CMPS 101
Algorithms and Abstract Data Types
Winter 2006

Programming Assignment 1

Due Wednesday January 18, 10:00 pm

The purpose of this assignment is threefold: to make sure everyone is up to speed with Java, to practice modularity and ADTs, and to build an ADT implementation which will be used (with minor modifications) in future programming assignments. You should therefore test your ADT carefully, even though all of its features may not be used here.

You are to write a Java program which takes as input a positive integer *n* from the command line, creates a deck of *n* cards, then repeatedly performs a *perfect shuffle* on this deck until the cards are again in their original order. Each card is labeled with a number from 1 to *n*. A perfect shuffle is performed by splitting the deck into a top part and a bottom part (if *n* is odd, the middle card goes into the top part) and then (starting with the top part) repeatedly taking the bottom card from each part and placing them on top of a new deck. Thus to perform a perfect shuffle on the deck **1 2 3 4 5 6 7 8 9** ($n = 9$), we begin by splitting it into a top part **1 2 3 4 5**, and a bottom part **6 7 8 9**. We start the new deck with **5** on the bottom, then **9**, then **4**, and so on until the new deck is **1 6 2 7 3 8 4 9 5**. (Note that if *n* is odd, then the first card remains in its original location after the shuffle.) Repeatedly shuffling a deck of size 9 results the following sequence of deck permutations:

```
#Shuffles   Deck Order
-------------------------------------------------------
    0       1 2 3 4 5 6 7 8 9
    1       1 6 2 7 3 8 4 9 5
    2       1 8 6 4 2 9 7 5 3
    3       1 9 8 7 6 5 4 3 2
    4       1 5 9 4 8 3 7 2 6
    5       1 3 5 7 9 2 4 6 8
    6       1 2 3 4 5 6 7 8 9
```

Repeatedly shuffling a deck of size 14 results in:

```
#Shuffles   Deck Order
-------------------------------------------------------
    0       1 2 3 4 5 6 7 8 9 10 11 12 13 14
    1       8 1 9 2 10 3 11 4 12 5 13 6 14 7
    2       4 8 12 1 5 9 13 2 6 10 14 3 7 11
    3       2 4 6 8 10 12 14 1 3 5 7 9 11 13
    4       1 2 3 4 5 6 7 8 9 10 11 12 13 14
```

The relationship between the deck size and the number of shuffles required to bring the deck back to it's original state is an interesting one. A deck of size 500 requires 166 shuffles, while one of size 508 requires 508 shuffles, and a deck of size 511 requires only 9 shuffles. Note that the above examples are for illustration only and do not represent program output.

Although there are many ways to shuffle cards, the purpose of this assignment is to implement and use a list of integers ADT. Your program will do the following:

1. Read the number of cards *n* from the command line.
2. Initialize a list (i.e. a deck) containing the *n* cards in sorted order. This is the only time you should need to use the value *n*.
3. Perform perfect shuffles (as described above) on this deck until the original sorted order is achieved.
4. Print out the number of shuffles performed.

To perform a perfect shuffle you should split your deck into two other decks by alternately "dealing" from the top and bottom of the original deck, and inserting the cards into the new decks, then combine these two decks by alternately "dealing" off their bottoms.

Your List ADT will actually be a double ended queue with a current-position marker. Thus the set of "mathematical structures" for this ADT consists of all finite sequences of integers, where one integer may be distinguished as the current element. (Note it is also possible in this ADT that no element is current.) The current element can be used by the client to traverse the lists. Your List module will support the following operations:

```
// Constructors
List()  // Creates an empty list

// Access functions
boolean isEmpty()  // Returns true if list contains no elements.
boolean atFirst()  // Returns true if current marker refers to first element.
boolean atLast()  // Returns true if current marker refers to last element.
boolean offEnd()  // Returns true if no element is current.
int getFirst()  // Returns first element in list. Pre: !isEmpty().
int getLast()  // Returns last element in list. Pre: !isEmpty().
int getCurrent()  // Returns current element in list. Pre: !isEmpty(), !offEnd().
int getLength()  // Returns length of list.
boolean equals(List L)  // Returns true if this list contains same elements
                        // as argument list.

// Manipulation Procedures
void moveFirst()  // Makes first element current.  Pre: !isEmpty(), Post: !offEnd().
void moveLast()   // Makes last element current. Pre: !isEmpty(), Post: !offEnd().
void moveNext()   // Steps the current marker toward end of list.
                  // Pre: !isEmpty(), !offEnd().
void movePrev()   // Steps the current marker toward beginning of list.
                  // Pre: !isEmpty(), !offEnd().
void insertBeforeFirst(int data) // Adds new element to beginning of list.
                                 // Post: !isEmpty().
void insertAfterLast(int data)   // Adds new element to end of list.
                                 // Post: !isEmpty().
void insertBeforeCurrent(int data) // Inserts new element just before current.
                                   // Pre: !isEmpty(), !offEnd().
void insertAfterCurrent(int data)  // Inserts new element just after current.
                                   // Pre: !isEmpty(), !offEnd().
void deleteFirst()  // Deletes first element.  Pre: !isEmpty().
void deleteLast()   // Deletes last element.  Pre: !isEmpty().
void deleteCurrent()  // Deletes current element.
                      // Pre: !isEmpty(), !offEnd(); Post: offEnd().

// Other Operations
List copy()  // Returns a new List containing the same elements as this list.
public String toString() // Overrides Object's toString method.
public static void main(String[] args) // Test driver for the List class.
```

The above operations are required for full credit, although it is not expected that all will be used by the client module in this project (i.e. the Shuffle class). The following operations are optional, and may come in handy in some subsequent assignment:

```
List cat(List L) //  Returns a new list which is the concatenation of
                 //  this List and the argument list.  This list and
                 //  the argument list are unchanged.
void makeEmpty() // Sets this List to the empty state.
```

Your List class should contain a private Node class which encapsulates one List element. This private class should contain fields for an int (the value stored at that node), a Node (the previous element in the list), and another Node (the next element in the list). It should also define an appropriate constructor, as well as a `toString()` method. The List class should contain private fields of type Node which refer to the first, last, and current Nodes in the List.

Your program will be structured in two files: a client module called Shuffle.java, and a List ADT module called List.java. Each file will contain one top level class, Shuffle and List respectively. Shuffle will represent a deck of cards as a List variable, and use the above methods to perform shuffling operations.

The following Makefile creates an executable jar file called Shuffle. Place it in a directory containing List.java and Shuffle.java, then type `gmake` to compile your program. You can now invoke your program by typing `Shuffle` (rather than `java Shuffle`.) For instance doing `Shuffle 17` should result in the integer `8` being printed to standard out.

```
# Makefile for CMPS 101 pa1 Winter 2006.

MAINCLASS  = Shuffle
JAVAC      = javac
JAVASRC    = $(wildcard *.java)
SOURCES    = $(JAVASRC) makefile README
CLASSES    = $(patsubst %.java, %.class, $(JAVASRC))
JARCLASSES = $(patsubst %.class, %*.class, $(CLASSES))
JARFILE    = $(MAINCLASS)

all: $(JARFILE)

$(JARFILE): $(CLASSES)
	echo Main-class: $(MAINCLASS) > Manifest
	jar cvfm $(JARFILE) Manifest $(JARCLASSES)
	chmod +x $(JARFILE)
	rm Manifest

%.class: %.java
	$(JAVAC) $<

clean:
	rm *.class $(JARFILE)
```

Note that this Makefile will compile all .java files in your current working directory. Also be aware that if you are using the bash shell and you type `make` (instead of `gmake`), this makefile may not work properly. To be safe always use `gmake`.

You must also submit a README file for this (and every) assignment describing the files created for the assignment, their purposes and relationships, along with any special notes to myself and the grader. Each file you turn in must begin with your name, user id, and assignment name. Thus you are to submit four files in all: List.java, Shuffle.java, Makefile, and README.

Start early and ask questions if anything is unclear. It is helpful to write simple test programs to make sure you understand each part of the problem. The main method in your List class is required because it is much easier to debug your List ADT in isolation before you use it in the Shuffle class. You should first design and build your List ADT, test it thoroughly, and only then start coding your Shuffle class. Information on how to turn in your program will be posted on the webpage.