CMPS 101 - Programming assignment 2 - S14
Implementing a List ADT
Handed out 04-07-14, revised 04-12 Due Friday 04-18-14

April 13, 2014

1 Introduction

Retailers like Amazon keep a lot of information on their customers. Amongst other things, this allows them to issue recommendations of the form ”other customers buying that book also bought this other book”. This assignment will build some of the groundwork for such an application.

The three goals of this assignment are:

- Further practice with ANSI C. For those of you less familiar with ANSI C, this will give you chance to get practice with it.
- To implement a List ADT in the style of the ADT handout. Not only will this give you a better idea of how ADTs work, but this List ADT should be something that will be useful in your future classes.
- To lay some groundwork for more interesting applications (like recommending books to customers).

These programming assignment is to be done in teams of two students. Students should rotate team members for different assignments. Team members should collaborate actively on the assignment, working closely together and ensuring that both team members understand why the program is being written the way it is. The resources directory has a handout on pair programming that might be helpful.

This assignment is due in two weeks. To be on track, you should have almost all of the List ADT working after one week.

Key Skills and concepts: pointers and pointer manipulations, malloc, free, assert, linked lists, ADTs and information hiding, `.h` and `.c` and makefiles, pre- and post-conditions.

2 Assignment Description

In this assignment, you will implement a List ADT and use it to develop a simple data processing application. Please read the ADT handout(ecommons→handouts) for more information on ADTs).

The mathematical set of objects associated with this ADT is a sequence of elements, where usually, but not always, one of the elements is marked as the *current element*. In principle, the elements could be any type, but for this assignment they are integers (use C type `long` to hold these integers as in the application they may be too large to fit in standard `int` variables).

The various access operations you are to implement include: checking if the list is empty, checking if there is a current element, and returning the value of the current element, and checking if the current element is the first or last element of the list. The manipulation operations you are to implement include: adding an additional element to the start or end of the list, inserting an element immediately after the current element, moving the mark so that a different element becomes the current element in the sequence, and deleting the current element.

In the first part you will play the role of the ADT implementor and implement for the List ADT specified by the operations given below. The list will need to be developed using a doubly linked list data structure (see section 10.2 of the text for an example doubly-linked list). The linked list should be defined in the `.c` file using a linked list header and a node structure. The `Node` type should at minimum contain the following. You may add others element as per your need.

```
typedef struct NodeStruct {
    long data;
    struct NodeStruct* next;
    struct NodeStruct* prev;
} NodeStruct;
```

You may find it helpful to define a `NodePtr` type and use that instead of `NodeStruct*`.

The List header at the minimum should contain the following. You can define additional members for the structure if you wish.

```
typedef struct ListStruct {
    NodePtr first;
    NodePtr last;
    NodePtr current;
} ListStruct
```

The `ListHndl` type is the handle for the list, it should be a pointer to the ListSturct structures. Note that the list handle `ListHndl` needs to be declared appropriately in the header file so clients can create variables to hold handles.

```
typedef struct ListStruct*  ListHndl;
```

2.1 List ADT operations

The required operations for your ADT are as follows.

```
/*** Constructors-Destructors ***/
ListHndl  newList( void );
void freeList(ListHndl* L);  // Pre: L has been created with newList.

/* NOTE:  all Access functions and Manipulation procedures also have the
   precondition that L has been initialized by newList().
   */

/*** Access functions ***/
int isEmpty(ListHndl L); // returns true is list is empty else returns false.
int offEnd(ListHndl L);  // returns true is current == NULL
int atFirst(ListHndl L); // returns true if current == first and !offEnd()
int atLast(ListHndl L);  // returns true if current == last and !offEnd()
long getFirst(ListHndl L); // return the first element; pre: !isEmpty()
long getLast(ListHndl L);  // return the last element; pre: !isEmpty()
long getCurrent(ListHndl L); // return the current element pre: !offEnd()

/*** Manipulation procedures ***/
void makeEmpty(ListHndl L); // delete all elements from the list,
                            // making it empty. Post: isEmpty(), offEnd() are both true
void moveFirst(ListHndl L); // without changing list order, make the first element
                            // the current one.  Pre: !isEmpty(); Post: !offEnd()
void moveLast(ListHndl L);  // without changing list order, make the last element
```

```
                            // the current one. Pre: !isEmpty(); Post: !offEnd()
void movePrev(ListHndl L);  // move the current marker one element earlier in the list
                            // Pre: !offEnd();   Post: offEnd() only if atFirst() was true
void moveNext(ListHndl L);  // move the current marker one element later in the list.
                            // Pre: !offEnd();   Post: offEnd() only if atLast() was true

void insertAtFront(ListHndl L, long data);  // Inserts new element before first
                            // Post: !isEmpty(); doesn't change current element
void insertAtBack(ListHndl L, long data);    // Inserts new element after last one
                            // Post: !isEmpty(); doesn't change current element
void insertBeforeCurrent(ListHndl L, long data); // Inserts new element before current one
                            // Pre: !offEnd(); Post: !isEmpty(), !offEnd(), !atFirst()

void deleteFirst(ListHndl L); // delete the first element. Pre: !isEmpty()
void deleteLast(ListHndl L);  // delete the last element. Pre: !isEmpty()
void deleteCurrent(ListHndl L); // delete the current element.
                             // Pre: !isEmpty(), !offEnd(); Post: offEnd()

/*** Other operations ***/
void printList(FILE* out, ListHndl L);  // prints out the list with the
                             // current element marked, helpful for debugging
```

You can "mark" the current element in printList by surrounding it with some kind of parens or brackets (like `[17]`) or follow it with a special symbol (like `17*`).

Feel free to support additional functions with your list ADT like `insertAfterCurrent(ListHndl L, long data)` if you want.

I hope the meaning of the operations is clear from the descriptions. For example, if list `L` is 11, 222, $\underline{7}$, 4 where underlining indicates the current element, then the call `insertBeforeCurrent(L, 5)` would result in `L` getting the state 11, 222, 5, $\underline{7}$, 4. Then a call to `moveNext(L)` would change the state of `L` to 11, 222, 5, 7 , $\underline{4}$. Another call to `moveNext(L)` would make `L` be 11, 222, 5, 7, 4, with no current element, so `offEnd(L)` would become true.

You will need to implement the List ADT as a Module. The implementation details should be in `List.c`. The interfaces listed above should be exported for public use through a header file, `List.h`. (Read carefully the ADT handout to understand the role of header files).

When implementing the ADT you can use the other ADT functions when helpful, and it can be *very* useful to add "helper functions" that are defined and used by the implementer in the ADT `.c` file, but not in exported to the ADT's client.

As described in lecture, you should incrementally construct a driver program, `Listdr.c`, which should test out the List functions as you implement them. The ADT has a lot to implement, but each function should be manageable if they are taken one at a time.

2.2 Application of List ADT

The application you will need to develop is as follows. There are *n* customers who purchase various books from a website. The customers are identified by a unique customer ID and books by a unique book ID (ISBN number or something similar to that). Your program should prompt for and accept a file name from standard input. That data file will contain which books were purchased by which customers as described below. The first line of the data file is *n*, the number of customers in the system, and the second line is the number of book purchases. After that each line contains two numbers: a customer ID (between 1 and *n*) and a book ID representing a purchase for that customer. The numbers on a line are separated by white space. You can consider that this information is produced "on-line" so that customers can buy books in any order.

An example input file is

```
3     // number of customers
9     // number of purchases
2  3453
1 3453
1 234443
2 41
1 30
3 11323
3 2
1 11323
3 24339
```

The book IDs may be quite long (Universal product codes are 13 digits, and to big to fit into an `int`, so you should use a 'long' data type to hold book IDs). The customer IDs will range from 1 to the number of customers.

In addition to the List ADT you should write a `store.c` file containing the main program. This main program should read the first two lines of the input file, create and initialize an array of linked lists indexed by customer ID. The linked list for a customer should should store the books purchased by that customer. After reading the entire input file (and storing all of the purchases), the main program should print out the purchases in condensed form using one line for each customer as in the following output. If the program encounters bad input (like lines without the required number of numbers, letters instead of numbers, or insufficient number of lines) then it should halt with a bad input error message. Extra characters after the number(s) on a line, or extra lines should be ignored.

```
customer#   books purchased
1      30      3453      11323      234443
2      41      3453
3      2      11323  24339
```

Note that the following output printed out the books in sorted order. To output the books in sorted order you should use insertion sort to insert the new purchase in the correct place in the list. For the basic assignment you can print out the books in any order, but it is worth 1 point to print out the books in sorted order. I **strongly** suggest you get the basic assignment working (and save your working code) before trying to output the books in sorted order, probably by using insertion sort to insert new purchases into the customer's list.

2.3 Submittal Information

You will need to submit the .c files, .h files, a makefile that correctly creates your store and driver programs, and a README file which describes the contents of all files. You will need to make sure that the program compiles and runs correctly on the UCSC unix systems. You will lose points for compile issues. Do not assume that graders will fix compile issues. If the program does not compile, graders may choose to grade the code on non-execution aspects and this will significantly cost your grade.

One partner should submit the entire assignment. That will include at least the following six files. List.c, List.h, Listdr.c, store.c, makefile and a README. If your program needs other files to compile (like myIncludes.h) then they must be submitted also. Each file should have a header comment indicating what the file is for and who wrote it (both partner's names and accounts@ucsc.edu). zip all the files together into a file named lastNameFirstInitialProg1.zip and then submit this zip file through ecommons.

The other partner should submit the README file as a text file (not zip'ed), and the header comment in the README file will identify the partner submitting the entire assignment. I suggest you watch you partner submit the assignment to ensure that its submitted correctly and on time.

2.4 Grading Guide

15 points total:

- 1 point for submitting separate ADT Files with information hiding, good comments (including header comments and pre and post conditions) and modularity
- 1 point for well organized, readable code and generally good style
- 1 point for good memory management (including free-ing the allocated storage for list nodes and list headers)
- 1 point for detecting incorrect input
- 10 points for correctly outputting the books purchased by each customer (assuming the List ADT and design requirements are met)
- 1 point for outputting the books purchased by each customer in sorted order