
|Assignment: Project 2 Part 3 |
|By: Lindsay Bloom |

Acknowledgements: Elena Cokova, Gabriella Bova, Bill Yung, Lucy Purinton,
Rebecca Newman, Vivek Bolikar, San Akdag, Matt Shenton,
Sam Usher, stackoverflow.com, cplusplus.com

Overall Purpose: to implement a program that indexes and searches a file tree for strings and indexes each string. When a query is entered, the program shall print out all occurences of that query along with its path and line number.

Files

DirNode.h: Written by Vivek, the header file for the DirNode class that makes up an N-ery file tree.

FSTree.h: Written by Vivek, the header file for the FSTree class which is the representation of a file system.

FSTreeTraversal.h: The header file for the FSTreeTraversal class, which includes the class declaration.

FSTreeTraversal.cpp: File that includes the tree traversal method that prints out the full paths of each file in the tree on seperate lines. The program takes the highest directory as a command line argument and then prints the full paths of all of the files accessible from that directory.

Indexer.h: The header file for the Indexer class, which includes the class definition.

Indexer.cpp: File that includes the hash table and indexing implementation. This file indexes each file from each directory into a vector. The hash table implementation is an array of Node that includes 2 strings - the case sensitive version and insensitive version - as well as a set of size_ts.

Searcher.h: The header file for the Searcher class, which includes the class definition.

Searcher.cpp: File that has run functions that run the implementation based on the type of sensitivity inputted by a user.

main.cpp: File that takes command line arguments and input from the user and calls the appropriate run function based on input from a user.

How to compile

With Makfile:
make

Without Makefile:
clang++ -Wall -Wextra Indexer.cpp FSTreeTraversal.cpp Searcher.cpp DirNode.o \
FSTree.o

How to run

./gerp [DirectoryToIndex]
Query? [mode][wordToFind] - note: mode is optional

For insensitive mode: @i or @insensitive
For sensitive mode: do not add anything
To quit: @q or @quit

Architectural Overview

My program is made up of 5 modules: the DirNode, the FSTree, the FSTreeTraversal, the Indexer and the Searcher. The DirNode class creates the nodes that make up the FSTree. The FSTree implementation keeps a pointer to the root DirNode. The FSTreeTraversal class then uses the FSTree class to create an instance of an FSTree and passes in the top directory. This implementation then traverses through all paths of each directory and stores them into a path vector. The Searcher class then creates an instance of the FSTreeTraversal and passes in the directory that is passed in from the command line to make the vector of paths. Once all of the paths are retrieved, an instance of the Indexer is created and the vector of paths is passed in. The Indexer class then uses the vector of paths that are passed in to access every file in the directory and index each word of each file. Once the words are indexed and a query has been entered by a user, the Searcher class then calls the respective print function from the Indexer class. main.cpp then calls the respective run function from the

Searcher class in order to run the entire implementation.

Data Structures and Algorithms Used

The first data structure that was used in the implementation was a File Search Tree, also known as an N-ery Search Tree. This data structure was selected by the COMP15 staff, however, I imagine it was selected because of its logn search time.

The second data structure that was used in this implementation was a vector in the FSTreeTraversal class. I chose to use a vector because every single path needed to be printed, which has the Big O of O(n). A vector also has O(n), which is why it was an efficient data structure to use in this implementation.

The third data structure that was used in this implementation was another vector in the Indexer class. This vector, called lines, was used to store every single line, along with its path and line number, from every file. This way, the files were only copied once, as opposed to storing each word and its corresponding lines, which would involve storing the files multiple times and would take too much memory.

The fourth data structure that was used in this implementation was a hash table. The hash table that I created is a pointer to an array of a struct called wordNode. wordNode is made up of 2 strings and a size_t set. One string is the word and the other string is the case insensitive version of that word. The set is made up of size_ts that are the indicies of that word in the vector of lines if that word appears more than once in a directory.
So, when a user enters a word, the program finds that word in the hash table, and then goes to that node's set of indicies and then prints out each index of that set from the lines vector. In order to solve collisions, linear probing was used by finding the next available slot in the array.

The fifth data structure that was used in this implementation was a linked list within the hash table. In order to deal with case sensitivities, each word is hashed by its insensitive version. If a word has already been stored, but has different letter cases, then it is chained onto the list of words within that index of the hash table.

Testing

In order to test my implementation, I would test each class seperately and then

would test classes combined. To test my FSTreeTraversal, I created a test file that created an instance of the FSTreeTraversal and passed in a directory that I made myself.
To test the Indexer class, I created a directory called "places" that included countries and within those countries were either directories of cities or files of cities. Within the city directory there were landmark files of landmarks that actually exist within those cities. The places directory made testing very easy because it was easy for me to know which directories had subdirectories and if the line that was printed was from the right file.
After I was confident that my Indexer was working, I tested with the data that was provided to students. When testing, I would compare my output with the_gerp to ensure that there were no differences between my output and the_gerp's output. I also tested the time it took to index the files and the amount of storage used to ensure that I did not go over the limit.

When I ran into bugs or Segmentation Faults, the best debugging method was to use cerr statements. Though it took a long time to find the source of the bug, cerr statements always led me to where the problem was.