CSE373 Spring 2015: Homework 3 - Priority Queues

Outline

- Due Dates, Turn-In, and Rules
- Provided Code
- Programming
- Write-Up Questions
- Above and Beyond
- What to Turn In

Due Dates, Turn-In, and Rules

Due 11:00PM Wednesday April 29, 2015

Read What to Turn In before you submit — poor submissions may lose points.

Turn in your assignment here

- Complete this project by yourself (i.e., without a partner). You may discuss the assignment with others in the class, but your solution must be entirely your own work.
- Remember Collaboration Policy, Grading Policy, and Programming Guidelines before you begin working on the project. In particular, note that the write-up you turn in is worth a substantial portion of the grade!

Provided Code

Download these files into a new directory:

- `PriorityQueue.java`: An interface that corresponds to a Priority Queue ADT for doubles. You will provide 3 different implementations of this interface.
- `EmptyPQException.java`: A class that defines a type of exception you can throw. You should not need to import anything to use `EmptyPQException`. Just place `EmptyPQException.java` in the same folder as your other Java files.

Programming

Implement three priority queues. All three implementations should implement the provided `PriorityQueue` interface (include `implements PriorityQueue` in your Java code), which means they should work with priorities that have type `double` and there are no corresponding items attached to the priorities. Your implementations should be as follows:

- A class `BinaryHeap` that implements a binary min-heap like we discussed in lecture, using an array to store the conceptual complete tree.
- A class `ThreeHeap` that implements a min-heap like `BinaryHeap` does except each tree node has 3 children (except for leaves and possibly one other node). You should still use a contiguous portion of an array to store the conceptual complete tree. We suggest you make a copy of your `BinaryHeap` class and make changes as necessary.
- A class `MyPQ` that implements a priority queue in another way of your choice. Feel free to get creative. The simplest possible implementation might just use an array (sorted or unsorted), a linked list (sorted or unsorted), or a tree. Whatever you do, be sure it implements the `PriorityQueue` interface provided, and does not use parts of the Java collections framework.

Put your three implementations in *three separate Java files*, `BinaryHeap.java`, `ThreeHeap.java`, and `MyPQ.java`.

Your priority queues should allow duplicates. That is, two or more copies of the same value should be allowed to exist in the heap at the same time. For example, if you call `deleteMin` and you have {3.0, 3.0, 6.0, 7.0} in the heap, it would just return one of the 3.0 values, then on the next `deleteMin` it would return the other 3.0. It does not matter "which" 3.0 is returned first. According to our definition of priority queue, what must be guaranteed is that both 3.0 values will be returned before a 6.0 or 7.0 is returned, and that the 6.0 would be returned before the 7.0.

Your implementations should automatically grow as necessary. (If interested, you may also have them shrink when appropriate; this is optional.) For any arrays, you should start with a small array (say, 10 elements) and resize to use an array twice as large whenever the array becomes full, copying over the elements in the smaller array. Do the copying with a `for` loop rather than any Java library methods (even though using the library is how one would normally do it). You may use the `length` field of an array as needed.

Be sure to test your solutions thoroughly and to turn in your testing code. Part of the grading will involve thorough testing including any difficult cases. For this assignment, we will be grading more strictly for things like style and efficiency than we did on Homework 1. However, your your `MyPQ` implementation does not need to be more efficient than a good array or linked-list implementation if that is your approach.

Write-Up Questions

The questions include comparing the actual run-time of your implementations. We would expect the reports to be at least a couple of pages long, quite possibly longer to have room for relevant graphs or tables.

Submit a `README.pdf` file, answering the questions in this template README file: ([README.docx]() or [README.tex]())

1. What is the worst case asymptotic running time of `isEmpty`, `size`, `insert`, `findMin`, and `deleteMin` operations on all three of your heap implementations? *For this analysis you should ignore the cost of growing the array. That is, assume that you have enough space when you are inserting a value.*
2. Timing your code: Perform several timing experiments (similar to what you did in Homework 2, where you timed pieces of code), to examine the running time of all three of your heap implementations. An

experiment will include running the same client code (that uses a Priority Queue) for your three different heap implementations *for at least four different values of N* and timing this. It is up to you to write and to determine what this client code should be. Just be sure that it exercises your `insert` and `deleteMin` operations in a reasonable manner, including eventually deleting everything that has been inserted into the heap. You are not required to explicitly measure calls to `findMin`, `size`, and `isEmpty` but feel free to do so if interested. Similar to Homework 2, graphing your results is recommended, but a table of results will work also. Please note that similar to Homework 2, you are required to turn in the code you used to do your timing experiments.

3. Compare what you see in your experiments, to what you expected to see based on a big-O analysis. (This is also similar to what you did in Homework 2.) In your discussion, answer these questions:
   a. How useful was the asymptotic analysis for predicting the measured run time of `insert` and `deleteMin` for your three implementations?
   b. If your predictions differed substantially from your measured times, gives reasons why this might have occurred.
   c. Which of your three implementations would you recommend to someone who needs to use a heap? Why is that one preferred? Are there any conditions under which you might suggest using your other implementations?
4. Briefly discuss how you went about testing your three heap implementations. Feel free to refer to your testing files, which you should submit.
5. You implemented a binary-heap and a three-heap. We could have also asked you to implement a four-heap, a five-heap, etc.
   a. In a short table, indicate for a binary heap, a three-heap, a four-heap and a five-heap, where the children for the node at array index `i` are. For example, the first row of the table would indicate that for a binary heap, the two children would be at `i*2` and `i*2+1`.
   b. For a *d*-heap where *d* is a variable representing the number of children (like two, three, four, five, ...), give an arithmetic formula for calculating where the left-most child for the node at array index `i` are. For example, a *wrong* answer in the right format would be *i*d+14*. Naturally, your formula should produce the right answer for all the rows in your table from part (a).

Above and Beyond

Note: If you use one of the fancier priority queue implementations below as your third implementation, that is fine, but be sure to test it carefully since it will be part of your main grade. Alternately, you may submit them in addition to your primary three implementations.

1. Implement a *d*-heap where *d* is the number of children for non-leaf nodes. Your class should implement the same priority queue interface and it should use a contiguous array portion as in your first two implementations. It should include an empty constructor **and** additional constructor that takes *d* as an argument, work correctly for any *d* greater than or equal to 2, and use *d* as the number of children for nodes.
2. Implement a binary heap that works for any type (not just doubles). It should use Java generic types to allow any priority type that implements an appropriate interface for comparing two priorities *and* your heap should allow items of a second generic type that are "attached" to each priority. Note this implementation will not implement the provided interface, so provide any additional comments necessary to explain how your class should be used.

What to Turn In

Turn-in: You should turn in the following files electronically, named as follows:

- `BinaryHeap.java`
- `ThreeHeap.java`
- `MyPQ.java`
- Any additional Java files needed, if any, for your three priority-queue implementations.
- The Java files you used to ***test*** your three implementations.
- The Java files you used to ***time*** your three implementations.
- `README.pdf`, containing answers to the Write-Up Questions.
- Any additonal files for the extra credit **in a zip file named** `extracredit.zip`. Please make sure that this zip file decompresses its contents into a folder called `extracredit` and *not* into a bunch of individual files.

Do **not** turn in `PriorityQueue.java` and `EmptyPQException.java`. You must not change these files. Your implementations must work with the code as provided to you.
