ReadMe

Our indexer consists of three distinct parts; a recursive function that walks through the directory structure provided to it, a tokenization portion that removes the tokens from individual .txt files, and a storage mechanism which would count the frequencies of those tokens.

The recursive portion of the program simulated walking through a general tree-type structure, so the total traversal time is O(n), where n is the number of items in the file system. In terms of spatial complexity, the recursive portion did not create or delete and anything. Since this portion of the program didn't really use any space, its space complexity was constant. The function worked by considering the root directory it was given. If that root item is a directory, the function would begin recursing through the system. If the root item is a file, then the program would directly process that file. If otherwise, then the program would exit because there is no defined behaviour for objects that are not either directories or .txt files.

The tokenization portion is a little more complex. The tokenization function opens the file and then checks to see if the file is empty. If it is empty, then the function returns because there is nothing to tokenize. If it is not empty then the function will open the file, extract its contents, and begin searching for tokens. The function does this by scanning the array of characters from the file took look for non-alphanumeric characters. Once an alphanumeric character has been found, then all the letters up to that non-alphanumeric character will be copied into a temporary holding array. Once the token is loaded separately, then the system will shift all the characters to the left by an amount of elements equal to the length of the current token. This allows the token-selection process to start at the beginning of the string every time. For non alphanumeric charactes the alogrightm will skip that character and continue. This algorithm is linear in time because of the time taken to access every token in the hash table. It is linear in space too because the number of remaining tokens decreases linearly s the algorithm progresses.

The insertion process relies upon a hash table and the SortedLists we implemented in assignment two. The hash table we used was from an open source location called uthash.h . It was written by Troy D. Hansen and can be found here.

UTHash made creating the hash table quite easy and allowed us to focus our efforts on setting up the SortedLists correctly.

Now the hash table is basically used to store the unique words, their frequencies, and the filename where each occurrence of the word is stored. The bucket of the hash table is the unique spelling of the word; for example, if the word “apple” appears in “file2.txt”, the bucket in the hash table will be “apple”. The bucket points to a SortedList that we implemented last assignment; the comparator function for this struct shall be CompareInts. What’s stored in this SortedList is a node that contains the filename of the file that contains the word, and the frequency as to how many times the word appears in the list. Following the previous example, the first time “apple” appears in the program, there will be a bucket in the hash table with the key “apple”, pointing to a SortedList whose head shall be a node pointing to a key of “file2.txt” and data pointing to an int of “1”.

Every time a word appears, the program searches the hash table to see if there is a bucket already created. If there isn’t, it creates a new bucket with the word as the key, and creates a new SortedList with that word’s filename and frequency of 1 as the head of the list. If the word has appeared before, the program accesses the SortedList and goes through it to find a node that matches its filename; if it doesn’t find the appropriate filename, it creates a new node with the filename as the key and a frequency of one. If it does find an existing node with the same filename, the program just adds one to the frequency, and the SortedList resorts itself based on frequency. (It is important to note that the HashTable will not store duplicates to the same bucket nor will the SortedList store duplicates. We had to specify this, or otherwise, the program broke.)

Finally, there is another SortedList that just contains the individual words that are found in the program. This SortedList’s comparator function is compareStrings, as it sorts all the words by alphabetical order. This exists for the printing function of the program; the hash table isn’t sorted alphabetically but rather by hash codes. The most time efficient way to print out all the words in alphabetical order is to simply keep a pointer on the head of this second SortedList. When we’re ready to print, we simply access the key stored in this node and search the hash table for the word. This requires O(1) time. When we access the bucket’s SortedList, we just print out the filenames and their frequency starting from the head and moving on to the next node in the list; since it’s sorted by frequency, we can just print the filenames knowing they’re in the right order. After that, we access the next node in the SortedList that’s keeping track of all the tokens in alphabetical order, and repeat until we hit the end.

The memory requirements are as follows: O(n) to store the hash table, where n is the number of buckets, O(m\*n) for the SortedLists where m is the number of nodes in the SortedList plus O(n) for the SortedList that stores all of the nodes for printing.