CSE12 - Spring 2014 HW #4

Deques, Stacks, Queues and Search

(200 points)

Due 11:59pm 29 April 2014

In this assignment you will implement data structures that provide an implementation for three abstract data types: A double-ended queue using a circular array, a stack and a queue. In addition, you'll use these data structures in a method that implements depth first search in support of the game of Minesweeper. Also, along the way you will get more experience with implementing Java interfaces, writing JUnit test cases, and using the Adapter design pattern.

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, including discussing testing strategies, concepts, implementation strategies, and bugs you ran into and how you fixed them. But don't look at or copy each others code or written answers.

START EARLY and remember the following rules for tutor hours:

- You must not put your name in the tutor queue more than once. If tutors find your name in the queue more than once, ALL instances of your name will be removed.
- You must indent your code correctly in order to get help from a tutor. If you code is not properly indented when the tutor comes to help you, you'll lose your spot and have to go to the end of the queue.

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

- BoundedDeque.java
- BoundedStack.java
- BoundedQueue.java
- MineSweeperGUI.java
- Stack12Tester.java
- Queue12Tester.java
- Javadocs for BoundedDeque, BoundedStack, and BoundedQueue

You will submit the following files for this assignment:

- **HW4.pdf:** This is a PDF file where you will place answers to questions throughout the assignment.

- **Deque12.java:** A Java class that implements the BoundedDeque interface using a circular array.
- **Stack12.java:** A Java class that implements the BoundedStack interface
- **Queue12.java:** A Java class that implements the BoundedQueue interface
- **BoundedDequeTester.java:** A JUnit tester for the Deque12 class.
- **MineSweeperGUI.java:** A Java class that implements a simple game of Minesweeper. This code is mostly provided for you. You just need to make a few additions.

Logistics:
In EACH AND EVERY FILE that you turn in, we need the following in comments at the top of each file.

NAME: <your name>
ID: <your student ID>
LOGIN: <your class login>

Part 1: Deque12 and BoundedDequeTester [120 points]

BoundedDequeTester.java
Read the documentation in the source code file for the BoundedDeque<E> interface, or view the javadoc page for BoundedDeque which is included in the HW4 resources. Understand the responsibilities of each method required by the interface. Sketch a test plan for a class that implements this interface. Define a class named BoundedDequeTester that extends junit.framework.TestCase and that implements your test plan, using an Deque12 object as a test fixture.

The class you will be testing is Deque12, which implements BoundedDeque, and which you will be defining as part of this assignment. You should have at least one test for every method, but it is better form to create multiple tests for some methods depending on the condition you are testing. For example, you'll probably want a separate test for adding to a full Deque from the test for adding to a non-full Deque.

As a practical matter, you do not need to completely write your BoundedDequeTester program before starting to define Deque12. In fact, an iterative test-driven development process can work well: write some tests in BoundedDequeTester, and implement the functionality in Deque12 that will be tested by those tests, test that functionality with BoundedDequeTester, write more tests, etc. The end result will (hopefully!) be the same: A good tester, and a good implementation of the class being tested.

As for previous (and future) assignments, make sure that your BoundedDequeTester does not depend on anything *not* specified by the documentation of the BoundedDeque interface and the Deque12 class. For example, do not make use of any other constructors or instance or static methods or variables that the ones required in the documentation.

Deque12.java

Define a generic class Deque12<E> that implements the BoundedDeque<E> interface. Besides the requirements for the methods and constructor documented in that interface, for this assignment there is the additional requirement that *this implementation must use a circular array to hold its elements*. (Note also that, as for other assignments, your Deque12 should not define any public methods or constructors other than those in the interface specification.)

A circular array can be rather tricky to implement correctly! It will be worth your time to sketch it out on paper before you start writing code.

You will of course need to create an array with length equal to the capacity of the Deque12 object. Because raw arrays don't play nicely with generics, you should use an ArrayList object for this purpose. However there are a couple of subtleties to look out for when using an ArrayList instead of an array:

- Be careful with the difference between the ArrayList's add method and its set method. You'll almost certainly want to use set, not add. But this means that you must add capacity items to your ArrayList before you do anything else (i.e. in the constructor), or set will throw an IndexOutOfBounds error.
- Remember that no matter what size you initialize your ArrayList, it can always be increased, so be careful not to accidentally grow your ArrayList beyond your BoundedDeque's capacity. If you only use `set` and not `add`, you should not run into this problem.

You will want to have instance variables for the size of the Deque12 (how many data elements it is currently holding), and to indicate which elements of the array are the current front and back elements. Think about what the values of these instance variables should be if the Deque12 is empty, or has one element, or is full (has size equal to its capacity). And in each case, think carefully about what needs to change for an element to be added or removed from the front or the back. Different solutions are possible, as long as all the design decisions you make are consistent with each other, and with the requirements of the interface you are implementing.

Part 2: Stack12 and Queue12 [40 points]

Now that you have your Deque built and fully tested, you will use it to implement a Stack and a Queue.

Stack12.java

Read and understand the interface specification in BoundedStack.java (or view the javadoc page for BoundedStack which is included in the HW4 resources) and then define a generic class Stack12<E> that implements the BoundedStack<E> interface. Once Deque12 is implemented and tested and debugged, defining Stack12 is quite easy. We've provided you with some unit tests that you can use to test your implementation. Note that the methods required by the

BoundedStack interface are different from, though closely related to, the BoundedDeque interface methods. Use the Adapter pattern! That is, in your Stack12 class, create an instance variable of type BoundedDeque (instantiated to a Deque12) and use it to perform all of the necessary BoundedStack methods. If this class is complicated, you're over-thinking it.

Queue12.java

Read and understand the interface specification in BoundedQueue.java (or view the javadoc page for BoundedQueue which is included in the HW4 resources) and then define a generic class Queue12<E> that implements the BoundedQueue<E> interface. As with Stack12, if Deque12 is already implemented and tested and debugged, defining Queue12 is quite easy. Again note that the methods required by the BoundedQueue interface are different from, though closely related to, the BoundedDeque interface methods, and so the Adapter pattern is applicable here as well.

As is often the case when the Adapter pattern is used, if the adapted class (Deque12 in this case) is tested and debugged, the adapting class shouldn't need much testing, because almost all of the work is being handled by delegation to the adapted class's methods. We provide you with a few simple tests which should be sufficient.

Part 3: Minesweeper, Depth First Search and Breadth First Search [40 points]

The last part of this assignment is to complete the implementation of a program that implements the popular game of Minesweeper (http://minesweeperonline.com/). If you've never played Minesweeper take a few minutes to familiarize yourself with the game by playing a little bit. But just a little bit… try not to get too distracted!

We've provided to you a very basic implementation of Minesweeper that is almost complete. However, it's missing the auto-expose functionality. When you click on a cell with 0 mined neighbors, the game is supposed to automatically expose all of the surrounding neighbors of that cell, and then if any of those have no neighbors, it auto-exposes all of its neighbors, etc, until all of the connected cells with no mined neighbors and all of their connected neighbors are exposed.

Implementing this functionality is done by searching outward from the first exposed cell, expanding the search until all connected neighbors that should be exposed have been explored. As we saw/will see in class on Thursday/Friday, this can be done using depth first search or breadth first search, and the only difference between the two algorithms is whether you use a stack or a queue as your data store during the search.

We have provided an implementation of BFS for you. Once you have your Queue12 class working, you simply need to uncomment the code in the appropriate method. You will add an

implementation of DFS in the method exposeCellsDFS. If you understand BFS and DFS and their relationship this will be trivial. You will then study the difference between the behavior of the two algorithms.

Part A

The first step is to understand the provided code. In your HW4.pdf file, in a section labeled Part 3a, answer the following questions:

1. Which method is called when the user clicks on one of the cells in the grid?
2. What are the two central classes used to implement Minesweeper? Roughly, what does each do?
3. What class implements the necessary MouseListener methods that are attached to each of the cells in the board?
4. What does the exposeSlowly method do? How do you control the animation speed?
5. What is the purpose of the actionPerformed method implemented in the MineSweeperGUI class?

These questions are a minimum starting point. If you don't understand what's happening in the code, talk about it with your classmates (this is allowed) or go see a tutor, TA or professor.

Part B

Now, uncomment the code in exposeCellsDFS and implement exposeCellsBFS in MineSweeperGUI. Notice that each method returns a BoundedQueue of the cells that are to be exposed, in the order in which to expose them according to the algorithm. It *should not* expose any cells itself. That is handled by the code that performs the animation. These methods simply perform a depth first search or a breadth first search starting from the initial location and make a list of the cells that need to be exposed.

Part C

Finally, explore the difference between the two algorithms. Play the game with each algorithm, and notice how the animation changes. In your HW4.pdf file in a section labeled Part 3c describe the difference you see between how the cells are exposed when you use depth first search vs. breadth first search.

Just for fun!

Our version of Minesweeper is very basic, lacking creativity and many of the bells and whistles of the original game or improvements that you could imagine. If you're interested, improve on the game. This is a boundlessly open ended extension, so have fun!

Turn in

Submit your files using the command **bundleHW4.**