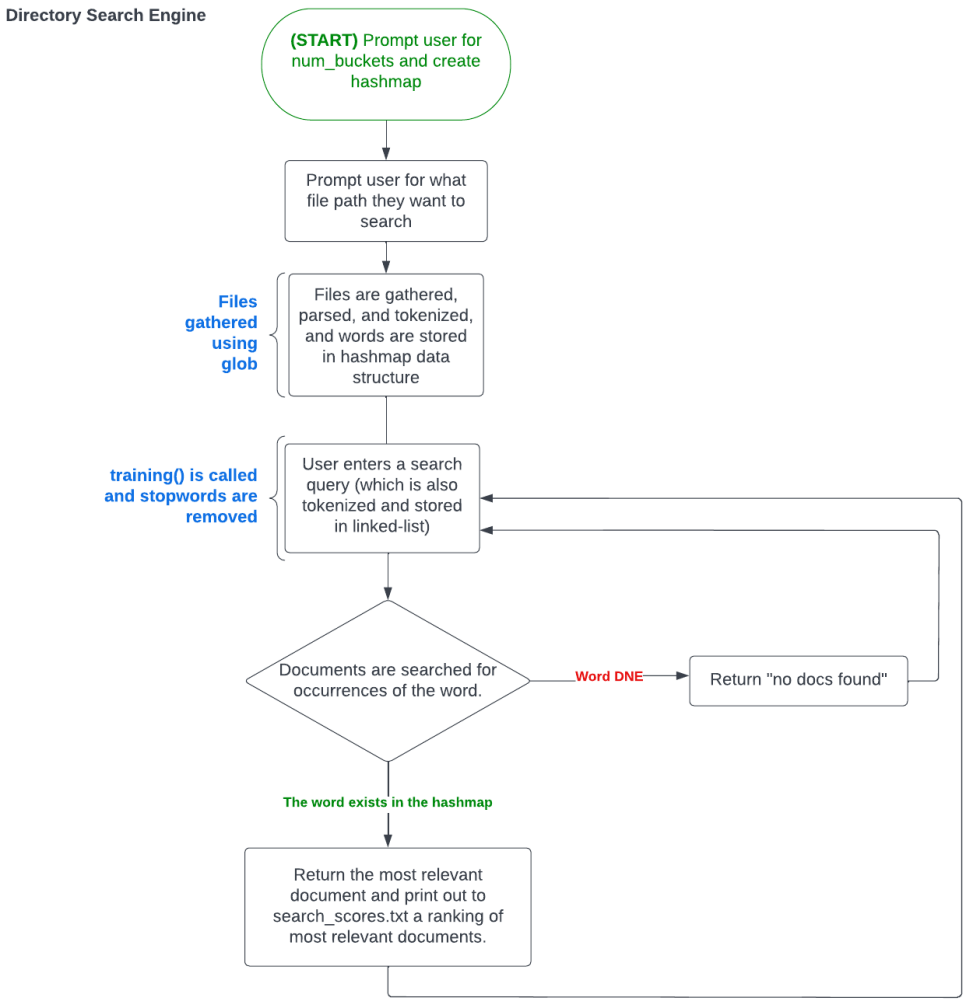
Project 5: Directory Search Engine

I. Flowchart

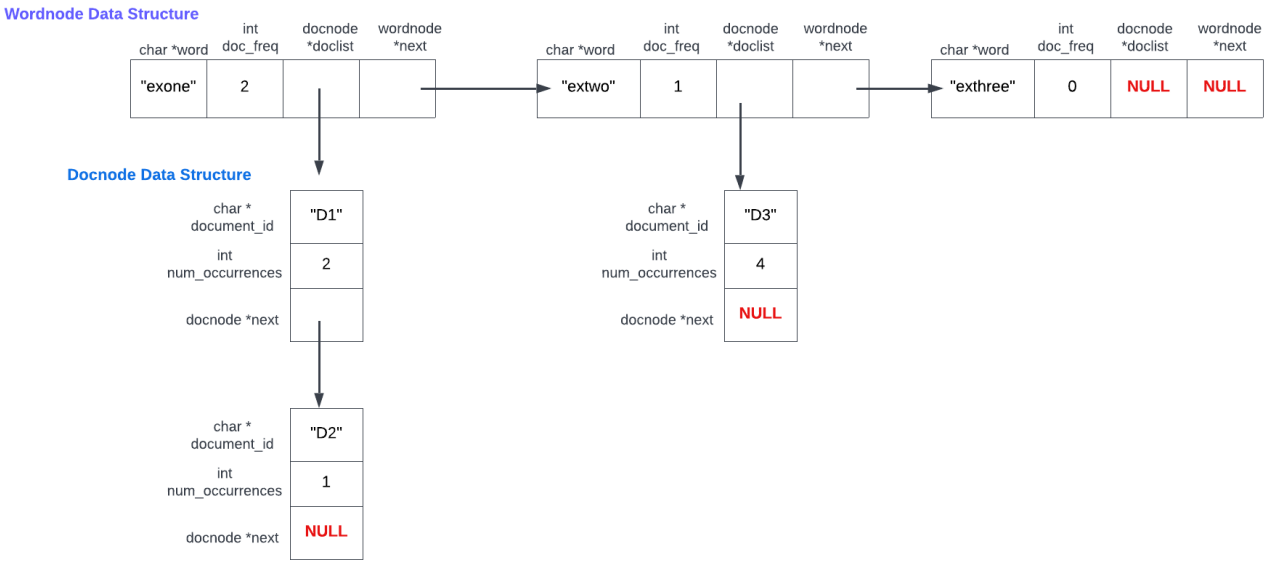


The flowchart above shows the general program flow, and various instructions/methods that are called at each point throughout the process.

II. File and Method Descriptions

A. hashmap.h/hashmap.c

The hashmap.h file contains a struct definition for both of the nodes in each linked-list structure within the word hashmap. The hashmap struct simply defines an integer variable for the number of buckets, and a double pointer to a 'wordnode'. Each wordnode contains a character array for the word, an integer doc_freq to store the number of times the word appears in all of the documents in the hashmap, a 'docnode' pointing to the secondary document linked-list listing individual occurrences of the word, and a wordnode 'next' that points to the next wordnode in the list. The overall structure of wordnodes and docnodes and their relation to each other is depicted in the diagram below:



The hashmap.h file also defines all of the functions implemented in the hashmap.c file, which include:

i) hm_create: Creates a new hashmap struct with a given number of buckets.

ii) hm_get_docs: Returns the number of documents that a specified word appears in.

iii) hm_get: Returns the term frequency of a given word/document pair, if it exists. Returns -1 for error cases, and 0 if the word/doc pair does not exist in the data structure.

iv) hash_table_insert: Insert a word into the hashmap at its correct place. Find the correct location using the word's KPV (found in hash_code).

v) hm_remove: Delete all occurrences of an entire word from the hashmap.

vi) hm_destroy: Deallocate the hashmap and all of its elements.

vii) hash_code: Use a hashing algorithm (sum of ASCII characters in each word % number of buckets in hashmap) to find the bucket/index of the hashmap that the word belongs in.

viii) hm_print: Prints out each element of the hashmap. Used for debugging purposes.

Altogether, the hashmap files implement the most important data structure of this program, which actually contains the words and frequencies of the documents that are read in.

B. querylist.h/querylist.c

The querylist.h file details the linked-list format that each user query will be stored in, in order to parse through words and determine the most relevant. Each 'querynode' has two pointers, one to a character array containing the word, and another to the next node. The querylist struct is made up of querynodes, with a pointer to the head (beginning) of the list and an integer value containing the size (number of nodes/words) in the query. The querylist.h file lists the three methods utilized in the querylist.c file:

i) ql_create: Initializes a list of query nodes and allocates the space in memory.
ii) ql_delete: Deletes a list of query nodes and deallocates space in memory.
iii) ql_insert: Inserts a node and its corresponding string into the list at the beginning (head) of the list.

C. scorelist.h/scorelist.c

The scorelist files function similarly to the querylist linked-list structure, but have different parameters and methods that align with the actions that a list of scores must be able to perform. Each scorenode has a double representing its score, a character array displaying the scored file's name, and a pointer to the next node in the list. The scorelist has a pointer to the first scorenode, and also has an integer representing its size (number of nodes). The methods outlined in the scorelist.h file are implemented in the scorelist.c file:

i) sl_create: Initializes a list of score nodes and allocates the space in memory.
ii) sl_delete: Deletes a list of score nodes and deallocates the space in memory. n
iii) sl_print: Prints out the list of scores (used for debugging purposes)
iv) sl_toFile: Writes the list of scores to search_scores.txt
v) sl_insert: Inserts a score, making sure that the scorelist is ranked highest to lowest.

D. search.h/search.c

The search.h file defines the methods that are called in search.c. Search.c is the executable file for this program, and so all of its methods and included files are outlined in the header file search.h. Search.c implements the actual functionality of the search directory, and contains all of the prompts for user input in its main method.

First, the user is prompted to enter a number of buckets, and then, a file path to search. After storing both of these inputs, and using the user file path input to run glob and return all files in that path, the training algorithm is run on a newly created hashmap. The program accepts a search query from the user, and continues to accept new queries as long as the input is not `X` (the exit code). The relevant files are ranked and displayed as output to the user. Below, each of the methods called in search.c is detailed further:

i) training: Adds each word in each file to a hashmap, and removes stopwords. Returns a file that has all words from the document entered, and all unnecessary

stopwords removed. “Cleans up” our bag of words, to help narrow down which will be most helpful in determining the most relevant document.
ii) read_query: Stores the user search query in a linked-list so that its words can be analyzed.
iii) rank: Uses the tf-idf algorithm provided by the specs to rank the list of scores.
iv) stop_word: Removes all unnecessary words from the document list in the hashmap. A word is considered “unnecessary” if it appears across all the documents, meaning either (a) it is a common article (ex: “the”, “a”, “and”) that is not helpful to our search or (b) it is common to the search query, and is not helpful in distinguishing the relevance of one document from another (ex: if “computer” appears in all of the documents, they are all equally as relevant – so it is difficult to return the most relevant).

III. Additional Implementations

A. Nested linked-list structure

1. I chose to implement **Hashmap Option 2** for this project, including an alternate data structure for an inverted index as listed in the project specifications.

B. Glob

1. I chose to implement glob in this project in order to accomplish **Option 2: Automatic reading of arbitrary number of documents** as listed in the project specifications.