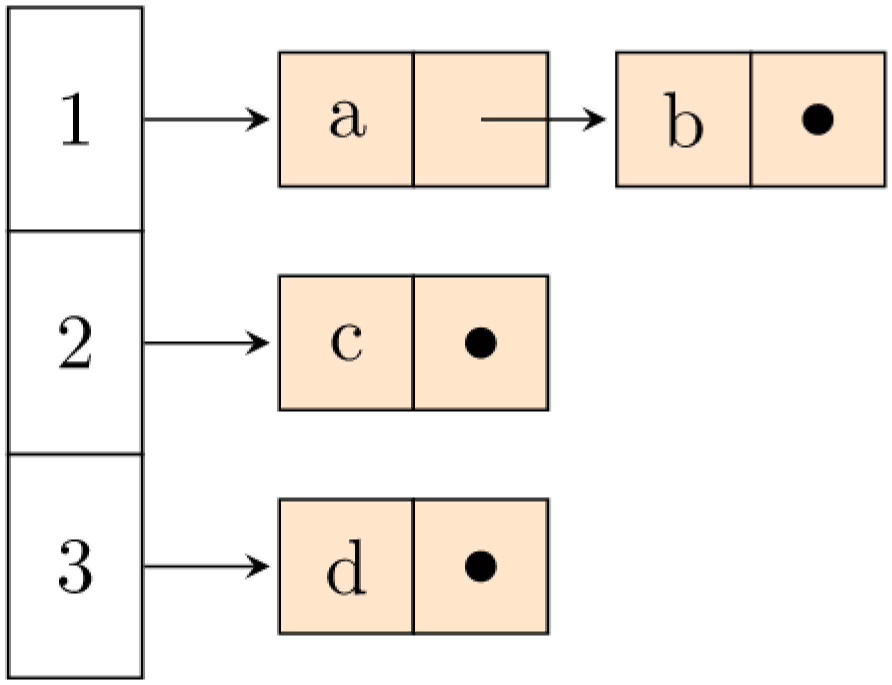
Assignmen 3: Indexer

Name: Gregory Grzymski

The primary data structure used for this assignment is a linked list, which was held responsible for storing tokens and file name records. My assignment design followed this diagram:



Source: http://tex.stackexchange.com/questions/86766/array-of-linked-lists-like-in-data-structure

This represents an Array Linked list, where the vertical list represented my token words, and each word had a horizontal linked list of records connected to it.

Each Word Node contained a char *field for the name of the token, a next Node field, and a pointer to the word's linked list of records.

Each File Node contained a char *field for the name of the file, and an integer frequency count to monitor how many times the name exists in the file.

Tokenizer.c

The tokenizer was constructed from utilizing the C stdio.h and ctype libraries, primarily using the functions: fgetpos, fgetc, fgets, and fsetpos. The ctype function isalnum allowed me to check for alphanumeric legitimacy in the construction of my tokens.

Fgetpos kept track of the position in my file stream, while fgetc would return an individual character read in the file. I would then allocate memory for a string to store the token once a non-alphanumeric character was encountered.

The tokenizer file had functions to create(load the file stream) , destroy (close the file stream) and to find the next token in the file stream.

Sorted-List.c

The sorted-list.c file handled the construction, maintenance, and insertion of the linked list data structures.

SLInsert handled the insertion of a token from the file stream. If the token didn't exist in the linked list, a new Node was created, along with a new File linked list, and it was inserted based on ascending order.

If the token did exist already, then FileInsert was called in order to just increase the frequency count of the token in the file that was being streamed

FileInsert handled the insertion of a file record into the file linked list. If the file existed in the list already, its frequency count was increased and then removed from the linked list. A new node was created with the removed node's information, and was properly sorted based on the descending order of the frequency count. If a file didn't exist, then a new Node was created and appended to the end of the list with a frequency count of 1.

SLCreate and FileCreate handled the memory allocation of creating the two linked lists. SLDestroy and FileDestroy handled the responsibility of properly freeing the allocated memory.

Note in SLDestroy, FileDestroy was called on the linked list before the word node itself was destroyed.

Index.c

The index file contains the main driver of the program with two compare functions to help facilitate the sorting of the linked lists. Note, the Word linked list is globalized to facilitate transition between functions of the index file.

Process is the main driver of the program, where the files/folders are processed. If the structure is a directory, then the directory is recursively searched for more directories and files. Once the files are found, each file is handled individually with a Tokenizer object created on the file stream. The tokenized word and the file name are then inserted into the word linked list, which subsequently calls the FIleInsert method. This process continues until each file is processed within the directory and its subdirectories.

Processdir handles the individual processing of a directory. Files and folders are concatenated to a malloced string field with the original path name, which gets returned back to process to check whether the field is a file or folder.

writefile handles the printing of the program's results into the inverted_index text file. Specific format is followed by assignment descriptions, and the stream is closed once all printing is done on the linked lists.

<u>Efficiency Analysis</u>

The most expensive operation in this program is the insertion of a token into the Word linked list, and that will be the operation focused on for running time calculation.

When you have w words and f files, the worst case scenario is when the word traverses through the whole list and ends up being the last node on the list, this requires O(w) time. Then, the file name is very the last file in the linked list with the lowest frequency count, O(f) time. The file is then removed in O(1) time, and the worst case is that the list is traversed again O(f-1) times. The file frequency count update operation costs O(f^2).

Since it would cost O(w) time for one word, the inserting all the words in the worst case would cost O(w^2) time, with O(f^2) time per word node.

So the worst case running time is: O(w^2 + w* f^2).