CS21120 Assignment 1

Double ended priority queue

Release date 2nd November 2015

Hand in date 23rd November 2015 (23:59 via Blackboard)

Feedback date 14th December 2016 (via Blackboard)

Aims and Objectives

The aim of this assignment is to give you experience in implementing an abstract data structure.

The objective is to implement a double ended priority queue.

Overview

This assignment asks you to implement a double ended priority queue (DEPQ)[1]. A double ended priority queue allows items to be added to the queue and for either the highest or lowest "priority" items to be removed. There are a number of ways this could be done[2] e.g.:

- using a binary search tree (with references to the min and max elements),
- using an interval heap,
- using a min-max heap[3],
- various versions of 2 linked heaps, or
- simpler but less efficient methods such as an ordered or unordered array.

In this assignment you are free to choose an implementation and you are awarded marks for correct functioning, the efficiency of your solution, documentation and analysis.

Requirements

You are asked to implement a DEPQ that satisfies the supplied DEPQ interface. Your solution should be submitted as a **single** *.java* file with the name *<user-id>DEPQ.java*, where *<user-id>* is replaced with your Aberystwyth user ID (so for me it would be *BptDEPQ.java*). This file should contain your main implementation class (*<user-id>DEPQ*) and any required supporting classes, which should be implemented as *inner* classes (i.e. contained within the scope of your main class). Your class should also include a no arguments constructor for testing purposes. The marking is organised as follows:

1) *Correct functioning of each method* (25%): The supplied interface should be implemented correctly, unit testing will be applied to your code to check that it is functioning correctly. The queue should be able to hold at least 1000 items.
2) *Overall efficiency* (25%): A good solution will typically have $O(\log N)$ complexity for the *add*, *getMost* and *getLeast* methods, and constant time complexity for other methods. Simpler

[1] https://en.wikipedia.org/wiki/Double-ended_priority_queue
[2] http://www.cise.ufl.edu/~sahni/dsaaj/enrich/c13/double.htm
[3] https://en.wikipedia.org/wiki/Min-max_heap

solutions such as an unordered array might have a lower cost for *add* but will pay the penalty for this with a higher cost for the *getMost* and *getLeast*.

3) *Documentation based report* (25%): Properly formatted and informative javadoc comments for the class and every method (including parameters and return type). The documentation will serve as your report, so you should explain what you have done in your implementation and why. You should reference any sources you have used and describe the algorithm implemented.
4) *Complexity analysis* (20%): As part of your documentation based report (above) you should include your own analysis of the efficiency of your solution. Simple solutions with a simple efficiency analysis will receive fewer marks than more complex methods that require more difficult analysis.
5) *Self evaluation* (5%): assess your own work on each of the listed criteria, giving an estimated mark for each and an estimated overall mark.

Resources

For this assignment you are **not** permitted to use classes from the java collections classes (java.util.* package). You are provided with the DEPQ interface, which you must implement and must not modify (including the package statement, so it needs to be placed in the right folder). For the rest of the code you must implement your own classes and methods. You should not use Threads in your code and you do not need to make your code thread safe. Any resources you use (including those provided in the module) must be acknowledged in your documentation based report. You are also provided with a basic set of unit tests to help guide your code development.

Academic conduct

As with all such assignments, the work must be your own. Do not share code with your colleagues and ensure that your own code is securely stored and not accessible by your classmates. Any sources you use should be properly credited with a citation, and any help you get (e.g. from demonstrators) should be acknowledged. Your report must accurately reflect what you have achieved with your code, any discrepancies between your code and report could be treated as academic fraud.

Marking grid

(Note a more detailed marking grid will be available on TurnItIn).

Aspect	70-100%	60-70%	50-60%	40-50%	0-40%
Correct functioning	All interface methods implemented and behave in the expected manner.	Most methods implemented and function in the expected manner, but there may be some bugs in more complex code.	Most methods function as expected, but may have bugs in simpler or more complex code.	Code compiles and some methods function as expected, but may have some serious bugs.	Code largely non-functional, may not compile or have serious flaws.
Efficiency	All methods operate in the most efficient way possible for overall performance i.e. *add*, *getMost* and *getLeast* O(logN), *inspectLeast*, *inspectMost*, *size* and *clear* O(1).	Most methods operate efficiently, but some compromises made e.g. *getLeast* O(logN) but *getMost* O(N).	Some methods operate efficiently, or a serious attempt to write efficient code that has some flaws.	Code functional, but at the lowest level of efficiency e.g. O(N) for *getMost* and *getLeast*.	Code largely non-functional, so efficiency impossible to judge.
Documentation based report	Clear and detailed description of algorithm, javadoc correctly formatted and complete, good use of English, good use of citations. If the algorithm is very simple it will be hard to achieve this grade.	Mostly clear and detailed description of algorithm, javadoc correctly formatted and complete, mostly good use of English, good use of citations. If the algorithm is very simple it will be hard to achieve this grade.	Description of algorithm that may be lacking in detail (or could be a very simple algorithm), javadoc mostly correctly formatted and complete, reasonable use of English, some citations.	Some documentation, but may have gaps or lack detail of the algorithm or implementation. Some poor use of English and citations may be limited or missing.	Documentation largely missing, incorrectly formatted or missing javadoc, poor use of English, citations missing.
Complexity analysis	Correct analysis of complexity, which matches the algorithm described and implemented. For simple algorithms (with simple complexity analysis) it will be hard to achieve this grade.	Mostly correct analysis of complexity, either some minor parts of a complex analysis may be incorrect, or some significant parts of the algorithm may have a very simple analysis.	Correct complexity analysis of a simple method or a mostly correct complexity analysis of a more complex algorithm.	Correct complexity analysis of a very simple method, which may have some mistakes.	Complexity analysis missing or mostly incorrect.
Self evaluation	Self evaluation complete with an estimated mark and comment for each part and overall.	Self evaluation mostly complete with some comments missing.	Self evaluation with some comments or marks missing.	Self evaluation with marks but no comments.	Self evaluation missing.

Bernie Tiddeman, 21st October 2015