# CS 305 Lab 7: Stacks and Queues
# Fall 2019 Cenek

The purpose of this lab is to give you experience with stacks and queues. In this lab, the stack is implemented with a linked list. The queue is implemented with an array.

This lab has a total of 100 possible points (30 points from pre-lab and 70 points from lab). You have **two lab sessions** to complete this lab. You have two lab sessions for this lab.

Sit with your assigned partner.

## Objectives

Upon completion of the laboratory exercise, you will be able to do the following:

- Read implementations for stacks and queues.
- Define a peek function for a stack.
- Define a function to determine if a string has properly nested ( ) and [ ], using a stack data structure
- Define a full function for a queue.
- Use stack and queue data structures to add and remove items.

## Part 1: Logging in (choose one person)

1. Follow the procedure from prior labs to log in and Mobaxterm. Go into your cs305 directory. Make a new `lab7` directory:

```
cd /drives/p/cs305
mkdir lab7
```

2. Get the lab 7 files. Download `stack.zip` and `queue.zip` from moodle to your lab7 folder. On Mobaxterm:

```
ls
```

Hopefully, you can see the file. Unzip both files:

```
unzip stack.zip
unzip queue.zip
```

## Part 2: Stacks

1. Go into the `stack` folder. Open `stack.h` to see the struct definitions and the function prototypes. The stack is implemented as a linked list, where the front of the list is the top element of the stack. The operations on a stack include: `initStack`, `empty`, `push`, and `pop`. You will implement `peek` as part of this lab.

2. Open `main.c` and look at the function `test1`. What are the values of the stack `s` just after 15 is pushed to the stack?

top of stack ---

bottom of stack ---

3. Compile and run the program to see if your answer to #2 is correct.

```
gcc –o s stack.c main.c
./s
```

4. Complete the function definition for `peek`. It should test if the stack is empty. If so, print `"Stack is empty. No data item to peek.\n"` and return BAD. BAD is a macro defined in `stack.h`. If there is data in the stack, return the top element of the stack (but don't remove it from the stack for the peek operation).

Comment out `test1()` in main and uncomment `test2()` in main. Compile and run your code to be sure that `peek` is working correctly.

5. Add a function prototype in `stack.h` and a function definition in `stack.c` to free the memory associated with the stack. It should look like:

```
void freeStack(Stack S);
```

Because the stack might have nodes in it, you should free the nodes by popping whatever is in the stack until it is empty. Once the stack is empty, you can free S (the pointer to the StackType).

Complete the function called `test3` in main that tests that freeing the stack works. You should put items in the stack and then free the stack.

**Checkpoint 1 [20 points]: Show your lab instructor/assistant the results of your program running with calls to test2 and test3 uncommented. Show your answer to the question above.**

## Part 3: Using Stacks

1. Now, you will complete the function `testMatch` in `main.c` that uses a stack to solve the nested delimiter problem for matching ( ) and [ ]. Here is the pseudocode:
While there is a unprocessed character c in input:
    if c is the left delimiter (, then push it to the stack
    if c is the left delimiter [, then push it to the stack
    if c is the right delimiter ), then pop the stack; if what is popped is (, keep going; if what is
        popped is not (, then print "Not in proper format"

if c is the right delimiter ], then pop the stack; if what is popped is [, keep going; if what is
popped is not [, then print “Not in proper format”
Else, move to next character
If stack is empty, print “In proper format”. Else, print “Not in proper format”

Uncomment `testMatch` in `main` and comment out the other tests.

2. Open `stack.h`. Change the `StackData` type to `char` and define `BAD` as `'!'`. Compile and run the code. Try different inputs for expressions to see if your function is working properly.

**<u>Checkpoint 2 [20 points]: Show your lab instructor/assistant the results of your program running for detecting properly formatted nested ( ) and [ ] for a variety of inputs. Show the lab instructor/assistant your code.</u>**

## Part 4: Queues

1. The queue data structure for this lab is implemented as an array. Go in the `queue` folder. Open `queue.h` to see the struct definitions and the function prototypes.

2. Open `main.c`. Look at the code and predict what the queue has in in before you test this function. What does the queue contain (in order such that the left-most item will be the next item dequeued) after 14 is enqueued?

front of queue_______________________________________________back of queue

3. Compile and run the code:

```
gcc –o q queue.c main.c
./q
```

4. Is your prediction in #2 correct? YES NO

5. Open `queue.c` to see how the functions are implemented. The `head` is the index in the array that is one less than the position of the first item in the queue. The `tail` is the index of the last item in the queue. When `head` is equal to `tail`, the queue is empty. Note that by keeping track of head and tail, we do not need to physically move items in the data array (copying/shifting them to the left as items are removed from the queue).

a. What is the value of `q->head` just after 4 in enqueued in main? ___________
(You can print this value if to confirm your prediction.)

b. What is the value of `q->tail` just after 4 is enqueued in main? _________
(You can print this value if to confirm your prediction.)

c. What is the value of `q->head` just after 20 is enqueued in main? ______

d. What is the value of `q->tail` just after 20 is enqueued in main? _____

9. Open `queue.h`. At the top of the file, you will see a macro definition for `MAX_Q`. Set it to 5. Recompile the code and run it.

What are the queue's contents after 14 is enqueued (this is what is printed after "`Queue contents`:")? _______________________

Hmmm, what happened to the rest of the items in the queue?

____________________________

What are the values of `q->data`, `q->head`, and `q->tail` just after 14 in enqueued in main? (you can check this by printing out these values.)

`q->data`:

| | | | | |
|---|---|---|---|---|
| | | | | |
| 0 | 1 | 2 | 3 | 4 |

`q->tail`: __________ `q->head`: _________

10. Well, what you saw is what happens when a queue is allowed to become overfull. This is a buggy implementation. One option is to re-allocate memory to store twice the size of data and copy everything over to this new array and store the new array as data. Another option is to not allow the queue to become over-full.

Open `queue.c`. At the bottom, there is a function called `full`. As implemented, `full` always returns 0 (meaning that is never full). Replace the return statement with a correct return statement. The queue is full when `tail` is one less than the value of `head` (it has wrapped around the array). Be sure that this works when `tail` is the largest position of the array and `head` is at 0. You can use modulus (%) to help you (see how it is used in `dequeue` and `enqueue`).

Compile and run your program.

Now what happens? _________________________________

What is in the queue after 14 is enqueued in main?

front of queue________________________________________ back of queue

11. Another thing that we should always do when writing C programs is to free any memory that was allocated on the heap before returning from main. We know the underlying representation of this queue is a static-sized array, but anyone else

using this queue ADT would not know that. Add a function to `queue.c` and a prototype to `queue.h` called:

```
void freeQueue(Queue Q);
```

The function definition should simply call `free(Q)`.

In main, just before the return statement, call:

```
freeQueue(q);
```

**Checkpoint 3 [15 points]: Show your lab instructor/assistant the result of your program running with MAX_Q set to 5 and your code. Show your lab instructor/assistant the result of your program running with MAX_Q set to 15. Show your answers to the questions above.**

Keep MAX_Q at 15. Another operation that could be implemented for queues is `length`. While this is not a "standard" queue operation, it could certainly be useful for a programmer. For example, knowing the length of a queue could help a program determine what should be done. At Disneyland, the length of the queue might trigger the staff to put more cars on a ride, for example.

1. Add a function to `queue.c` and a prototype to `queue.h` called:

```
int length(Queue Q);
```

2. Using the values of head and tail, return the current length of the queue. Think about the two cases:

If tail >= head, this is calculated by:

______________________________________________
(queue does not wrap around end of array)

If tail < head, this is calculated by: ______________________________________________
(queue wraps around end of array)

3. Add code to `main.c` to test that length is working properly. Print the length of the queue before and after enqueuing and dequeuing items.

**Checkpoint 4 [15 points]: Show your lab instructor/assistant the result of your program running and your code. Show your answers to the questions above.**

If you finish early and want more practice, try writing code to:

- use two stacks to mimic the behavior of a queue
- use two queues to mimic the behavior of a stack

- use an extra stack to keep a stack in sorted order

You are finished. Close Mobaxterm and any other applications. Log off the computer.

If any checkpoints are not finished, they are due March 15. You may submit screenshots, code files, and answers to questions electronically on moodle.