CMPS 101
Algorithms and Abstract Data Types

Programming Assignment 1

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 some 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 the set S to be $\{1, 2, \ldots, n\}$. 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

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

Upon composing again we see that $\sigma^3 = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 5 & 4 & 2 & 1 & 3 \end{pmatrix}$, and again $\sigma^4 = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 5 & 1 & 3 & 4 \end{pmatrix}$, 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 particular, the order of the identity permutation is 1. 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 some other 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 $i - 1$, which therefore has only i spaces to fill. 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 $\sigma = (3\ 1\ 4\ 5\ 2)$, has the operator representation $\sigma = [1\ 2\ 1\ 3\ 4]$.

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

The course website (under the 'examples' link) will contain a program called FileIO.java, which illustrates how to do file input and output operations in java. The first line of an input file for this project will contain a single integer N, giving the number of permutations in the 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 a current element 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. Observe that it is a valid state for this ADT to have *no* current element. When in such a state, the current element marker is deemed to be *undefined*, which is its default state. The finite integer sequence represented by this ADT has two distinguishable ends referred to as "front" and "back" respectively. The current element will be used by the client to traverse a List in either direction. Your List module will define the following operations.

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

// Access functions
int getLength()   // Returns length of this List.
boolean isEmpty() // Returns true if this List is empty, false otherwise.
```

```
boolean offEnd()  // Returns true if current is undefined.
int getIndex()    // If current element is defined, returns its position in
                  // this List, ranging from 0 to getLength()-1 inclusive.
                  // If current element is undefined, returns -1.
int getFront()    // Returns front element. Pre: !isEmpty().
int getBack()     // Returns back element. Pre: !isEmpty().
int getCurrent()  // Returns current element. Pre: !isEmpty(), !offEnd().
boolean equals(List L) // Returns true if this List and L are the same integer
                       // sequence.  Ignores the current element in both Lists.

// Manipulation Procedures
void makeEmpty()   // Sets this List to the empty state. Post: isEmpty().
void moveTo(int i) // If 0 <= i <= getLength()-1, moves current element
                   // marker to position i in this List.  Otherwise current
                   // element becomes undefined.
void movePrev()    // Moves current one step toward front element.  If the current
                   // element is already the front element, current element becomes
                   // undefined.  Pre: !isEmpty(), !offEnd().
void moveNext()    // Moves current one step toward back element.  If the current
                   // element is already the back element, current element becomes
                   // undefined.  Pre: !isEmpty(), !offEnd().
void insertFront(int data) // Inserts new element before front element.
                           // Post: !isEmpty().
void insertBack(int data)  // Inserts new element after back 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 deleteFront() // Deletes front element. Pre: !isEmpty().
void deleteBack()  // Deletes back element. Pre: !isEmpty().
void deleteCurrent() // Deletes current element, which then becomes undefined.
                     // Pre: !isEmpty(), !offEnd(); Post: offEnd()

// Other methods
List copy()  // Returns a new list which is identical to this list.  The current
             // element in the new list is undefined, regardless of the state of
             // the current element in this List.  The state of this List is
             // unchanged.

public String toString() // Overrides Object's toString method.  Returns a string
                         // representation of this List consisting of a space
                         // separated sequence of integers, with no trailing space.
```

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 the argument list.  The current element marker in
                 // the new list is undefined, regardless of the states of the
                 // current elements in the two lists.  The states of this list and
                 // the argument list are unchanged.
```

The underlying data structure for the List ADT is a doubly linked list. The List class should therefore contain a private inner Node class, which itself contains fields to store 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). The Node class should also define an appropriate constructor, as well as a `toString()` method. The List

class should contain three private fields of type Node which refer to the front, back, and current elements, respectively. The List class should also contain an int field to store the index (i.e. the position, counting from 0) of the current element. When the current element is undefined, an appropriate value for this field is `-1`, since that is what is to be returned by `getIndex()` in this case.

All of the above classes, fields, and methods are to be defined in a file called List.java. Your project will also include a separate file called ListTest.java defining a class called ListTest, which serves as a test client module for the List ADT. The main program for this project will be called Shuffle.java.

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

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 -f *.class $(JARFILE)
```

Note that this Makefile will compile all .java files in your current working directory, so it is a good idea to separate your programming projects into different directories. 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. The Makefile you turn in however must create an executable jar file called Shuffle, and must include a clean utility that removes all class files and the jar file.

You must also submit a README file for this (and every) assignment. README should list each file submitted, together with a brief description of its role in the project, and any special notes to myself and the grader. README is essentially a table of contents for the project, and nothing more. You are therefore to submit five files in all: List.java, ListTest.java, Shuffle.java, Makefile, and README. Points will be deducted if you misspell these file names, or if you submit jar files, class files, input or output files, or any other extra files for the project. Each file you turn in must begin with your name, user id, and assignment name (as comments if it is a source file).

Advice

The examples Queue.java and Stack.java on the website are good starting points for the List ADT module in this project. You are welcome to simply start with one of those files, rename things, then add functionality until the specifications for the List ADT are met. Students are often confused as to what to put in the program ListTest.java, since it is a required file with no specific contents. The rule is this: put enough calls to List operations in ListTest.java to convince the grader that you really did completely test the List module in isolation before using it in the Client module Shuffle.java. Better yet, actually use the file ListTest.java to test your List ADT, commenting out old test code rather than deleting it. You should first design and build your List ADT, test it thoroughly, and only then start coding your Shuffle class. Start early and ask questions if anything is unclear. Information on how to turn in your program is posted on the webpage.