Amar Bakir
CS 214
Project Assignment 3: Indexer
Monday, October 20th, 2014

Design:

My program has four main parts: the tokenizer, the main sorted list, sorted lists, and the indexer. The indexer uses the other three parts (which have their own data structures and functions) to complete the task of indexing all files given as input. The tokenizer is responsible for creating tokens from a file stream, while also making sure that every token is properly formatted (all lowercase) and does not contain separator characters. This is achieved by using a buffer to read in large chunks of the file at once and then parsing through the buffer to get the token when asked.

The main sorted list consists of the nodes with a token and corresponding sorted list, which will store the files the token appears in and that token's frequency. The sorted list contains nodes which hold a filename (represented as the path from the input directory, if there is one) and a frequency. Both lists are in the form of linked lists with a front and each node holding a next value. The front is stored in the main sorted list or sorted list structs, which allows the program to perform various search and reorganizing functions.

Finally the indexer uses the previous parts in order to create the lists. It creates the main sorted list by creating tokens using tokenizer and then inserting (or searching) for these tokens in the list. When the node is created or found it searches through the normal sorted list associated with that main sorted list node for the filename, and if found it will increment the frequency count. Otherwise it will create the proper node and set the frequency to 1.

The indexer contains the function indexDir which can recursively index directories if they happen to have directories of their own. It also calls indexFile if it happens across a file. IndexFile uses the tokenizer to actually build the required parts of the main sorted list tree, which indexDir is more responsible for the file I/O and maintaining path names, all the while keeping an eye out for errors.

When everything is created, a function names output in indexer creates the output file in the requested file, after which the structure is freed from memory, and the program closes.

Running Time:

The worst case running time for this program is fairly simple if one is to ignore all the details that only lead to coefficients that are removed anyways. When you consider the main sorted list structure, there is a node for every unique token. Each token then has a sorted list which has a node for each file that token appears in. Assume you have m unique tokens and k files. Also assume there are n tokens altogether. The program searches through the list first to find the token, then through the token's sorted list to find the file entry. With that in mind, in the worst case the program must run through the entire main list of m nodes, and then run through the sub-list of whatever token it finds for at most k nodes, since there are only k files. In between the program performs memory allocations and pointer reassignments, and these will amount to cn, where c is some constant. So in total, the program, in the worst case, performs m*k operations n times, plus cn => nmk + cn => n(mk + c). Since this is the worst case and I am computing O(), the cn is irrelevant in the face of the nmk, and so O(n) in the absolute worst case, since mk is also a constant. On the other hand, this worst case is extremely unlikely to happen, as the computations overestimate by a lot in order to achieve O(). One thing to note, is that for enormous files, n will be much larger than m*k, so O(n) is reasonable, but if somehow m*k is larger than n then O() will be closer to $O(n^2)$ or simply O(nmk).

Memory Usage:

Memory usage is much easier to compute. Using the variables from the running time analysis, n tokens, m unique tokens, and k files, there will be at most m allocated nodes for the main sorted list, each of which can have a max of k allocated nodes, one for each file. Counting each node as a single unit of memory (not an actual bit or byte, just a unit for the sake of the analysis), this will take at most m*k nodes. Each node holds a certain amount of data, so the it would be (m*c)(k*i) where c is how many bytes a single main sorted list node has, and i how many bytes a sorted list node has. So the final memory usage is mkci bytes of memory.