CS 208, Winter 2020
Lab 0: Welcome to C
Assigned: Jan. 10

Due: Monday, Jan. 20, 9:00pm

1 Introduction

This lab will give you practice in the style of C programming you will need to be able to do proficiently, especially for the later assignments in the class. Specific skills include

- 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

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

2 Logistics

Download the starter code for this lab from the course website:
`http://cs.carleton.edu/faculty/awb/cs208/w20/lab0-handout.tar` If you're working on `mirage`, you can download the tar file to your current directory by running

```
> wget http://cs.carleton.edu/faculty/awb/cs208/w20/lab0-handout.tar
```

This is an individual project. All submissions are electronic. You can do this assignment on any machine you choose. The testing for your code will be done on a CS department Ubuntu Linux machine. We advise you to test your code on `mirage` before submitting it. Before you begin, please take the time to review the course policy on academic honesty and collaboration.

Note that for this lab (and every lab) you will be asked to report how many hours you spent working on it, so please try and keep track.

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.* One copy is on reserve at Gould. For this assignment, Chapters 5 and 6 are especially important. There are good online resources as well.
2. *Linked lists.* Here are a couple good descriptions for you to consult:
   - `https://people.engr.ncsu.edu/efg/210/s99/Notes/LinkedList.1.html`
   - `https://medium.com/basecs/whats-a-linked-list-anyway-part-1-d8b7e6508b9d`
3. *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!)

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

4 Overview

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

```
/* Linked list element */
typedef struct list_ele {
  char *value;
  struct list_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.

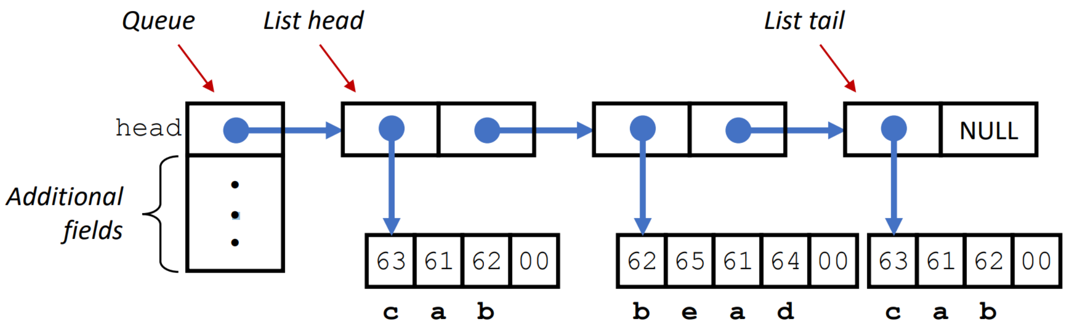

Figure 1: **Linked-list implementation of a queue**. Each list element has a `value` field, pointing to an array of characters (C's representation of strings), and a `next` field pointing to the next list element. Characters are encoded according to the ASCII encoding (shown in hexadecimal.)

Recall that a string is represented in C as an array of values of type `char`. On 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.

5 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.

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

`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.

`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.

6 Testing

You can compile your code using the command:

```
> 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:

```
> ./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:

```
> ./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.

7 Evaluation

Your program will be evaluated using the fifteen traces described above. You will given credit (either 2, 4, or 7 points, depending on the trace) for each one that executes correctly. This plus 3 points for submitting `feedback.txt` will be your score for the the lab. 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 grade. You can invoke the driver directly with the command:

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

or with the command:

```
> make test
```

8 What to Turn In

We will use the COURSES network drive for lab submissions. You can find instructions on how to access it here. If you are on `mirage`, you can mount it in your home directory by running

```
linux> sudo map-courses
```

It will prompt you for your password, your username, and then your password again. A `COURSES` directory will appear with a `cs208-01-w20` directory (possibly in addition to those for other courses) inside it. The `Hand-In` directory in the `cs208` directory should contain a directory with your username—this is where you will copy files for submission.

Using `make` to generate `qtest` also has the effect of generating a file `handin.tar` (you should do this on a Linux machine). Inside your `Hand-In` directory, create a `lab0` directory. Copy `handin.tar` and a text file called `feedback.txt` there.

Your `feedback.txt` is a way for me to check in and get a sense of how this lab went for you. Please include the following information:

- How many hours you spent outside of class on this homework.
- The difficulty of this homework: too easy, easy, moderate, challenging, or too hard.
- What you learned on this homework (very briefly). Rate the educational value relative to the time invested from 1 (low) to 5 (high).

This lab is adapted from the C Programming Lab developed for CMU's 15-213 (Introduction to Computer Systems), available here.