Jeff DiPaola

CS 214

11/10/14

Assignment 4 – Overview

I would like to begin by calling your attention to some assumptions I made throughout the development of this assignment:

- **Case Sensitivity is not an issue** – One of the requirements for the indexer assignment was that it should be case insensitive, that is, it should convert all characters to lower case to provide case insensitivity. I am assuming that the user of the search program knows this and therefore knows to enter all search terms in lowercase.
- **Inverted Index File will be setup appropriately** – Being that this program, in its description, states that it is to be used in conjunction with the indexer, I am assuming that the inverted index file passed to the program will be of the proper format. Therefore, no error checking was done on the inverted index file itself because the user should not be editing the inverted index file in anyway.
- **The counts from the inverted index file are not needed** – The count of how many times each word occurs in the each respective file does not come into play here. Therefore, when reading the inverted index file into memory I omit the count associated with each file name as it would just take up more memory and be unused here.

I developed my program as two separate files, according to the program description. To begin, I developed the "reader.c" file, which is responsible for the actual reading of the inverted index file into memory. Our main C file, "search.c" starts by taking the argument passed to it's main, the inverted index file's name, and passing it to ReadIn, the primary function inside the reader c file. The reader file then parses through this line by line, identifying the words and their associated file names and storing them in the appropriate parts of a linked list structure very similar to the one I used in the indexer assignment.

The structure I am using to store the file contents in memory is basically an elaborate linked list. The primary structure, called List has a 50 char array called "word", a FileList object called "files" and a next pointer. The FileList object is a secondary structure that contains a 200 char array called FileName and a next pointer. Basically, the idea here is that each word has its own list structure and they're linked in a linked-list like manner. Additionally, each of these "word lists" has its own sub list of the files it contains. It is also important to remember at this point, that per the indexer assignment instructions, the words are read and therefore placed into the structure in alphabetical order. However, the file names are not as that wasn't a requirement in the original indexer assignment. This data structure is probably our largest memory taxing object at the moment. If large files containing multiple words were read in, the structure would undoubtedly fail.

Once the reader has finished reading all the contents of the inverted index file into memory, the start of our memory structure is passed back to the main function in "search.c" where rest of the user's requests will be completed. Once we receive this, we enter a while loop that prompts the user for a search term. Here, I'd like to take the opportunity to point out that although I did make the assumption that the search terms themselves would be input in lowercase, I did not

make that assumption with the search functions themselves. For example, “sa”, “SA”, “sA”, and “Sa” are all appropriate search terms and will function as they are supposed to. The while loop checks the user’s input, and as long as it matches one of the three terms we are to look for, it takes the proper action.

In the case of the “sa” command, the function “AndSearch” is called with the argument of the string that the user passed. I begin by removing the “sa” command from the string and then indexing the next term up until the space. When the first word is found, a FileList (basically a string linked list) called output is populated with all the files that the first word is found in. From there, if there are more words, a different aspect of the loop is called into play where it will check each linked list entry against each file entry in the word we are adding. If the linked list entry is not found within the file list for our subsequent words, it is removed and then we proceed. If all entries are removed a 0 is returned to indicate an appropriate response. If, after all words have been processed, there are words remaining in our linked list we know that these are matches and we output them to the user. Finally, the function returns a 1 allowing main to resume the loop of prompting the user.

If a “so” command is used, the function “OrSearch” is called upon and acts pretty much in the same manner. It is passed the entire query issued by the user and processes it into words and searches for each word in our inverted index file. Additionally, it also stores the output in the same FileList structure. The only difference here is that, instead of checking for duplicates in the file names it is checking for missing files to add. If a file name is already in the structure, the program moves on to the next file name. However, if it is not, the program will reach the end of the linked list structure, realize that, and add the new file name before continuing on. Once all files have been processed, this function too outputs the results and then returns a 1 to the main program to indicate a successful run. This function is basically the same except for how it handles the addition/removal of filenames.

As previously stated, the aspect of this program that is the most taxing on memory is the structure in which the data from the initial inverted index read is stored. It is important to mention, however, that the search queries also generate small string linked lists to store and manipulate the results. This too could probably grow to be pretty large in a case where multiple file names are present and multiple words are passed with a “so” query.