I chose option 1 which utilized a similar structure as HW6 for the hashmap. . The hashmap essentially stores nodes with values: word,,document Id, and int value, numOccurences. THe hashmap holds an array, called map, that holds the linked lists. Each map index is a bucket. The nodes are hashed based on their ascii values of the word value. Therefore, if duplicate words are entered into the hashmap, they will always be placed in the same bucket. They are not the only words in that bucket, as there is a lot of collision with words that have the same ascii value. This means that my search algorithm needed to implement a way to first, locate the correct bucket, and then find the correct word and document id. Therefore, my program first had to hash the node to get the correct index for the map, and then iterate through the linked list, while comparing the iterator node to the target word and document id. I had to utilize this strategy in the hm_remove, and hm_get functions. This hashmap implementation is contained within the file, hasmap.c. There is another file, called search.c, that contains the main method and the search functions. The search.c file holds the methods: training, read_query, stop_word,find idf, get weight and get rank. The main method contains a prompt to the user for a number of buckets for the hashmap. The hashmap is then created with that amount of buckets. The main method then enters a while loop that will end when the user ends X or #. THe while loop calls the methods, training, stop_word and read_query. The training method took in a file and a hashmap as a parameter and would use hm_table_instert on each word contained in the file. This means my main method calls it 3 times for each of the three files(hardcoded 3 files). The stop_word method would then remove all of the words in the hashmap that are contained in every file. This process was completed using the findIdf method. The find IDF method took the log() of: the amount of documents that the word was divided by the total number of documents, in this case 3. I must note that I implemented code under the premise that there were only 3 documents. I also hard coded the file paths of D1.txt, D2.txt, D3.txt. After stop_word is called, we then call read_query, to receive a search query from the user. This is done by reading the search query by character, I read the characters inputted and add them to an array called word until I find a space or a new line character. At this pointli copy and send the word into getRank which uses the getweight function and the FindIdf to rank the node based on its word and documentID. This score is then added to a 3 length array that contains the total score of the 3 documents respectively. Once a word is sent over, only if the character was a space (not a new line character), it will keep reading for a new word, essentially saving its contents over the previous word. If it is a new line character, the loop ends and so the read_query function prints the results to the search_results.txt file. It is hard coded to find the correct order of the 3 documents(it must be 3 documents, called D#.txt from 1-3) and prints that order to the file. There is also a hasmap.h and search.h that hold the methods implemented in hashmap.c and search.c respectively. The make file contains the compile statements with the correct flags, files,

and a clean function that removes the search_scores file.