Stephen Morris

Search Engine Project

**Writeup:**

This is a search engine program in C that is designed to perform a basic search function akin to what a search engine like Google might perform. The purpose of this program is to take a number of text documents and use the word strings from the text documents to populate a hash map of a user specified number of buckets. The program then searches through all of the words for stop words, which are words that appear in all the documents and have an idf score of 0.000, therefore providing no information to distinguish the documents if the words were to be queried. Once all of the stop words are calculated, they are removed from the hash map to prevent them from affecting the document scores. The next step would be to get a user input of query words. These query words are words that the program uses to search through the populated hash map and determine their relevance score. After the relevance score is calculated, the program prints out the documents in order of relevance and prints a statement if the program has no relevance to the query words.

**Method Explanations:**

The program starts in the test class and within this class, in the main method. Main takes in no parameters and returns an int. In the beginning of main, an integer ‘buckets’ is initialized and the user instantly prompted to fill the variable with the number of desired buckets. This must be a two digit number so that the getchar() works properly and doesn’t cause an issue with the number of keys typed by the user. The number of files is hardcoded into the program, this number is used to create an array of char\* (strings) named ‘files’ that is filled with the names of the files being input. Both ‘buckets’ and ‘files’ are then sent to the training method in the hashmap class.

The training method takes the number of buckets and the ‘files’ array as parameters and returns a hashmap. The first thing the training method does is create a hashmap named ‘hm’ by calling the hm\_create function. hm\_create allocates and mallocs memory for a new hashmap with the specified number of buckets, which are all set to NULL intially and returns this hashmap to the training method. Once, hm\_create returns to training, the training method creates a file reader (to open the desired text files) and a char\* ‘str’ (to hold the char strings pulled from the files). Next comes a for loop that will loop a number of times equal to the length of the ‘files’ array. In each loop, a different files will be opened (checking to make sure they exist) and a while loop will read in every line of the text files, breaking each line into the individual strings it contains, checking whether the word exists in the hashmap using the hm\_get method (description in code comments); if the word doesn’t exist then it is added to the hashmap with its document\_id in a specified bucket (using hm\_put method) and if it does the num\_occurances of the node the word belongs to gets incremented. After the while loop finishes and takes in all the words, the files that was just read is closed and the for loop moves on to its next iteration. Once all the files have been read in, the training method returns the filled hashmap to main. The main function will now get more user input, asking if the use wants to search through the hashmap or exit, specified by the S and X keys (only these keys will be accepted and the program will loop until it receives them), if the user chooses exit then the word “Exit” is printed and the program is immediately exited; if the choice is S then the program continues.

The next step is sending ‘hm’ to the stop method in hashmap, which takes the hashmap as a parameter and removes specified stop words. A stop word is a word that is contained in all documents and have an idf score of 0 (we will explain how idf is calculated in this methed). At the top of stop, several variables are created, stopWords is a char\* array that will hold all of the collected stop words, stopCount is an int that tracks the number of stopWords that have been flagged, n is an int that is equal to the number of documents that are being search (this has been decided by now), df is an int that holds the number of documents that contain the specified query word, and idfW is a double used to calculate the other double idf. The next for loop containing a while loop will go through every element of hm to check if that element is a stop word. While checking each individual element, we set its df to 0 and send the word and document id to the hm\_get function to check if the word and id pair exist and if each does, we increment df by 1. After this we check if the word has been marked as a stop word already (the checking loop will only start after the first stop word has been identified); if the word has already been marked as a stop word, we use the int bool to continue to the next while iteration so the word isn’t added again. After this, all of the information necessary has been collected to calculate the idfW and idf. IdfW is (n/df) or the number of documents that have been checked divided by the number of documents that contain the word. From there, we calculate the idf by doing log10(idf). If the idf = 0 then we know the word is a stop word and add it to the stopWords array. After the whole hash map has been checked and all the stop words put into the stopWords array, we loop through this entire array and remove all of the stopWords nodes using the word and all the document ids and putting them in the hm\_remove functions (which is explained in the program comments). After this, the hash map is returned to the main method, devoid of all stop words.

Now the edited hash map will be sent with the ‘files’ array to the read\_query method. This method takes in the hash map and ‘files’ array and uses user input to get and search query words in the hash map, compute the document ranking score using tf-idf scoring and ranks and prints the documents. This starts by making a char\* array userQueries to hold the char strings of all the user queries and a char array to hold the string that will contain the query line before it is broken into individual char strings. Next, the user is prompted to enter a single line that contains all of the words they want to query, followed by the designated finish word ‘END’, once they have typed all of this, they will hit the enter key and the program uses and while loop and the strtok function to break the full string into its individual parts. After this, we initialize essentially the same variables from the beginning of the stop method that will be used to calculate the ranking score of the documents using tf-idf scoring; this is essentially the same thing as idf in stop but then we multiply idf by tf, which is the number of times the specified word appears in a document. The scores are put in the array of doubles docScore. We use nested for loops to calculate the scores, the first loop runs through all of the text files document\_ids and the second loop runs through all of the query words. After this the idf score is computer and the tf-idf score computer by multiplying tf and if. This determines the document score a bubble sorting loop will correctly rank the document names using the docScore array and pring the document names in the correct order. This returns to main and main’s final act is to call the destroy fuction to de-allocated the entire hashmap.