Mid-Semester Exam

CS 306/491-003 Fall 2019

Total Marks: 100

*Submit your C source files *.c and *.h in a single zip file called* ***midsem.zip*** *to the "****Mid-sem Exam****" folder in D2L. Your code must compile and run without any error (e.g., Segmentation fault) or you will get an automatic zero. Other rules pertaining to cheating and copying of code from external sources apply – refer to the syllabus for the course policies.*

NOTE: All your submissions MUST check for the appropriate number of command line arguments expected to be passed via argv[] and argc variables. If not, then your codes MUST print an error message of the form: `printf("\n\tUsage %s ...\n", argv[0], ...)` and terminate using `exit(EXIT_FAILURE)`.

1. Write a C program called **readword.c** which does the following: your program should take as command line argument a filename and then count all words in the file. We will define a word/token as a string of characters that are bounded by one or more whitespace characters (space, tab or newline) on both left and right sides. Use the skeleton file provides **readword.c** to complete your assignment and test against one of the .txt files. A sample output file **readword_out** is also provided for your reference. As you can see from the code, you can assume that a word can be maximum 50 characters long and hence there is no need of dynamic memory allocation in this assignment. **[25]**

Example invocation: readword testfile1.txt

2. Use the `get_token()` function code from **readword.c** file to create a hash table of tokens/words. The structure of the hash table will be as follows: the hash table has 26 entries, one for each character 'A' to 'Z'. For each entry there will be a singly linked list, containing a list of tokens that all start with the same letter. The token returned by the `get_token()` function is first converted into all Upper Case and then passed to the `insert_in_hash_table()` function. Only tokens which start with letters 'A' to 'Z' will be inserted into the hash table. Depending on the starting letter of the token, you need to first find the appropriate location in the hash table and then insert the token at the head of the linked list corresponding to that location. For example, if the first token is "AND", then it will be inserted in the linked list pointed to by hash_table[0]; if next token is "ABLE", it again be inserted at the head of the linked list pointed to by hash_table[0]; if token = "BODY", it will be inserted into hash_table[1]; if token = "CHOICE", it will be inserted into hash_table[2]; if token = "ZEBRA", it will be inserted into hash_table[25]. Every time an insertion happens at hash_table location 'i', the corresponding hash_table[i].count is incremented by 1. Use the skeleton files **wordhash.h** and **wordhash.c** for this assignment. A sample output is provided in

wordhash_out file. Your code MUST free all dynamically allocated memory before terminating by completing the `delete_hash_table()` function and calling it appropriately in your code. **[30]**

Example innovation: wordhash testfile1.txt

3. Your third assignment involves enhancing the **wordhash.c** file from above to create a new file **wordhash2.c**. The additional enhancement is to avoid the insertion of duplicate tokens into the hash_table. For example, if the first token is "AND", then it will be inserted in the linked list pointed to by hash_table[0] and hash_table[0].count as well as the node→freq (corresponding to the node containing the token "AND" are incremented by 1; if at some point later we encounter the token "AND" again, instead of inserting it in hash_table[0], we will increment the node→freq by 1, and leave hash_table[0].count unchanged. Thus, for every new token that we attempt to insert into the hash table, we need to first check for duplicates using the `find_dup_token()` function. Finally, you need to print out the complete hash table using the `print_hash_table()` function in the format specified in the sample output file **wordhash2_out**. You have been provided with the **wordhash2.h** and **wordhash2.c** files to start your assignment. Your code MUST free all dynamically allocated memory before terminating by completing the `delete_hash_table()` function and calling it appropriately in your code. **[25]**

Example innovation: wordhash2 testfile1.txt

*As part of the next problem, you will need to use of a couple of functions from the C String Library. Specifically, you will need to make use of **strlen()**. However, for the sake of this assignment, you are not allowed to use other functions like the **strncpy()** function directly, rather you must implement your own version of these functions. This new function should be named "**my_strncpy()** respectively and should work exactly like the original functions. You will have to make sure that the output matches as well. For example, does **strlen()** count the null character? If it does, your function should. If it doesn't, your function should not, etc. You will have to look over the manual pages for both of these functions to make sure that the ones you write have the same parameters, output, and behavior.*

4. Write a C program that takes in 3 command line arguments in the following order: a string, and integer, and another string. As with the previous problem, if an incorrect number of command line arguments are submitted, the program should print an error message and stop running.

You will also have to make sure you properly convert the second command line argument to an integer that C can manipulate, since by default, command line arguments are saved as strings. The function of this program will be to insert the second string into the first string, with the input integer being the index where the string should be inserted. For example, if the user calls the program like this: **./my_program**

“aaa” 1 “bbbb” then the new string would look like “**abbbbaa**”. Once the new string has been formed, the program is to print the new string to the terminal.

A few important things to keep in mind for this program is that dynamic memory allocation will be needed to store the new string. One of the tricky problems to solve for this question is figuring out how much space the new string needs. Remember that all valid C strings end in a null character! You will also need to use your version of **my_strncpy()** here when generating the new output string. Also note that if the user inputs an index that is out of bounds of the first string, a segmentation fault will occur. Your program should be able to prevent this from happening by checking the user input and exiting with an error message if the index is out of bounds. **[20]**

Hints/advice for this assignment

- Meet or email the TA or the Professor for any questions.
- Start early as the difficulty level for this assignment is a bit higher than the last one.
- Program in stages. Have small goals and build upon them to reach the final product.
- Your code must be your own. DO NOT COPY code snippets/blocks from other students or from the Internet. Any plagiarism will result in an automatic zero.
- For your versions of **strncpy()**, it is recommended that you get your program working with the standard versions of these functions first, to ensure that your main program logic is correct. After your program is working, then you can start working on your version of those functions and insert them into your code.