CS 2461 Project 5: Information Retrieval (a 'Search Engine' ?)

In this project you will implement a document retrieval system that searches through a set of documents to determine the relevance of the documents with respect to a search phrase/query; aka a *search engine* for text documents in a directory.

The goal of this project is to further expose you to working with a large C code base to build a realistic software system. It builds on your knowledge of data structures and hash tables and linked lists in particular – and utilizes the hashmap data structures that you built in previous homework. This project has an extra credit option –first focus on completing the basic specifications before moving to the extra credit component.

You must work on your own on this project – no code or algorithm collaboration of any kind allowed, and you **CANNOT** use source code from any other source apart from the textbook or lecture notes. You can use your code from CSCI2113 and your code from Homework 6 – but make sure you have fixed any problems you had with the code (you don't to end up losing points twice for the same mistake). Note: We will be running your code through a code plagiarism detection tool to detect similarities. You can discuss the general architecture of the system and C/Unix questions. *Failure to adhere to these policies will constitute a violation of GW's Academic Integrity code and you will be charged with a violation – a grade of 0 on the project and at least one grade lower on the course.*

Important: This project requires planning and writing a substantial amount of C code as well as significant amount of time testing. It is highly recommended that you start designing your solution first on paper (using a flowchart to describe the modules in your system), then determining what data structures you need, and then building the functions to manipulate your data structures. This is not a project that you can put off till the weekend before it is due and then expect to have a working project by the deadline. Section 1 describes the problem and the ranking (relevance) algorithm. Section 2 describes a simple example. Section 3 provides the project specifications and requirements, and Section 4 describes some extra credit options. *It is very important that you follow the specifications – i.e., your project must meet specifications in order to earn any credit***.**

Problem Statement: John Nedwons is interested in the task of searching through (secret) documents in a directory and identify "documents of interest" which are documents that contain specific keywords or query words. However, the directory has a large number of files including files which have no relevance to the search parameters (i.e., the query words), and manually inspecting every file's contents will take an unacceptable amount of time. Fortunately, he has decided to outsource the job to you since he knows that you have the necessary skills and knowledge to write an efficient program to automate the search process and give him a set of relevant documents. His plan rests on the assumption that all relevant documents (stored as plain text files) are stored in one directory. And luckily for you, this problem reduces to a special case of the document retrieval problem that can be solved using techniques that you are familiar with!

1. Algorithm for search and retrieval using the tf-idf ranking function

Your task is searching through documents (i.e., files/webpages) in a directory and identify "documents of relevance" for a search phrase (i.e., search query) submitted by a user. For example, if the search phrase is "computer architecture GW", you need to find the documents (i.e., files) that not only contain (some of) the words in the query but you would like to rank the documents in order of relevance. The most relevant document would be ranked first, and the least relevant file would be ranked last. This is analogous to how you search the web using a search engine (such as Google) – the results from a search engine are sorted in order of relevance. (Google's page rank algorithm used a very different technique from what we describe here.)

1.1. Overall Architecture of the System

This search problem is an instance of a document retrieval problem (or document search) and a solution could be architected by composing two main phases (steps) – the (1) *training* phase and (2) *search* phase. The process is outlined in Figure 1.

```
// 1. Training Phase
// 1(a). Read documents, create hash index
For each document:
    For each word:
        hashmap_add(word)
// 1.(b). Remove Stop words
    For each word:
        removeStopWord(doc)

// 2. Search/Retrieval and Ranking phase
    // search query is string_of-words
    For each word in string_of_words
        ranking_score(word)
    // Find overall ranking and return sorted Doc IDs
        return document_rank(string_of_words)
```

Figure 1

The first step, the training phase, consists of reading in all the documents (which are in one directory) and creating an effective data structure (called an *inverted index* structure in search algorithms) that will be used during the search process. The training step is the pre-processing phase of your system, and the data structure you can use to help speed up the search is a hash table. Since we will be searching for words in a search query, we create a hash table that stores information of the form `<word, pointer to bucket>`. Each bucket is a linked list where a node in the linked list must contain (i) the word, (ii) document ID, and (iii) number of times that the word appears in that document. Thus, a document may appear in multiple buckets; but a word appears in a single bucket and a word can appear multiple times in a document. Towards the end of the training phase, the program has read all the documents and created the hash table index. The final step of the training

phase is removal of "*stop words*". Once all documents have been read and the hash index created, the system determines "stop words" for this set of documents and then removes them from the index. Stop words are words that occur frequently (such as articles and prepositions in English) or words that do not help us in the relevance ranking since they occur in most documents. Removing stop words can not only help improve relevance ranking but can also help speed up the search process since the size of the data structure is reduced. An algorithm for identifying and removing stop words is described later in this document.

Once the program has completed the training phase, it enters the second phase which is the search/retrieval and ranking phase. The user provides a search query consisting of a number of words/search-term, and the program must return the document IDs in order of relevance. If the query contains multiple words, we perform a hash table lookup for each word. A table lookup for a word gives us the corresponding bucket, and by searching through the bucket we can determine if the word exists in any of the documents and if so then its frequency of occurrence (i.e., the count of the number of times it appeared in that document). By performing this table lookup for each word in the search query, we can compute the *score* or *relevance* of each document for the query. The higher the score the more relevant the document, and the result lists the documents in decreasing order of relevance. This is where the method used to determine the relevance ranking of a document comes into play. In this project, we use the term frequency-inverse document frequency (*tf-idf*) score to determine the relevance of a document. This method is described next.

1.2 The *tf-idf* Algorithm: A Ranking Algorithm

The simplest way to process a search query is to interpret it as a Boolean query – either a document contains all the words in the search query or they do not. However, this usually results in too few or too many results and further does not provide a ranking that returns the documents that may be most likely to be useful to the user. We wish to assign a 'score' to how well a document matched the query, and to do this we need a way to assign a score to a document-query pair. To do this, consider some questions such as "how many times did a word occur in the document", "where the word occurs and how important the word is". The goal of relevance functions (which is the 'secret sauce' of search engines) is to determine a score that co-relates to the relevance of a document.

The *term frequency-inverse document frequency* (*tf-idf*) method is one of the most common weighting algorithms used in many information retrieval systems as well as in many Natural language processing (NLP) systems. This starts with a *bag of words* (BOW) model – the query is represented by the occurrence counts of each word in the query (which means the order is lost – for example, "john is quicker than mary" and "mary is quicker than john" both have the same representation in this model). Thus a query of size m can be viewed as a set of m search terms/words $w_1, w_2, \ldots w_m$ and the bag of words model vectorizes this query string by counting how many times each word appears in the document.

Term Frequency (*tf*): The **term frequency** $tf_{w,i}$ of a term (word) w in document i is a measure of the frequency of the word in the document. Using raw frequency, this is the number of times that word)appears in the document i. Note: There are variations of term frequency that are used in different search algorithms; for example, since raw frequency may not always relate to relevance they divide the frequency by the number of words in the document to get a normalized raw frequency. Further, some compute the log frequency weight of term w as the log of tf.

- *For this project*, you will use the raw frequency as the definition: The term frequency $tf_{w,i}$ of a word w in document i is the number of times the word w appears in document i.

(Changing your code to handle other models, such as the normalized frequency or log scaled frequency is straightforward). One of the problems with the *tf* score is that common (and stop) words can get a high score – for example, terms like "and" "of" etc. In addition, if a writer of a document wants a high score they can 'spam' the search engine by replicating words in the document.

Inverse Document Frequency (*idf)*: Many times a rare term/word is more informative than frequent terms – for example, stop words (such as "the" "for" etc.). So we consider how frequent the term/word occurs across the documents being searched (i.e., in the database of documents), and the **document frequency** df_w captures this aspect in the score.

- The document frequency df_w is the number of documents that contain the word w.

The inverse document frequency idf_w of term w is defined as:

- Inverse document frequency*:* $idf_w = log\ (N / df_w)$, where N is the total number of documents in the database (that are being searched). If $df_w=0$ then 1 is added to the denominator to handle the divide by zero case, i.e., for this case $idf_w = log\ (N/(1+df_w))$. And log is the natural log (the `log` function in C).
    - The natural logarithm is used, instead of N/df_w to dampen the effect of *idf*. (The natural log could be replaced by any base.)

Term Frequency-Inverse Document Frequency (*tf-idf*) score: The *tf-idf* score gives us a measure of how important is a word to a document among a set of documents. It uses local (term frequency) and global (inverse document frequency) to scale down the frequency of common terms and scale up the frequency/score of rare terms.

The *tf-idf(w,i)* weight of a term (word) w is the product of its term frequency and inverse document frequency.

- $tf\text{-}idf(w,i) = tf_{w,i} \times idf_w$

A search query (i.e., search phrase), submitted by a user of the 'search engine', typically consists of a number of words/terms*, i.e.,* the bag of words (BOW). Therefore, we have to determine the relevance, or rank, of the document for the entire search phrase consisting of some m number of words w_1, w_2....w_m , using the *tf-idf* scores for each word. The **relevance**, or rank, $\boldsymbol{R_i}$ of document i for this search phrase consisting of m words, is defined as the sum of the *tf-idf* scores for each of the m words.

$$R_i = \sum_{j=1}^{m} tf - idf(w_j, i)$$

Based on the above definitions, the outline of the training phase is shown in Figure 2 below.

Printing out tf-idf weighting

For grading purposes, you must print/output your *tf-idf* ranking scores to a file in the same directory as your program. The file must be named **"`search_scores.txt`,"** and it will contain the filename:score for your query on each document in descending order (delimited by new lines). **Failure to do so will result in major points lost**. For instance:

`search_scores.txt`*:*

```
<filename_with_highest_score>:<highest_score>
<filename_with_second_highest_score>:<second_highest_score>
....
<filename_with_lowest_score>:<lowest_score>
```

```
// 2. Search/Ranking Phase (Calculate Scores)
function calcRanking(string_of_words):
   score[num_docs]
   For each document:
      For each word in string_of_words:
      tf = termFreq(word, doc) // how many times
                                    word appears in doc
      df = docFreq(word) // number of docs that
                                    contain word
      idf = log(num_docs/(1+df)) // inverse doc frequency

      score[doc] += tf*idf // score incremented by
                          tf-idf weighting
   writeScoresToFile("search_scores.txt",score[num_docs]) //
  scores should be sorted highest to lowest
   return
```

Figure 2

References: Some references for more information on *tf-idf* method for document retrieval.

- H. Wu and R. Luk and K. Wong and K. Kwok. *"Interpreting TF-IDF term weights as making relevance decisions"*. ACM Transactions on Information Systems, 26 (3). 2008.
- J. Ramos, *"Using TF-IDF to determine word relevance in document queries".*

1.3 Algorithm to remove Stop words.

An important part of the information retrieval algorithms involves dealing (removing) stop words. Stop words are words which do not play a role in determining the significance or relevance of a document – these could be either insignificant words (for example, articles, prepositions etc.) or are very common in the context of the documents being processed. Stop words are language dependent, as well as context dependent, and there are a number of methods discussed in the literature to identify stop words and to create a list of stop words for the English language.

In this project, you are required to use a simple heuristic (described in what follows) that identifies stop words based on the context (i.e., the set of documents). Simply stated, the words that occur frequently across all documents could be tagged as a stop word since they will have little value in helping rank this set of documents for a user query. In terms of our metrics, term frequency and inverse document frequency, the lower the *idf* the greater the probability that the word is a stop word. Simplifying this further, we can tag a word as a stop word if it has a $idf=0$ (i.e., it appears in all documents). (Note: A better solution would be to combine words with low *idf* with a list of stop words

consisting of articles, prepositions etc. which may or may not have a low idf score for the set of documents you are processing.)

- In this project you will implement a stop word removal function that will remove words from your hash index based on an *idf* score of 0 – i.e., after the hash index is built the words with *idf=0* will be removed from that bucket thus resulting in a final hash index that has no words with *idf=0*.

Removing stop words will lead to a more efficient search/query process – think about why leads to a more efficient search process. A more realistic system would combine such a heuristic algorithm along with a statically provided stop word list (i.e., common prepositions etc.). Once again, extending your code to include this realistic solution is relatively straightforward (i.e., build a list of common words and then search this list to check if a word is a stop word).

2. An Example

Assume we have the following three documents, in Table 1, in a directory (called p5docs).

Document	Content
D1.txt	computer architecture at gw is both torture and fun
D2.txt	computer architecture refers to the hardware and software architecture of a computer
D3.txt	greco roman architecture is influenced by both greek architecture and roman architecture

Table 1: Documents

The inverted index, mapping words to documents, can be constructed as shown below in Table2:

Word	Documents the word appears in
computer	D1.txt, D2.txt
architecture	D1.txt, D2.txt, D3.txt
at	D1.txt
gw	D1.txt
torture	D1.txt
hardware	D2.txt
roman	D3.txt
and	D1.txt, D2.txt, D3.txt
…	…

Table 2: An Inverted Index

Next let the search query (consisting of three words) be: `computer architecture gw`

If we had to construct a bag of words vector for the search words, it would be as shown in Table 3

:

Word	No. of occurences in D1.txt (term freq *tf*)	No. of occurences in D1.txt (term freq *tf*)	No. of occurences in D3.txt (term freq. *tf*)
computer	1	1	0
architecture	1	2	3
gw	1	0	0

Table 3: Bag of Words Vectorization showing term frequency

2.1 Computing tf-idf scores

During the search phase, the system takes a user query and searches for relevant documents. To determine the relevance of each document (with N=3 documents), it uses the *tf-idf* scoring (ranking) technique. Based on the above data (in the bag of words table) we can derive the following scores for the example.

- The term frequency for the search term "computer" for each document is:
  - $tf_{computer,1}=1$, $tf_{computer,2}=2$, $tf_{computer,3}=0$ (since computer does not occur in document D3).
- The term frequency for the search term "architecture" for each document is:
  - $tf_{architecture,1}=1$, $tf_{architecture,2}=2$, $tf_{architecture,3}=3$
- The term frequency for the search term "gw" for each document is:
  - $tf_{gw,1}=1$, $tf_{gw,2}=0$, $tf_{gw,3}=0$
- The document frequency for "computer" for each word is:
  - $df_{computer}=2$ $df_{architecture}=3$ $df_{gw}=1$
- Thus, the inverse document frequency idf scores are:
  - $idf_{computer}= log(3/2)= 0.405$ (Note: *log* is the natural log *ln*).
  - $idf_{architecture} = log\ (3/3)=0$
  - $Idf_{gw}= log(3/1)= 1.09$

Next, compute the *tf-idf* scores for each search term and document:

- The *tf-idf* score for the term "computer" for the three documents are:
  - $tf\text{-}idf(computer,1)= 1*0.405 = 0.405$
  - $tf\text{-}idf(computer,2)= 2*0.405= 0.81$
  - $tf\text{-}idf(computer,3)= 0$
- *tf-idf* score for the term "architecture"
  - $tf\text{-}idf(architecture,1) = 1* (log\ (3/3))=0$
  - $tf\text{-}idf(architecture,2) = 2 * log\ (3/3)=0$

- $tf\text{-}idf(architecture,3) = 3 * \log(3/3) = 0$

- *tf-idf* score for the term "gw"
  - $tf\text{-}idf(GW,1) = 1 * \log(3/1) = 1 * 1.09 = 1.09$
  - $tf\text{-}idf(GW,2) = 0$
  - $tf\text{-}idf(GW,3) = 0$

Using the above *tf-idf* scores for each term we can compute the rank/relevance score (for the entire query "computer architecture GW") for each document as:

- $R_1 = 0.405 + 0 + 1.09 = 1.495$
- $R_2 = 0.81 + 0 + 0 = 0.81$
- $R_3 = 0 + 0 + 0 = 0$

Based on the above relevance ranking, the system would return D1,D2,and D3 in order of relevance. If the term frequency for all search terms is zero, then that document should not be returned since none of the terms in the search query appear in the document (in the example, D3 does contain "architecture"). We can also have a perfect match if all words in the query appear in a document (i.e., no-zero term frequency for all words in the query for a document). You have to figure out how to keep track of the score and what data structure to use for this purpose.

2.2 Removing Stop Words

In the previous subsection, the example derived the *tf-idf* scores for a search query. However, the search engine must first remove stop words before entering the search phase. Removing stop words (in this project) requires computing the idf scores for every word in the documents. In the example, the word "and" appears in all three documents and its document frequency $df = 3$ and therefore $idf_{and} = \log(3/3) = 0$ and is identified as a stop word and needs to be removed from the index.Note that the word "architecture" also appears in all three documents and thus ends up with $idf_{architecture} = 0$ and is hence removed from the index.

The stop word removal algorithm would start after all documents have been read. It first computes the document frequency for each word (across all documents), as shown in table 4 below, and then removes all words w with $idf_w = 0$, i.e., with $df_w = N$ (where N is the number of documents; in our example $N=3$).

Word	Document Frequency *df*
computer	2
architecture	3
at	1
gw	1
torture	1
hardware	1
roman	1

and	3
…	…

Table 4: Document Frequency and identifying Stop Words

Important Observation: If the search and ranking function were applied to the data without removing stop words, the output was D1.txt, D2.txt, D3.txt (as shown in our example). However, the project requires that stop words be removed before the search function. Therefore, for our example, the word "architecture" was identified as a stop word and would have been removed from the index (data) structure. Consequently, the $tf_{architecture}$, $idf_{architecture}$ and $tf\text{-}idf_{architecture}$ terms would be zero (since "architecture" is no longer in the index).

3. Implementation – Data Structures

The algorithm(s) presented have several different implementation paths. We describe two specific options for organizing the data (the inverted index); both are hashmaps but differ in how the data in each bucket is organized.

The high level data structure, to store the inverted index and process the search query, is a hash table. The hash table maps each word to a bucket; i.e., the key to the hash function is the word in the document and the data to be inserted must store the document ID in which the word appears.

For our example, suppose we have a hash function with 4 buckets (this is only an example – we are not using the actual hash function you will implement) that hashes some these words as shown in the table below (note that there are collisions – for example, both "computer" and "torture" are hashed to the same bucket):

Bucket	Words
0	computer, torture,roman
1	is, fun, and, greek, greco, gw
2	architecture, refers,hardware,
3	a, at, by, influenced, both,

Table 5: Example Hash Table

The pre-processing of the documents D1. D2 and D3, will result in each word scanned being placed into a linked list in its bucket. In other words, each bucket contains a pointer to a linked list; there are as many linked lists as there are buckets in the hash table. Note the mapping of words to buckets is strictly dependent on the hash function being used, but each bucket will contain entries with the information about the word and the document. Furthermore, we want our data structure to support the tf-idf ranking function. Therefore, if a word is repeated in a document then its count should be incremented; *i.e.,* the term frequency $tf_{w,i}$ of word w in document i (number of times the word occurs in the document should also be stored in the linked list node. For example, the word "computer" appears twice in D2.txt. The data structure corresponding to this option (i.e., "Option 1- simple

hashmap") is shown in Figure 3 below.

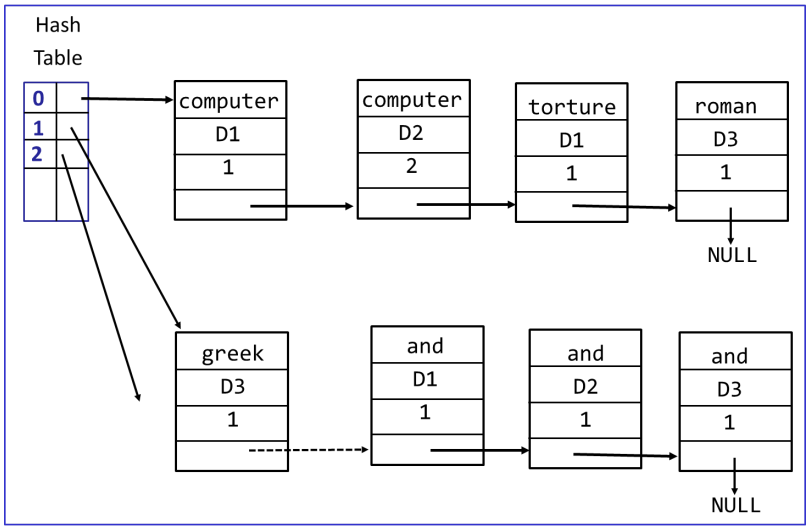


Figure 3: A simple hashmap data structure for the inverted index

In this data structure, the term frequency *tf* for each word and document is stored at the nodes in the list. However, to compute the *idf* score for each word, we need to traverse the linked list and compute the idf score. And if $idf=0$ then it should remove all occurrences (from all documents) of that word from the list. For example, the word "and" appears in all the documents (as does the word "architecture") and therefore $idf_{and} = 0$. Therefore during the stop word removal step, the function `stop_word` would remove the nodes containing "and" and would result in the new index structure shown in Figure 4.

In our example, our query set contains the words "computer" and "architecture" and "GW". The retrieval process starts off by searching for the first word in the query – "computer" and computes its score. Since "computer" gets hashed to the first bucket (bucket 0). We search through this bucket and compute the *tf-idf* score for every document for the word "computer".

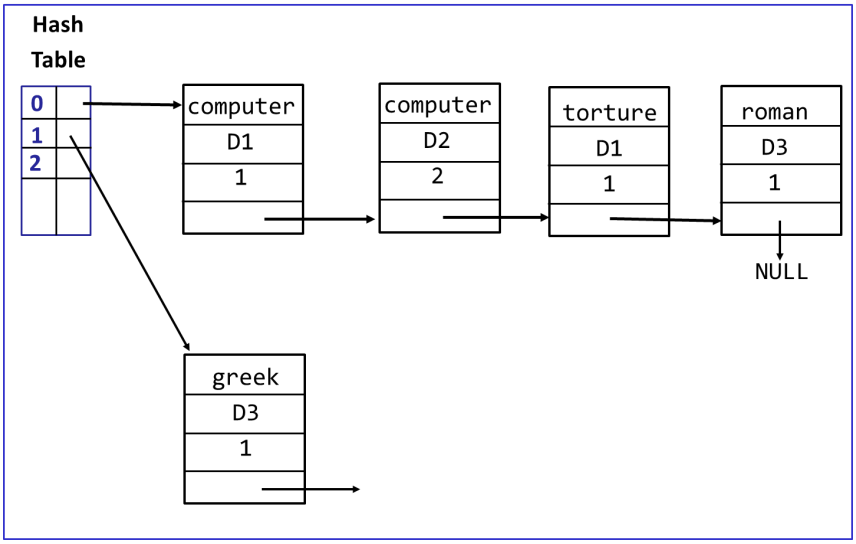
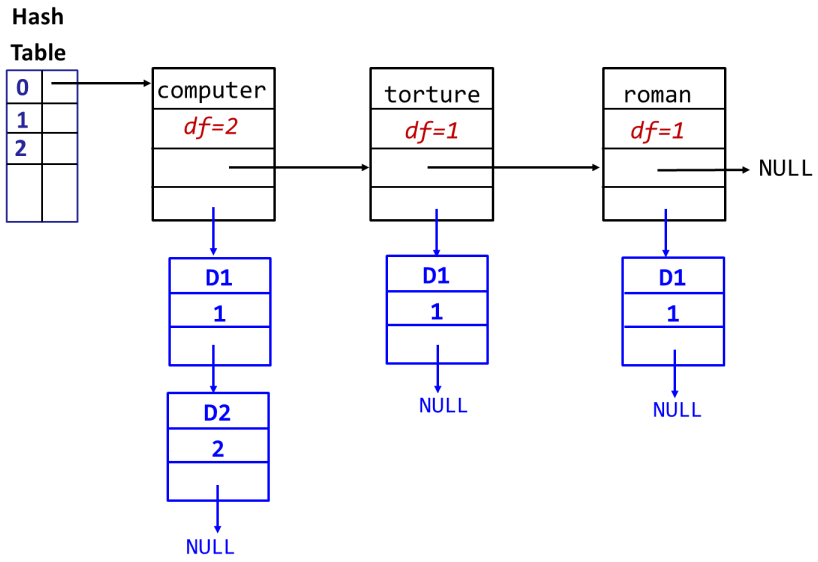


Figure 4: After deleting Stop words

A modified hashmap: The hashmap data structure can be modified to better support the *tf-idf*

algorithm by storing both the term frequencies and the document frequencies. Furthermore, the search (for a word) can be sped up if the linked list did not store duplicates of the word. In figure 3 we have to traverse two nodes of the linked list when searching for the word. This modified data structure, called Option 2 Hashmap, is shown in Figure 5 – the figure shows entries for bucket 0 and bucket 1.

In the modified hashmap, we have a linked list for each word and the document frequency *df* score for that word is stored in the root node of this list for that bucket. The linked list for each word would contain the document where the word appeared and the number of times that word appeared in that document (i.e., the term frequency tf). Searching for a word would entail traversing the upper level linked list which does not have duplicate entries for a word, and the document frequency is easily obtained by examining the node at the upper level. Removing stop words can be done by examining the document frequency *df* scores at each node (in the upper level linked list) and computing the *idf* score to determine if that entire linked list needs to be deleted. In the example, the word "and" which is mapped to bucket 1 appears in all three documents and its document frequency $df = 3$ and therefore $idf = log\ (3/3) = 0;$ therefore the entire linked list rooted at the node labeled "and" needs to be removed from the index.



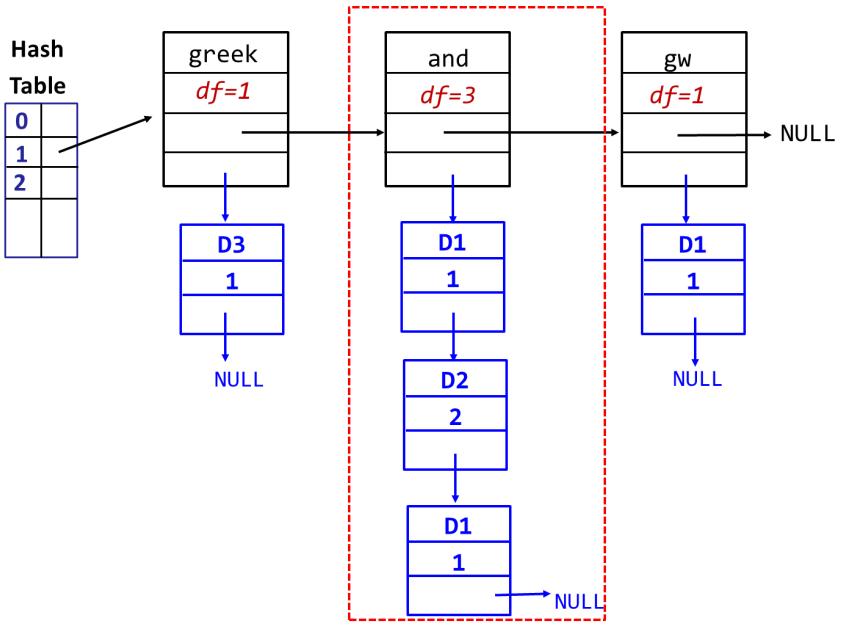


Figure 5: Hashmap Option2 – an alternate data structure for inverted index

3. Specification and Requirements

A formal description of the problem can be stated as:
Task : Given a search query consisting of a number of words, retrieve and rank documents in their relevance to the search query. Display the contents of the most relevant document to the screen and store the details of your rankings to a file.

Input : A search query string passed in as a command line argument. Set of documents labelled D1.txt, D2.txt,…D_i.txt etc. in a subdirectory titled p5docs

Output : The contents of the most relevant text file (to standard out). The file output of search_scores.txt as described.

3.1 Assumptions and Requirements

Following are some assumptions and conditions that your solution must satisfy:

- Documents to be searched:
  - Option 1 - Simpler option but lose up to 7.5%. You can design your system assuming there are three documents D1.txt, D2.txt and D3.txt in a directory p5docs (which is in the same directory as your program).
  - Option 2 - general case. You can design your solution with an arbitrary number of documents – labeled D1.txt through Dn.txt all of which are in a directory `p5docs` (which is in the same directory as your program). More information on implementing this general option is provided in the Appendix at the end of this document.
- You can assume each document contains several words, and you can assume no word is longer than 20 characters (it is possible to design a solution without these assumptions).
- The document only contains words from the English alphabet (i.e., no digits or special characters from the keyboard).

- For simplicity, you can assume all words are in lower case. But see if you can write a program that is case insensitive – if you implement this option, then please indicate this clearly in your documentation (code comments).
- The query set (i.e., the search phrase/query) can be of an arbitrary length (and you can again assume no word is longer than 20 characters). If you feel the need to simplify this and assume a maximum number of words in the query, then clearly state this assumption – you will lose 2.5 points for this assumption.
- The query set (of keywords) is entered by the user at run-time (after the pre-processing phase when all the documents have been read) and you can assume they are entered on one line. The program must prompt the user for the query keywords and then return the result of the search. After returning the results the program will return to prompt for the next query set or for special symbol # to exit the program. If you set a maximum size to the query set then include this in your prompt.
- The number of buckets in the hash table is specified at run-time by the user.
    - You can make a simplifying assumption (with a 1 point penalty) and assume that this size is specified statically at compile time in the program.
- You should not make any assumptions on the contents of the documents or the query words.

3.2 What you have to implement: Requirements and Specifications

- Your program must print the documents in order of relevance after each search.
- Your program must write the *tf-idf* scores to the file `search_scores.txt` as described earlier.
- A hashing function `hash_code(w)` that takes a string *w* as input and converts it into a number. A simple (and general) process is: Sum up the ASCII values of all the characters in the string to get a number S and then apply the hash function to get bucket *b*. For the hash function, you can choose the simple function $b = S\%\ N$ (i.e., *S modulo N*) function where *N* is the number of buckets in the hash table. You should explore if there are other, better, hash functions you can choose for this application, and if you choose a different hash function, you must then define that function in your documentation and why you chose that function. Note: Hash functions using a modulo N function typically use a prime number for N (the number of buckets). Why do you think this is the case ?
- A function `hash_table_insert(w,i)` that inserts a string w and the associated document number *i* in the hash table (into bucket `hash_code(w)`). Take care to ensure that the frequency of the string in that document is updated if the string has appeared before in the document (i.e., if it has already been inserted into the table). This function will need to call some of the functions you need to implement linked lists.
- A function `training` for the "training" process, i.e., pre-processing, that takes a set of documents as input and returns the populated hash table as output.
- A function `read_query` to read the search query.
- A function `rank` in the search/retrieval process that computes the score for each document and ranks them based on the *tf-idf* ranking algorithm. Your system should also determine if there is no document with a match – i.e., if none of the words in the search query appear in any of the documents.
- A function `stop_word` that is part of (last step of) the training process that identifies stop words and removes the stop words and adjusts the hash table and lists accordingly.
- A `main` function that first calls the training process to read all the documents and create the hash table. Note that main must first prompt user for the size of the hash table, i.e., number of

buckets. Once the training phase is over, it enters the search (retrieval) phase to search for the keywords and find the documents that contain these keywords. Main will then prompt the user for the query set (search phrase entered on one line) or prompt to exit: "Enter Search String or `X to Exit`". If the user enters "X" main will exit the program, otherwise it calls the `read_query` function to read the query set. The program then computes the score, prints out the documents in order of relevance (i.e., descending order of scores), prints out the document *tf-idf* scores to `search_scores.txt`, and returns to the main prompt (i.e., to prompt for another search or to exit).

- A `makefile`. Think carefully about how you want to construct the different modules and therefore how you set up the makefile. However you decide to construct your makefile, make sure that everything compiles correctly on shell by evoking make with no arguments. The resulting executable should be named "search."

Example program execution (user input in bold) for the example:

```
⇒ ./search
⇒ How many buckets?:
⇒ 10
⇒ Enter search string or X to exit:
⇒ computer architecture gw
⇒ D1.txt
⇒ D2.txt
⇒ D3.txt
⇒ Enter search string or X to exit:
⇒ X
```

The file `search_scores.txt` would contain:

```
D1.txt     1.495
D2.txt      0.81
```

(If the user entered another search query, then those scores would be printed to `search_scores.txt` below these scores.)

3.3 Grading and Submission Instructions:

You must submit, on GitHub, (1) the source code files, (2) the makefile, and (3) a short document (report) describing your implementation – first state whether you chose option 1 or option 2 (i.e, reading arbitrary files from a directory), you must show the flow chart, and algorithms and how the different functions interact with each other, and name your report first_initial-Last_Name.pdf.

- We will test the code on shell.seas.gwu.edu – so be sure your code works on shell (and gcc) before submission.
- If your code does not compile, you will receive a zero for the project.

- If your code crashes during normal operation (i.e., the specifications of the project), then it can result in a 50% penalty depending on the severity of the reason for the crash.
- You are required to provide the makefile. How you break up your code into different files will play a role in your grade. To run the code, we will use the make command – so make sure you test your makefile before submitting.
- You will be graded on both correctness (50%) as well as efficiency (40%) of your solution, in addition to documentation and code style (10%).
    - Efficiency refers to the efficiency of your overall system (yes, you can earn extra points)
        - 20: the time complexity of your algorithm; i.e.,the data structure you chose to use.
            - 12.5 for Option 1 (simple hashmap) or 20 for Option2 (hashmap with list of lists).
        - 10: memory management (no memory leaks!) – you can lose up to 10 points.
        - 15 reading three files or an arbitrary number of files (see Appendix)
            - 15 for Option 2 or 7.5 for Option 1.
    - You must document your code – if you provide poor documentation and no report then you lose up to 10% of the grade.
- Any assumptions you make on the specification of the input and search process (if the provided specifications do not cover your question) then you should state these clearly in the report and in the comments in your code (in function main). Failure to do so may lead to your program being graded as one that does not meet specifications. Additionally, if you make an assumption that contradicts the specifications we provided then you are not meeting the project specifications and points will be deducted.
- Read the next section for the two options on reading files from the search directory.

If you choose to use your one time late submission, you have 36 hours extra but will incur a 10% late penalty (in addition to any other points taken off during grading). However, you should be cognizant of the fact that the final Project 6 will be due on the final exam date and there are no extensions for that project.

Appendix - Processing files from a Directory and other Options

There are two options to process your documents (during the training phase) - (1) assume there are three files in a subdirectory (of your current directory with the code) called `p5docs` or (2) arbitrary number of files in the subdirectory called `p5docs` but assume all the files have `.txt` extention.

Option 1: There are three documents/files named `D1.txt, D2.txt, and D3.txt` in subdirectory `p5docs`. Your program must read in these files during the training phase, and your search query returns one of these as the relevant document. While you can choose to build option 1, you can earn a maximum of 95% on the project.

Option 2 Automatic reading of arbitrary number of documents: In option 1, we hard coded the names of the file we are interested into the program itself. This can be quite cumbersome when we have too many files that we need to search in and requires changing the code everytime the number of files changes. In a realistic scenario, we could just specify a directory (`p5docs`) and the program just reads all the text files from that directory and uses them to build the hash table. In order to do this, we make the following assumptions:

1. We are given a wild card string to search for within the directory.
2. The directory does not have sub-directories.

In Unix the function we use is `glob` which is defined in `glob.h`. So all you need to do is include it in the same way as `stdio.h` and then you can call it as

```
glob_t result;

retval = glob( search_string, flags, error_func, result );
```

For further details on this function and a simple code snippet, please use the command "`man glob`" on the command line. The results of the function call are in the variable result and the pathnames of the files found in this directory can be found in the `gl_pathv` member variable of this structure.

As a part of option 2, you are expected to use this function in your code to read in a directory and a search string from the user code and use this to read in all the specified files. For example, if there is a directory called "`p5docs`" and it contains three files "a.txt" "b.txt" and "c.txt" and suppose that the user enters the search string "`p5docs/*.txt`", you should use glob to obtain the filenames `"p5docs/a.txt" "p5docs/b.txt" "p5docs/c.txt"`. Your code should then use these file names and read in their contents. The rest of the code remains unchanged.

Recommendation: You should consider adding Option 2 after you have completed the basic project with option 1. You can get an idea of how to query directories by running some sample code before integrating it into your project.