CSE 2231 – Software 2: Software Development and Design

Professor: Rob LaTour

Project #5: SortingMachine with Heapsort

The Ohio State University

College of Engineering

Columbus, Ohio

```
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 Danny Kan (kan.74@osu.edu)
* @author Jatin Mamtani (mamtani.6@osu.edu)
```

```
 *
 */
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|";

    if (i != j) {
        T k = array[i];
        array[i] = array[j];
        array[j] = k;
    }
}

/**
 * Given an array that represents a complete binary tree and an index
```

```
* referring to the root of a subtree that would be a heap except for its
* root, sifts the root down to turn that whole subtree into a heap.
*
* @param <T>
*          type of array entries
* @param array
*          the complete binary tree
* @param top
*          the index of the root of the "subtree"
* @param last
*          the index of the last entry in the heap
* @param order
*          total preorder for sorting
* @updates array
* @requires <pre>
* 0 <= top  and  last < |array|  and
* for all i: integer
*     where (0 <= i  and  i < |array|)
*   ([entry at position i in array is not null])  and
* [subtree rooted at {@code top} is a complete binary tree]  and
* SUBTREE_IS_HEAP(array, 2 * top + 1, last,
*     [relation computed by order.compare method])  and
* SUBTREE_IS_HEAP(array, 2 * top + 2, last,
*     [relation computed by order.compare method])  and
* IS_TOTAL_PREORDER([relation computed by order.compare method])
* </pre>
* @ensures <pre>
* SUBTREE_IS_HEAP(array, top, last,
*     [relation computed by order.compare method])  and
* perms(array, #array)  and
* SUBTREE_ARRAY_ENTRIES(array, top, last) =
*  SUBTREE_ARRAY_ENTRIES(#array, top, last)  and
```

```
 * [the other entries in array are the same as in #array]
 * </pre>
 */
private static <T> void siftDown(T[] array, int top, int last,
        Comparator<T> order) {
    assert array != null : "Violation of: array is not null";
    assert order != null : "Violation of: order is not null";
    assert 0 <= top : "Violation of: 0 <= top";
    assert last < array.length : "Violation of: last < |array|";
    for (int i = 0; i < array.length; i++) {
        assert array[i] != null : ""
                + "Violation of: all entries in array are not null";
    }
    assert isHeap(array, 2 * top + 1, last, order) : ""
            + "Violation of: SUBTREE_IS_HEAP(array, 2 * top + 1, last,"
            + " [relation computed by order.compare method])";
    assert isHeap(array, 2 * top + 2, last, order) : ""
            + "Violation of: SUBTREE_IS_HEAP(array, 2 * top + 2, last,"
            + " [relation computed by order.compare method])";

    /*
     * Impractical to check last requires clause; no need to check the other
     * requires clause, because it must be true when using the array
     * representation for a complete binary tree.
     */

    int leftChildIndex = 2 * top + 1; // the def. of odd.
    int rightChildIndex = leftChildIndex + 1; // the def. of even.
    if (rightChildIndex <= last) {
        int index = leftChildIndex;
        if (order.compare(array[leftChildIndex],
                array[rightChildIndex]) > 0) {
```

```
            index = rightChildIndex;
        }
        if (order.compare(array[top], array[index]) > 0) {
            exchangeEntries(array, top, index);
            siftDown(array, index, last, order);
        }
    } else if (leftChildIndex <= last
            && order.compare(array[top], array[leftChildIndex]) > 0) {
        exchangeEntries(array, top, leftChildIndex);
        siftDown(array, leftChildIndex, last, order);
    }
}

/**
 * Heapifies the subtree of the given array rooted at the given {@code top}.
 *
 * @param <T>
 *            type of array entries
 * @param array
 *            the complete binary tree
 * @param top
 *            the index of the root of the "subtree" to heapify
 * @param order
 *            the total preorder for sorting
 * @updates array
 * @requires <pre>
 * 0 <= top  and
 * for all i: integer
 *     where (0 <= i  and  i < |array|)
 *   ([entry at position i in array is not null])  and
 * [subtree rooted at {@code top} is a complete binary tree]  and
 * IS_TOTAL_PREORDER([relation computed by order.compare method])
```

```
 * </pre>
 * @ensures <pre>
 * SUBTREE_IS_HEAP(array, top, |array| - 1,
 *     [relation computed by order.compare method])  and
 * perms(array, #array)
 * </pre>
 */
private static <T> void heapify(T[] array, int top, Comparator<T> order) {
    assert array != null : "Violation of: array is not null";
    assert order != null : "Violation of: order is not null";
    assert 0 <= top : "Violation of: 0 <= top";
    for (int i = 0; i < array.length; i++) {
        assert array[i] != null : ""
                + "Violation of: all entries in array are not null";
    }

    /*
     * Impractical to check last requires clause; no need to check the other
     * requires clause, because it must be true when using the array
     * representation for a complete binary tree.
     */

    int leftChildIndex = 2 * top + 1; // the def. of odd.
    int rightChildIndex = leftChildIndex + 1; // the def. of even.
    if (leftChildIndex <= array.length) {
        if (!isHeap(array, leftChildIndex, array.length - 1, order)) {
            heapify(array, leftChildIndex, order);
        }
        if (rightChildIndex <= array.length) {
            if (!isHeap(array, leftChildIndex, array.length - 1, order)) {
                heapify(array, leftChildIndex, order);
            }
```

```
        }
    }
    siftDown(array, top, array.length - 1, order);
}

/**
 * Constructs and returns an array representing a heap with the entries from
 * the given {@code Queue}.
 *
 * @param <T>
 *            type of {@code Queue} and array entries
 * @param q
 *            the {@code Queue} with the entries for the heap
 * @param order
 *            the total preorder for sorting
 * @return the array representation of a heap
 * @clears q
 * @requires IS_TOTAL_PREORDER([relation computed by order.compare method])
 * @ensures <pre>
 * SUBTREE_IS_HEAP(buildHeap, 0, |buildHeap| - 1)  and
 * perms(buildHeap, #q)  and
 * for all i: integer
 *     where (0 <= i  and  i < |buildHeap|)
 *   ([entry at position i in buildHeap is not null])  and
 * </pre>
 */
@SuppressWarnings("unchecked")
private static <T> T[] buildHeap(Queue<T> q, Comparator<T> order) {
    assert q != null : "Violation of: q is not null";
    assert order != null : "Violation of: order is not null";

    /*
```

```
     * Impractical to check the requires clause.
     */

    /*
     * With "new T[...]" in place of "new Object[...]" it does not compile;
     * as shown, it results in a warning about an unchecked cast, though it
     * cannot fail.
     */

    T[] heap = (T[]) (new Object[q.length()]);
    int i = 0;
    while (q.length() > 0) {
        heap[i] = q.dequeue();
        i++;
    }
    heapify(heap, 0, order);
    return heap;
}

/**
 * Checks if the subtree of the given {@code array} rooted at the given
 * {@code top} is a heap.
 *
 * @param <T>
 *            type of array entries
 * @param array
 *            the complete binary tree
 * @param top
 *            the index of the root of the "subtree"
 * @param last
 *            the index of the last entry in the heap
 * @param order
```

```
*            total preorder for sorting
* @return true if the subtree of the given {@code array} rooted at the
*         given {@code top} is a heap; false otherwise
* @requires <pre>
* 0 <= top  and  last < |array|  and
* for all i: integer
*     where (0 <= i  and  i < |array|)
*   ([entry at position i in array is not null])  and
* [subtree rooted at {@code top} is a complete binary tree]
* </pre>
* @ensures <pre>
* isHeap = SUBTREE_IS_HEAP(array, top, last,
*     [relation computed by order.compare method])
* </pre>
*/
private static <T> boolean isHeap(T[] array, int top, int last,
        Comparator<T> order) {
    assert array != null : "Violation of: array is not null";
    assert 0 <= top : "Violation of: 0 <= top";
    assert last < array.length : "Violation of: last < |array|";
    for (int i = 0; i < array.length; i++) {
        assert array[i] != null : ""
                + "Violation of: all entries in array are not null";
    }

    /*
     * No need to check the other requires clause, because it must be true
     * when using the Array representation for a complete binary tree.
     */

    int left = 2 * top + 1;
    boolean isHeap = true;
```

```
    if (left <= last) {
        isHeap = (order.compare(array[top], array[left]) <= 0)
                && isHeap(array, left, last, order);
        int right = left + 1;
        if (isHeap && (right <= last)) {
            isHeap = (order.compare(array[top], array[right]) <= 0)
                    && isHeap(array, right, last, order);
        }
    }
    return isHeap;
}

/**
 * Checks that the part of the convention repeated below holds for the
 * current representation.
 *
 * @return true if the convention holds (or if assertion checking is off);
 *         otherwise reports a violated assertion
 * @convention <pre>
 * 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>
 */
private boolean conventionHolds() {
```

```
    if (this.insertionMode) {
        assert this.heapSize == 0 : ""
                + "Violation of: if $this.insertionMode then $this.heapSize = 0";
    } else {
        assert this.entries.length() == 0 : ""
                + "Violation of: if not $this.insertionMode then $this.entries = <>";
        assert 0 <= this.heapSize : ""
                + "Violation of: if not $this.insertionMode then 0 <= $this.heapSize";
        assert this.heapSize <= this.heap.length : ""
                + "Violation of: if not $this.insertionMode then"
                + " $this.heapSize <= |$this.heap|";
        for (int i = 0; i < this.heap.length; i++) {
            assert this.heap[i] != null : ""
                    + "Violation of: if not $this.insertionMode then"
                    + " all entries in $this.heap are not null";
        }
        assert isHeap(this.heap, 0, this.heapSize - 1,
                this.machineOrder) : ""
                        + "Violation of: if not $this.insertionMode then"
                        + " SUBTREE_IS_HEAP($this.heap, 0, $this.heapSize - 1,"
                        + " [relation computed by $this.machineOrder.compare"
                        + " method])";
    }
    return true;
}

/**
 * Creator of initial representation.
 *
 * @param order
 *            total preorder for sorting
 * @requires IS_TOTAL_PREORDER([relation computed by order.compare method]
```

```
 * @ensures <pre>
 * $this.insertionMode = true  and
 * $this.machineOrder = order  and
 * $this.entries = <>  and
 * $this.heapSize = 0
 * </pre>
 */
private void createNewRep(Comparator<T> order) {
    this.insertionMode = true;
    this.machineOrder = order;
    this.entries = new Queue1L<>();
    this.heapSize = 0;
}

/*
 * 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 {
        return this.getClass().getConstructor(Comparator.class)
                .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();
    assert localSource.conventionHolds();
}

/*
 * Kernel methods ------------------------------------------------------
 */

@Override
public final void add(T x) {
    assert x != null : "Violation of: x is not null";
    assert this.isInInsertionMode() : "Violation of: this.insertion_mode";

    this.entries.enqueue(x);
    assert this.conventionHolds();
}

@Override
public final void changeToExtractionMode() {
    assert this.isInInsertionMode() : "Violation of: this.insertion_mode";

    this.heap = buildHeap(this.entries, this.machineOrder);
    this.heapSize = this.heap.length;
    this.insertionMode = false;
```

```
    assert this.conventionHolds();
}

@Override
public final T removeFirst() {
    assert !this
            .isInInsertionMode() : "Violation of: not this.insertion_mode";
    assert this.size() > 0 : "Violation of: this.contents /= {}";

    T removed = this.heap[0];
    this.heap[0] = this.heap[this.heapSize - 1];
    this.heapSize--;
    siftDown(this.heap, 0, this.heapSize, this.machineOrder);
    assert this.conventionHolds();
    return removed;
}

@Override
public final boolean isInInsertionMode() {
    assert this.conventionHolds();
    return this.insertionMode;
}

@Override
public final Comparator<T> order() {
    assert this.conventionHolds();
    return this.machineOrder;
}

@Override
public final int size() {
    int size = this.heapSize;
```

```
    if (this.insertionMode) {
        size = this.entries.length();
    }
    assert this.conventionHolds();
    return size;
}

@Override
public final Iterator<T> iterator() {
    return new SortingMachine5aIterator();
}

/**
 * Implementation of {@code Iterator} interface for
 * {@code SortingMachine5a}.
 */
private final class SortingMachine5aIterator implements Iterator<T> {

    /**
     * Representation iterator when in insertion mode.
     */
    private Iterator<T> queueIterator;

    /**
     * Representation iterator count when in extraction mode.
     */
    private int arrayCurrentIndex;

    /**
     * No-argument constructor.
     */
    private SortingMachine5aIterator() {
```

```
    if (SortingMachine5a.this.insertionMode) {
        this.queueIterator = SortingMachine5a.this.entries.iterator();
    } else {
        this.arrayCurrentIndex = 0;
    }
    assert SortingMachine5a.this.conventionHolds();
}

@Override
public boolean hasNext() {
    boolean hasNext;
    if (SortingMachine5a.this.insertionMode) {
        hasNext = this.queueIterator.hasNext();
    } else {
        hasNext = this.arrayCurrentIndex < SortingMachine5a.this.heapSize;
    }
    assert SortingMachine5a.this.conventionHolds();
    return hasNext;
}

@Override
public T next() {
    assert this.hasNext() : "Violation of: ~this.unseen /= <>";
    if (!this.hasNext()) {

        /*
         * Exception is supposed to be thrown in this case, but with
         * assertion-checking enabled it cannot happen because of assert
         * above.
         */

        throw new NoSuchElementException();
```

```
            }
            T next;
            if (SortingMachine5a.this.insertionMode) {
                next = this.queueIterator.next();
            } else {
                next = SortingMachine5a.this.heap[this.arrayCurrentIndex];
                this.arrayCurrentIndex++;
            }
            assert SortingMachine5a.this.conventionHolds();
            return next;
        }

        @Override
        public void remove() {
            throw new UnsupportedOperationException(
                    "remove operation not supported");
        }

    }

}
```