Assignment #3: Dynamic Data structures (linked lists)

Tweet Manager

1.0 Introduction

Social media such as Twitter is a well-known example of a dataset that has a countably infinite potential for growth, in that there is a hard-minimum number of tweets that a user might create (i.e. zero), but there is no maximum to the number of tweets that a user might create. In this case, you need to consider dynamic allocations when accessing such a dataset, since no fixed value can be guaranteed to work for all cases.

2.0 The Task

Via a menu-driven solution, create a program that allows a user to run basic tasks on a set of tweet data that emulates the basic functionality of Twitter. Users should be able to create tweets, display tweets, search tweets based on a keyword or mention, save and load tweets from a local file, and calculate the number of stop words are present across all of the tweets (background information on stop words: http://bit.ly/stopWords). The menu will be command-line based and will have the following options:

1. Create a new tweet
2. Display tweets
3. Search a tweet (by keyword)
4. Find how many words are "stop words" (stop words are explained in a later section)
5. Delete the nth tweet
6. Save tweets to a file
7. Load tweets from a file
8. Exit

After the completed execution of each of the menu options (except for the Exit option), the user should be brought back to the menu and re-prompted to enter a new command.

Technical Components

While it is likely that Twitter uses a much more efficient means of storing their data, one of the most basic structures to hold an endless set of data is a **linked list**. We will create a linked list to store the data for all the tweets that we create, edit, delete, search, and store. The linked list will rely on a user-defined structure and will follow this structure definition for it:

```
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
};
```

For simplicity, all of the components inside the struct will be statically sized, apart from the **next* pointer that will be used to create and link the nodes in the linked list. This pointer will be used to hold the memory address for the next tweet.

The given struct must be defined in a separate header file. You may add other definitions in it as well.

You must write a separate function for each menu option (except option 8). Each function should be contained in its own function file. For example, create a function called createTweets for menu option 1 in a file called createTweets.c. You must create a main (**mainA3.c**) that displays the menu and executes the menu options based on user choice. All function **prototypes** are given in a file called **headerA3.h**. Besides these 7 functions, you may create other functions and store them in a separate file (e.g. helper.c).

Input

Since this is a menu-driven program, there are several potential input streams for the tweets. The first input stream is from the command line itself (Menu Option 1). Alternatively, the user can use the seventh option to load a CSV file of tweets into the program. A sample CSV file has been provided for you to test your programs with. Both of these inputs will feed into the same linked list structure, as defined above.

Menu Option 1: Creating a new tweet

If the user chooses to manually enter a new tweet, the following format should be followed (black is system text, red is user input):

Choose a menu option: 1
Enter userid: 1236
Enter a username: uog
Enter the user's tweet: The quick brown fox jumped over the lazy dog\n

Your computer-generated userid is 375.
<Reprint the menu>

Through the menu, the user should have the option to create a new tweet and fill in some of the parameters for the struct manually (namely, the username and user's tweet). Userid is automatically generated using the following rule:

userid = (sum of ascii values of characters in the username) + (length of the user's tweet). For example, for the sample given above, the generated userid = (117 + 103 + 111) + 44 = 375. If this userid already exists in the current linked list, then add random numbers between 1 and 999, repeatedly, until a unique id is generated.

You must store new tweets at the end of the current list.

/*\Note: Use the prototype given in headerA3.h

Menu Option 2: Displaying Tweets

If option 2 is selected, all of the tweets that are currently in memory (ie. in the linked list) will be displayed to the user. The output must include the full static contents of the struct (id, user, text), in the following format:

Choose a menu option: 2
<id>: Created by <user>: <text>
<id>: Created by <user>: <text>
<id>: Created by <user>: <text>
<Reprint the menu>

/*\Note: Use the prototype given in headerA3.h

Menu Option 3: Searching Tweets

Searching for tweets in this assignment will be done by keyword. To search for tweets, the linked list should be traversed to find all tweets that have the user-provided substring inside them. If a match is found, that tweet should be printed to console along with its associated user value (example below: black is system text, red is user input):

Choose a menu option: 3
Enter a keyword to search: bread
Match found for 'bread': uogsocis wrote: "If you put two breadboards together, is it a bread-sandwhich?"
Match found for 'bread': uogsocis wrote: "We apologize for that bread-related pun."
<Reprint the menu>

Note that the search is **case-insensitive**. And so, the same result is achieved for keyword Bread.

/*\Note: Use the prototype given in headerA3.h

Menu Option 4: Counting Stop Words

Stop words are an interesting phenomenon in language. When the user chooses option 5, the program will traverse the entire linked list and calculate the number of stop words that are present across all tweets, outputting the summation of all tweets. For this program, only the top 25 stop words found at http://bit.ly/stopWords will be used as our list of stop words for this assignment, also given here for convenience:

```
"a","an","and","are","as","at","be","by","for","from","has","he","in","is","it","its",
"of","on","that","the","to","was","were","will","with"
```

The format of the output will be a single line formatted as follows (black text is system text, red text is user input):

> Choose a menu option: 4
> Across **27** tweets, **43** stop words were found.
> <Reprint the menu>

/*\Note: Use the prototype given in headerA3.h

Menu Option 5: Delete the nth tweet

In this option, the user is prompted to give a value between 1 and the total number of tweets your linked list has – note that this number must be equal to the total number of tweets at that time.

> Choose a menu option: 5
> Currently there are 5 tweets.
>
> Which tweet do you wish to delete – enter a value between 1 and 5: 2
>
> Tweet 1234 deleted. There are now 4 tweets left.
>
> <Reprint the menu>

/*\Note: Use the prototype given in headerA3.h

Menu Option 6: Save tweets to a file

Should the user wish to output their tweets, they should be first prompted for the filename to output their tweets to. For example (black is system text, red is user input):

> Choose a menu option: 6
> Enter the filename where you would like to store your tweets: tweetsStore.csv
> Output successful!
> <Reprint the menu>

Unless chosen by the user through the menu (Option 6), the tweet data will not be output to file; that is, if the user chooses *Exit* before saving, then the data that they input is lost. Tweet storage should follow the CSV format of the input file, such that the file can be reloaded using the load menu option. The format of the CSV is as follows: id, *user, text*. For example, a tweet that was created by the @uofg Twitter account will be stored in this format:

> 1246, uog, The quick brown fox jumped over the lazy dog\n

/*\Note: Use the prototype given in headerA3.h

Menu Option 7: Load tweets from a file

If the user chooses to load tweets from a file, the following format should be followed (black is system text, red is user input):

> Choose a menu option: 7
> Enter a filename to load from: tweetsStore.csv
> Tweets imported!
> <Reprint the menu>

If an error occurs while inputting from the file, an error message should be output to the screen to inform the user that the action failed.

You may assume that the userids of all tweets in the file are unique.

/*\Note: A comma will not be found in the usernames, but CAN exist in the text of the tweet!

/*\Note: Use the prototype given in headerA3.h

3.0 Error Checking

- No error checking is expected for tweets read from a file: it can be assumed that these tweets will be free from errors. However, new tweets generated by the user, much like on the real Twitter platform, must be checked for errors:
    - Both the username and the tweet must be a non-zero length.
    - A username should not be longer than the maximum username length defined by the struct. The same is true for the tweet text.
- An error message should be produced if an invalid value is provided as a menu option, and the user should be re-prompted.

4.0 Technical Criteria

- All code must compile without warnings or errors while using the -Wall -std=c99 flags.
- The microtweet struct **MUST** be used to store the tweet information.
- All microtweets should be stored in a **single linked list**. A dynamically sized array is not an acceptable means of storing the microtweets for this assignment.
- The program should run without stopping until the exit option is chosen by the user.
- If the user creates some tweets, then loads more from a file, you are required to retain the original list: it is assumed that the tweets from the file are to be appended onto the pre-existing list in memory (ie. within the linked list). The inverse is also true (i.e. if you are loading from a file first, then adding more tweets, the linked list must have tweets from the file added before the tweets added using menu option 1).
- If a user exits the program without saving the tweets, the program will **not** be expected to save the tweets.
- All tweets in memory must be freed upon completion of the program execution.

5.0 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?

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.

6.0 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) Given headerA3.h and other user-defined header files (if any).
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.

Note that the README file is new to CIS2500 assignment submission requirement – you were not required to submit any such file for A1 and A2. Make sure that this is in your checklist of the files to submit to gitlab.

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.