Assignment #4: Tweet Manager – Part II

1.0 The Task and its technical requirement

The technical requirements and the linked list structure definition that you need for this assignment are similar to given in A3.

```
typedef struct microtweet {
   int id;                          //non-unique integer value
   char user [51];                  // the username / userid of the person who wrote the tweet
   char text [141];                 // the text of the tweet
   struct microtweet *next;         //dynamic connection to the next tweet
} tweet;
```

You are required to implement the list that you used in A3 as a queue now. For example, queues follow the FIFO principle (First-In-First-Out), so new tweets are added at the end of the queue (enqueue operation), but tweets are always deleted from the front of the queue (dequeue operation).

Write the following C programs:

1. (Required) queueFunctions.c - this file must have at least the following queue functions:

a. Enqueue - adds a node to the queue

b. Dequeue - deletes a node.

c. isEmpty - checks if the given queue is empty - returns true if empty, 0 if not empty

d. printQueue - takes the front of the queue and prints every element of the queue.

Here are the prototypes:

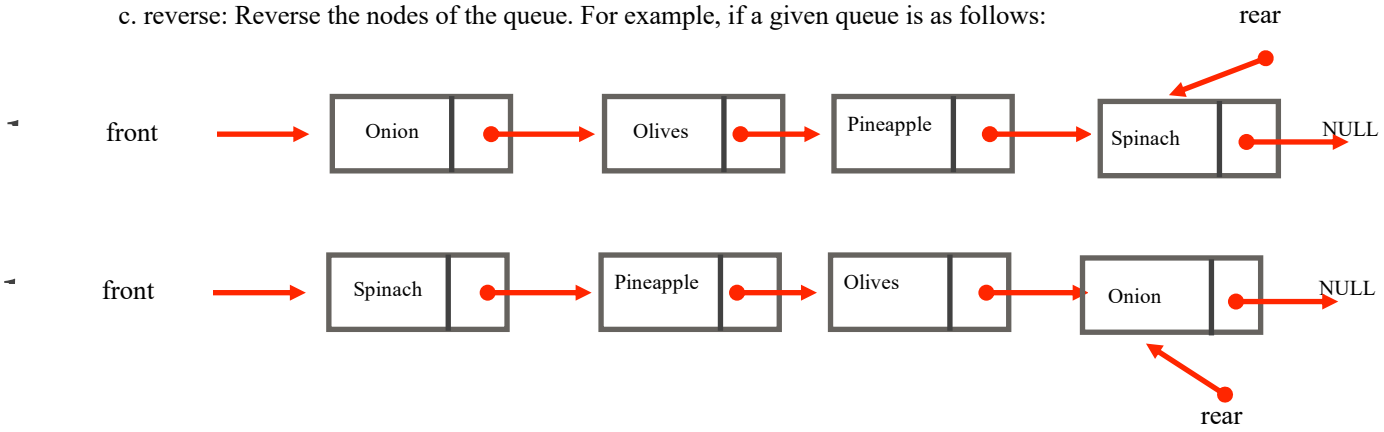
```
void enqueue (tweet ** head, tweet ** tail, tweet * node);

void dequeue (tweet ** head, tweet ** tail);

int isEmpty (tweet * head);

void printQueue (tweet * head);
```

2. (Required) miscFunctions.c - this file has definitions for 3 functions:

a. sortID: Sort the queue in ascending order by id – you may use any sorting method discussed in class

b. sortUsername: Sort the queue in ascending order by username - you may use any sorting method discussed in class

c. reverse: Reverse the nodes of the queue. For example, if a given queue is as follows:



Here are the prototypes:

```
void sortID (tweet ** head, tweet ** tail);

void reverse (tweet ** head, tweet ** tail);

void sortUsername (tweet ** head, tweet ** tail);
```

3. (Optional) helper.c

If you have any additional functions that you would like to write and use for A4, other than the required ones listed above, then write them in this program.

4. (Required) testA4.c

This program must have at least 3 test cases for each of the required 7 functions listed above.

5. (Required) headerA4.h

This includes all structure definitions and function prototypes used in any program written for this assignment.

2.0 Grading Criteria

Grading will focus on three major components: technical completion, code style, and memory management. Some questions to ask yourself to verify that you have completed the components has been given below:

Technical Completion

- Does the submission compile without issue?
- Is the program working as intended?
- Are there any parts of the program that are missing or incomplete?
- Where necessary, is a suitable amount of error-checking completed?
- Does the program work for edge cases, such as printing an empty tweet list?
- Did the compiler flag anything as either a warning or an error?

Code Style and Folder Structure

- Are the source files in the correct location? Header files? Makefile?
- Does the makefile have all of the functionality necessary?
- Does the code have a consistent format? Is it well-written, documented, properly indented, etc.?
- If I showed this code to someone else with a similar level of programming skill, would they be able to understand the code's behaviour?

Memory Management

- Are there any memory leaks in code and, if so, how can I resolve them?
- Am I allocating a proper amount of memory for certain items, or am I over-allocating (ex. 10,000 bytes for an input buffer versus 500bytes)?
- Is there a corresponding *free* statement for each *malloc* statement?
- Note that there is a penalty if leaks are found

Testing will occur across all parts of the assignment, with greater attention given to the more complex components of the code. There will be a mark given for style, and deductions for compiler warnings and errors.

3.0 Submission Instructions

Once you are satisfied with your work, upload your work to the GitLab repository for this assignment. We suggest that you upload iterations throughout the development process as a precaution, but the final submission will be the submission that is graded. Your final submission should include:

1) A **makefile** that compiles your program and generates an executable by running 'make', and has a target 'clean', which removes all intermediate .o files, as well as the final executable.
2) All source files.
3) All header files
4) A **README** file, alongside the makefile. It should include AT LEAST the following information:
   a) A section that includes your name and username, the assignment name, the course code, and the date of last revision.
   b) A section explaining how to compile and run your program.
   c) A section explaining each of the completed components.
   d) A section outlining any known limitations of your program (ex. missing features, inefficiencies, edge cases that you couldn't figure out, etc.).
   e) A section outlining future improvements, if any, to your assignment.

Submissions that do not compile using the makefile will be given a zero, and a penalty may be incurred for compiler warnings. If no makefile is submitted, the TA will attempt to compile your program, but reserves the right to assign a zero. We will test you code for memory leaks – there will be penalty if leaks are found.

4.0 Marking

- Programs that don't compile receive an immediate zero (0).
- Programs that produce compilation warnings will receive a deduction of 1 mark per unique warning (i.e. 3 'unused variable' warnings will only deduct 1 mark).
- Loss of marks may also result from poor style (e.g. lack of comments, structure)
- Programs that use goto statements receive an immediate zero (0)
- Programs that use global variables statements receive an immediate zero (0)