CS 0445 Spring 2023 Assignment 1

Online: Thursday, January 19, 2023
Due: All source files plus a completed Assignment Information Sheet zipped into a single .zip file and submitted properly to the submission site by **11:59PM on Friday, February 3, 2023** (Note: see the submission information page for submission details)
Late Due Date: 11:59PM on Monday, February 6, 2023

Purpose: To refresh your Java programming skills and to emphasize the object-oriented programming approach used in Java. Specifically, you will work with control structures, class-building, inheritance, interfaces and generics to **implement** some simple array-based data structures.

Background: In Lecture 4 we discussed the author's DequeInterface<T>, which is an ADT that allows for adding and removing at both sides of a queue (hence the name double-ended queue). The methods in this interface are specified in file DequeInterface.java. See the Lecture 4 Powerpoint presentation for some background and ideas about the DequeInterface<T>.

In Recitation Exercise 1 you implemented (or will implement) a simple class called SimpleDeque<T> that satisfies DequeInterface<T> but in an inefficient way. Specifically, it uses an array and maintains the logical "front" of the deque at index 0 and thus requires shifting in order to add or remove data at the front of the deque. We will discuss specific run-time analysis of this implementation a bit later in the course, but it is intuitive that there is a lot of overhead when adding or removing at the front of the deque.

A deque can be implemented in a more efficient way with an array if we allow both logical sides of the deque to move along the array in a circular fashion. For example, consider the array below:

0	1	2	3	4	5	6	7
		10	20	30	40	50	

front back

In this deque, both front and back will move in both directions within the array as we add or remove from either end. For example, if we add 55 at the front, the deque will look as follows:

0	1	2	3	4	5	6	7
	55	10	20	30	40	50	

front back

If we then remove from the back, the deque will look as follows:

0	1	2	3	4	5	6	7
	55	10	20	30	40		

front back

Note that for this approach to work effectively, both back and front will need to "wrap" around either end of the array when necessary. For example, if we now add 66 and 77 (in that order) at the front, our array will appear as follows:

0	1	2	3	4	5	6	7
66	55	10	20	30	40		77

back front

Note that when the deque is implemented in this way, we do not have to shift data in the array and any of the add / remove methods can be implemented with just a few statements.

Goal 1: To design and implement a class MyDeque<T> that will be an efficient, **array-based** implementation of **a modified version of author's DequeInterface<T>**. The data within your MyDeque<T> must be stored in a regular Java array **(no Java classes may be used for your MyDeque<T> data – it must be a regular array)**. None of your DequeInterface<T> methods should require any shifting of data within the array. I have added a couple of extra methods to the DequeInterface<T> and changed a couple of special cases. **Be sure to download the version of DequeInterface<T> posted on the Assignments page for this project.** The details of this interface are explained in the file DequeInterface.java and in Lecture 4. **Review these items very carefully before implementing your MyDeque<T> class.**

Goal 2: To write a subclass of MyDeque<T> called IndexDeque<T> that will have **no new instance data** but will additionally implement the interface **Indexable<T>**. The Indexable<T> interface has 5 methods:

```
public T getFront(int i);

public T getBack(int i);

public void setFront(int i, T item);

public void setBack(int i, T item);

public int size();  // same size() as in DequeInterface<T>
```

These methods will allow us to access an arbitrary index within the deque relative to either the front or back of the deque. The "get" methods will return the value at that location and the "set" methods will allow a new object to be assigned at that location. Note that **none of these methods will increase or decrease the size of the deque**. For example, given the deque above, if we were to call setFront(3, 44) the deque would appear as shown:

0	1	2	3	4	5	6	7
66	55	44	20	30	40		77

back front

Note that the indexing for getFront() and setFront() is opposite that of getBack() and setBack() – they move in opposite directions within the deque. Note also that getFront(0) is identical to getFront() in DequeInterface<T>, and getBack(0) is identical to getBack() in DequeInterface<T>.

The details of this interface are explained in the file Indexable.java. **Read this file over very carefully before implementing your IndexDeque<T> class.**

Goal 3: To write a subclass of IndexDeque<T> called IndexAddRemoveDeque<T> that will have **no new instance data** but will additionally implement the interface IndexableAddRemove<T>. The IndexableAddRemove<T> interface extends Indexable<T> and adds the following methods:

```
public void addToFront(int i, T item);

public void addToBack(int i, T item);

public T removeFront(int i);

public T removeBack(int i);
```

These methods allow new items to be inserted into the deque with position relative to the front or back. Unlike the previous two interfaces, the methods in this interface may require shifting of the data. This is necessary to create a space for a new item or to fill in a gap created by removal of an item (since they may be in the middle of the existing data). For example, in the deque shown above, if we were to call removeBack(2), the deque may appear as shown:

0	1	2	3	4	5	6	7
66	55	44	30	40			77
				back			front

The underlying data within this class must still be a circular array and you should not redefine any of the methods from the previous interfaces. Think carefully about how you will implement the IndexAddRemoveDeque<T> methods – you may want to figure these out with a pencil and paper before coding them. Note also that the shifting required for these methods can be done in relative to the front or the back of the deque. Either is ok as long as the result of the operation yields a correctly configured deque.

The details of this interface is explained in the file IndexableAddRemove.java. **Read this file over very carefully before implementing your IndexAddRemoveDeque<T> class.**

Implementation Details:
Your MyDeque<T> class should have the following header:

```
public class MyDeque<T> implements DequeInterface<T>
```

Your IndexDeque<T> class should have the following header:

```
public class IndexDeque<T> extends MyDeque<T> implements Indexable<T>
```

Your IndexAddRemoveDeque<T> class should have the following header:

```
public class IndexAddRemoveDeque<T> extends IndexDeque<T>
                                    implements IndexableAddRemove<T>
```

Note that each class is extending the previous one. You should not have any additional instance data in your two subclasses – the only data you need should be declared in MyDeque<T>. However, you should make the data in MyDeque<T> protected so that it can be accessed from within your subclasses. For example, you may have the following data in MyDeque<T>:

```
protected T [] data;
protected int front, back, N; // N is number of items
```

Think carefully about each of the methods you are implementing. Some of them may be easy but some may be more complicated. Also think carefully about **special cases**. For example, adding to an empty

deque may be a special case. Due to the inheritance chain of these classes, you clearly must complete them in the order that they are presented above.

The array in your MyDeque<T> class should be initialized of a specific length in a MyDeque<T> constructor that takes an int argument:

```
public MyDeque(int sz)
```

This will create an array of length sz to be used for your MyDeque<T>. If this array fills you should **dynamically resize it** by creating a new array of double the size, and copying the data into the new array. **Be very careful with your array resizing.** Because of the circular nature of your array access, you cannot just copy the data in the old array into the same locations in the new array. Think about why this is the case. The important issue is that the deque after you resize the array is logically equivalent to the deque prior to resizing. I recommend writing a protected method in class MyDeque<T> to do the resizing. Then you can also call this method from IndexAddRemoveDeque<T> when needed.

To allow for testing and more functionality in your classes, you must also implement the following 3 methods in your MyDeque<T> class:

```
public MyDeque(MyDeque<T> old)
public boolean equals(MyDeque<T> rhs)
public String toString()
```

The first method is a copy constructor, which should make a new MyDeque<T> that is logically equivalent to the argument. The copy constructor should not be shallow – the arrays should not be shared. However, it does not have to be completely deep either (you don't need to make new copies of the individual items in the deque). Note also that the copy does not have to have the same values for front and back as the original – the important thing is that both contain the same data in the same relative ordering from front to back.

The second method is an equals method that returns true if the deques are logically equivalent. In this case logically equivalent means that the individual items in both deques are equal() and in the same relative positions within the deques (based on front and back). However, they do not have to be located in the same index values in the arrays.

The third method is a toString() method which will make a and return single String of all the data in the MyDeque<T> from front to back and return it. This method will assume that a reasonable toString() method exists for whatever type T is being used for the MyDeque<T>.

After you have written all of your classes, test them with the following two programs:

Assig1A.java – this program will test the MyDeque<T> class and its implementation of the DequeInterface<T>. Read over the code in this program carefully so that you understand what all of the methods are doing. This program should compile and run without change using your implementation of the MyDeque<T> class. The output should match that provided in file Assig1A-out.txt.

Assig1B.java – this program will test the IndexDeque<T> and IndexAddRemove<T> classes. Read over the code in this program carefully so that you also understand that this program is doing. This program should compile and run without change using your implementation of the IndexDeque<T> and IndexAddRemove<T> classes. The output should match that provided in file Assig1B-out.txt.

Make sure you **put comments into all of your files**. Minimally this should include your name and the course, and also a brief explanation of each file (at the top of the file). Also include comments to explain any code that is not obvious.

You **must submit in a single .zip file** containing (minimally) the following 8 complete, working source files for full credit:

DequeInterface.java Indexable.java IndexableAddRemove.java
Assig1A.java Assig1B.java

the **above five files** are given to you and must not be altered in any way.

MyDeque.java IndexDeque.java IndexAddRemoveDeque.java

the **above three files** must be created so that they work as described. If you create any additional files, be sure to include those as well. You must also submit in your .zip file an Assignment Information Sheet, as explained below.

The idea from your submission is that your TA can unzip your .zip file, then compile and run both of the main programs (Assig1A.java and Assig1B.java) **from the command line** WITHOUT ANY additional files or changes, so be sure to test it thoroughly before submitting it. If you cannot get the programs working as given, clearly indicate any changes you made and clearly indicate why (ex: "I could not get method X to work, so I eliminated code that used it") on your Assignment Information Sheet. You will lose some credit for not getting it to work properly, but getting the main programs to work with modifications is better than not getting them to work at all. See the CS 0445 Web site for an Assignment Information Sheet template – you do not have to use this template but your sheet should contain the same information. **Note: If you use an IDE such as Eclipse to develop your programs, make sure they will compile and run on the command line before submitting – this may require some modifications to your program (such as removing some package information).**

Extra Credit:

- If you can think of another subclass of MyDeque<T> that has reasonable functionality, implement it and write a short driver program to test it. If your new functionality is non-trivial, you can receive up to 10 extra credit points for it.