Implementing a Deque

Summary: In this assignment, you will implement a deque ADT from an interface, while meeting performance constraints by using a list data structure.

1 Background

In this assignment, we will be implementing a Deque ADT (see Requirements). Your job is to implement the interface that is provided for this ADT, given a performance requirement. **All operations (except toString) must be O(1)**.

In previous courses, the primary technique we used to store collections of information was arrays. Recently, we have been introduced to linked lists. Lists are a way to structure data that lets us quickly add and remove elements. Unlike arrays - where we have to create an entirely new array and copy the contents of the existing array into it. Of course, we need a way to determine which of these data structures we should use for a particular problem. It is not always the case that we should choose to use a list, as an example: they are slower than arrays when it comes to index-based access. There are always trade-offs. Considering the task of implementing a deque (aka a double-ended queue), we might suspect that a list will be appropriate. The reason for that is that we need to support enqueue and dequeue in constant time. Arrays won't do that for us - they have a fixed size and require all the elements to be copied to a new array when it needs to be resized. On the flip side, we also know that lists are slower for index-based access. Fortunately, Deques won't require index-based access. The user can only add/remove elements from the head or the tail. They cannot access arbitrary nodes. Thus, the drawback of lists won't impact us. For this assignment, you should plan to use a doubly linked list.

This document is separated into four sections: Background, Requirements, Testing, and Submission. You have almost finished reading the Background section already. In Requirements, we will discuss what is expected of you in this homework. In Testing, we give a quick overview of the test code that is already provided. Lastly, Submission discusses how your source code should be submitted on Canvas/Gradescope.

2 Requirements [32 points]

For this assignment, you will be writing a Deque ADT (i.e., CompletedDeque). A deque is closely related to a queue - the name deque stands for "double-ended queue." The difference between the two is that with a deque, you can insert, remove, or preview, from either end of the structure. Attached to this assignment are the two source files to get you started:

- Deque.java - the Deque interface; the specification you must implement.
- CompletedDeque.java - The driver for Deque testing; includes some simple testing code. Despite the name, it's not complete yet! If you look inside, you will see a TODO to implement the interface. Right now, this file won't compile, since the methods required by the interface haven't even been outlined (which might be your first task, then leading you to fully implement each interface method).

Your first step should be to review these files and satisfy yourself that you understand the specification for a Deque. Your goal is to implement this interface. Implementing this ADT will involve creating 9 methods:

- public CompletedDeque() - a constructor [1 points]
- public void enqueueFront(Item element) - see interface [5 points]
- public void enqueueBack(Item element) - see interface [5 points]

- public Item dequeueFront() - see interface [5 points]
- public Item dequeueBack() - see interface [5 points]
- public Item first() - see interface [1 points]
- public Item last() - see interface [1 points]
- public boolean isEmpty() - see interface [1 points]
- public int size() - see interface [1 points]
- public String toString() - see interface [3 points]
- There are no required contents for main(), and it will not be used during grading.

From Section 1, be sure that you:

- Make all operations (except toString) work in O(1). Hint: Use a doubly linked list data structure. You may need to create a node class - you can use Java's inner class feature to put it inside of your main file or create a separate file/class called DoubleLinearNode.

In addition to per method evaluation, other aspects of the class will be evaluated:

- Additional Tests: combinations of enqueue and dequeue. [2 points]
- Additional Tests: proper generics support. [2 points]

2.1 Packages

Do not import any packages other than java.util.NoSuchElementException. (Do not use any star imports.)

3 Testing

A few simple test operations have already been provided. These are very simple tests - passing them is *necessary but not sufficient* for knowing that your program works properly. (Note that these tests are separate from those performed by Gradescope for grading.) It will be up to you to perform additional testing. The sample output from these tests is:

```
size: 3
contents:
8 9 4
4
8
9
1
11
size: 2
contents:
1 11
```

When you set about writing your tests, try to focus on testing the methods in terms of the integrity of the overall linked list. If you compare the tests that you are given with the parts of your program that they use (the "code coverage"), you'll see that these tests only use a fraction of the conditionals that occur in your program. Consider writing tests that use the specific code paths that seem likely to hide a bug. You should also consider testing things that are not readily apparent from the interface specification. For example, write a test that prints your list starting at the head, another that starts at the end, and see if they give the same result.

Required Files	Optional Files
CompletedDeque.java	DoubleLinearNode.java

Table 1: Submission ZIP file contents.

4 Submission

The submission for this assignment has only one part: a source code submission.

Writeup: For this assignment, no write up is required.

Source Code: The source file must be named as "CompletedDeque.java", and then added to a ZIP file (which can be called anything). Be sure that you are directly ZIPing the file, rather than ZIPPing a folder that contains it. The class must be in the "edu.ser222.m01_03" package, as already done in the provided base file (do not change it!). You will submit the ZIP on Gradescope. Optionally, you may include a file called "DoubleLinearNode.java" in your submission.

4.1 Gradescope

This assignment will be graded using the Gradescope platform. Gradescope enables cloud-based assessment of your programming assignments. Our implementation of Gradescope works by downloading your assignment to a virtual machine in the cloud, running a suite of test cases, and then computing a tentative grade for your assignment. A few key points:

- **Grades computed after uploading a submission to Gradescope are not final.** We have final say over the grade, and may adjust it upwards or downwards. (That said, for this assignment, we don't expect to make many changes.)
- **Additional information on the test cases used for grading is not available, all the information you need (plus some commonsense and attention to detail) is provided in the assignment specification.** Note that each test case will show a small hint in its title about what it is testing that can help you to target what needs to be investigated.

If you have a hard time passing a test case in Gradescope, you should consider if you have made any additional assumptions during development that were not listed in the assignment, and then try to make your submission more general to remove those assumptions.

Protip: the Gradescope tests should be seen as a way to get immediate feedback on your program. This helps you both to make sure you are meeting the assignment's requirements, and to check that you are applying your knowledge correctly. Food for thought: if you start on the assignment early, check against our suite often, and use its feedback, there's no reason why you can't both get full credit and know that you'll get full credit even before the deadline.