CSE 101
Introduction to Data Structures and Algorithms
Programming Assignment 1

Our goal in this project is to build an Integer List ADT in C and use it to indirectly alphabetize the lines in a file. This ADT module will also be used (with some modifications) in future programming assignments, so you should test it thoroughly, even though not all of its features will 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 for implementing ADTs in C in this class.

Program Operation

The main program for this project will be called Lex.c. Your List ADT module will be contained in files called List.h and List.c, and will export its services to the client module Lex.c. The required List operations are specified in detail below. Lex.c will take two command line arguments giving the names of an input file and an output file, respectively.

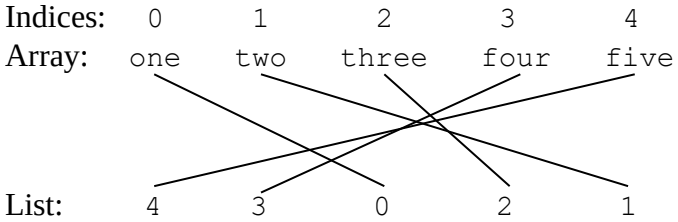
```
Lex <input file> <output file>
```

The input can be any text file. The output file will contain the same lines as the input, but arranged in lexicographic (i.e. alphabetical) order. For example:

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

Lex.c will follow the sketch given below.

1. Check that there are two command line arguments (other than the program name `Lex`). Quit with a usage message to `stderr` if more than or less than two command line arguments are given.
2. Count the number of lines *n* in the input file. Create a string array of length *n* and read in the lines of the file as strings, placing them into the array. (Allocate this array from heap memory using functions `calloc()` or `malloc()` defined in the header file `stdlib.h`. Do not use a variable length array. See the comments here for more on this topic.)
3. Create a List whose elements are the indices of the above string array. These indices should be arranged in an order that indirectly 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 Insertion Sort algorithm (section 2.1 of the text CLRS) 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 Insertion Sort.) You may use only the List ADT operations defined below to manipulate

the List. Note that the C standard library string.h provides a function called `strcmp()` that determines the lexicographic ordering of two Strings. If `s1` and `s2` are strings then:

`strcmp(s1, s2)<0` is true if and only if `s1` comes before `s2`
`strcmp(s1, s2)>0` is true if and only if `s1` comes after `s2`
`strcmp(s1, 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 output file. 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.c to learn about file input-output operations in C 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. You may 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 the list", 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 export a `List` type along with the following operations.

```
// Constructors-Destructors ---------------------------------------------------
List newList(void);       // Creates and returns a new empty List.
void freeList(List* pL);  // Frees all heap memory associated with *pL, and sets
                          // *pL to NULL.

// Access functions -----------------------------------------------------------
int length(List L); // Returns the number of elements in L.
int index(List L);  // Returns index of cursor element if defined, -1 otherwise.
int front(List L);  // Returns front element of L. Pre: length()>0
int back(List L);   // Returns back element of L. Pre: length()>0
int get(List L);    // Returns cursor element of L. Pre: length()>0, index()>=0
bool equals(List A, List B);  // Returns true iff Lists A and B are in same
                              // state, and returns false otherwise.

// Manipulation procedures ----------------------------------------------------
void clear(List L);      // Resets L to its original empty state.
void set(List L, int x); // Overwrites the cursor element's data with x.
                         // Pre: length()>0, index()>=0
void moveFront(List L);  // If L is non-empty, sets cursor under the front element,
                         // otherwise does nothing.
void moveBack(List L);   // If L is non-empty, sets cursor under the back element,
                         // otherwise does nothing.
void movePrev(List L);   // If cursor is defined and not at front, move cursor one
                         // step toward the front of L; if cursor is defined and at
                         // front, cursor becomes undefined; if cursor is undefined
                         // do nothing
void moveNext(List L);   // If cursor is defined and not at back, move cursor one
                         // step toward the back of L; if cursor is defined and at
                         // back, cursor becomes undefined; if cursor is undefined
                         // do nothing
```

```
void prepend(List L, int x); // Insert new element into L.  If L is non-empty,
                             // insertion takes place before front element.
void append(List L, int x);  // Insert new element into L.  If L is non-empty,
                             // insertion takes place after back element.
void insertBefore(List L, int x);  // Insert new element before cursor.
                                   // Pre: length()>0, index()>=0
void insertAfter(List L, int x);   // Insert new element after cursor.
                                   // Pre: length()>0, index()>=0
void deleteFront(List L);  // Delete the front element. Pre: length()>0
void deleteBack(List L);   // Delete the back element. Pre: length()>0
void delete(List L);       // Delete cursor element, making cursor undefined.
                           // Pre: length()>0, index()>=0

// Other operations -----------------------------------------------------------
void printList(FILE* out, List L);  // Prints to the file pointed to by out, a
                                    // string representation of L consisting
                                    // of a space separated sequence of integers,
                                    // with front on left.
List copyList(List L);  // Returns a new List representing the same integer
                        // sequence as L. The cursor in the new list is undefined,
                        // regardless of the state of the cursor in L. The state
                        // of L 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 concatList(List A, List B);  // Returns a new List which is the concatenation of
                                  // A and B.  The cursor in the new List is undefined,
                                  // regardless of the states of the cursors in A and B.
                                  // The states of A and B are unchanged.
```

Notice that the above operations offer a standard 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.

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

To iterate from back to front, replace `moveFront()` by `moveBack()` and `moveNext()` by `movePrev()`. One could just as well set this up as a for loop. Observe that in the special case where `L` is empty, the cursor is necessarily undefined, so that `index(L)` returns `-1`, making the loop repetition condition initially false. Therefore the loop executes zero times, as it should on an empty List. It is required that function `index()` be implemented efficiently, meaning that it should not itself contain a loop.

The underlying data structure for the List ADT will be a doubly linked list. The file List.c should therefore contain a private (non-exported) struct called NodeObj and a pointer to that struct called Node. The struct NodeObj should contain fields for an int (the data), and two Node references (the previous and next Nodes, respectively.) You should also include a constructor and destructor for the private Node type. The private (non-exported) struct ListObj should contain fields of type Node referring to the front, back and cursor elements, respectively. ListObj should also contain int fields for the length of a List, and the index of the cursor element. When the cursor is undefined, an appropriate value for the index field is `-1`, since that is what is returned by function `index()` in such a case. Study the examples Queue.c and Stack.c on the course webpage, and feel free to use either file as a starting point for List.c.

A sample test client will be placed on the webpage in Examples/pa1 called ListClient.c, which you will not submit. This program should be considered to be a weak test of your List ADT. Its correct output is included as a comment at the end of the file. Create your own test client for the List ADT called ListTest.c, and submit it with this project. It should contain your own tests of all ADT operations.

You are required to submit a Makefile that creates an executable binary file called Lex, which is the main program for this project. Include a `clean` target in your Makefile that removes Lex and any associated .o 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 lab assignment 1 from my (now defunct) CMPS 12B https://classes.soe.ucsc.edu/cmps012b/Spring19/lab1.pdf to learn basic information about Makefiles.

Note that the compile operations mentioned in the above Makefile call the gcc compiler with the flag `-std=c17`. It is a requirement of this project (and all other C programs) that it compile without warnings or errors under gcc (with the c17 flag), and run properly in the Linux computing environment on the UNIX Timeshare unix.ucsc.edu provided by ITS. Your C programs must also run without memory leaks. Test them using `valgrind` on unix.ucsc.edu by doing

```
valgrind program_name argument_list.
```

You must also submit a README file for this (and every) assignment. README will list each file submitted, together with a brief description of its role in the project, along with 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 six files in all:

List.h	written by you
List.c	written by you
ListTest.c	written by you
Lex.c	written by you
Makefile	provided on webpage, alter as needed
README	written by you

Points will be deducted if you misspell these file names, or if you submit .o files, executable binary files, input-output files, or any other files not specified above. Each source file you submit must begin with a comment block containing your name, CruzID, and the assignment name (pa1 in this case).

Advice

The examples Queue.c and Stack.c 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. You should first design and build your List ADT, test it thoroughly, and only then start coding Lex.c. Start early and ask questions if anything is unclear. Information on how to turn in your project is posted on the class webpage.