/* CS 214: Systems Programming Spring 2018
 * Professor Francisco
 * Assignment 2: Keyspace Construction
 * Richard Farrington netid: raf172
 * Anirudh Tunoori netid: at813
 */

readme.pdf

Our implementation of the invertedIndex has a reasonably efficient time and space complexity. In terms of the space requirements, we utilized a few primitive character buffers to read from files and to construct tokens. These data structures were reusable, and to save space we made liberal use of malloc() and realloc() to in order to only use as much memory as is needed. Our overall data structure, which was used to build this inverted index was a single (large) doubly linked list. This linked list would consist of nodes that contain unique tokens associated with each file. Recurrences of a token within the same file did not demand new nodes; a counter kept track of the frequency of the tokens. We went with a linked list data structure as opposed to a hash table or multiple linked lists in order to avoid complexities in space, implementation, and etc. Furthermore, our space complexity, as a result of our reusability is O(n) where n is the total number of tokens across all the files. This linked list afforded us with other advantages. We were able to sort as we built the list, thereby saving time and space and once all the files were processed, being equipped with a sorted list enabled easy printing and dynamic freeing. Other implementations could have greatly influenced the modularity and complexity of our code. Proceeding with a doubly-linked list our a singly linked list helped us traverse our data structure more effectively.

When it comes to the time analysis, our implementation operates under a runtime of O(mn), where m is the number of files our indexer has to process, and n is approximately the number of tokens/words found within each file. We also feel that using read() and write() in our implementation helped our program run (even marginally) faster. Separating/differentiating the read process between files and directories helped us process the entire directory more effectively in addition to using recursive read calls for directories in order to ensure that every file was parsed. We also had a reasonably efficient process of parsing tokens, which was inspired by an earlier assignment.

Finally, one thing that we noticed was that regardless of the number of files and directories we used, the index was built relatively quickly. In particular, we did not notice a significant difference in runtime between multiple files with a few words and a single file with a lot of text.