15-213/18-213/15-513, Fall 2018
C Programming Lab: Assessing Your C Programming Skills

Assigned:	Tues., Aug. 28
Due:	Thurs., Sept. 6, 11:59 pm
Last possible hand in:	Thurs., Sept. 6, 11:59 pm

1   Logistics

There are no late days, grace days, or extensions for this assignment. Start early enough to get it done before the due date. Assume things will not go according to plan, and so you must allow extra time for heavily loaded systems, dropped Internet connections, corrupted files, traffic delays, minor health problems, etc. Do not request an extension—it will not be granted. A well-prepared student can complete the assignment in 1–2 hours, but it may require longer if you have not done much C programming. If you are not yet registered for the course, you can request an Autolab account and complete the assignment on schedule.

This is an individual project. All handins are electronic. You can do this assignment on any machine you choose. It has been tested on machines running several versions of Linux and OS X. The testing for your code will be done using Autolab, using OS and compiler configurations similar to those on the Andrew Linux machines. We advise you to test your code on an Andrew Linux machine before submitting it.

Before you begin, please take the time to review the course policy on academic integrity at:

> http://www.cs.cmu.edu/~213/academicintegrity.html

2   Overview

This lab will give you practice in the style of programming you will need to be able to do proficiently, especially for the later assignments in the class. The material covered *should* all be review for you. Some of the skills tested are:

- Explicit memory management, as required in C.

- Creating and manipulating pointer-based data structures.

- Working with strings.

- Enhancing the performance of key operations by storing redundant information in data structures.

- Implementing robust code that operates correctly with invalid arguments, including NULL pointers.

The lab involves implementing a queue, supporting both last-in, first-out (LIFO) and first-in-first-out (FIFO) queueing disciplines. The underlying data structure is a singly-linked list, enhanced to make some of the operations more efficient.

3 Resources

Here are some sources of material you may find useful:

1. *C programming*. Our recommended text is Kernighan and Ritchie, *The C Programming Language, second edition*. Copies are on reserve in the Sorrells Engineering Library. For this assignment, Chapters 5 and 6 are especially important. There are good online resources as well, including: https://en.wikibooks.org/wiki/C_Programming.

2. *Linked lists*. You can consult the lecture materials for 15-122: http://www.cs.cmu.edu/~iliano/courses/18S-CMU-CS122/handouts/10-linkedlist.pdf.

3. *Asympotic (big-Oh) notation*. If you are unsure what "$O(n)$" means, consult the lecture materials for 15-122: http://www.cs.cmu.edu/~iliano/courses/18S-CMU-CS122/handouts/05-sort.pdf.

4. *Linux Man pages*. The authoritative documentation on a library function *FUN* can be retrieved via the command "man *FUN*." Some useful functions for this lab include:

   Memory management: Functions `malloc` and `free`.

   String operations: Functions `strlen`, `strcpy`, and `strncpy`. (Beware of the behavior of `strncpy` when truncating strings!)

Be sure to follow posts on Piazza providing clarifications and updates on the assignment.

As the Academic Integrity Policy states, you should not search the web or ask others for solutions or advice on the lab. That means that search queries such as "linked-list implementation in C" are off limits.

4 Logging in to Autolab

All 213/513 labs are being offered this term through a Web service developed by CMU students and faculty called *Autolab*. Before you can download your lab materials, you will need to update your Autolab account. Point your browser at the Autolab front page: https://autolab.andrew.cmu.edu.

You will be asked to authenticate via Shibboleth. After you authenticate the first time, Autolab will prompt you to update your account information with a *nickname*. Your nickname is the external name that identifies

you on the public scoreboards that Autolab maintains for each assignment, so pick something interesting! You can change your nickname as often as you like. Once you have updated your account information, click on "Save Changes" button, and then select the "Home" link to proceed to the main Autolab page.

You must be enrolled to receive an Autolab account. If you added the class late, you might not be included in Autolab's list of valid students. In this case, you won't see the 213 course listed on your Autolab home page. If this happens, contact the staff and ask for an account.

If you are still on the waitlist for the course, then download a copy of the archive file (described below) from the course schedule web page. You can get working on the lab and then get an Autolab account once you are enrolled.

5 Downloading the assignment

Your lab materials are contained in a Linux archive file called `cprogramminglab-handout.tar`, which you can download from Autolab or from the class web page. Start by copying the file to a protected directory in Andrew in which you plan to do your work. Then login to a Linux machine and give the command

```
linux> tar xvf cprogramminglab-handout.tar
```

This will create a directory called `cprogramminglab-handout` that contains a number of files. Consult the file `README` for descriptions of the files. You will modify the files `queue.h` and `queue.c`.

6 Overview

The file `queue.h` contains declarations of the following structures:

```
/* Linked list element */
typedef struct ELE {
    char *value;
    struct ELE *next;
} list_ele_t;

/* Queue structure */
typedef struct {
    list_ele_t *head;  /* Linked list of elements */
} queue_t;
```

These are combined to implement a queue of strings, as illustrated in Figure 1. The top-level representation of a queue is a structure of type `queue_t`. In the starter code, this structure contains only a single field `head`, but you will want to add other fields. The queue contents are represented as a singly-linked list, with each element represented by a structure of type `list_ele_t`, having fields `value` and `next`, storing a pointer to a string and a pointer to the next list element, respectively. The final list element has its next pointer set to `NULL`. You may add other fields to the structure `list_ele`, although you need not do so.
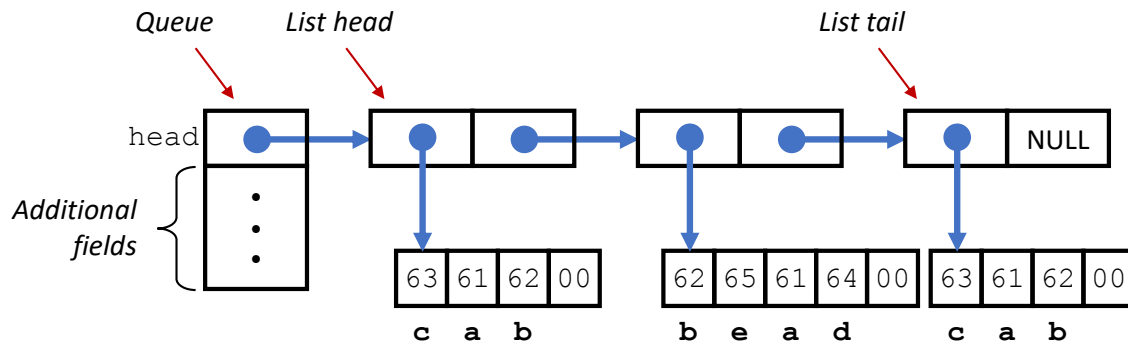
Figure 1: **Linked-list implementation of a queue.** Each list element has a `value` field, pointing to an array of characters (C's representation of strings), and a `next` field pointing to the next list element. Characters are encoded according to the ASCII encoding (shown in hexadecimal.)

Recall that a string is represented in C as an array of values of type `char`. In most machines, data type `char` is represented as a single byte. To store a string of length l, the array has $l + 1$ elements, with the first l storing the codes (typically ASCII[1] format) for the characters and the final one being set to 0. The `value` field of the list element is a pointer to the array of characters. The figure indicates the representation of the list ["`cab`", "`bead`", "`cab`"], with characters `a`–`e` represented in hexadecimal as ASCII codes `61`–`65`. Observe how the two instances of the string "`cab`" are represented by separate arrays—each list element should have a separate copy of its string.

In our C code, a queue is a pointer of type `queue_t *`. We distinguish two special cases: a *NULL* queue is one for which the pointer has value `NULL`. An *empty* queue is one pointing to a valid structure, but the `head` field has value `NULL`. Your code will need to deal properly with both of these cases, as well as queues containing one or more elements.

7   Programming Task

Your task is to modify the code in `queue.h` and `queue.c` to fully implement the following functions.

`q_new`: Create a new, empty queue.

`q_free`: Free all storage used by a queue.

`q_insert_head`: Attempt to insert a new element at the head of the queue (LIFO discipline).

`q_insert_tail`: Attempt to insert a new element at the tail of the queue (FIFO discipline).

`q_remove_head`: Attempt to remove the element at the head of the queue.

`q_size`: Compute the number of elements in the queue.

---

[1]Short for "American Standard Code for Information Interchange," developed for communicating via teletype machines.

`q_reverse`: Reorder the list so that the queue elements are reversed in order. This function should not allocate or free any list elements (either directly or via calls to other functions that allocate or free list elements.) Instead, it should rearrange the existing elements.

More details can be found in the comments in these two files, including how to handle invalid operations (e.g., removing from an empty or NULL queue), and what side effects and return values the functions should have.

For functions that provide strings as arguments, you must create and store a copy of the string by calling `malloc` to allocate space (remember to include space for the terminating character) and then copying from the source to the newly allocated space. When it comes time to free a list element, you must also free the space used by the string. You cannot assume any fixed upper bound on the length of a string—you must allocate space for each string based on its length.

Two of the functions: `q_insert_tail` and `q_size` will require some effort on your part to meet the required performance standards. Naive implementations would require $O(n)$ steps for a queue with n elements. We require that your implementations operate in time $O(1)$, i.e., that the operation will require only a fixed number of steps, regardless of the queue size. You can do this by including other fields in the `queue_t` data structure and managing these values properly as list elements are inserted, removed and reversed.

Your program will be tested on queues with over 1,000,000 elements. You will find that you cannot operate on such long lists using recursive functions, since that would require too much stack space. Instead, you need to use a loop to traverse the elements in a list.

8   Testing

You can compile your code using the command:

```
linux> make
```

If there are no errors, the compiler will generate an executable program `qtest`, providing a command interface with which you can create, modify, and examine queues. Documentation on the available commands can be found by starting this program and running the `help` command:

```
linux> ./qtest
cmd>help
```

The following file (`traces/trace-eg.cmd`) illustrates an example command sequence:

```
# Demonstration of queue testing framework
# Initial queue is NULL.
show
# Create empty queue
new
# Fill it with some values.  First at the head
ih dolphin
```

5

```
ih bear
ih gerbil
# Now at the tail
it meerkat
it bear
# Reverse it
reverse
# See how long it is
size
# Delete queue.  Goes back to a NULL queue.
free
# Exit program
quit
```

You can see the effect of these commands by operating qtest in batch mode:

```
linux> ./qtest -f traces/trace-eg.cmd
```

With the starter code, you will see that many of these operations are not implemented properly.

The traces directory contains 15 trace files, with names of the form trace-k-*cat*.txt, where k is the trace number, and *cat* specifies the category of properties being tested. Each trace consists of a sequence of commands, similar to those shown above. They test different aspects of the correctness, robustness, and performance of your program. You can use these, your own trace files, and direct interactions with qtest to test and debug your program.

9   Evaluation

Your program will be evaluated using the fifteen traces described above. You will given credit (either 6 or 7 points, depending on the trace) for each one that executes correctly, summing to a maximum score of 100. This will be your score for the assignment—the grading is completely automated.

The driver program driver.py runs qtest on the traces and computes the score. This is the same program that will be used to compute your score with Autolab. You can invoke the driver directly with the command:

```
linux> ./driver.py
```

or with the command:

```
linux> make test
```

10   Handin

Using make to generate qtest also has the effect of generating a file handin.tar. You should do this on a Linux machine. You can upload this file (and only this file!) to Autolab, which will autograde your submission and record your scores. You may handin as often as you like until the due date.

IMPORTANT: Do not upload files in other archive formats, such as those with extensions `.zip`, `.gzip`, or `.tgz`.

11 Reflection

This should be a straightforward assignment for students who are fully prepared to take this course. If you found you had trouble writing this code or getting it to work properly, this may be an indication that you need to upgrade your C programming skills over the next month. Starting with Lab 4 in early October, you will need to write programs that require a mastery of the skills tested in this assignment.

A good place to start is to carefully study Kernighan and Ritchie. This book documents the features of the language and also includes a number of examples illustrating good programming style. The book is a bit dated, and so it doesn't contain some more modern features of the language, such at the `bool` data type, but it is still considered one of the best books on how to program in C.