CSCI 401 Operating Systems
Instructor: Gedare Bloom
Project 1 (100 points)

You must do this assignment by yourself. You may discuss the problem or solutions with your classmates or instructors, but **you may not share code with anyone**. You may not use code downloaded from the Internet.

Read all instructions carefully.

Due Saturday, September 10

Part 0 (0 points)

You will benefit from having a Linux development environment for this course. The easiest approach is to install Virtualbox and download the virtual machine available from the text book's companion site http://os-book.com/ -- You have to select the "Operating Systems Concepts" book and find the link to the Linux Virtual Machine. While you're on the site I highly recommend you browse around. The VM is a big file (almost 3 GB) so you will want to start the download immediately.

If you need some help with the C language, you should definitely check out the http://cslibrary.stanford.edu/ resources especially "Essential C", "Pointers and Memory", "Linked List Basics", and "Linked List Problems". If you need help with using *nix for software development then you should go through the "Unix Programming Tools".

Part 1 (50 points)

Create a C-string queue (FIFO) data structure for storing character strings using a header file (queue.h) and separate source file (queue.c) for all functions. You need to implement the following functions, as well as define the struct(s) necessary for your queue to work correctly.

```
/* Allocates and initializes a new queue */
queue* create_queue( );

/* Adds item to end of the queue. This function allocates a new buffer and copies the
string from item (use malloc, strlen, and strncpy; or try strdup) */
void enqueue(queue* Q, char* item);

/* Pops and returns the string from the front of the queue. The calling function is
expected to free the string returned when finished with it. */
char* dequeue(queue* Q);

/* Prints every string in the queue, with a new line character at the end of each string */
void print_queue(queue *Q);
```

/* Flushes (clears) the entire queue and re-initializes the queue. The passed pointer Q should still point to a valid queue when this function returns. Any memory allocated to store items in the queue should be freed. */
void flush_queue(queue* Q);

/* De-allocates all data for the queue. Note: this function should ensure all memory allocated for this queue is freed. */
void free_queue(queue *Q);

Put any struct definitions, typedefs and the above function prototypes in a header file "queue.h" and function bodies (implementations) in a source file "queue.c". Make a test file "queue_test.c" that tests your queue implementation. Your test file should try various tests of your create_queue, enqueue, dequeue, flush_queue, print_queue, and free_queue functions. Write a Makefile that compiles queue.c and queue_test.c to create a binary executable named `queue_test`.

Important elements of the grading will be the quality and coverage (number of cases) that are tested and the cleanliness of your implementation and documentation, and of course for correctness and efficiency.

Part 2 (50 points)

This program will have you create a simple, but useful tool that can analyze a list of words and calculate which words appear most frequently. Your program will have to read in a list of words in a text file, sort it alphabetically as a linked list of the words, and save the word frequencies. A small sample file is provided below; however, your program should not have any limit on the size of the file besides the amount of available memory.

Once the word list is created, your program should calculate the N most frequent words in the list and print them on the screen in a simple format displaying both the word and the frequency that it appeared. The program should ignore any invalid input lines and should not crash under any input. The maximum line length of any line in the file is 255 characters. Each line is terminated with a Unix style EOL character `\n` not a DOS style `\n\r`.

A sample input file:

```
foofoo
dog
cat
dog
moom
csci401isfun
moon$
foofoo
moom.
dog
```

```
    moom
    doggod
    dog3
    f34rm3
    foofoo
    cat
```
So, for the above input, the word list would have 10 distinct words (and no invalid input) in it with the most frequent word a tie between “dog” and “foofoo” which each appear 3 times.

Make sure you only allocate as many nodes as needed for any lists you create, and only as many characters as necessary for each string. The structure you will use for the linked list should resemble this (feel free to use it verbatim):
```
    struct node
    {
        char *word;
        unsigned int freq;
        struct node *next;
    } node;
```

Required Functions

You must implement the following 6 functions. Feel free to make more helper functions if you want, but **do not maintain a second list of all words sorted by frequency!**

```
    struct node *readStringList(FILE *infile);
```
This function reads strings from the file and puts them into a sorted (alphabetically) word list with each word node recording the frequency of the word. Each node and string should be dynamically allocated on the heap (using malloc).

```
    int readLine(FILE *infile, char * line_buffer);
```
This function reads one line at a time from the file and stores the contents of the line into the line_buffer string. The line_buffer string is dynamically allocated outside of the readLine function. If the function succeeds in reading a line, it should return 0, otherwise it should return some non-zero error code.

```
    struct node *getMostFrequent(struct node *head,
        unsigned int num_to_select);
```
This function takes the head pointer of the word list and a number which is how many of the most frequent words to return (for example if num to select is 5 then the function should return the 5 most frequent words). If several words have the same frequency and should be output, what order they are output in is arbitrary. The function should return a new head list node which points to a new list of “num_to_select” nodes sorted by order of frequency; each node should contain one of the most frequent words. This function must be able to be called multiple times with different values of “num_to_select” without corrupting or damaging previously output lists.

```
    void printStringList(struct node *head);
```

This function can be used to print out a list (displaying the words and frequency).

```
    void freeStringList(struct node *head)
```

This function takes the head node of a list and de-allocates the memory for each of its nodes and strings.

```
    int main(int argc, int argv[]);
```

The main function of your program. Calls the other functions. The main function (and your program) should accept 2 required command line parameters. The first (argv[1]) is a number indicating how many of the most frequent words to output, the second (argv[2]) is the name of a file to read the word list from. So your program will be run as follows:

```
      > ./most_freq 7 filename
```

And it should print the 7 most frequent words occurring in the file named filename.

Requirements

1. Your program must be contained completely in a file called most_freq.c and up to one header file that you create called most_freq.h. You can not create any other files or name your program something else. The compiled program your makefile generates must be called `most_freq`.
2. Use malloc and free (Be efficient with memory allocation: don't leak memory!) You must free all memory you allocated before calling exit() or reaching the end of main(). Even though the OS does clean up any non-freed memory when a process terminates, when you are writing the OS code (as you will do later in the course) you do not have this luxury, so you need to practice now.
3. Use `make`. If your project does not make with your Makefile - we won't fix it or grade it. The 'make' command should compile both most_freq and queue_test.
4. Your program must compile without any warnings or errors when using all standard GCC warnings in -Wall.
5. Do not use arrays. Do not declare arrays. In this project, there should be no need to directly access parts of a string (if you use the helpful functions below), so don't use [] anywhere.
6. Do not use C++ Standard Template Library. Do not use cin or cout.
7. COMMENT YOUR CODE. If something doesn't work, you may get partial credit if your comments show that you were on the right track. Remember to put a comment block at the top of every file. **Document any places where you received help from others.**
8. You may not use the qsort function (or any other sorting library). If you need to sort (or partially sort something) implement it yourself.
9. You should check return values on any function that can possibly fail.
10. Your source code must use UNIX style EOL characters `\n` not DOS style `\n\r`. You can write code on Windows, but you must convert it to Unix text files before submitting. The `dos2unix` command may be useful for this.
11. You must submit a plain text (i.e. not WYSIWYG word processed) README.txt file that explains how to build and run your program, gives a brief description of the

pieces of your assignment, and contains any notes from discussions you had with others. **I suggest starting work on your assignment with this file.**

Hints

1. Some helpful C functions are `strlen, strncmp, fscanf, atoi, printf, strncpy` (`man` these if necessary. Pay attention to return values!)
2. When building the word list from the file, you may find it easier to keep the list in sorted order by name as you create it.
3. In the getMostFrequent() function you do **not** need to sort the full word list by frequency in order to compute the N most frequent words.

Always watch Piazza for updates on the assignments.

Submission Instructions (read carefully)

Make a tar and gzip file with your HowardU username (email address without the @.edu part) as the name.
– tar -zcvf HowardU.tgz HowardU-project1/
Where all of your files are in the HowardU-project1 directory. For example I would replace HowardU above with gedare.bloom. Your submission must consist of your source code, README.txt, and Makefile; do not include compiled output! Upload the tgz file to Project 1 on Blackboard.