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. The goal of this project is to further expose you to working with pointers, memory allocation and file I/O operations in C. It builds on your knowledge of data structures and hash tables and linked lists in particular – be sure to go over homework 7 and read the example on linked lists in the textbook (Chapter 19) very carefully. 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. (Note: We will be running your code through a code plagiarism detection tool to detect similarities.) You can discuss general architecture of the system and C/Unix questions. You can use the linked list and Hash table codes you implemented in Homework 7 (and linked list code from Software Engineering CS 2113)– but make sure you have fixed any problems you had with the code (you don’t want points taken off multiple times for the same errors!). *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 an extra credit option that leads to a more realistic system.

Problem Statement: Alice 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 her search parameters, and manually inspecting every file’s contents will take an unacceptable amount of time. Fortunately, she has decided to hire you for the job to you since she knows that you have the necessary skills and knowledge to write an efficient program to automate the search process and give her a set of relevant documents. Her 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 have already studied!

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 similar (but not necessarily the same method) 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 uses a very different technique from what we describe here.)

1.1 Overall Architecture of the System

A naïve solution to the problem is to read all the documents (i.e., words in the documents) and create a single (large) linked list. Then for each query keyword, you can search through the linked list. **Question (a):** If we have N documents, and document D_i consists of m_i words, then how long does this simple solution take to search for a query consisting of K words. Give your answer in big O notation. A more efficient solution can be constructed using other data structures.

This problem is an instance of a document retrieval problem and a solution could be architected bya composition of two main phases (steps) – the (1) *training* phase and (2) *search* phase. The first step, of training, consists of reading in all the documents and creating an effective data structure that will be used during the search process. The training step is nothing but a 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 can 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, all documents have been read by the program and the hash table created. 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.) ***Question (b)*****:** If we assume that the hash function maps words evenly across the buckets (i.e., each bucket gets same number of words), then what is the time complexity (big O notation) of this improved solution? Use the same parameters as Question (a) for the size of each document and size of the query set.

The second step is the search/retrieval and ranking phase. The user provides a search query consisting of a number of words/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 bucsket, 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 for Ranking

The simplest way to process a search query is to interpret it as a Boolean query – either a document contains all the words 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 and text search problems. This starts with a *bag of words* 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 $w_1, w_2, \ldots w_m$.

Note: We use the 'word' and 'term' interchangeably in what follows.

<u>Term Frequency</u>: is a measure to quantify the frequency of occurrence of a word within a particular document.

- The **term frequency** $tf_{w,i}$ of a term (word) *w* in document *i* is a measure of frequency of the word in the document.

Using raw frequency, this is the number of times that term (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 use the simple definition of number of times the word appears in the document. *You could, if you prefer, use the normalized (divide by number of words in the document) or logarithm scaled but clearly indicate in your report and code documentation which metric you are using IF you are not using the simple definition.* 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 'bias' the search engine by replicating words in the document.

<u>Inverse Document Frequency</u>: 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 occurs across the documents being searched (i.e., in the database), 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 term *w*.
- The inverse document frequency idf_w of term *w* is defined as $idf_w = log\ (N/df_w)$, where N is the total number of documents in the database (i.e., 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 $idf_w = log\ (N/(1+df_w))$. (The logarithm is used, instead of N/df_w to dampen the effect of *idf.*)

<u>tf-idf weighting</u>: 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* in document *i* 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, typically consists of a number of words/terms. 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, ***Ri*** 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.

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

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 Dealing with Stop words.

An important part of the information retrieval algorithms involves dealing (removing) with 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 will 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 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*. This will lead to a more efficient search/query process – ***Question (c)*** why does this lead to a more efficient search process ? If you want to build a more realistic system, you can combine this algorithm along with a statically provided stop word list (i.e., common prepositions etc.) and look up the stop word list during insertion into the hash table.

2. A Simple Example

Suppose you are given documents D1.txt, D2.txt and D3.txt whose contents are as follows:

```
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
```

The search query, consisting of three words/terms is:

```
computer architecture GW
```

2.1 Training (Pre-Processing) phase

D1 contains 9 words, and D2 contains 12 words, D3 contains 12 words. The word architecture is common to all three documents, and computer is common to D1 and D2, while GW appears only in D1.

Suppose a hash function with 4 buckets (this is only an example – we are not using the actual hash function you will implement) will hash some these words as follows (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,

The pre-processing of the documents D1. D2 and D3, will result in entries being placed into the hash table. 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 of the type described earlier and shown in the figure below. Note that if a word from D2 gets mapped to the same bucket, it should be "added" to the data structures (lists) for that bucket. If a word is repeated then its count should be incremented – for example, the count of "computer" in D2 is 2 since "computer" appears twice in document D2.

2.2 Dealing with stop words:

There are two options for organizing data to facilitate both the search process as well as removing stop words. The first (straightforward) approach is shown in Figure 1. In this case, each bucket has a linked list and the same word can appear multiple times in the list but only once for each document. Further, the term frequency of the word in that document is stored at the node. To compute the *idf* score for each word (after reading in all the documents), the algorithm needs 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 linked list. This option will let you use the hash map you created in Homework 7 to be used directly without major modifications.

The second approach is shown in Figure 2. In this case, we have a linked list for each word and the document frequency *df* score for that word can be stored in the node of the linked list for that bucket. Thus, after reading in all documents, removing stop words can be done by examining the *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. If you choose to implement this option then you will need to change your hash table implementation from Homework 7.

In the example, the word "and" appears in all three documents and its document frequency *df =3* and therefore *idf = log (3/3)=0* and is identified as a stop word and needs to be removed from the index.

Question 1(d): Which of the two approaches is more efficient in terms of removing stop words and why. Which option did you choose to implement.

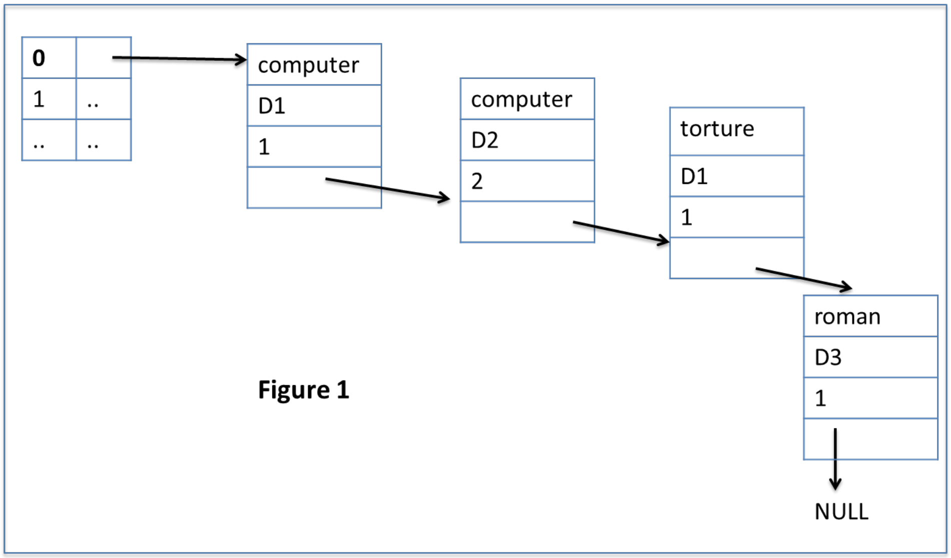

Figure 1

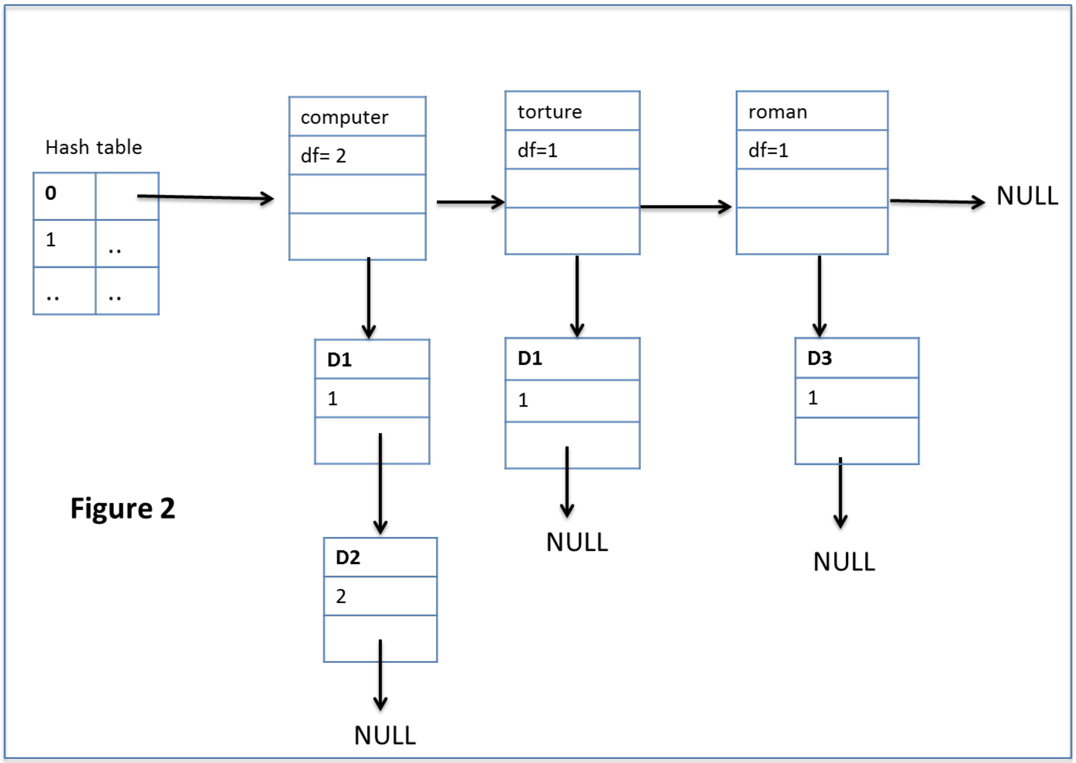

Figure 2

2.2 Search/Retrieval phase:

During the search phase, the system takes a user query and searches for relevant documents. To determine the relevance of each document, it uses the *tf-idf* scoring (ranking) technique. 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". In this example, I am using the raw frequency (i.e., count) to determine the *tf* score.

- 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). This step simply searches for the word (in the appropriate bucket) and retrieves the frequency count stored at the node in the list (if the word is found, else it is zero).

- For the document frequency: the word "computer" occurs in 2 out of 3 documents therefore $df_{computer}=2$.
- Inverse document frequency: $idf_{computer}= log(3/2)= 0.17$
- The tf-idf score for the term "computer" for the three documents are:
  - *tf-idf(computer,1)= 1*0.17 = 0.17*
  - *tf-idf(computer,2)= 2*0.17= 0.34*
  - *tf-idf(computer,3)= 0*

We can similarly compute the tf-idf terms for the other two terms in the search query – "architecture" and "GW" and this gives us:

- for the term "architecture"
  - *tf-idf(architecture,1) = 1* (log (3/3))=0*
  - *tf-idf(architecture,2) = 2 * log (3/3)=0*
  - *tf-idf(architecture,3) = 3* log (3/3)=0*
- for the term "GW"
  - *tf-idf(GW,1)= 1*log(3/1)= 1* 0.48= 0.48*
  - *tf-idf(GW,2)= 0*
  - *tf-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") of each document as:

- $R_1 = 0.17 +0 + 0.48 = 0.65$
- $R_2 = 0.34+0 = 0.34$
- $R_3 = 0 + 0 = 0$

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.

3. Specification and Requirements

This project is worth 150 points.
A formal description of the problem can be stated as:
Task : Given a search query consisting of a number of words, retrieve (i.e., list) and rank documents in that are relevant to this search query.

Input : A set of plain text documents (D_1, D_2... D_n) in a single directory that need to be stored and indexed. A search query consisting of query words w_1,.. w_q.

Output : Listing of the (names) of documents ordered by descending order of relevance/score (i.e., the most relevant document with the highest score first).

3.1 Assumptions

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

- For the sake of this assignment, assume there are at least three documents D1.txt, D2.txt and D3.txt. You can design your solution with more than 3 documents – labeled D1.txt through Dn.txt. If you choose to implement the extra credit version (reading files from a directory) then the number of documents depends on the number of files in the directory.
- 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.
  - If you make a simplifying assumption and assume that this size is specified statically at compile time in the program you will incur a penalty of 5 points.
- 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

- A hashing function that takes a string *w* as input and converts it into a number. Refer to Homework 7 for the function specifications. 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 *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 that inserts a string w and the associated document number *i* in the hash table (into bucket) – refer to homework 7 for the function specification (**hm_put**). 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. If you need to refer to code to implement linked list functions, then read Chapter 19 and you can use the code provided in the book.

- 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. Figure out the specifications for the function.
- 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 from the hash table 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., "Enter number of buckets: "
  - Once the training phase is over, it will enter the retrieval (search) phase to search for the keywords and find the documents that contain these keywords. main will prompt user and asks for "Enter S for search" or "X to Exit".
  - If user enters S, then prompts for the query set (keywords entered on one line) and then calls the `read_query` function to read the query set. The program then computes the score (call function `rank`) prints out the documents in order of relevance (i.e., descending order of scores), 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.

3.3 Grading and Submission Instructions:

You must turn in, using blackboard, a tar (or zip) file containing (1) a short document (report) describing your implementation – show the flow chart, algorithms and how the different functions interact with each other, (2) the source code files and (3) the makefile.

- 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 (60%) as well as efficiency (30%) of your solution, in addition to documentation and code style (10%).

    - Efficiency refers to the time complexity of your algorithm and also includes data structures you use and memory management (no memory leaks!).
    - You must document your code – if you provide poor documentation then you lose 10% of the grade. **However, if you provide no documentation whatsoever then you will be penalized 20%.**
- 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 extra credit options.

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).

4. Extra Credit Options

Consider adding this extra credit option after you have completed the basic project. It will help you learn a few more C/Unix utilities (you can try to get an idea of how to query directories by running some sample code before integrating into your project). **There is no partial credit on the extra credit options – you must implement the specified functionality fully, and to meet the specifications.**

Option: Automatic reading of arbitrary number of documents: (10 points) In this project, we hard coded the names of the file we are interested in, in the program itself. This can be quite cumbersome when we have too many files that we need to search in. In a realistic scenario, we could just specify a directory 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 your extra credit assignment, you are expected to use this function in your code to read in a directory and a search string from the user and use this to read in all the specified files. For example, if there is a directory called "sample" and it contains three files "a.txt" "b.txt" and "c.txt" and suppose that the user enters the search string "sample/*.txt", you should use glob to obtain the filenames "sample/a.txt" "sample/b.txt" "sample/c.txt". Your code should then use these file names and read in their contents. The rest of the code remains unchanged.