**I.  Introduction:**

The purpose of this assignment is to replicate some of the behaviors of a search engine. Basically the concept is to index each relevant word, keeping track of its occurrences per input file. The input files should be considered as movies; therefore the search engine is in order words, a search engine for movies. The program starts off by going through every file (movie) and indexing every word and movie it belongs to. After the indexing is done, the user will have the opportunity to insert multiple words to conduct against the search engine. The words provided by the user will get matched in the engine based off on relevance (occurrence in the file/movie) and eventually sorted out using the same criteria. After looking for all the possible matches, the top results will get presented to the user. The user can perform multiple searches with multiple words at the time.

**II. Contributions by each team member**

This project was done independently. But acknowledgements should be given to Alex Santos and Christian Rodriguez for comparisons and discussions of ideas.

**III. Summary of Design:**

**High Level Overview:**

**Search Engine Buildup:**

I will describe this part as a list of steps that occur during the creation of the search engine itself.

1. Create a “Set” of “stop-words” by reading the words from an input file.

2. Going through every file (movie), insert/append every word that is not a stop-word in the engine.

a. The word should be sanitized (remove non-alpha chars, uppercase letters, and possessive indicators)

b. Insert happens when a word was not previously indexed in the engine. If this is invoked, the word will be added and the current movie will be used to create its reverse index.

c. Appending occurs when the word is already on the engine. When this happens, a search inside the contents of the current word is done. If a match with the movie title is found, then the counter for the movie title will be increase. Otherwise, the movie will be added to the contents of the word with a counter value of one.

3. The engine is built.

**User search:**

The following list will explain the steps that occur inside when the process of user search is triggered.

1. Identify if the user wanted to exit.

2. Obtain the words that the user gave as input and by using a delimiter, split them in a list (vector).

3. The list should be compared against the engine in order to remove those words that are not indexed.

4. With the reduced list, perform an intersection between the movies of each word.

5. The movies in the intersection shall be sorted in descending order.

6. The top movies will be shown to the user.

7. Wait for another user input.

**Design of Classes**:

**Engine class**

This class extends a HashMap (unordered map) and is the main class of the program. This class will keep track of the “stop-words” that should not appear in the engine. It also performs the additions of the words and the search. And more importantly it keeps track of all the words that have been indexed.

Since it’s a HashMap, its keys are the words. That how the index of words is contained. The value per key is a Histogram (will talk about this later) that will keep track of all the movies and the words occurrences per movie.

Its methods are:

addToEngine

search

buildStopWords

addToEngine will check if the word being processed is not indexed already and it will add it to the engine and starting its Histogram. If it’s already indexed, then the Histogram will make sure to increase the counter in the specified movie if it exists, if it doesn’t exist it will add it to the Histogram.

Search will create a List based off the intersection of all the movies in the specified words to be searched.

buidStopWords will read the words from the stop-words file and will generate a “Set” based of off them.

**Histogram class**

This class contains a HashMap (unordered map) that will serve as a reverse index for the movies. The keys for the HashMap will be the name of the movie. The value is a counter for repetitions of a word.

Its main methods are:

getIntersection

add

toList

add will basically take the movie title and use it as the index for its location on the reverse index while setting the counter up to one repetition. If the title is already indexed then the counter will be increased by one.

toList will convert the current Histogram into a LinkedList of Books which will be sorted in reverse order. It’s used when the intersection histogram is built. The produced list will be sorted.

getIntersection is the main function when doing searches. It will find all the movies the specified words have in common and it will generate a hybrid histogram of the movies.

**Book class**

This class is more of a helper class. It was created in order to visualize and understand better the concept of a Movie outside of the maps. When searches are performed, the result is a List containing “Books”. The Books are the representations of a movie outside of an index. It contains the name of the movie and a counter. The counter is the total repetitions of the specified words in the search query.

**Design Decisions**

The selection of HashMap in the “stop-words” was to emulate the behavior of the “Set” of my favorite course, Discrete Math. (Implicit sarcasm). The “Set” behavior would allow the search to be close to O(1).

Same thing applied for the Engine and the Histogram which both use a HashMap. The HashMap would provide a fast method for searching and inserting which are the main methods used in the program. Putting the Histogram inside the Engine instead of a parallel Map seemed like an obvious reason for Space Complexity and simplicity of the management of the resources.

The methods of the Histogram and Engine were designed so they could be modified at any given time. For example, if the project was to be modified to add more files on the users command. The code could easily be modified to do so.

The sanitation of words was something carefully studied. Compared to how the operating system performs searches and how it would be more logical to clean the engine of unnecessary characters and fluctuations between words.

In order to accept multiple search words the code was modified to calculate intersections of two words at the time. For the time being this is the best implementation so far, but more research could be done to improve the search time and flexibility of the engine. I just tried to exceed the requirement stated at the time by allowing multiple word searches.

The use of the Book class was merely to further simplify the management of moving data (a movie for example) to another data structure. In this case a LinkedList was the preferred data structure. Sorting a LL is less tedious and faster than attempting to sort a HashMap which doesn’t actually have an order.

**IV. Debugging the program:**

The major issues were the initial discrepancies in the search before the words were not sanitized. After that, everything was very straight forward.

**V. Testing of the Software:**

Most of the validation was done with small sets of the data. The indexer would print all the values and they would be manually checked in order to see the data was accurate.

Then the operating system search (Finder) was used to validate that the big data set was accurate in search results.

**VI. Manual:**

The folder architecture must be the following for now.

./moviesdb

./stopwords.txt

./engine/source files

Compile using the Makefile.

After running the program will guide you through the process.

**VII. Requirements not implemented by the program:**

The requirements were all met.