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 in Java and C.

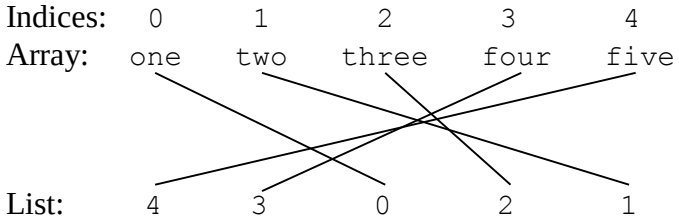
Program Operation

The main program for this project will be called Lex.java. Your List ADT module will be contained in a file called List.java, and will export its services to the client module Lex.java. Each file will define one top level class, `List` and `Lex` 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. The input can be any text file. The output file will contain the same lines as the input arranged in lexicographic (i.e. 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 `args[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.



To build the integer List in the correct order, begin with an initially empty List, then insert the indices of the array one by one into the appropriate positions of the List. Use the InsertionSort algorithm as a guide to your thinking on how to accomplish this. (Please read the preceding two sentences several times so that you understand what is required. You are *not* being asked to sort the input array using InsertionSort.) You may use only the List ADT operations defined below to manipulate the List. 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 constructed in (3) to print the array in alphabetical order to the file named by `args[1]`. Note that at no time is the array ever sorted. Instead you are *indirectly* sorting the array by building a List of indices in a certain order.

See the example FileIO.java to learn about file input-output operations in Java if you are not already familiar with them. 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 the number of elements in this List.
int index()   // If cursor is defined, returns the index of the cursor element,
              // otherwise returns -1.
int front()   // Returns front element. Pre: length()>0
int back()    // Returns back element. Pre: length()>0
int get()     // Returns cursor element. Pre: length()>0, index()>=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()     // Resets this List to its original empty state.
void moveFront() // If List is non-empty, places the cursor under the front element,
                 // otherwise does nothing.
void moveBack()  // If List is non-empty, places the cursor under the back element,
                 // otherwise does nothing.
void movePrev()  // If cursor is defined and not at front, moves cursor one step toward
                 // front of this List, if cursor is defined and at front, cursor becomes
                 // undefined, if cursor is undefined does nothing.
void moveNext()  // If cursor is defined and not at back, moves cursor one step toward
                 // back of this List, if cursor is defined and at back, cursor becomes
                 // undefined, if cursor is undefined does nothing.
void prepend(int data) // Insert new element into this List.  If List is non-empty,
                       // insertion takes place before front element.
void append(int data)  // Insert new element into this List.  If List is non-empty,
                       // insertion takes place after back element.
void insertBefore(int data) // Insert new element before cursor.
                            // Pre: length()>0, index()>=0
void insertAfter(int data)  // Inserts new element after cursor.
                            // Pre: length()>0, index()>=0
void deleteFront() // Deletes the front element. Pre: length()>0
void deleteBack()  // Deletes the back element. Pre: length()>0
void delete()      // Deletes cursor element, making cursor undefined.
                   // Pre: length()>0, index()>=0
```

```
// Other methods
public String toString() // Overrides Object's toString method. Returns a String
                         // representation of this List consisting of a space
                         // separated sequence of integers, with front on left.
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.
```

Notice that the above operations offer a coherent method for the client to iterate in either direction over the elements in a List. A typical loop in the client might appear as follows.

```
L.moveFront();
while(L.index()>=0){
   x = L.get();
   // do something with x
   L.moveNext();
}
```

To iterate back to front, simply replace `moveFront()` by `moveBack()` and `moveNext()` by `movePrev()`. One could just as well set this up as a for loop in either direction. Observe that in the special case where `L` is empty, the cursor is necessarily undefined so that `L.index()` returns `-1`, making the loop repetition condition initially false, and the loop executes zero times as it should on an empty List.

The underlying data structure for the List ADT will be a doubly linked list. The List class should therefore contain a private inner Node class which itself contains fields for an int (the data), and two Node references (the previous and next Nodes, respectively) which may be null. 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 referring to the front, back, and cursor elements, respectively. The List class should also contain private int fields storing the length of the List and the index of the cursor element. When the cursor is undefined, an appropriate value the index field is `-1`, since that is what is returned by `index()` 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 as a comment. You will submit this file unchanged with your project.

You are required to submit a Makefile that creates an executable jar file called Lex, which is the main program for this project. Include a clean target in your Makefile that removes Lex and any associated .class files to aid the grader in cleaning the submit directory. One possible Makefile will be included on the course webpage under `Examples/pa1`. You may alter this Makefile as you see fit to perform other tasks such as submit. See my CMPS 12B website (https://classes.soe.ucsc.edu/cmps012b/Spring16/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 will therefore submit five files in all:

List.java	written by you
ListClient.java	provided on webpage, do not alter
Lex.java	written by you
Makefile	provided on webpage, you may alter
README	written by you

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 source file you submit must begin with a comment block containing your name, CruzID, and the assignment name.

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 Lex.java. Start early and ask questions if anything is unclear. Information on how to turn in your project is posted on the class webpage.