PA 2
Due Date: Friday, 16 February 2024 by 11:59 PM

General Guidelines.

The instructions given below describe the required methods in each class. You may need to write additional methods or classes that will not be directly tested but may be necessary to complete the assignment.

In general, you are not allowed to import or use any additional classes in your code without explicit permission from your instructor!

Note: It is okay to use the Math class if you want.

Project Overview

In this project, you must implement an array-based double-ended queue (a.k.a. a Deque). This should be a dynamically resizable structure, and you will do some experimentation comparing two different resizing strategies to see which one is better.

Part 0. Before you start coding.

Before you start coding, it is a good idea to make sure you have a current account on lectura and you know your password. You should also go through the process of transferring files to lectura to make sure you know how to do it. The following are resources that may help with this process.

- mac to lectura tutorial
- lectura instructions for pc
- CS Help Desk (in case you need help with account information)
- TAs & instructors (We can walk you through the process in office hours or SI.)

Part 1. Deque.java

Recall that a Stack is a LIFO data structure which allows easy access to the “top” element, which means you add and remove items at the same end. A Queue, on the other hand, is a FIFO data structure that means that you add to one end and remove from the other. The idea of a Deque, or double-ended queue, is to combine these two structures together so that you can easily add or remove from either end. It turns out that adding and removing from the ends of a list is usually much easier (and less expensive) than adding or removing from the middle of the list, and if you implement it correctly, this is true even if the list is built with an array.

For this part of the assignment, you must implement a *generic, efficient, array-based Deque.* The API with the required methods is given below. Please pay close attention to the method signatures, and keep in mind that like most APIs, this assumes you know that public-facing methods must be declared as public. Please pay special attention to the required runtimes. Also, note that you must use the provided wrapper class to back your Deque so that we can count your accesses to assess efficiency, and so that you can use the access count to compare the resizing strategies.

Resizing

When the array is full, instead of throwing an exception, resize the array to be twice as big. This means creating a new Array and copying all the elements over in the appropriate order. In order to save space, you should also resize the array to be smaller when it becomes less than ¼ full, but do not resize it to be less than the original capacity.

Required Methods for Deque.java

Method Signature	Description
`Deque<T>()`	`constructor: default size of Deque should be 16`
`Deque<T>(int cap)`	`constructor: starting Deque size is <cap>`
`void addLast(T item)`	`adds a new element to the end of the Deque//should be O(1) except when resizing is necessary`
`void addFirst(T item)`	`adds a new element to the front of the Deque//should be O(1) except when resizing is necessary`
`boolean isEmpty()`	`return true if the Deque is empty and false otherwise//should be O(1)`
`T removeFirst() throws EmptyDequeException`	`remove and return the first element in the Deque; throw the exception if the list is empty//should be O(1) unless resizing is necessary`

`T removeLast() throws EmptyDequeException`	`remove and return the last element in the Deque; throw the exception if the list is empty//should be O(1) unless resizing is necessary`
`T peekFirst() throws EmptyDequeException`	`return but do not remove the first element in the Deque; throw the exception if the list is empty//should be O(1) unless resizing is necessary`
`T peekLast() throws EmptyDequeException`	`return but do not remove the last element in the Deque; throw the exception if the list is empty//should be O(1) unless resizing is necessary`

Guidelines:

- Implement your solution in a class called *Deque.java*
- You are not allowed to use any additional data structures (besides regular single-value variables)–only the underlying Array (but this will need to be resized).
- Familiarize yourself with the Array class if you haven't already.
- The way to get O(1) runtime for these methods is by resizing correctly and by using a "circular". In other words, instead of moving items around, you can just wrap around to the other end of the array.
    - Resize your underlying Array by multiplying it by 2 when it gets full or by cutting it in half when it drops below ¼ full. This ensures that you aren't wasting too much space while still providing a reasonably efficient runtime. The justification for this is called *amortized cost analysis* and can be shown mathematically. Also, resize when you are trying to add an element to a full Array–that means you should check to see if the Array needs to be resized at the beginning of any "add" method.
    - When removing an element, you should remove the element first, then check if the Array is less than ¼ full. In that case, resize to half the capacity *unless that would put the capacity below the original capacity.*
    - Some of these methods are required to "throw" an exception. The Exception code is provided for you, but if you are unfamiliar with exceptions in Java, you may want to look up how to use exceptions. Some of you may wonder if the test code is incorrect in how it is handling

exceptions. It is not. It is handling the exceptions with try-catch blocks because it is expecting your method to “throw” the exception in the appropriate place.

- Similarly, this implementation needs to be generic and work any any type of object, so if you are unfamiliar with generics in Java, you should get familiar with it.
- Use a wrap-around method for adding/removing from the front and back of the list.
  - This will require you to keep track of where the *front* and *back* elements of the Deque are.
  - Consider the example below and think about where the *front* and *back* of the Deque are in each case.

`addLast(5)`

5						

`addLast(6)`

5	6					

`addLast(7)`

5	6	7				

`addFirst(8)`

5	6	7				8

`addFirst(9)`

5	6	7			9	8

`addLast(10)`

5	6	7	10		9	8

`removeFirst()`

5	6	7	10			8

`removeLast()`

5	6	7				8

Part 2. Deque1.java and DequeCompare.java

In this part, you are going to run an experiment to compare two resizing methods. First, you need to copy your Deque.java code into a new class called Deque1. Then rewrite the code so that when you resize the array, you are adding or subtracting some

constant amount rather than multiplying by 2. You should make this a variable so that you can easily change the constant.

Next, in a file called `DequeCompare.java`, write a simple main method that gets an integer *N* as a command-line argument and then runs the experiment. The goal of the experiment is to compare the performance of the two Deques. To do this, you should call your `addLast` method *N* times in a row on each of the two Deques.

Once you have this code running, run it several times on increasingly large values of N. The goal is to see the difference in performance as N gets larger and larger. Keep in mind that you may have to get to pretty large values of N in order to see the difference. If you try larger and larger values and you still are not seeing a difference in performance, try making your resize constant in Deque1 a bit smaller.

Testing & Submission Procedure.

Your code must compile and run on lectura, and it is up to you to check this before submitting.

To test your code on lectura:

- Transfer all the files (including test files) to lectura.
- Run *javac *.java* to compile all the java files. (You can also use *javac* with an individual java file to compile just the one file: *javac ArrList.java*)
- Run *java <filename>* to run a specific java file. (Example: *java ArrListTest.java*)

After you are confident that your code compiles and runs properly on lectura, you can submit the required files using the following **turnin** command. Please do not submit any other files than the ones that are listed.

```
turnin csc345pa2 Deque.java Deque1.java DequeCompare.java
```

Upon successful submission, you will see this message:

```
Turning in:
    Deque.java - ok
    Deque1.java - ok
    DequeCompare.java - ok
All done.
```

Note: Only properly submitted projects are graded! No projects will be accepted by email or in any other way except as described in this handout. If you are worried about your submission, feel free to ask us to check that it got into the right folder.

Grading.

Auto-grading

Part	Total Points	Details
Deque.java	46	Checks both accuracy and efficiency of the required methods.
Deque1.java & DequeCompare .java	4	We will make sure these files compile, run, and do what they are supposed to do.

In addition to the auto-grading, the following lists items we could potentially take points off for.

- Code does not compile or does not run on lectura (up to 100% of the points).
- Bad Coding Style (up to 5 points) – this is generally only deducted if your code is hard to read, which could be due to bad documentation, lack of helpful comments, bad indentation, repeating code instead of abstracting out methods, writing long methods instead of smaller, more reusable ones, etc.
- Not following directions (up to 100% of the points depending on the situation)
- Late Submission (5 points per day)--see the policy in the syllabus for specifics
- Inefficient code (up to 50% of the points)

Your score will be determined by the tests we do on your code minus any deductions that are applied when your code is manually graded. In addition to late deductions, coding style, and deductions for not following directions, you may also receive deductions for inefficient code.

See the late submission and resubmission policies in the syllabus.