Problem Set Five

Programming Problem on Bitsets (25 - 35 points)

See chapter 16 in the Reges textbook, which covers linked-list data structures.[1] Consult also the CSCI E10b lecture notes!

[1] Copy the files **`Bitset.java`** and **`TestSets.java`** from the Unit 7 Lecture Files section of the Java Resources page.

Part (a) **10 points**

The *cardinality* of a set is defined as the number of elements in the set. Implement **`public int cardinality()`** as an instance method of the ***Bitset*** class, and modify program ***TestSets*** to adequately test this new feature.

Part (b) **15 points**

A set named **`a`** is considered *subset* of the set named **`b`** if and only if every element of **`a`** is also a member of the set **`b`**. Implement **`public boolean isSubset (Bitset b)`** as an instance method of the ***Bitset*** class, and modify program ***TestSets*** to adequately test this new feature. (Within *TestSets* be sure to test whether **`a`** is a subset of **`b`** and also whether **`b`** is a subset of **`a`**.)

Part (c) **up to 10 points — for "extra credit" only**

The user-interface presented by program **`TestSets.java`** is pretty lame, given the graphical user interface components provided by Java's **`AWT`** and **`Swing`** classes. At present, the user is presented with a console-window that presents a list of choices that looks something like the following:

---

[1] The Horstmann textbook, *Core Java, Volume 1* (10th edition), contains some interesting descriptions on how linked lists work in chapter 9. The bit manipulation operators in Java are described in chapter 3. Linked lists are thoroughly explained in the 4th edition of the Reges & Stepp textbook in chapters 11 and 16.

```
Type 1 to CREATE SET A
Type 2 to CREATE SET B
Type 3 to CREATE INTERSECTION (A * B)
Type 4 to CREATE UNION (A + B)
Type 5 to CREATE DIFFERENCE (A - B)
Type any OTHER # to EXIT PROGRAM

Command: 2
TYPE SOME SMALL INTEGERS each < 8, and type DONE when all done!
3 3 3 7 5 6 2  DONE
     SET B = {   2  3  5  6  7  }
Type 1 to CREATE SET A
Type 2 to CREATE SET B
Type 3 to CREATE INTERSECTION (A * B)
Type 4 to CREATE UNION (A + B)
Type 5 to CREATE DIFFERENCE (A - B)
Type any OTHER # to EXIT PROGRAM

Command: 1
TYPE SOME SMALL INTEGERS each < 16, and type DONE when all done!
11 12 3 4 4
4 5 2  DONE
     SET A = {   2  3  4  5  11  12  }
Type 1 to CREATE SET A
Type 2 to CREATE SET B
Type 3 to CREATE INTERSECTION (A * B)
Type 4 to CREATE UNION (A + B)
Type 5 to CREATE DIFFERENCE (A - B)
Type any OTHER # to EXIT PROGRAM

Command: 3
     Intersection (A * B) = {   2  3  5  }
Type 1 to CREATE SET A
Type 2 to CREATE SET B
Type 3 to CREATE INTERSECTION (A * B)
Type 4 to CREATE UNION (A + B)
Type 5 to CREATE DIFFERENCE (A - B)
Type any OTHER # to EXIT PROGRAM
```

Your job is to rewrite the existing code so that the user interface is provided through a combination of ***JMenus***, ***JButtons***, ***JTextFields***, ***JTextAreas***, ***JPanels***, ***JFrames***, ***JCheckBoxes*** and other GUI components you are familiar with. Note: you do not need to use all of the above-mentioned elements; create a tasteful and easy-to-use interface using a subset of the *Swing* components that make the most sense.

Programming Problem on Linked-Lists (25 — 30 points)

[2] 25 points

Recall the **LinkedQueue.java** code from the Unit 7 Lecture Files section of the Java Resources page.

A simple variation of the ordinary queue data type is a ***dequeue*** (short for "double-ended queue"). Such a data structure allows a user to add new elements to either end (the head or the tail), as well as to delete an element from either end.

The **`LinkedDequeue`** will be singly linked (as in **`LinkedQueue`**), and data members should be kept **`private`**. Modify the code in **`LinkedQueue.java`** to create **`LinkedDequeue.java`**, with the following *public* methods.

- **`LinkedDequeue ()`**
  Constructor that creates an empty *Dequeue* .

- **`void headAdd (Object o)`**
  Insert an item into the *Dequeue* at the head.

- **`Object headPeek ()`**
  Peek at the head of the *Dequeue*.

- **`Object headRemove ()`**
  Remove an item from head of the *Dequeue*, just like method **`delete`** in **`LinkedQueue.java`**

- **`boolean isEmpty ()`**
  Returns **true** if *Dequeue* contains no elements; else returns **false**

- **`int size()`**
  Returns the number of elements currently in the *Dequeue*

- **`void tailAdd (Object o)`**
  Insert an item into the *Dequeue* at the tail, just like the method **`add`** in **`LinkedQueue.java`**

✦ **`Object tailPeek()`**

Peek at the tail of the *Dequeue*. In other words, return the value at the tail, without removing it.

✦ **`Object tailRemove()`**

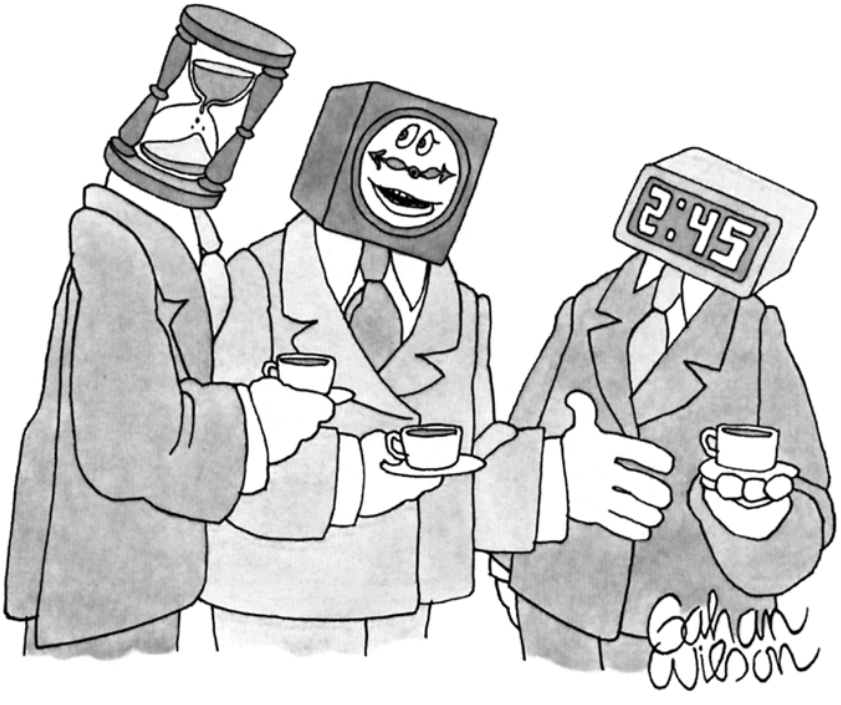
Remove an item from the tail of the *Dequeue*.

✦ **`String toString()`**

Outputs, in a neatly formatted fashion, all objects in the queue, from the head through the tail. The objects should print out one per line!

You must write a main program to demonstrate that all of the above methods work correctly, perhaps by adding and deleting *String* objects from both ends of a dequeue. Note that the two "peek" methods let the user see whatever object is at the head or tail of the queue, without actually removing the object. Be aware that some of the methods that already exist in **`LinkedQueue.java`** can simply be renamed as part of your solution.

For 5 points of *extra credit*: invent a new **`Exception`** type named **`DequeueUnderFlowException`** that gets thrown when one tries to remove or peek at an element from an already empty queue. This feature should be demonstrated in your main program.



"Basically, we're all trying to say the same thing."