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 unique words in a text 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 ADTs-in-C.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 Words.c. Your List ADT module will be implemented in files called List.h and List.c, and will export its operations to the client module Words.c. The required List operations are specified in detail below. Words.c will take two command line arguments giving the names of an input file and an output file, respectively.

```
Words <input file> <output file>
```

The input can be any text file. Words.c will parse the input file into individual words, discarding duplicate words. It will then place the unique words into a string array by order of first occurrence in the input file. On the first line of the output file will be a list containing the indices of that array arranged in an order that indirectly sorts the array. The second line will be blank, and the remaing lines will contain all of the unique words from the input file in alphabetical order, one word per line. For example:

Input file:
```
One, two + :)
two::One,,,three
three/two
Four
one, five=three
```

```
        0    1     2     3     4     5
```
Array: `[One, two, three, Four, one, five]`

Output file:
```
(3, 0, 5, 4, 2, 1)

Four
One
five
one
three
two
```

Observe that Words.c determines the words in a text file by discarding all non-alphabetic characters. Specifically, a word is defined to be a maximal substring of a line not containing any of the characters in the string literal:

```
" \t\n\r\\\"\',<.>/?;:[{]}|`~!@#$%^&*()-_=+0123456789"
```

It will be very helpful to study the example FileIO.c posted in Examples/C/FileIO on the class webpage, which illustrates, among other things, tokenization of a text file using the C function strtok().

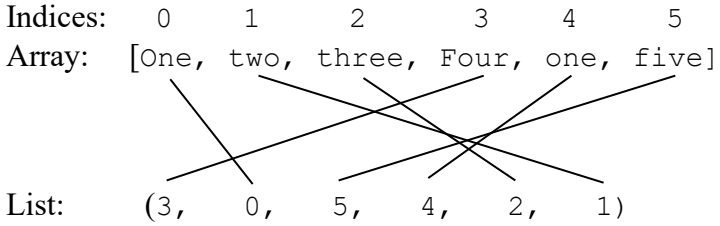
Note that we distinguish between upper and lower case letters, so that '`one`' and '`One`' are regarded as different words. The alphabetization will be done using the C function strcmp(), also illustrated in FileIO.c. In particular, upper case letters come before lower case in the alphabetization.

Words.c should follow the outline given below.

1. Check that there are two command line arguments (other than the program name `Words`). Quit with a usage message to `stderr` if more than two or less than two command line arguments are given.

2. Open the input file and parse each line into individual words. Place the new words in a string array and discard any repeated words. Expand the string array as needed to accommodate all unique words in the file. Study the array based Queue example at Examples/C/Queue/Array to see how to expand an array by using the C function realloc().

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 List of indices, begin with an 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. *Do not* sort the string array using Insertion Sort, or any other algorithm.) You may use only the List ADT operations defined below to manipulate the List. Note that the C function strcmp() determines the alphabetic ordering of two strings `s1` and `s2` as follows:

   `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. Open the output file, print the List constructed in (3), print a blank line, then use the List to print the string array in alphabetical order. Note again that at no time is the string array sorted. Instead, you *indirectly* sort the array by building a List of array indices in a certain order.

A number of matched pairs of input-output files have been placed in Examples/pa1, along with a Makefile for the project which you may alter as you see fit. Use these tools to help test your program once it is up and running.

List ADT Specifications

Your List ADT 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 distinguished. 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 position 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 ----------------------------------------------------

// newList()
// Creates a new empty list.
List newList();

// freeList()
// Frees heap memory associated with *pL, sets *pL to NULL.
void freeList(List* pL);


// Access functions -----------------------------------------------------------

// length()
// Returns the number of elements in L.
int length(List L);

// position()
// If cursor is defined, returns the position of the cursor element, otherwise
// returns -1.
int position(List L);

// front()
// Returns front element. Pre: length()>0
ListElement front(List L);

// back()
// Returns back element. Pre: length()>0
ListElement back(List L);

// get()
// Returns cursor element. Pre: length()>0, position()>=0
ListElement get(List L);

// equals()
// Returns true if A and B are the same integer sequence, false otherwise. The
// cursor is not altered in either List.
bool equals(List A, List B);


// Manipulation procedures ----------------------------------------------------

// clear()
// Resets L to its original empty state.
void clear(List L);

// set()
// Overwrites the cursor element's data with x. Pre: length()>0, position()>=0
void set(List L, ListElement x);

// moveFront()
// If L is non-empty, places the cursor under the front element, otherwise does
// nothing.
void moveFront(List L);

// moveBack()
// If List is non-empty, places the cursor under the back element, otherwise
// does nothing.
void moveBack(List L);

// movePrev()
// If cursor is defined and not at front, moves cursor one step toward front of
// L, if cursor is defined and at front, cursor becomes undefined, if cursor is
// undefined does nothing.
void movePrev(List L);
```

```
// moveNext()
// If cursor is defined and not at back, moves cursor one step toward back of
// L, if cursor is defined and at back, cursor becomes undefined, if cursor is
// undefined does nothing.
void moveNext(List L);

// prepend()
// Insert new element into L.  If List is non-empty, insertion takes place
// before front element.
void prepend(List L, ListElement data);

// append()
// Insert new element into L.  If List is non-empty, insertion takes place
// after back element.
void append(List L, ListElement data);

// insertBefore()
// Insert new element before cursor. Pre: length()>0, position()>=0
void insertBefore(List L, ListElement data);

// insertAfter()
// Inserts new element after cursor. Pre: length()>0, position()>=0
void insertAfter(List L, ListElement data);

// deleteFront()
// Deletes the front element. Pre: length()>0
void deleteFront(List L);

// deleteBack()
// Deletes the back element. Pre: length()>0
void deleteBack(List L);

// delete()
// Deletes cursor element, making cursor undefined. Pre: length()>0, position()>=0
void delete(List L);


// Other operations -----------------------------------------------------------

// printList()
// Prints a string representation of L consisting of a comma separated sequence
// of integers, surrounded by parentheses, with front on left, to the stream
// pointed to by out.
void printList(FILE* out, List L);

// copyList()
// 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
// List L is unchanged.
List copyList(List L);

// join()
// Returns the concatenation of A followed by B. The cursor in the new List is
// undefined, regardless of the states of the cursors A in and B. The states of
// A and B are unchanged.
List join(List A, List B);

// split()
// Removes all elements before (in front of but not equal to) the cursor element
// in L. The cursor element in L is unchanged. Returns a new List consisting of
// all the removed elements. The cursor in the returned list is undefined.
// Pre: length(L)>0, position(L)>=0
List split(List L);
```

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

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

To iterate back-to-front, replace `moveFront()` by `moveBack()` and `moveNext()` by `movePrev()`. One could also set this up as a for loop. Observe that in the special case where `L` is empty, the cursor is necessarily undefined, so that `position(L)` returns `-1`, making the loop repetition condition initially false. Therefore the loop executes zero times, as should be the case on an empty List. It is required that function `position()` 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. List.c will therefore contain a private (i.e. non-exported) struct called Node that contains fields for an int (the data), and two Node* pointers (references to the previous and next Nodes, respectively.) You may also include a constructor 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 position of the cursor element. When the cursor is undefined, an appropriate value for the position field is `-1`, since that is what is returned by function `position()` 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 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 Words, which is the main program for this project. Include a `clean` target in your Makefile that removes all binary files, including the executables Words, ListClient and any associated .o files. A Makefile is also included in Examples/pa1 which you may taylor to your own needs. The course webpage contains some links to some basic Makefile tutorials.

The compile operations included in the given Makefile call the gcc compiler with the flags `-std=c17 -Wall`. It is a requirement of this project (and all other C programs in this class) that it compile without warnings or errors under gcc (with the above flags), and run properly in a Linux computing environment. Your C programs must also run without memory errors. Test using the Linux program `valgrind` by doing

```
valgrind program_name argument_list.
```

You must also submit a README.md file for this, and every programming assignment. README.md will list each file submitted, together with a (very) brief description of its role in the project, along with any special notes to the grader. README.md 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
Words.c	written by you
Makefile	provided on webpage, alter as needed
README.md	written by you

Points will be deducted if you misspell these file names, 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 in Examples/C are good starting points for the List.c, and 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. The example FileIO.c is also a good starting point for Words.c. First design and build your List ADT, test it thoroughly, and only then start coding Words.c. Start early and ask questions of myself and the TAs if anything is unclear. Information on how to turn in your project is posted on the class webpage.