AC21008 2018 Assignment 2 Specification: Doubly-linked List with Iterator

Due: Beginning of Week 8: Monday, 29th October 2018 @ 2359 (11:59pm).
This assignment is worth 15% of your final grade.

All submissions are to be made to My Dundee. You must upload a single .c file that contains the implementation of your code as described below.

In this assignment, you will provide an implementation of the functions required for the following data structure: a doubly-linked list with iterator.

To complete your assignment you have been provided with some starter files to help you:

- A header file for the doubly-linked list (`list.h`) which defines the data structures you require for the `List` and `ListNode` along with prototypes / declarations of the functions that you are being asked to implement. The header file also defines the error codes that you can use when returning values from your functions.
- A tester program (`list_tester.c`) that you can use during your development to run a (fairly exhaustive) set of tests on your code to make sure it is working OK.
- A simple tester program (`list_simple_tester.c`) which runs a basic set of tests on your code and provides a general example of how to instantiate and use the doubly-linked list data structure.

Your task is to provide an implementation of the functions defined in the header file (and also described below). You will implement these in a single source file: **`list.c`**.

When you submit your assignment, submit your list.c file only. We will ONLY look at your list.c file. You may modify the list.h and tester files provided, but we do not recommend this. We will use our own copies of these, not yours, to grade your submission

We will be grading your submission automatically using our tester file (extended to provide detailed feedback and grades) by building your *list.c* file (with our *list.h*) and running our tests on it (with our *list_tester.c*). Lack of adherence to this spec will cost you a lot of marks. Full adherence will get you 100%.

You will recall from our lectures that a doubly linked list data structure allows both forward and backward traversal of the list (using 'next' and 'prev' fields inside each ListNode). It is also possible to store the current position in the list and move the current position backwards and forwards (using a 'curr' pointer). Using 'curr', your list will provide functionality for inserting before and after the current position, moving the current position within the list, getting data at the current position, and removing a node from the list at the current position. To help you get started, please review the following:

1. Lecture slides. In lectures, we walked through the design of how this data structure is expected to work. From this you should be able to infer the steps that you need to go through to implement the list functions in code as well as considering any checks you need to do for special cases. The lectures provided some hints to get you started with your code too.
2. The Stack sample code and your code from your first assignment: double-ended queue. The code that you require for the doubly-linked list will have great similarities to the code of these other data structures so these will be an effective starting point for you to build upon.

The header file *list.h* contains a number of *#define* statements for error codes. You should use these in the functions that you implement. So, when your function is called, it must return a value to indicate its success or otherwise. Use the values defined in the header file for this – our automated testing system relies on you doing this. Please see the sample code for the Stack data structure for an example of using these error codes.

We expect you to practice defensive C programming skills. In particular, you will not get far in our tester program without verifying parameters that are passed to your function, e.g. to make sure they are not null, etc. The first tests we run in our marking system is to check for illegal or invalid input parameters. Remember to check for special cases too, e.g. what if the list is empty when you try to operate on it? Remember to initialise your pointer variables correctly before using them. Please see the Stack sample code for examples of error checking.

You will implement the following functions. In order to make the most use of our tester program, we recommend implementing '`listInit()`' first, and moving down from there. We also recommend opening the *list.h* file to understand the *#defines* and *structs* provided while you read through the specification below.

[10 marks] *int listInit(List **listPtr)*: This function initializes a new, empty List and records its pointer in (*listPtr). **NOTE:** use the 'myMalloc' function to allocate memory instead of the standard malloc function – see details later below.

[15 marks] *int insertBeforeCurr(List *listPtr, int data)*: This function creates a new ListNode, stores 'data' in it, and adds the new node immediately **before** the current position in the list. Use 'myMalloc' to request memory.

[15 marks] *int insertAfterCurr(List *listPtr, int data)*: This function creates a new ListNode, stores 'data' in it, and adds the new node immediately **after** the current position in the list. Use 'myMalloc' to request memory.

[10 marks] *int currToPrev(List *listPtr)*: This function moves 'curr' one node backward in the List.

[10 marks] *int currToNext(List *listPtr)*: This function moves 'curr' one node forward in the list.

[10 marks] *int getDataAtCurr(List *listPtr, int *dataPtr)*: This function obtains the data from the current node in the list and stores it into the memory address of the variable provided by 'dataPtr'.

[20 marks] *int removeAtCurr(List *listPtr, int *dataPtr, int moveForward)*: This function obtains the data from the current node in the list and stores it into the memory address of the variable provided by 'dataPtr'. Then, it removes the current node from the list. The final parameter determines what to do with the 'curr' pointer after the node is removed. If 'moveForward' is true (i.e., not 0), then 'curr' is shifted one node forward in the list after removal of the current node, otherwise 'curr' is shifted one node backward.

[10 marks] *int listFree(List *listPtr)*: This function frees all memory used by the list including any Nodes that it contains and the list itself. No need to reinvent the wheel – feel free to use your 'removeAtCurr()' function from above to help you if you wish.

You will notice one more function prototype at the bottom of *list.h*:

void myMalloc(size_t size);*

We define this function in *list_tester.c*, so you should not. ‘myMalloc’ allows us to simulate malloc failures so we can thoroughly test your programs. You MUST use ‘myMalloc’ instead of using ‘malloc’ any time you need allocate memory for your List or ListNodes. ‘myMalloc’ has exactly the same signature as the regular ‘malloc’ (it takes a *size_t* and returns a *void**, including NULL if allocation failed).

NOTE: please make sure that you place a comment at the top of your list.c file before you submit it which includes your full name, matriculation number and module code. So, it may appear thus for example:

```
/*
   Name: Craig Ramsay
   Matric number: 234545656
   Module code: AC21008
*/
```

You can place this before any <include> statements.

Additional Notes:

- Please use the ‘-Wall –Wextra and –pedantic’ options with gcc to help you identify errors and warnings. Consider using a Makefile to make that easy

Further details about the ‘curr’ pointer
In case it isn’t clear what value the ‘curr’ variable should have after certain list operations have taken place, please find a summary below.

- Initialising the list. ‘curr’ will be NULL (as will the head and tail of list too) – an empty list.
- Inserting the first node into the list. ‘curr’ will point to the newly added node, it is the only node in the list.
- Inserting a new node *before* the current position. ‘curr’ will remain unchanged after the insertion, pointing to the same node it was pointing to before the insertion.
- Inserting a new node *after* the current position. ‘curr’ will remain unchanged after the insertion, pointing to the same node it was pointing to before the insertion
- Moving the current position backwards and forwards. ‘curr’ will move one node position backwards or forwards accordingly.
- Getting data from the current node. No change to ‘curr’.
- Removing a node from the current position. Several cases to consider:
    - Removal from a single item list. The list will become empty after. ‘curr’ and other fields will be set to NULL to indicate an empty List.
    - Two or more items in the list and ‘curr’ is currently at the head of the list. The head node will be removed. Both ‘curr’ and ‘head’ will be shifted to next node in the list after the current head is removed.
    - Two or more items in the list and ‘curr’ is currently at the tail of the list. The tail node will be removed. Both ‘curr’ and ‘tail’ will be shifted to the previous node in the list before the current tail is removed.

    - Two or more items in the list and ‘curr’ is neither at the head or tail of the list. When the node is removed, the ‘curr’ pointer will have to shift to now point to the previous node or next node instead. A parameter to the ‘remove’ function (*moveForward*) indicates what to do with ‘curr’.
- Free the list. This deletes the entire list. ‘curr’ will not exist afterwards.

How to user the tester files
You are provided with a header file (list.h) and two tester files (list_tester.c and list_simple_tester.c).

You need to implement code for the doubly-linked list functions in a file ‘list.c’ – remember to include list.h into it. NOTE: you should place the function implementations in your file only, you don’t need anything else in there (e.g. such as main() function).

Both of the tester files contain a main() function which will automatically run a series of tests on your code and then report the results. You just need to compile the different files together.

Example: to use the simple tester. Make sure all of the above files are in the same folder on your PC then run the following command:

```
gcc list.c list_simple_tester.c
```

This will compile your own code (list.c) and the simple tester program and combine them together to product an output file of ‘a.out’. You can then run ./a.out to run your program and see the test results.