Pset 5 Notes (CSCI E-10B)

General Information

a. Problem set 5 is due Monday, Dec 4, 2023 at 9:00 AM Eastern Time. You can resubmit your assignment as often as desired, but each submission's zip must include every file that you want to be graded.

b. We will deduct 10% for a homework assignment that is turned in up to 3 days late. 20% will be deducted if the homework is more than 3 days late. No homework will be accepted more than 7 days late. The last submission controls the late penalty for the entire assignment.

c. Don't procrastinate! Note, though, that the pset covers material from lectures 11 and 12 so you will probably need to defer some questions until you've viewed future lectures and attended or viewed future sections.

d. Your Java code must compile. Programs that do not compile will automatically have their score divided by two.

e. Your programs must behave as specified. Do everything the specs say to do. Do not do more than the specs say to do. Precisely follow the directions in the pset (e.g. use the exact filenames specified in the pset), except make adjustments specified by the staff in these notes or on the Ed discussion board.

f. Some of the work is designated as "extra credit." In this course, extra credit points are kept separate from regular credit points. They only come into play at the end of the semester when Dr. Leitner is assigning final grades. If you are on the cusp between, say, a B+ and an A-, extra credit points can influence that decision.[1]

   If the pset says "identify which problem(s) you want treated as extra credit," ignore that! We automatically allocate points to maximize your regular credit score because that contributes the most to your grade in the course.

   If your submission gets a late penalty, then you won't get any extra credit points for that submission.

g. This course emphasizes programming style. See the Ed posts titled *Reasonably-commented programs*, *Javadoc commenting* and *Java Style Guide*. Inadequately-commented or -styled programs will lose points.

Pencil-and-Paper Exercises

This assignment has no Pencil-and- Paper Exercises.

Programming Problems

To minimize complications when developing Swing programs with VS Code, you are advised to run VS Code on your local PC instead of running the cloud-hosted VS Code.

Submit complete and correct programs that behave exactly as specified, except make adjustments specified by the course's staff in these notes or on the Ed discussion board.

Your programs' interaction with the user should match what's in the assignment, and when sample cases are shown in the problem set, your program must produce the same output when given the sample case's input.

Whenever you are asked to write a specific method that's not `main()`, calling that method must not cause any output to be sent to the terminal unless the instructions say otherwise.

Unless otherwise specified, programs that read or write files must behave gracefully if input files cannot be found, cannot be read, are empty, contain unexpected data, ... or if output files cannot be written.

[1] The decision is also influenced by your teaching assistant's feedback, so it's a good idea to get to know that person.

Programs that accept command line arguments must behave gracefully if an unexpected number of command line arguments are provided.

If you are told to provide code that demonstrates your method, you must follow these rules:

1. You must not require your TA to define the test cases.
2. You must not require your TA to read your source code in order to decipher your code's output.

Instead, you need to implement a robust set of self-documenting test cases that convincingly demonstrate your method. Here's how I would do that: For a well-chosen set of test cases, my demo would contain statements analogous to:

```
System.out.printf( "someMethod( %d, %d ) = %d\n", int1, int2,
                                                   someMethod( int1, int2 ) );
```

You will replace "**someMethod**" with the actual name of the method being demonstrated, and you'll adjust the number of arguments, their data types, the method's return type, etc.

a. Problem 1a (**Bitset.java**, **TestSets.java** or **TestSetsSwing.java**): Create a **Bitset** instance method named **cardinality()** that behaves like this:

```
Bitset foo = new Bitset( 8 );
foo.include( 1 ); foo.include( 3 ); foo.include( 5 );
int bar = foo.cardinality(); // bar receives the value 3
```

cardinality() should use existing **Bitset** instance methods to compute the **Bitset**'s cardinality. It should not directly access the bitset's **byteArray** instance variable.

Add support for **cardinality()** to **TestSets.java** or **TestSetsSwing.java**.

b. Problem 1b (**Bitset.java**, **TestSets.java** or **TestSetsSwing.java**): Create a **Bitset** instance method named **isSubset()** that behaves like this:

```
Bitset foo1 = new Bitset(); Bitset foo2 = new Bitset();
// Add members to foo1 and to foo2 ...

boolean s = foo1.isSubset(foo2);
// 1. foo1.isSubset(foo2) returns true if every member of foo1
//    is a member of foo2.
// 2. Another (equivalent) way to express that:
//    foo1.isSubset(foo2) returns false if at least one member of foo1
//    is NOT a member of foo2.
```

isSubset () should use existing **Bitset** instance methods to compute its result. It should not directly access the implicit and explicit arguments' **byteArray** instance variables.

Add support for **isSubset()** to **TestSets.java** or **TestSetsSwing.java**.

Note that the empty set is a subset of *every* set, so if **foo1** is empty, then **foo1.isSubset(foo2)** evaluates to **true** no matter what's in **foo2**. If you choose wisely between definitions 1 and 2 in the above code snippet, your **isSubset()** implementation doesn't even have to treat "**foo1** is empty" as a special case!

c. Problem 1c (**TestSetsSwing.java**): For up to 10 points of extra credit, create a tasteful and easy-to-use Swing implementation of **TestSets.java**. If you implement **TestSetsSwing.java**, then you do not need to modify **TestSets.java**.

d. Problem 2 (**LinkedDeque.java**, **LinkedDequeTest.java**): The **LinkedQueue.java** template class [2] implements a single-ended queue that allows a user to add an element to the rear of the queue and to delete an element from the front of the queue.

   You must modify **LinkedQueue.java** to create **LinkedDeque.java**, that implements a "double-ended queue" that allows a user to add new elements to either end of the double-ended queue, as well as to delete elements from either end of the double-ended queue.

   Like **LinkedQueue.java**, **LinkedDeque.java** must implement a singly-linked list. In other words, **LinkedDeque.java** must use the same **QueueNode** inner class that is in **LinkedQueue.java**:

```
/**
 *  The QueueNode class is an inner class implemented to model a
 *  queue node; it can contain an Object type of data, and also
 *  holds the link to the next node in the queue.  If there are no
 *  other nodes, the link will be null.
 */
class QueueNode {         // an inner class
    private Object item;
    private QueueNode link;
}
```

   LinkedDeque.java must use instance variables analogous to those that are in **LinkedQueue.java**:

```
private QueueNode tail;  // analogous to LinkedQueue's rear
private QueueNode head;  // analogous to LinkedQueue's front;
private int count;
```

   LinkedDeque.java must contain these (and only these) **public** methods.

```
// Constructor
public LinkedDeque() { … }

// add, peek, remove object at head (front) of double-ended queue
public void headAdd( Object o ) { … }
public Object headPeek() { … }
public Object headRemove() { … }

// add, peek, remove object at tail (rear) of double-ended queue
public void tailAdd( Object o ) { … }
public Object tailPeek() { … }
public Object tailRemove() { … }

public boolean isEmpty() { … }
public int size() { … }
public String toString() { … }
```

   You can include additional **private** helper methods and **private** variables at your discretion.

   You must write a **main()** program in a **LinkedDequeTest** program class, that demonstrates that all of the above methods work correctly.

[2] Located on the Java Resources page.

For five points of extra credit, invent a new **`Exception`** type named **`DequeueUnderFlowException`** that gets thrown when a program class tries to remove or peek at an element from an already-empty double-ended queue. This feature should be demonstrated in your main program.

For the extra credit part of the problem, pay attention to what gets done in the **`LinkedDequeue`** template class, and what gets done in the **`LinkedDequeueTest`** program class. In particular:

- The **`LinkedDequeue`** template class's **`headRemove()`**, **`tailremove()`**, **`headPeek()`**, and **`tailPeek()`** methods throw the **`DequeueUnderflowException`** when they detect that they've been called on an empty double-ended queue.
- The **`LinkedDequeueTest`** program class only catches the exception that was thrown by the **`LinkedDequeue`** template class.

I'll say it again:

- The **`LinkedDequeue`** template class does throw the exception.
- The **`LinkedDequeueTest`** program class does not throw the exception.
- The **`LinkedDequeue`** template class does not catch the exception.
- The **`LinkedDequeueTest`** program class does catch the exception.

Here it is in table format:

	throws the exception	**catches** the exception
`LinkedDequeue` template class	Yes	No
`LinkedDequeueTest` program class	No	Yes