CS 200: Introduction to Programming

Dr. Naveed Arshad

Muhammad Kamran Nishat

Assignment 5

(Due Date: Tuesday, December 1, 2014)

This assignment is due on Tuesday, 1st December. The usual late submission policy of 10% per day deduction for up to 3 days applies. The assignment needs to be submitted on LMS under the 'Assignments' tab.

The course policy about plagiarism is as follows:

1. Students must not share actual program code with other students.
2. Students must be prepared to explain any program code they submit.
3. Students cannot copy code from the Internet.
4. Students must indicate with their submission any assistance they received.
5. All submissions are subject to automated plagiarism detection.

Students are strongly advised that any act of plagiarism will be reported to the Disciplinary Committee.

Search Engine

In this assignment you will be required to make a search engine. Don't worry, not a full blown online version, for that you would need to know some network programming as well ☺ You will be rigorously tested on hash tables, linked lists and pointers (important concepts) but in the process you will be doing so by seeing how it can be applied to make an efficient search engine. Now let's start off.

Introduction/Requirements

PART I [50 points]

You are all familiar with search engines. In this task you will be required to implement it using the linked list data structure (which you have also now had practice with in the lab). A very important point of this assignment is to see how efficient you can make your search engine. Your code will be timed. To test your own code, take the start/stop timer functions that you were given in the lab.

You will be provided with the file, data.txt. This contains the information to be used in your search engine. Its format is that the first line will be a URL then an n number of lines which will be the information/text associated with that webpage. Then there will be a blank line. Then again another link and some lines of text then a blank line and so on. 2 blank lines signify that we have reached the end of the file. Each URL may have different number of lines of text. You are being given a relatively small "data.txt" but while grading, your code will be tested with a much larger file.

Your search engine should:

- On startup, read the data from data.txt and create a linked list based on the information in that file
- 1 node in the linked list should have one URL plus associated data stored in it (at least).
- After that, a menu should be displayed that should show all the functions that can be applied on the search engine.
- These functions should be to "search", "reload data", "save data", "quit and save data", "quit without saving".

Requirements of each function are:

1) search: should ask the user to enter a word or a phrase. Then it should search through your data and show the URL and associated data if URL and/or associated data contain the word/phrase. And the phrase should be broken down like Google does it. If I enter "how to

fly", it should show all searches with "how", "to" and "fly" (all the keywords of the phrase need not be present for the data to be shown, if it contains only "how" then it should be displayed). And data should be displayed according to higher rank on top. Rank is defined below.

2) reload data: this should clear your linked list of all present data and then load "data.txt" again.
3) save data: should ask the user for input to give an output filename. Then for keyword/phrase. It should store all data that contains the keyword/phrase in a new file with the initially inputted filename.
4) quit and save data: quit should de-allocate all the memory and close the program. But before quitting, it should save data that the user asks for.
5) quit without saving: it should de-allocate all memory and close the program.

Rank: if the user inputs a search phrase of "how to fly", a URL containing "how", "to" and fly" would be of higher rank as compared to a URL containing just "how". URLs having the search word appear more often would have higher rank. E.g. URL with "fly" appearing 3 times would have higher priority as compared to a URL with "fly" appearing twice. But you can assume that "fly" appearing 3 times is the same rank as "how to fly" appearing just once. You have also been given a list of words in a file "lowRank.txt" which contain a list of words that would only increase priority by one, regardless of how many times that word may appear in the data. These words are basically often used words like "the", "and", "he", "she", etc.

PART II [50-points]

In this part you will be required to do inverse-indexing to improve efficiency of your search. The simplest way to visualize an inverted list is to think of the index at the back of a book, each word in the index is followed by a list of page numbers where it occurs, this is a classic inverted list.

You will implement the inverted list by using the hash table that you just learned about in class. You will still have the linked list as the main search engine. Make the hash table by using an array. Each slot should be storing a word and the corresponding indexes that contain the word. These indexes should be the corresponding node in the linked list (**with index starting from 0**). Note that final output should still be displayed in higher rank order.

Use any hash function that you would like to map the word to hash table index. However, if there is a word that is mapped to a table slot that is already occupied by a word and its index values, then store the clashing word and its corresponding indexes in the next free slot of the hash table.

Make sure that you do all the appropriate book-keeping. Just leaving dangling pointers can cause massive errors as you progress. Even if it doesn't, it is bad programming practice. Marks ARE also allotted for neat display of all information. This assignment is open-ended; you are allowed to do it any way you like. As long as you fulfill the above mentioned requirements, feel free to try any method that you may have learnt in class (or for that matter any way that you may have read up on by yourself).

Tips:

Now we don't mean to leave you in the lurch. So here are some tips and guidance in case you are stuck on how to go about it.

- First of all, though it is mentioned that you can also implement other methods learnt outside class, a very important point to note is that you do NOT need anything from outside class material to do this assignment. In fact our advice is to not use anything excessively outside what has been taught in class otherwise you may confuse yourself in unnecessary syntax.
- Though it is written as a tip, it should by now be practically a requirement: make functions. What this helps you do is break your code down into parts. And we don't mean just make functions for "search", "quit without saving", etc but also use your own functions inside these. This way, if a bug comes about, it becomes easier to pinpoint where the problem could be as opposed to just looking at a big block of code.
- Write one function and see if it works. Always test in parts. Don't write a massive chunk and expect it to work right the first time, that has been rare for even your TAs.
- One often used way to debug is to use cout statements. A logical problem can't be just fixed by looking at the code. In the case of segmentation faults, using cout statements can help to pinpoint till how far the code is running correctly. Then you can see where exactly you made the error. And even otherwise, using them to see whether your variables are holding the right values at that certain point can make debugging a quicker process.
- Note you are allowed to use any type of linked list. Singly linked lists, doubly linked lists, with or without tail pointers. It is all up to you. Just note one thing though, using doubly linked lists would mean that you have to do more book-keeping when you add and delete nodes and that is thanks to the "node *prev" pointer (which is to the previous node) in your node struct. However, if implemented correctly it may be faster than singly linked lists. Not to say that singly linked lists will be slower, it all depends on how you implement your search engine.
- For your inverse-index you might be confused as to what type of exact hash function to use. Use a function that you feel would get you more distinct and well distributed values. This would help in the case of getting fewer collisions (finding that there is another word already occupying the hashed value/index). Think over it and you will realize that why this would help in efficiency.
- Think simple. Complicated algorithms are not required. And more often than not, simple methods tend to go faster than complicated ones.