Michael Wexler

Wordstat Readme

Instructions to run program

Open up a terminal. Navigate to directory holding source code. If you didn't already do so, enter “make” to generate output file.

To run this program, enter

./wordstat <option> <filename>

Option refers to the following:
-l : number of lines
-w: number of words
-p: palindrome’s statistics.
-pf: palindrome’s frequency

filename refers to the input file which should be located in the same directory as the source code.

Summary of design

The design is very modularized so as to minimize the difficulty of debugging. In main method, the option chosen is examined, and branches out to different helper functions based on what was chosen. The main part of the project was parsing and storing the palindromes from the input file. The design used was a linked list consisting of self-referential structs. Nodes were kept in alphabetical order based on word, and their counts were updated for repeats.

Running time analysis

Counting lines/words in the input file was O(n), as was printing the linked list. The runtime is directly proportional to the amount of lines/words/nodes. Also converting the input file to a character array was O(n), as were several other tasks. Checking for a word being a palindrome was O(c*n) = O(n).

The most time-consuming part of the design was the storage of palindromes in a linked list. This was $O(n^2)$. Essentially, as palindromes were found, they were put into nodes, and a linked list was searched for the correct alphabetical position. If there were n palindromes, then the average searching was n/2, so the run-time was $O(n*n/2) = O(n^2/2) = O(n^2)$.

Space requirement analysis

Essentially, the three most data-storage-intensive parts of the program consist of:

1) Getting all the text from the input file to an array. Could be very high if the text file is large.
2) Creating a linked list, with each node having a count and a pointer to a palindrome (see (3))
3) Allocating memory to hold the palindrome. If there are a lot of unique palindromes in text file, this memory could be quite high.

Challenges encountered

I often had to deal with “segmentation fault” errors. The meaning and cause of the errors were very elusive, and the compiler did not tell me what line was causing this. I had to use GDB (debug) to step through the program and examine what was causing this error each time.

I was under the impression that I could instantiate nodes using WORDNODE node. However, this was causing those segmentation faults. When I actually used dynamic memory allocation, i.e. WORDNODE *node = (WORDNODE *) malloc(sizeof(WORDNODE)), this got rid of that problem. I think this was because static allocation is only temporary to the scope that you were in when you declared it. I think dynamic allocation is more appropriate for linked lists.

I am very used to using Eclipse with Java, which provides a lot of tools to make coding easier. In this project, I only used a text editor to write the code, and wrote commands to the terminal to compile it. This made it much more challenging to write code, and also made it much more difficult to debug.