CMPS 101
Algorithms and Abstract Data Types
Summer 2006

Programming Assignment 1

Due Wednesday July 5, 10:00 pm

Introduction

The purpose of this assignment is threefold: to make sure everyone is up to speed with Java, to practice modularity and ADTs, and to implement an Integer List ADT which will be used (perhaps 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.

In this project you will write a Java program which performs shuffles (i.e. permutations) on lists of integers. Observe that there are many ways to shuffle a given list of integers. For example the list (1 2 3) can be shuffled in 6 distinct ways: (1 2 3), (1 3 2), (2 1 3), (2 3 1), (3 1 2), (3 2 1). In general a list of length *n* has *n*! arrangements or permutations. Formally, a permutation of a set *S* is a bijection (i.e. a one-to-one onto mapping) from *S* to *S*. From now on we take *S* to be the set $S = \{1, 2, \ldots, n\}$. One way to denote a permutation of *S* is to simply list its elements twice, side by side, showing the image of each element under the permutation. For instance, let $n = 5$ and write

$$\sigma = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 3 & 1 & 4 & 5 & 2 \end{pmatrix}$$

to stand for the mapping $\sigma : S \to S$ which takes $1 \to 3$, $2 \to 1$, $3 \to 4$, $4 \to 5$, and $5 \to 2$. Such a mapping can be composed with itself to obtain another bijection $\sigma \circ \sigma = \sigma^2$ from *S* to *S*. One may verify in this case that

$$\sigma^2 = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 4 & 3 & 5 & 2 & 1 \end{pmatrix}.$$

Composing again we see $\sigma^3 = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 5 & 4 & 2 & 1 & 3 \end{pmatrix}$, $\sigma^4 = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 5 & 1 & 3 & 4 \end{pmatrix}$, and $\sigma^5 = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 \end{pmatrix}$.

Observe that σ^5 is the *identity* mapping, which takes $i \to i$ for all $i \in S$. The *order* of a permutation σ is the smallest positive integer *k* such that $\sigma^k = \text{identity}$. Thus in the above example, the order of σ is 5. In this project you will compute the order of a permutation by applying it repeatedly to a list (i.e. shuffling the list) until the list is brought back into its original order.

Our notation for permutations is still somewhat cumbersome. If we fix an ordering of *S* (such as increasing numerical order), then we can leave out the top row entirely. The permutation σ in the preceding example is then written as $\sigma = (3 \quad 1 \quad 4 \quad 5 \quad 2)$. In this way any arrangement of the elements of *S* is understood to denote the permutation which takes the standard arrangement (i.e. the one in increasing order) into the given arrangement. We will call this method for specifying permutations the *arrangement* representation.

There is yet another way to specify a permutation of $S = \{1, \ldots, n\}$ as a list of *n* integers. In this notation a list is interpreted to be a set of instructions for transforming the standard arrangement of *S* into a given

arrangement. To distinguish this representation from the arrangement representation, we will surround the list with square brackets [ ] rather than round brackets ( ). We refer to this scheme as the *operator* representation of a permutation, and illustrate with the following example. The list [1 2 1] instructs us to place elements from the standard arrangement (1 2 3) into an initially empty list ( ) in three steps:

1. Place element 1 in position 1: (1)
2. Place element 2 in position 2: (1 2)
3. Place element 3 in position 1: (3 1 2)

The result is the arrangement (3 1 2). Similarly [1 1 3] applied to (1 2 3) gives (2 1 3). Below are the 6 permutations of $S = \{1, 2, 3\}$ written in both operator and arrangement representation.

[1 2 3]	(1 2 3)
[1 2 2]	(1 3 2)
[1 1 3]	(2 1 3)
[1 1 2]	(2 3 1)
[1 2 1]	(3 1 2)
[1 1 1]	(3 2 1)

Observe that in the operator representation of a permutation, the integer at position i must be in the range 1 to i, since it is literally an instruction to insert something into a list of length $i-1$. Thus [1 3 2] is not a valid operator representation of any permutation. (What would it say? Starting with an empty list, insert 1 into position 1 to get the list (1), then insert 2 into position 3 to get …? But inserting anything into the list (1) results in a list of length 2, which therefore *has no* position 3.) One nice thing about the operator representation of a permutation is that it can be easily applied to any list of length n, not just the standard arrangement of S. For instance [1 2 2] applied to (a b c) gives (a c b), and [1 1 1] applied to (x y z) gives (z y x). The reader is urged at this point to write out all 24 permutations of $S = \{1, 2, 3, 4\}$ in both representations. Also check that the permutation σ from the preceding example, which was given in arrangement representation as $\sigma = (3 \quad 1 \quad 4 \quad 5 \quad 2)$, has the operator representation $\sigma = [1 \quad 2 \quad 1 \quad 3 \quad 4]$.

Program Operation

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. The client Shuffle will use List variables in two ways. On the one hand, a List will represent a permutation in operator representation. Such a List will itself be made to operate on a second List by splicing and dicing according to the 'instructions' in the first List. These shuffling operations will be performed in the client by calling the methods in the List module. Shuffle will be invoked at the command line by doing: `Shuffle input_file output_file`. Notice that one does not type `java Shuffle` at the command line. A Makefile is included at the end of this handout which places all .class files for this project in an executable jar file called Shuffle, making it possible to leave out `java` when invoking the program.

The input file will contain a number of permutations (of various sizes) in operator representation. For each such permutation your program will do the following.

1. Read the permutation from the input file and store it in a List P.
2. Initialize a List L consisting of the integers (1 2 … n), where n is the length P.
3. Shuffle L once by applying the permutation P. The list L now gives the arrangement representation of the permutation. Print L to the output file.

4. Continue to shuffle *L* by applying the operator *P* until *L* returns to the original standard order (1 2 … *n*). Count the number of shuffles performed, including the one in step (3). This count gives the order of the permutation. Print this count to the output file.

It is strongly recommended that Shuffle.java contain a function with the prototype

```
static void shuffle(List L, List P);
```

which performs one shuffle on the List *L* by applying the operator permutation *P*. This function will splice and dice the List *L* by applying the operations exported by the List ADT module List.java, described below. Note that the input Lists *L* and *P* must be of the same length, and *P* should be a valid operator representation of a permutation. These conditions should be checked by function shuffle().

File Formats

Go to the course website and follow the 'examples' link to see the program FileIO.java, which illustrates how to do file input and output. The first line of the input file will contain a single integer *N*, giving the number of permutations in the input file. The next *N* lines of the input file each contain exactly one permutation in operator representation, given as a space separated list of integers. Your program will read the first line, parse the integer *N*, then enter a loop which reads the next *N* lines. Each iteration of the loop will execute steps 1-4 above. The output file will contain exactly *N* lines giving the arrangement representation of each permutation, along with its order. These formats are illustrated below.

Input File:

```
6
1 1 2 2 3 5 4 3 8
1 2 1 3 4
1 2 3 3 5 4 7 2
1 2 1
1 1 1 4 2 6 2
1 1 1 1 1
```

Output File:

```
(2 4 8 5 7 3 6 9 1)  order=9
(3 1 4 5 2)  order=5
(1 8 2 4 6 3 5 7)  order=6
(3 1 2)  order=3
(3 7 5 2 1 4 6)  order=12
(5 4 3 2 1)  order=2
```

You may assume that your program will be tested only on correctly formatted input files. In particular, lines 2 through $N+1$ will contain only the valid operator representation of a permutation. You may also assume that the number of permutations in a file will not exceed 1000, and that each permutation will be of length at most 100.

List ADT Specifications

Your List ADT for this project will be a double ended queue with current-position marker. Thus the set of "mathematical structures" for this ADT consists of all finite sequences of integers, in which one integer may be distinguished as the current element. (Note that it is a valid state for this ADT to have *no* element be current. When in such a state, the current marker is considered to be *undefined*.) The current marker is used by the client to traverse Lists. Your List module will support the following operations:

```
// Constructors
List()  // Creates new empty List.

// Access functions
boolean isEmpty() // Returns true if this List is empty, false otherwise.
boolean offEnd() // Returns true if current is undefined.
boolean atFirst() // Returns true if first element is current. Pre: !isEmpty().
boolean atLast() // Returns true if last element is current. Pre: !isEmpty().
int getFirst() // Returns first element. Pre: !isEmpty().
```

```
int getLast() // Returns last element. Pre: !isEmpty().
int getCurrent() // Returns current element. Pre: !isEmpty(), !offEnd().
int getLength() // Returns length of this List.
boolean equals(List L) // Returns true if this List has same elements as L in the
                       // same order.  Ignores the current marker in both Lists.

// Manipulation Procedures
void makeEmpty() // Sets this List to the empty state. Post: isEmpty().
void moveFirst() // Sets current marker to first element.
                 // Pre: !isEmpty(); Post: !offEnd().
void moveLast() // Sets current marker to last element
                // Pre !isEmpty(); Post: !offEnd().
void movePrev() // Moves current marker one step toward first element.
                // Pre: !isEmpty(), !offEnd().
void moveNext() // Moves current marker one step toward last element.
                // Pre: !isEmpty(), !offEnd().
void insertBeforeFirst(int data) // Inserts new element before first element.
                                 // Post: !isEmpty().
void insertAfterLast(int data) // Inserts new element after last element.
                               // Post: !isEmpty().
void insertBeforeCurrent(int data) // Inserts new element before current element.
                                   // Pre: !isEmpty(), !offEnd().
void insertAfterCurrent(int data) // Inserts new element after current element.
                                  // 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 methods
List copy() // Returns a new list which contains the same elements as this List, and
            // in the same order. The current marker in the new list is undefined,
            // regardless of the state of the current marker in this list.  The state
            // of this list is unchanged.
public String toString() // Overrides Object's toString method and returns a string
                         // representation of this List consisting of a space
                         // separated list of integers.
public static void main(String[] args)  // Used as a 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. The following operation is optional, and may come in handy in some future assignment:

```
List cat(List L) // Returns a new List which is the concatenation of this list
                 // followed by L.  The current marker in the new list is undefined,
                 // regardless of the states of the current markers in the two lists.
                 // The states of the two Lists are unchanged.
```

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.

Makefile

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.

```
# Makefile for CMPS 101 pa1 Summer 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 may of course alter this Makefile as you see fit to perform other tasks (such as submit), but the Makefile your turn in must make an executable jar called Shuffle, and must include a clean utility.

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. README is essentially a table of contents for the project. Each file you turn in must begin with your name, user id, and assignment name. Thus, for this project you are to submit four files in all: List.java, Shuffle.java, Makefile, and README.

Advice

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 is posted on the webpage.