CSE12 - Spring 2014 HW #6

Heaps and Heapsort

(200 points)

Due 11:59pm 14 May 2014 (Wednesday!)

In this assignment you create a heap (backed by an array, actually an ArrayList), a JUnit framework to help you develop and validate your heap class, a sorting algorithm that uses the heap (AKA heapsort) and some timing tests the extends the work completed in Homework #5.

This assignment is an individual assignment. You may ask Professors/TAs/Tutors for some guidance and help, but you can't copy code. You can, of course, discuss the assignment with your classmates, but don't look at or copy each others code or written answers.

The following files are provided for you and can be found on the HW6 page:

- HeapSort12.java
- SortTimer2.java
- Heap12.java

In addition you will want to re-download the files from the HW5 page

- Sort12.java
- Insertion12.java
- Merge12.java
- Bubble12.java
- Quick12.java
- random-strings.txt
- random-strings-sorted.txt

You will submit the following file for this assignment:

- **Heap12.java**
- **Heap12Tester.java**
- **HeapSort12.java**
- **HW6.txt**

HW6.txt will be a plain text file (NOT .docx or .pdf or anything else). You should use the ^ symbol to indicate exponentiation (e.g. n^2 should be written n^2) and log(n) where appropriate. At the top of your HW6.txt file, please include the following header:

```
CSE 12 Homework 6
```

Your Name
Your PID
Section [Your section] (A00 for Alvarado, B00 for Papadopoulos)
The date

Part 1 - Implementing Heap12, Heap12Tester and HeapSort12

(175 points)

Read the Javadoc documentation of the supplied Heap12.java (you will need to generate the javadoc itself). Make certain you understand the responsibilities of each public method defined. of each method required by the interface. Sketch a test plan for testing Heap12, which you will be coding completely for part of this assignment Your plan should involve testing all the public methods defined in the Heap12 class, special cases, and the iterator.

Define a class named Heap12Tester that extends junit.framework.TestCase and that implements your test plan. Keep in mind that while you should definitely think about testing first, you do not need to write a complete tester program before starting to define Heap12. In fact, an iterative process can work well: define some tests in Heap12Tester, and implement the functionality in Heap12 that will be tested by those tests, test that functionality with Heap12Tester, write more tests, etc. The end result will (hopefully!) be the same: A good tester, and a good implementation of the class being tested.

You might want to begin defining Heap12Tester by first testing the Java Collections Framework PriorityQueue class. That way you can develop some tests that you know work on a PriorityQueue because it also extends AbstractQueue. Please note that if you take this approach to start, that you only need to be writing tests for methods that are defined both for Heap12 and PriorityQueue. Heap12 has many fewer public methods than PriorityQueue.

Also keep in mind that your Heap12Tester must not make use of any Heap12 methods *not* specified in the Heap12 documentation. This is because we will validate your Heap12Tester on a know good implementation of Heap12.

Heap12.java

Define a generic class `Heap12<E extends Comparable<? super E>>`. Your Heap12 must properly implement all the public methods defined in the javadoc you generate from the Heap12 starter code. In addition, your Heap12 must meet these requirements:

1. Heap12 must extend the AbstractQueue class of the Java Collections Framework (You cannot, for example adapt the already-existing PriorityQueue)

2. Heap12 **must use** a heap-structured array as its backing store. (For simplicity use an ArrayList object wherever your assignment simply says “array”. In what follows, when it says array, we really mean ArrayList and array as interchangeable)
3. Heap12 must define a public default constructor which creates an array of length 5 as its initial backing store.
4. Heap12 must define a copy constructor, which takes another Heap12 as an argument, and then initializes the new Heap12 to be a “deep copy” of the argument Heap12. Here "deep copy" means that the new Heap12 and the argument Heap12 will have different backing store arrays, but the data objects pointed to by elements of those arrays will not be copied. (only the references to the objects are copied)
5. When the offer() method is called with a non-null argument and the current size of the Heap12 is equal to the length of its backing store array, the length of its backing store array must double. (That is, a new array with length twice that of the existing full array must be allocated, elements from the old array copied into it, and the new array then used as the backing store. While ArrayLists will extend automatically, you should explicitly perform a copy) Thus a Heap12 object appears to a client as a priority queue with unlimited capacity.

Some suggestions for Heap12 implementation.

Think about what other methods it might be useful to define in order to implement the required public methods. Since these other methods are not part of the public interface of your Heap12 class, they should have private or protected visibility. For example, methods like:

```
private void trickleDown(int indx)
private void bubbleUp(int indx)
parent(int indx)
```

are just a few that are likely convenient helper methods. You almost certainly will want others. Helper methods are defined to make your code more readable, easier to understand, and reduce “cut-and-paste” code replication. We are particularly interested in reducing code replication.

As you get a little deeper into your implementation, you likely will find it useful to define 'trickledown' and 'bubbleup' heap operations so that they can start from any element in the array not just the first or last. These methods can be defined recursively, or iteratively.

(An additional hint: think about a helper method that is a wrapper around “compareTo” to reduce the amount of code needed to differentiate between a min-heap and a max-heap -- A good approach is to *first write and test a min-heap implementation*. Then think about how comparisons would need to change to turn this code into a max-heap. This is where a simple wrapper around compareTo can be useful If you approach this correctly, you should only need slightly recode existing comparisons in your bubbleUp() and trickleDown() methods to use a comparison helper method

A side note: You could internally build a Comparator object, but that is more complicated than necessary to solve the problem at hand. If you find yourself doing a lot of cut-and-paste code replication to code a max-heap, you need to think harder about the problem).

Heap12Tester.java

We are NOT supplying you with any starting code for Heap12Tester.java, but you are required to supply this JUnit-based tester as part of your solution. We will run your Heap12Tester against a known, good, implementation. Your Tester should help you with the implementation of the Heap12 class, you certainly will want to test all the public methods, your iterator, and places where exceptions should be thrown. We would expect that *at least* 10 meaningful tests would indicate a baseline of reasonable coverage. (You probably need a few more tests for so-called edge conditions). In the iterative approach to testing/development, when you find a bug in your implementation that your tests did not uncover, write another test that would find that bug.

Some ideas on testing

- Don't forget that your code should support both min-heaps and max-heaps, you will eventually want test fixtures for both min-heaps and max-heaps (see note above about first developing a min-heap only implementation and then extending that to support both min-heaps and max-heaps)
- Don't overlook the remove() method of the Iterator interface. This supports the "strange" operation of removing an element that is possibly in the middle of the heap.
- While it's convenient to test with elements offered to the heap in sorted order, try some test cases where the data isn't presorted, and then verify that you retrieve the proper min (or max) value.
- It's helpful to define your iterator to simply return elements in index order, you can print these out in a loop and verify visually that your internal array IS heap-ordered

HeapSort12.java

Using your Homework #5 as a guide. Implement HeapSort12.java using your Heap12 class. This should be fewer than 20 additional lines of code and must implement the Sort12 interface. If your HeapSort is long and complex, you are not using your Heap12 class appropriately You can test your HeapSort12.java for both correctness and performance in the same manner as you did in Homework #5 (see the next section)

Testing Heap Sort

The supplied timing code SortTimer2.java is a modification of SortTimer from HW5. It now includes the option to call HeapSort12 as algorithm 4. The modification that enables you to sort a text file and print the results with the -p option is still present. The -p option will print for each iteration, so you likely want to use for a single iteration, and a single timing repetition.

For example, sort the first 100 words of the file random-strings.txt using heap sort and print the resulting sorted list,, you would type in

```
% java SortTimer2 -p random-strings.txt 4 100 1 1 1
```

This will use sort algorithm “4”, sort the first 100 strings, extend each test by 1, do just 1 test, and 1 rep/iteration. For SortTimer2, algorithm 4 is heap sort. You can use this to verify that your heap sort is sorting properly. By redirecting the output to a file and verifying that the result is properly sorted (We WILL test that your implemented heap sort actually sorts correctly).

Part 2: Running Time of Heap Sort (25 points)

In this section, you will test actual efficiency of your Heap sort algorithm and attempt to empirically verify that its computational complexity is “reasonable” given what you have learned about the actual algorithms. You should download/copy the Sort12, Merge12, Insertion12, Quick12, Bubble12 from the HW5 files. *These are unchanged from the previous assignment*

To help you, SortTimer2 has a usage statement if you invoke with no arguments

$ java SortTimer

```
Usage: java  SortTimer2 [-p] <document> <sortAlg> [start] [increment]
[steps] [reps]
      -p      - print the sorted list
      document  - text file to sort
      Algorithm - 0:bubble, 1:insertion: 2:merge, 3:quick, 4:heap
      start     - number of words read in from  document
      increment - number of words to add for each iteration
      steps         - number of iterations
      reps      - number of times to repeat for timing
```

There are five (5) sort algorithms: Bubble, a modified insertion sort, merge sort,quicksort, and your heapsort.

You have already run the above to find good the defaults for each of the sort algorithms. Recall that you can get a feeling for how quickly they run using random-strings.txt as the document, e.g.,

```
$ java SortTimer random-strings.txt 1
```

will perform a test with the modified insertion sort and would have output similar to

```
$ java SortTimer random-strings.txt 1
Document: random-strings.txt
 sortAlg: 1
=======================================
```

```
1:    5000 words in       5 milliseconds
2:   10000 words in      11 milliseconds
3:   15000 words in      21 milliseconds
4:   20000 words in      35 milliseconds
5:   25000 words in      53 milliseconds
```

Note that you are still working with a *modified* insertion sort in this homework (same as HW #5), But for this section, *do not be alarmed* that it will not necessarily have the n^2 behavior of standard insertion sort that we discussed in class (that's why it has been modified!)

Place the answers to the following in our HW5.txt file:

In what follows please answer the following questions

I. Answer following questions for heap sort

What testing parameters did you select so that you could gain insight as to how this algorithm performs as N increases. Explain briefly why you made your choices (one or two sentences). Feel free to use the numbers/explanations you used for insertion sort from your last assignment, or simply run it again.

A. Copy the output of your actual test run for the algorithm
B. Given the numbers you selected, What is the apparent complexity of each sorting algorithm. Which data points did you use to come to your conclusion?

(please section each of your answers as)

I. Heap
    A. answer to part A
    B. answer to part B
II. Insertion (You can reuse your answers from HW5, if they are correct. If they are not correct, fix them here).
    A. answer to part A
    B. answer to part B

II. Repeat the above questions, but use the random-strings-sorted.txt as the document.
Use the same format as in part I for your answers

III. Compare the behaviour of modified insertion sort to heap sort in the pre-sorted case, fundamentally, why are they different?

Turning in your assignment

For this assignment you will turn in your **HW6.txt** and **Heap12.java, Heap12Tester.java, HeapSort12.java** files using the command **bundleHW6**.