General Implementation:

Functions of search.c

int main();

- The main method uses argv to acquire as an array the command line inputs. Index 1 of this array is the number of buckets as a string. This is converted to an int using the stringToInt() method. The number of search terms is then acquired by using the num_query() method. The query is read by the read_query method if the number of search terms is greater than 0. An array of strings is returned by the read_query method, splitting the single string search query into its individual terms. Glob is used to parse through the files of p5docs and acquire the array of filepaths stored in paths. The num_files variable is also acquired through glob, indicating the number of files. At this point, if the number of files is NOT 0, the training method is called to acquire the populated hashmap. The rank method is then called, which ranks the documents in descending order of relevance, writes this information to a text file, and prints the contents of the most relevant file.

void printDoc(char **paths, int N, int id);

- The printDoc method takes the array of paths generated from glob, N, the number of files, as well as the id of the file that you want to print the contents of. It parses through the filepaths until it finds the filepath that matches with the file id. This involves parsing through each string to isolate the number, then converting that number to an integer before comparing it with the id passed in to see if we've located the filepath for the correct document. After we've located the filepath, we open the file and read the file one line at a time, printing it out each time. After we've printed the file, close the file using fclose(). This method is used to print the contents of the most relevant file to screen at the end of the main method.

int num_query(char *query);

- num_query is a helper method for determining the number of words in the search query passed in by the user through the command line argument. It takes the query (e.g. "computer architecture is fun") as one string and parses through the string character by character in order to determine the number of words in the search query, which it then returns.

char** read_query(char *query, int n);

- The read_query method takes the query itself as one string (query) and the number of words in the search query (n). It then parses through the string character by character and stores each search word detected in a dynamically allocated array, which it then returns.

int* rank(struct hashmap* myMap, char** query, char** paths, int Q, int N);

- Takes the hashmap itself (myMap), the array of search terms (query), the array of filepaths (paths), the number of terms in the query (Q), as well as the number of filepaths (N) as inputs. The rank() method is split up into three parts:
- ranks is a temporary dynamically allocated 2D array that is N by 2. Each row represents a document. For example, ranks[i][0] represents the document_id and ranks[i][1] represents the calculated ranking.
- In part 1, each document id is derived from the corresponding paths[i] string and converted to an integer from string by calling the helper method stringToInt(char* str, int j). This integer is then converted to a double and stored in the dynamically allocated double array ranks in ranks[i][0]. The corresponding ranks[i][1], the ranking for id, is initialized to 0.
- In part 2, the method then calculates the rankings for each document and stores it in the corresponding ranks[i][1]. It does this by using the rankHelper() method to calculate idf for each word. After the completion of part 1, ranks[][] is fully updated with each document id (ranks[i][0]) and its corresponding tf-idf ranking (ranks[i][1]). This can now be sorted in descending order from left to right in part 3.
- In part 3, the method increments through the array ranks and writes each document id and its tf-idf score line by line in descending order to the search_scores.txt file. Also, the largest score and its corresponding document id is recorded by highestRank and highestID respectively. Finally, the method calls printDoc to print the contents of this document. This information is only printed when there are more than 0 search terms in the query and when the highestRank is greater than 0. If the highest rank is 0, there are no matching documents. If the search query has no terms, there are no matching documents. The only exception is if there is only 1 document in the directory. In this case, a message is printed that there is only one document in the directory and the contents of that document is printed. This is because due to the method of calculating tf-idf rankings, if there is only one document in the directory, its score will always be 0 (log(1/1) = 0). Because of this, there is no way of classifying its relevance.

double rankHelper(struct hashmap* myMap, char* word, int N);

- rankHelper is a helper method that takes the hashmap, a word from the user input search query, as well as the number of documents/filepaths. It then parses through the hashmap and calculates the idf for that query word. This is then returned.

struct hashmap* training(int num_buckets, char** paths, int num_files);

- The training method takes the number of buckets (num_buckets), the array of filepaths (paths), as well as the number of filepaths as inputs. A hashmap is then instantiated. It parses through the list of filepaths and isolates the document id from the filepath string. For example in (D1.txt), the document id isolated is 1. This document id is then converted from a string to an int by calling the stringToInt(num, j) method. The num variable is the document id as a string. The j variable is the number of digits. For example, for “12”, j would be 2. For “3”, j would be 1. For “12345”, j would be 5. This is

so stringToInt can use j as a measure of the weight (significance) of a digit. After the document id is isolated and converted to an int, wordExtract(map, document_id, paths[i]) to update the hashmap with the words from the current document. Finally, after the words from every document had been loaded into the hashmap, stop words are removed by calling stop_word(map, idArray, num_files);

int stringToInt(char* str, int j);

- This method takes the document id as a string (str) as well as the number of digits in the document id (j). It uses j to measure the significance of each digit so the total decimal value can be calculated from the provided ascii values of the digit characters. The integer is returned.

void stop_word(struct hashmap* myMap, int* idArray, int N);

- This method takes the hashmap, the array of document id's, and the number of files as inputs. It then parses through the map and calculates the idf for each word. If the idf is equal to 0, every instance of the word is removed from the hashmap. This method calls hm_remove to remove every document instance of this word, which in turn removes the word entirely from the map.

void wordExtract(struct hashmap* myMap, int doc_id, char* filename);

- This method takes the hashmap, the document id, and a filepath as inputs. It reads every word from the document and inserts it into the hashmap. It first uses hm_get to acquire the current num_occurrences of the word in that document. It then adds 1 to that number and updates the hashmap using hash_table_insert(myMap, inputWord, document_id, (count+1)).

void printPaths(int document_id, char *paths);

- A helper method for printing the paths

void printMap(struct hashmap* map);

- A helper method for printing the entire hashmap.

Functions of hashmap.c

struct hashmap* hm_create(int num_buckets);

- This method takes the number of buckets as input. It declares and instantiates a new instance of the hashmap (myMap). Initialize all buckets to NULL to avoid infinite loop errors.

int hm_get(struct hashmap* hm, char* word, int document_id);

- This method takes the hashmap, a word, and a document_id as an input. It then parses through the map to locate the word, then locate the document in the linkedlist in

wordNode. Once the document/word is found, it returns the current number of occurrences of that word in that document. If the document/word is not found, return -1.

void hash_table_insert(struct hashmap* hm, char* word, int document_id, int num_occurrences);

- This method takes the hashmap, a word to be inserted, the document_id to be inserted, as well as the new number of occurrences of that word in that document. Parses through the hashmap and inserts the word and document if it does not currently exist. If the wordNode exists but the document under that word does not exist, simply add the document to the linkedlist and increase the document frequency. If the word/document already exists, increment num_occurrences in the document llnode.

void hm_remove(struct hashmap* hm, char* word, int document_id);

- Removes all instances of a word from a document in the hashmap. If the document frequency is 0 after this removal, remove the wordNode completely from the hashmap.

void hm_destroy(struct hashmap* hm);

- Destroys the map by freeing the map and all of its contents.

int hash_code(struct hashmap* hm, char* word);

- Takes the word and returns the hash_code. This algorithm adds the ascii value of the character in the word together then computes modulo num_buckets and returns that as the index for the bucket in the hashmap.