COMS W3157 Advanced Programming, Lab #3

Please refer our instructions for lab retrieval and submission. You must include the required information in your README.txt. This lab consists of multiple parts, so your code for each part must be in a subdirectory (named "part1", "part2", etc.). Like before, you are required to have 5 Git commits with meaningful commit messages, a Makefile for each part where you write code, and Valgrind output in your README.

Part 1: Implement a singly linked list

(a)

Implement a library for generic singly linked lists that can hold any data type. The interface has been specified and provided in a header file named mylist.h. Your job is to write mylist.c that implements each function whose prototype is included in mylist.h.

Specifically, you are asked to write the following functions:

```
struct Node *addFront(struct List *list, void *data);

void traverseList(struct List *list, void (*f)(void *));

void flipSignDouble(void *data);

int compareDouble(const void *data1, const void *data2);

struct Node *findNode(struct List *list, const void *dataSought,
        int (*compar)(const void *, const void *));

void *popFront(struct List *list);

void removeAllNodes(struct List *list);

struct Node *addAfter(struct List *list,
        struct Node *prevNode, void *data);

void reverseList(struct List *list);
```

The header file contains detailed comments specifying each function's behavior. Your implementation should follow the specified behavior.

The skeleton code also provides a test driver program, mylist-test.c, which produces the following output for a correctly implemented linked list:

```
testing addFront(): 9.0 8.0 7.0 6.0 5.0 4.0 3.0 2.0 1.0
testing flipSignDouble(): -9.0 -8.0 -7.0 -6.0 -5.0 -4.0 -3.0 -2.0 -1.0
testing flipSignDouble() again: 9.0 8.0 7.0 6.0 5.0 4.0 3.0 2.0 1.0
testing findNode(): OK
popped 9.0, the rest is: [ 8.0 7.0 6.0 5.0 4.0 3.0 2.0 1.0 ]
popped 8.0, the rest is: [ 7.0 6.0 5.0 4.0 3.0 2.0 1.0 ]
popped 7.0, the rest is: [ 6.0 5.0 4.0 3.0 2.0 1.0 ]
popped 6.0, the rest is: [ 5.0 4.0 3.0 2.0 1.0 ]
popped 5.0, the rest is: [ 4.0 3.0 2.0 1.0 ]
popped 4.0, the rest is: [ 3.0 2.0 1.0 ]
popped 3.0, the rest is: [ 2.0 1.0 ]
```

```
    popped 2.0, the rest is: [ 1.0 ]
    popped 1.0, the rest is: [ ]
    testing addAfter(): 1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0
    popped 1.0, and reversed the rest: [ 9.0 8.0 7.0 6.0 5.0 4.0 3.0 2.0 ]
    popped 9.0, and reversed the rest: [ 2.0 3.0 4.0 5.0 6.0 7.0 8.0 ]
    popped 2.0, and reversed the rest: [ 8.0 7.0 6.0 5.0 4.0 3.0 ]
    popped 8.0, and reversed the rest: [ 3.0 4.0 5.0 6.0 7.0 ]
    popped 3.0, and reversed the rest: [ 7.0 6.0 5.0 4.0 ]
    popped 7.0, and reversed the rest: [ 4.0 5.0 6.0 ]
    popped 4.0, and reversed the rest: [ 6.0 5.0 ]
    popped 6.0, and reversed the rest: [ 5.0 ]
    popped 5.0, and reversed the rest: [ ]
```

This model output is also provided to you in mylist-test-output.txt.

I recommend you implement the functions in the order listed, and test each function as you go. You can start by commenting out the code in main() of mylist-test.c and uncomment the code one block at a time to test each list function you implemented, comparing your output with that of mylist-test-output.txt. The diff command may come in handy.

Note that mylist-test.c may not test every single function. You are still responsible for correctly implementing all functions. You may modify the test driver with your own additional tests, but do not commit changes to the test driver.

Don't forget to run Valgrind at each step to make sure you don't have memory errors, and don't forget to include the Valgrind output in your README.txt when you're done.

(b)

Modify your Makefile to produce a static library named "libmylist.a" that contains your linked list object files. Your test program, mylist-test, must link with the library file, NOT the mylist.o file.

Part 2: Using the linked list library for strings

In this part, you will use the linked list library that you implemented in part1 to write a program called revecho that prints out the command line arguments in reverse order. It will also look for the word "dude" (case-sensitive) among the command line arguments you passed, and report whether it's there.

For example,

```
    ./revecho hello world dude
    dude
    world
    hello

    dude found
```

Another example:

```
    ./revecho hello world friend
    friend
    world
    hello
```

dude not found

Here are the program requirements and hints:

- Your program should simply put all the argument strings into a list WITHOUT duplicating them. There should be no malloc() in your code. Just call addFront() for all strings.

- To print out the strings, you can either use traverseList(), or you can traverse the list yourself by following the .next pointers, and print out each string as you go.

- Don't forget to initialize the list and remove all nodes at the end to prevent memory errors or leaks. Make sure you include Valgrind output for this part in your README.txt.

- To find "dude", you can either traverse the list yourself, or use findNode(). In either case, the strcmp() function will come in handy. If you want to pass strcmp() to findNode(), you will have to account for the function pointer type, because the type signature of strcmp() is slightly different from that of the compar argument of findNode(). One solution for the type mismatch is to cast the function pointer; see K&R2 section 5.11 for an example of doing so.

- You must use mylist.h and libmylist.a directly from the part1 directory. Do not copy any files from part1 directory to part2 directory. The part2 directory should contain only 2 tracked files: Makefile and revecho.c. revecho.c should #include <mylist.h> instead of #include "mylist.h".

  You should tell the compiler and linker to look for mylist.h and libmylist.a in ../part1, using the -I and -L flags, respectively (see gcc's man pages). You should also tell the linker to link in libmylist.a using the -l flag. You should put these flags into your Makefile; if you are using Make's implicit rules for compilation and linking, you will need to include the -I, -L, and -l flags in CFLAGS, LDFLAGS, and LDLIBS, respectively.

  The search path you give to -I and -L must be expressed as a relative path (i.e., ../part1) from the part2 directory. It should not include "~" anywhere, nor should it begin with "/"; failure to follow this requirement will cause your code to fail to build when we try to grade it.

--

Good luck!