Kinh Hoang khh39
Tim Goetjen tag135

Indexer-Readme

Introduction:

We have created an indexing program: Indexer, which creates an inverted index of mapped to the files which they were found in. The program takes in an output file and a directory or file name. It then traverses and indexes the files underneath content, and returns the output in JSON format via the output file.

Design:

For file traversal and indexing, we use the stdlib nftw() file tree walk function. The file tree walk function walks through the directory tree that is located under the directory or file given to it, and calls a function passed to it once for each entry in the tree. It is here that we open each valid file and tokenize its content; creating, hashing, and updating record objects as necessary in the hash tree where we store our tokens. (Note: we do not allow "." and ".." as a starting index for the traversal.)

We use a Hash table to insert and lookup with constant time, since these operations will be performed the most, and because our data contains only unique items. Each Record object represents a token-file pair and contains a count of the number of occurrences in each file.

After nftw() tokenizes and stores the data in our hash table, we then pull out our data into an array of record objects, and sort our data using the highly optimized C stdlib qsort() function. We pass qsort our unique comparison function, which sorts our records by word, then count, then filepath.

Lastly we print our data to the output file given, in JSON format, using the sorted Record array. (Note: In traversal, if we encounter the output file we notify the user and ask if they wish to overwrite the file.)

Big-O:

Let **"t"** be the number of tokens in our entire file tree and **"r"** be the number of unique tokens in our file tree:

Our program on average traverses and tokenizes at constant time, **t** times [O(t)], qsorts **r** records [O(rlogr)], and prints **r** records in JSON format [O(r)]. This gives us a runtime of O(t+ rlogr +r).

Our program stores **r** records in a hash table, and **r** records in an array to sort and print, giving us an average space complexity of O(r).