2 * top + 1, last,
149     *     [relation computed by order.compare method])  and
150     * SUBTREE_IS_HEAP(array, 2 * top + 2, last,
151     *     [relation computed by order.compare method])  and
152     * IS_TOTAL_PREORDER([relation computed by order.compare
    method])
153     * </pre>
154     * @ensures <pre>
155     * SUBTREE_IS_HEAP(array, top, last,
156     *     [relation computed by order.compare method])  and
157     * perms(array, #array)  and
158     * SUBTREE_ARRAY_ENTRIES(array, top, last) =
159     *  SUBTREE_ARRAY_ENTRIES(#array, top, last)  and
160     * [the other entries in array are the same as in #array]
161     * </pre>
162     */
163    private static <T> void siftDown(T[] array, int top, int
    last,
164            Comparator<T> order) {
165        assert array != null : "Violation of: array is not
    null";
166        assert order != null : "Violation of: order is not
    null";
167        assert 0 <= top : "Violation of: 0 <= top";
168        assert last < array.length : "Violation of: last < |
    array|";
169        for (int i = 0; i < array.length; i++) {
170            assert array[i] != null : ""
```

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

```
200                     exchangeEntries(array, top, smallest);
201                     siftDown(array, right, last, order);
202                 }
203             } else {
204                 if (order.compare(array[top], array[left]) > 0)
    {
205                     exchangeEntries(array, top, (2 * top + 1));
206                 }
207             }
208         }
209
210     }
211
212     /**
213      * Heapifies the subtree of the given array rooted at the
    given {@code top}.
214      *
215      * @param <T>
216      *            type of array entries
217      * @param array
218      *            the complete binary tree
219      * @param top
220      *            the index of the root of the "subtree" to
    heapify
221      * @param order
222      *            the total preorder for sorting
223      * @updates array
224      * @requires <pre>
225      * 0 <= top  and
226      * for all i: integer
227      *     where (0 <= i  and  i < |array|)
228      *   ([entry at position i in array is not null])  and
229      * [subtree rooted at {@code top} is a complete binary
    tree]  and
230      * IS_TOTAL_PREORDER([relation computed by order.compare
    method])
231      * </pre>
232      * @ensures <pre>
233      * SUBTREE_IS_HEAP(array, top, |array| - 1,
```

```
234     *     [relation computed by order.compare method])  and
235     * perms(array, #array)
236     * </pre>
237     */
238    private static <T> void heapify(T[] array, int top,
   Comparator<T> order) {
239        assert array != null : "Violation of: array is not
   null";
240        assert order != null : "Violation of: order is not
   null";
241        assert 0 <= top : "Violation of: 0 <= top";
242        for (int i = 0; i < array.length; i++) {
243            assert array[i] != null : ""
244                    + "Violation of: all entries in array are
   not null";
245        }
246        /*
247         * Impractical to check last requires clause; no need
   to check the other
248         * requires clause, because it must be true when using
   the array
249         * representation for a complete binary tree.
250         */
251
252        int left = 2 * top + 1;
253        int right = 2 * top + 2;
254
255        if (left < array.length) {
256            heapify(array, left, order);
257            heapify(array, right, order);
258            siftDown(array, top, array.length - 1, order);
259        }
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         // Copy elements from the queue to the heap array
```

```
299         int index = 0;
300         while (q.length() > 0) {
301             T element = q.dequeue();
302             heap[index] = element;
303             index++;
304         }
305
306         // Build the heap by ensuring the heap property
307         for (int i = (heap.length - 1) / 2; i >= 0; i--) {
308             heapify(heap, i, order);
309         }
310
311         return heap;
312     }
313
314     /**
315      * Checks if the subtree of the given {@code array} rooted
    at the given
316      * {@code top} is a heap.
317      *
318      * @param <T>
319      *            type of array entries
320      * @param array
321      *            the complete binary tree
322      * @param top
323      *            the index of the root of the "subtree"
324      * @param last
325      *            the index of the last entry in the heap
326      * @param order
327      *            total preorder for sorting
328      * @return true if the subtree of the given {@code array}
    rooted at the
329      *         given {@code top} is a heap; false otherwise
330      * @requires <pre>
331      * 0 <= top  and  last < |array|  and
332      * for all i: integer
333      *     where (0 <= i  and  i < |array|)
334      *   ([entry at position i in array is not null])  and
335      * [subtree rooted at {@code top} is a complete binary
```

```
    tree]
336      * </pre>
337      * @ensures <pre>
338      * isHeap = SUBTREE_IS_HEAP(array, top, last,
339      *     [relation computed by order.compare method])
340      * </pre>
341      */
342     private static <T> boolean isHeap(T[] array, int top, int
    last,
343             Comparator<T> order) {
344         assert array != null : "Violation of: array is not
    null";
345         assert 0 <= top : "Violation of: 0 <= top";
346         assert last < array.length : "Violation of: last < |
    array|";
347         for (int i = 0; i < array.length; i++) {
348             assert array[i] != null : ""
349                     + "Violation of: all entries in array are
    not null";
350         }
351         /*
352          * No need to check the other requires clause, because
    it must be true
353          * when using the Array representation for a complete
    binary tree.
354          */
355         int left = 2 * top + 1;
356         boolean isHeap = true;
357         if (left <= last) {
358             isHeap = (order.compare(array[top], array[left]) <=
    0)
359                     && isHeap(array, left, last, order);
360             int right = left + 1;
361             if (isHeap && (right <= last)) {
362                 isHeap = (order.compare(array[top],
    array[right]) <= 0)
363                         && isHeap(array, right, last, order);
364             }
365         }
```

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

```
398                     + "Violation of: if not $this.insertionMode
    then"
399                     + " $this.heapSize <= |$this.heap|";
400             for (int i = 0; i < this.heap.length; i++) {
401                 assert this.heap[i] != null : ""
402                         + "Violation of: if not
    $this.insertionMode then"
403                         + " all entries in $this.heap are not
    null";
404             }
405             assert isHeap(this.heap, 0, this.heapSize - 1,
406                     this.machineOrder) : ""
407                             + "Violation of: if not
    $this.insertionMode then"
408                             + " SUBTREE_IS_HEAP($this.heap, 0,
    $this.heapSize - 1,"
409                             + " [relation computed by
    $this.machineOrder.compare"
410                             + " method])";
411         }
412         return true;
413     }
414
415     /**
416      * Creator of initial representation.
417      *
418      * @param order
419      *            total preorder for sorting
420      * @requires IS_TOTAL_PREORDER([relation computed by
    order.compare method]
421      * @ensures <pre>
422      * $this.insertionMode = true  and
423      * $this.machineOrder = order  and
424      * $this.entries = <>  and
425      * $this.heapSize = 0
426      * </pre>
427      */
428     private void createNewRep(Comparator<T> order) {
429
```

```
430         // Set up insertion mode according to contract
431         this.insertionMode = true;
432         // Set up comparator
433         this.machineOrder = order;
434         // Create representation variable
435         this.entries = new Queue1L<T>();
436         // Set the number of elements in a queue
437         this.heapSize = 0;
438         // Build heap
439         this.heap = buildHeap(this.entries, order);
440
441     }
442
443     /*
444      * Constructors
    ------------------------------------------------------------
445      */
446
447     /**
448      * Constructor from order.
449      *
450      * @param order
451      *            total preorder for sorting
452      */
453     public SortingMachine5a(Comparator<T> order) {
454         this.createNewRep(order);
455         assert this.conventionHolds();
456     }
457
458     /*
459      * Standard methods
    -----------------------------------------------------------
460      */
461
462     @SuppressWarnings("unchecked")
463     @Override
464     public final SortingMachine<T> newInstance() {
465         try {
466             return
```

```
    this.getClass().getConstructor(Comparator.class)
467                     .newInstance(this.machineOrder);
468         } catch (ReflectiveOperationException e) {
469             throw new AssertionError(
470                     "Cannot construct object of type " +
    this.getClass());
471         }
472     }
473
474     @Override
475     public final void clear() {
476         this.createNewRep(this.machineOrder);
477         assert this.conventionHolds();
478     }
479
480     @Override
481     public final void transferFrom(SortingMachine<T> source) {
482         assert source != null : "Violation of: source is not
    null";
483         assert source != this : "Violation of: source is not
    this";
484         assert source instanceof SortingMachine5a<?> : ""
485                 + "Violation of: source is of dynamic type
    SortingMachine5a<?>";
486         /*
487          * This cast cannot fail since the assert above would
    have stopped
488          * execution in that case: source must be of dynamic
    type
489          * SortingMachine5a<?>, and the ? must be T or the call
    would not have
490          * compiled.
491          */
492         SortingMachine5a<T> localSource = (SortingMachine5a<T>)
    source;
493         this.insertionMode = localSource.insertionMode;
494         this.machineOrder = localSource.machineOrder;
495         this.entries = localSource.entries;
496         this.heap = localSource.heap;
```

```
497         this.heapSize = localSource.heapSize;
498         localSource.createNewRep(localSource.machineOrder);
499         assert this.conventionHolds();
500         assert localSource.conventionHolds();
501     }
502
503     /*
504      * Kernel methods
    ------------------------------------------------------------
505      */
506
507     @Override
508     public final void add(T x) {
509         assert x != null : "Violation of: x is not null";
510         assert this.isInInsertionMode() : "Violation of:
    this.insertion_mode";
511
512         this.entries.enqueue(x);
513
514         assert this.conventionHolds();
515     }
516
517     @Override
518     public final void changeToExtractionMode() {
519         assert this.isInInsertionMode() : "Violation of:
    this.insertion_mode";
520
521         this.insertionMode = false;
522
523         this.heap = buildHeap(this.entries, this.machineOrder);
524         this.heapSize = this.heap.length;
525
526         assert this.conventionHolds();
527     }
528
529     @Override
530     public final T removeFirst() {
531         assert !this
532                 .isInInsertionMode() : "Violation of: not
```

```
   this.insertion_mode";
533         assert this.size() > 0 : "Violation of: this.contents /
   = {}";
534
535         T x = this.heap[0];
536         // switch root with last entry in heap
537         exchangeEntries(this.heap, 0, this.heapSize - 1);
538         this.heapSize--;
539         siftDown(this.heap, 0, this.heapSize - 1,
   this.machineOrder);
540
541         assert this.conventionHolds();
542         // Fix this line to return the result after checking
   the convention.
543
544         return x;
545
546     }
547
548     @Override
549     public final boolean isInInsertionMode() {
550         assert this.conventionHolds();
551         return this.insertionMode;
552     }
553
554     @Override
555     public final Comparator<T> order() {
556         assert this.conventionHolds();
557         return this.machineOrder;
558     }
559
560     @Override
561     public final int size() {
562
563         assert this.conventionHolds();
564
565         int result = this.heapSize;
566         if (this.insertionMode) {
567             result = this.entries.length();
```

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

```
604             assert SortingMachine5a.this.conventionHolds();
605         }
606 
607         @Override
608         public boolean hasNext() {
609             boolean hasNext;
610             if (SortingMachine5a.this.insertionMode) {
611                 hasNext = this.queueIterator.hasNext();
612             } else {
613                 hasNext = this.arrayCurrentIndex <
    SortingMachine5a.this.heapSize;
614             }
615             assert SortingMachine5a.this.conventionHolds();
616             return hasNext;
617         }
618 
619         @Override
620         public T next() {
621             assert this.hasNext() : "Violation of:
    ~this.unseen /= <>";
622             if (!this.hasNext()) {
623                 /*
624                  * Exception is supposed to be thrown in this
    case, but with
625                  * assertion-checking enabled it cannot happen
    because of assert
626                  * above.
627                  */
628                 throw new NoSuchElementException();
629             }
630             T next;
631             if (SortingMachine5a.this.insertionMode) {
632                 next = this.queueIterator.next();
633             } else {
634                 next =
    SortingMachine5a.this.heap[this.arrayCurrentIndex];
635                 this.arrayCurrentIndex++;
636             }
637             assert SortingMachine5a.this.conventionHolds();
```

```
638             return next;
639         }
640 
641         @Override
642         public void remove() {
643             throw new UnsupportedOperationException(
644                     "remove operation not supported");
645         }
646 
647     }
648 
649 }
650 
```