CS 34841 – A Linked List in C

1 Introduction

The purpose of this lab is to design, code, and test a classic data structure within the confines of the C programming language. While not strictly and "operating systems" exercise, it is intended to familiarize you with the development process on Linux and provide additional practice with dynamic memory.

2 Prerequisites

Your VM should be operational, and you should have a development environment (programmer's editor or IDE) setup and tested (i.e. you have written a hello world program). This assignment will also make use of the program 'valgrind' which may need to be installed. You can do this at the command line with the commands:

```
sudo apt update
```

```
sudo apt install valgrind
```

3 References

It will certainly help to review the concepts of linked lists. Dust of your data structures text, or, there are plenty of online resources. Do be careful, however, as there are many variations and approaches to implementing linked lists. Be sure your solution meets the requirements of the assignment.

4 The Exercise

This linked list will be doubly linked with a head and tail reference. Thus, we can push and pop to either end of the list, and, insert and remove any node from any position. The list can also be used as a queue as well as a stack. The list will be designed to only hold strings (c-style strings, that is), and will have ownership semantics. This means that any strings submitted to be stored in the list must be copied into memory dedicated to the list.

A header file is supplied (in Box folder) that represents the "public" interface that you must implement. Each method is documented within that header file. You are not allowed to change this header file in any way. You should supply ONE source file with all of the list implementation (llist.c). Within that .c file you may declare additional 'static' helper functions if needed.

In addition to the list implementation, supply a "driver" that will completely test your list implementation.

Upon reviewing the header file, you may have some impressions. First, the list is rather crude. The methods in this "public" interface only manage the list's memory and organization. The user has full access to the list internals. Of course, some of this is an artifact of the C language that does not include the protections provided by languages such as Java and C++. Also note that the user must pass either a list pointer or a node pointer to every list management function. Again, a necessity due to the lack of object-oriented language features.

Another observation is that the user is responsible for iterating the list. Could we have provided an iterator? Certainly. You are welcome to build this feature if you wish, but it is not required. Note, if you do implement an iterator, you may need additional source files, of course. However, you may not change the design of the list itself. It should still operate as documented without the iterator.

How is this list to be used? Below is some sample code you can use to get started, but this should not be the extents of your testing efforts.

```
#include <stdio.h>
#include "llist.h"

/* A simple function to dump the contents of a list to the console */
void dump(list* myList)
{
	// node reference - start at head
	node *aNode = myList->head;

	// print size
	printf("\nList Has: %d\n",llSize(myList));

	// iterate list
	while(aNode)
	{
		printf("%s\n",aNode->string);
		aNode = aNode->next;
	}

}


int main(void)
{
	list myList;

	llInit(&myList);

	llPushBack(&myList,"Four");
	llPushFront(&myList,"Two");
	llInsertAfter(&myList,myList.head,"Three");
	llInsertBefore(&myList,myList.head,"One");

	dump(&myList);

	llPopBack(&myList);

	dump(&myList);

	llRemove(&myList,myList.head->next);

	dump(&myList);

	llPopFront(&myList);

	dump(&myList);

	printf("\nClearing List\n");

	llClear(&myList);

	dump(&myList);

	return 0;
}
```

5 Development

Let's keep this as simple as possible. You should have already picked an editor or IDE to use. I suggest compiling at the command line for now even though a good IDE is arguably more productive. I do not suggest editing in Windows. I highly suggest using git to create a repository for your code.

You can invoke the compiler at the command line directly:

`gcc hello.c`

This command will build hello.c and product an executable name a.out in the same directory. Errors and warning will be reported to the console. You can run a.out by issuing the command:

`./a.out`

Here, the '.' refers to the current directory.

We can get fancier. Instead of the default output name, we can specify the name of the executable:

`gcc -o hello hello.c`

If we have more than one source file, just list them all:

`gcc -o hello hello.c goodbye.c`

By default, gcc doesn't report all possible warnings. If you want more warnings (hint, you do), do this:

`gcc -Wall -o hello hello.c goodbye.c`

Better yet, we should be using a makefile. Once we have a makefile, all we must do is type 'make' at the command line and the compiler is invoked only for source files that have changed since the last build. A sample makefile is shown below, and must be customized for your project if you wish to use it. The necessary customizations should be obvious. Note that this makefile will not react to changes to header files. If you change a local header file, you will need to **`make clean`**, then **`make`**.

```
CC=gcc
CFLAGS=-c -Wall
LDFLAGS=
SOURCES=source1.c source2.c
OBJECTS=$(SOURCES:.c=.o)
EXECUTABLE=exename

all: $(SOURCES) $(EXECUTABLE)

$(EXECUTABLE): $(OBJECTS)
	$(CC) $(LDFLAGS) $(OBJECTS) -o $@

.c.o:
	$(CC) $(CFLAGS) $< -o $@

clean:
	 rm -rf $(OBJECTS) $(EXECUTABLE)
```

6 Testing and Debugging

As stated above, you must write a driver that completely tests your linked list. So what is complete testing? How do you go about this? You will have to put some thought into this and use your experience to guide you. It would make sense that you will have to call every method at least once, probably many times under different conditions. Look for boundary cases. For example, be sure to test **`llPopFront()`** with an empty list as well as a non-empty list.

Given that we are using C and not Java, there are few protections we can build in to prevent misuse of the list. For example, there is no perfect way to make sure **`llInit()`** is called before adding items to the list.

So, you are calling all of the methods, and your program crashes. What now? Well, you have to track down where it is crashing. An IDE with debugging will be helpful, but you can also do this with gdb from the command line (you might need to install). To use gdb, you should build your program with the -g option to build in debugging information.

See the sample session below:

```
rothede@debian99:~/Box/CourseNotes/CS3841_new/code/lab$ ./a.out
Segmentation fault
rothede@debian99:~/Box/CourseNotes/CS3841_new/code/lab$ gdb ./a.out
GNU gdb (Debian 7.12-6) 7.12.0.20161007-git

== chopped out copyright stuff ==

Reading symbols from ./a.out...done.
(gdb) run
Starting program: /media/sf_Box/CourseNotes/CS3841_new/code/lab/a.out

Program received signal SIGSEGV, Segmentation fault.
0x0000555555554b3b in llPushBack (myList=0x7fffffffe120, toStore=0x555555554f63 "Four") at llist.c:123
123                     myList->tail->next = newNode;
(gdb) bt
#0  0x0000555555554b3b in llPushBack (myList=0x7fffffffe120, toStore=0x555555554f63 "Four") at llist.c:123
#1  0x0000555555548a4 in main () at lltester.c:30

(gdb)
```

Crashed! Segfault (we will talk about those later).

Fire up gdb

Run the program

Crashed! Now we know where.

bt = backtrace – lets you know where program flow that led to issue.

Another tool I highly recommend you use is valgrind. Valgrind will monitor the heap and report memory leaks. The program must run to completion for this to be useful. Sample output is below:

```
==1739==
==1739== HEAP SUMMARY:
==1739==     in use at exit: 30 bytes in 2 blocks
==1739==   total heap usage: 9 allocs, 7 frees, 1,139 bytes allocated
==1739==
==1739== LEAK SUMMARY:
==1739==    definitely lost: 24 bytes in 1 blocks
==1739==    indirectly lost: 6 bytes in 1 blocks
==1739==      possibly lost: 0 bytes in 0 blocks
==1739==    still reachable: 0 bytes in 0 blocks
==1739==         suppressed: 0 bytes in 0 blocks
==1739== Rerun with --leak-check=full to see details of leaked memory
==1739==
==1739== For counts of detected and suppressed errors, rerun with: -v
==1739== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
rothede@debian99:~/Box/CourseNotes/CS3841_new/code/lab$
```

7 Questions

These questions are to be answered in a comment block at the top of your "driver" code:

1. All of the list functions need a "reference" to the list structure, and according to this design, that list reference is passed as a pointer. Why is this necessary? Do all of the list functions need this to be passed as a pointer? Any exceptions? Be specific in your answer.
2. Unlike a Java or C++ implementation, this implementation cannot "hide" any of the internal structure of the list. That is, users of the list could mess up the next and prev pointers if they are careless. Can you think of any way we could hide the structure of the list to lessen the chances a user will mess up the list? Describe in brief detail.
3. What if all llClear() did was assign NULL to head and tail in the list structure and nothing else. Would the program crash? Would there be any side effects? Try it and report results.
4. This design requires the user to iterate the list somewhat manually as demonstrated in the sample driver. Propose the design of an iterator for this list. What data items would the iterator need to store (in a structure, perhaps)? What functions would the iterator supply?

8 Deliverables

You should have exactly three files in your project. lltester.c, llist.c, llist.h. The first file contains main (the driver) and should have the questions posed above along with the answers (yes, repeat the question in the comment and then provide your answer). A makefile is useful, but optional for this assignment.

Also, in a block comment at the top of the lltester.c file, write a short paragraph relating your experiences with this project. Discuss specific experiences using dynamic memory and your debugging efforts.

Prepare a well-documented pdf printout of your 'C' source code that implements the above requirements. Make sure your lines are not too long as to cause word-wrap on the output. The files should appear in the pdf in this order: lltester.c (with "report" comments"), llist.h (unchanged), and llist.c.

Prepare a zip file with your source code. For this lab, there should be only be the three files detailed above and the optional makefile.

Name the two files "code.pdf" and "code.zip" and submit to Lab#2 in Blackboard.

You should ensure that this program compiles without warning (-Wall and -Wextra) prior to submitting. Using gnu90 standard is allowed (gcc default). If your implementation requires C99 or gnu99, please make that clear in the comments.

9 Grading Checklist

Approximate!

Grade	Criteria
0	Not turned in, or left lab prior to completion and submittal.
<C	Not completely functional, poorly documented, turned-in late.
C	Mostly functional, marginally documented, turned-in late.
B	Completely functional, well-documented, submitted on time, basic testing.
A	Exceptional implementation with well-factored code, static helper method(s), excellent testing and documentation(comments).
A+ (100%)	'A' level plus an implemented and tested iterator.