CMPS 101
Algorithms and Abstract Data Types
Programming Assignment 1

Our goal in this project is to build an Integer List ADT that will be used to alphabetize the lines in a file. This ADT module will also be used (perhaps with some modifications) in future programming assignments so you should test it thoroughly, even though all of its features may not be used here. Begin by reading the handout ADT.pdf posted on the class webpage for a thorough explanation of the programming practices and conventions required in this class for implementing ADTs.

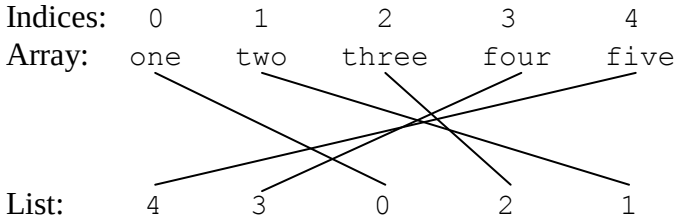
Program Operation

The main program for this project will be called Lex.java (for lexicographic i.e. alphabetical order.) Your List ADT module will be housed in a file called List.java, and will export its services to the client Lex.java. Each file will define one top level class, `Lex` and `List` respectively. The required List operations are specified in detail below. Lex.java will take two command line arguments giving the names of an input file and an output file respectively. The input file can be any text file. The output file will contain the same lines as the input, but re-arranged into alphabetical order. For example:

Input file:	Output file:
one	five
two	four
three	one
four	three
five	two

Lex.java will follow the sketch given below.

1. Check that there are two command line arguments. Quit with a usage message to `stderr` if more than or less than two strings are given on the command line.
2. Count the number of lines *n* in the file named by `arg[0]`. Create a String array of length *n* and read in the lines of the file as Strings, placing them into the array.
3. Create a List whose elements are the indices of the above String array. These indices should be arranged in an order that effectively sorts the array. Using the above input file as an example we would have.



Building the list in the right order can be accomplished by using a variation of Insertion Sort and the List operations defined below. Note that the String class provides a method called `compareTo()` that determines the lexicographic ordering of two Strings. If `s1` and `s2` are strings then:

`s1.compareTo(s2)<0` is true if and only if `s1` comes before `s2`
`s1.compareTo(s2)>0` is true if and only if `s1` comes after `s2`
`s1.compareTo(s2)==0` is true if and only if `s1` is identical to `s2`

4. Use the List to print the array to the file named by `arg[1]` in alphabetical order.

See the example FileIO.java to learn about file input/output operations. I will place a number of matched pairs of input-output files in the examples section, along with a python script that creates random input files along with their matched output files. Use these tools to test your program once it is up and running.

List ADT Specifications

Your list module for this project will be a bi-directional queue that includes a "cursor" to be used for iteration. Think of the cursor as highlighting or underscoring a distinguished element in the list. Note that it is a valid state for this ADT to have *no* distinguished element, i.e. the cursor may be undefined or "off", which is in fact its default state. Thus the set of "mathematical structures" for this ADT consists of all finite sequences of integers in which at most one element is underscored. A list has two ends referred to as "front" and "back" respectively. The cursor will be used by the client to traverse the list in either direction. Each list element is associated with an index ranging from 0 (front) to *n*-1 (back), where *n* is the length of the list. Your list module will define the following operations.

```
// Constructor
List() // Creates a new empty list.

// Access functions
int length()   // Returns number of elements in this list.
int getIndex() // Returns the index of the cursor element in this list, or
               // returns -1 if the cursor element is undefined.
int front() // Returns front element in this list. Pre: length()>0
int back()  // Returns back element in this List. Pre: length()>0
int getElement() // Returns cursor element in this list.
                 // Pre: length()>0, getIndex()>=0
boolean equals(List L) // Returns true if this List and L are the same integer
                       // sequence. The cursor is ignored in both lists.

// Manipulation procedures
void clear()  // Re-sets this List to the empty state.
void moveTo(int i) // If 0<=i<=length()-1, moves the cursor to the element
                   // at index i, otherwise the cursor becomes undefined.
void movePrev() // If 0<getIndex()<=length()-1, moves the cursor one step toward the
                // front of the list. If getIndex()==0, cursor becomes undefined.
                // If getIndex()==-1, cursor remains undefined. This operation is
                // equivalent to moveTo(getIndex()-1).
void moveNext() // If 0<=getIndex()<length()-1, moves the cursor one step toward the
                // back of the list. If getIndex()==length()-1, cursor becomes
                // undefined.  If index==-1, cursor remains undefined.  This
                // operation is equivalent to moveTo(getIndex()+1).
void prepend(int data) // Inserts new element before front element in this List.
void append(int data)  // Inserts new element after back element in this List.
void insertBefore(int data) // Inserts new element before cursor element in this
                            // List. Pre: length()>0, getIndex()>=0
void insertAfter(int data)  // Inserts new element after cursor element in this
                            // List. Pre: length()>0, getIndex()>=0
void deleteFront() // Deletes the front element in this List. Pre: length()>0
void deleteBack()  // Deletes the back element in this List. Pre: length()>0
void delete() // Deletes cursor element in this List. Cursor is undefined after this
              // operation. Pre: length()>0, getIndex()>=0

// Other methods
public String toString() // Overides Object's toString method. Creates a string
                         // consisting of a space separated sequence of integers
                         // with front on the left and back on the right.  The
                         // cursor is ignored.

List copy() // Returns a new list representing the same integer sequence as this
            // list.  The cursor in the new list is undefined, regardless of the
            // state of the cursor in this List. This List is unchanged.
```

The above operations are required for full credit, though 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 concat(List L) // Returns a new List which is the concatenation of
                    // this list followed by L.  The cursor in the new list
                    // is undefined, regardless of the states of the cursors
                    // in this list and L. The states of this list and L are
                    // unchanged.
```

The underlying data structure for this ADT will be a doubly linked list. The List class should therefore contain a private inner Node class. Node should contain fields for an int (the data), and two Node references (previous and next Nodes, respectively) which may be null. The Node class should also define an appropriate constructor and a `toString()` method. The List class should contain three private fields of type Node which refer to the front, back, and cursor elements, respectively. The List class should also contain an int field for the index of the cursor element. When the cursor is undefined, an appropriate value for this field is `-1` since that is what is to be returned by `getIndex()` in such a case.

All of the above classes, fields and methods will be placed List.java. Create a separate file called ListTest.java to serve as a test client for your ADT. Do not submit this file, just use it for your own tests. I will place another test client on the webpage called ListClient.java. Place this file in the directory containing your completed List.java, then compile and run it. The correct output is included in ListClient.java. You will submit this file (unchanged) with your project.

Makefile

The following Makefile should create an executable jar file called Lex. Place it in a directory containing List.java and Lex.java, then type `make` or `gmake` to compile your program.

```
# Makefile for CMPS 101 pa1
MAINCLASS  = Lex
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. You may 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 Lex, and must include a clean utility that removes all class files and the jar file from the current directory. See

my CMPS 12B website (http://ic.ucsc.edu/~ptantalo/cmps12b/Summer13/lab1.pdf) to learn basic information about makefiles.

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,
ListClient.java (unaltered)
Lex.java
Makefile
README

Points will be deducted if you misspell these file names, or if you submit jar files, class files, input/ output files, or any other extra files not specified above. 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 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. You should first design and build your List ADT, test it thoroughly, and only then start coding your Lex class. Start early and ask questions if anything is unclear. Information on how to turn in your program is posted on the webpage.