COSC2320: Data Structures and Algorithms
HW6: Hash Tables

1 Introduction

In this homework you will create a C++ program to act as a primitive search engine, first searching for keywords in a series of files, storing those keywords in a hash table (indexing) and implementing a quick way of determining what files contain certain keywords(searching).

1.1 Keywords

To find the keywords in a text document you will first need to load the file into memory, as was done in previous homeworks. From this list you should delete words that cannot be keywords such as ”a”, ”the”, ”is”, etc. This list of non-keywords, commonly called *stopwords*, will be provided for you in a special file that we will call dictionary. The remaining words will be the keywords in your file. This can be added to a common repository in memory.

1.2 Indexing

You will need to create a special data structure to hold your results. This structure should have a field for a single keyword, that will act as the index, and a list of the files where this keyword is present. For instance if the word ”blue” can be found only in text files ”rainbow.txt” and ”colors.txt”, your data structure might look like:

```
{ keyword='blue', files = {'rainbow.txt','colors.txt'}}
```

This data structure should be stored in a hash table and you should allow for more files to be added to the structure. In essence after you process the words in a file (as in 1.1) you will search your hash table for each keyword. If the word is found, then you should add the file to the list. If it is a new keyword, you should add it to the hash table.

1.3 Searching

The last part of the program should be a function to efficiently search the hash table and answer conjunctive queries such as searching for all the files where the words ”blue” AND ”red” are present. Since the results of a keyword search will be a list of filenames, stored as words, you are encouraged to use the intersection function that was developed previously.

2 Examples

INPUTS

```
rainbow.txt
----------------------------------------------
red orange yellow
green

blue indigo
violet


<EOF>

roses.txt
----------------------------------------------
Roses are red,
Violets are blue,
Sugar is sweet,
And so as you.
<EOF>

puppy.txt
----------------------------------------------
I've got a yellow puppy,
And I've got a speckled hen,
I've got a lot of little
Spotted piggies in a pen.

<EOF>

spectrum.txt
----------------------------------------------
The color table should not be interpreted as a definitive list,
the pure spectral   colors form a continuous spectrum, and
how it is divided into  distinct colors linguistically is a matter
of culture and historical contingency. A common list identifies
six main bands: red, orange, yellow, green, blue, and purple.
Newton's conception included a seventh color, indigo, between
blue and purple. It is possible that what Newton referred to as
blue is nearer to what today we call cyan, and that indigo was
simply the dark blue of the indigo dye that was being imported
at the time.   <EOF>
```

```
filelist.txt
----------------------------------------------
rainbow.txt roses.txt
puppy.txt
spectrum.txt
<EOF>

remove.txt
----------------------------------------------
a    the   is   was   will   would   be
not   if   then   else   to   it   they
you   and   or   in
<EOF>

input.script
----------------------------------------------
load(corpus,‘filelist.txt’)
filter(corpus,‘remove.txt’)
index(corpus)
search(R1,‘blue’,corpus)
write(R1,‘out1.txt’)
search(R2,‘blue,red’,corpus)
write(R2,‘out2.txt’)
search(R3,‘amber’,corpus)
write(R3,‘out3.txt’)
<EOF>
```

RESULTS

```
out1.txt
----------------------------------------------
rainbow.txt
roses.txt
spectrum.txt

out2.txt
----------------------------------------------
rainbow.txt
roses.txt
spectrum.txt

out3.txt
----------------------------------------------
<EOF>
```

3 Program and output specification

The main program should be called ”keywordsearch”. Syntax:

```
keywordsearch script=input.script
```

where `script.script`) is the file where the instructions will be stored.

In the input script you can have calls to read a set of files (`load(corpus,filelist.txt)`) into a list or array of lists (`corpus`), calls to filter the list `load(corpus,remove.txt)`) where the second argument points to the file where the dictionary of redundant words can be found) and calls to index the list: `index(corpus)`. Lastly the call `search` should find all the files where the keywords in the parenthesis are found and store the results in a list. The search call has three arguments: The name of the list where the results should be stored, the keywords that should be searched and the name of the corpus to be searched.

4 Requirements

- **As discussed in class, when you output the result of the search, you can either put a count of 1 for each file (preferred method) or no count at all (allowed). Please make sure to let us know what option you took in the README file.**
- **Even if we ask for more than one corpus to be loaded, we will not require any operations between search results in different corpuses**
- You should write a function to find keywords in a text using a hash table. It is recommended that you use some variation of folding (adding the ASCII values of each letter of the keyword) as your hash function. Remember that some words could be anagrams of each other (‘looped’ and ‘poodle’ for example) and could have the same hash.
- You are free to create your hash table in any way you see fit, using any of the algorithms and hash functions described in class or in the text book. That said, we recommend that each member of the table be a pointer to a linked list, containing the data structures defined above. In essence it would be a linked list of linked lists. Remember that the efficiency of a hash table decreases if all entries are clustered around a few indexes, so it would be best if you experimented with different functions and table sizes in order to find an optimal combination.
- The list of files will be between 50 and 100 files long, so you should take this into account when allocating space for the hash table. All files will have at least 100 different words (not necessarily keywords). You can estimate the number of keywords pessimistically or optimistically. You could even determine the number of different keywords exactly before defining your hash table size.
- It is possible that you will be asked to load more than one corpus into memory. Your searches must take this into account. This would be equivalent to searching different collection of files for different keywords.
- The dictionary file will be provided. You can find a sample in the Appendix.
- Your hash table should handle collisions efficiently, using separate chaining or any other strategy you deem appropriate.
- Your search function should allow you to search from 1 to 5 keywords at the same time. We will not ask for more than that.

- The numbers should be eliminated from the input files. For example ‘12349’ cannot be a keyword, but ‘SHA1’ is a keyword. Your program should also be able to recognize that ‘wesTeros’ is the same word as ‘westeros’ and ‘Westeros’. Lastly punctuation at the end of a word or at the beginning should be eliminated, but not if it is in the middle of the word. For instance ‘(Rand)’ is the same as ‘rand’ but ‘low-budget’ is one single word, not two.
- You should implement an efficient function to search your hash table. Remember that in theory, the hash table can be searched in time $O(1)$.
- When writing your results to the output file you should write the list of files, one per line in alphabetical order.
- As in previous homeworks, words in the input files (dictionary and corpus) will be separated by a space or a carriage return. No other separators will be used.
- The program should not halt when encountering errors in the script. It should just send a message to the log file and continue with the next line. The only error that is unrecoverable is a missing script file, a missing dictionary or a missing corpus.
- Do not use the STL library.
- Your program should write error messages to a log file (and optionally to the screen). Your program should not crash, halt unexpectedly or produce unhandled exceptions. Consider empty input, zeroes and inconsistent information. Each exception will be -10.
- Test cases: Your program will be tested with 10 test scripts, going from easy to difficult. You can assume 80% of test cases will be clean, valid input files. If your program fails an easy test script 10-20 points will be deducted. A medium difficulty test case is 10 points off. Difficult cases with specific input issues or complex algorithmic aspects are worth 5 points.
- A program not submitted by the deadline is zero points. A non-working program is worth 10 points. A program that does some computations correctly, but fails several test cases (especially the easy ones) is worth 50 points. Only programs that work correctly with most input files that produce correct results will get a score of 80 or higher. In general, correctness is more important than speed.

5 Appendix: Sample list of connecting words

From IBM Research:

a	an	and	are	as
at	be	but	by	for
if	in	into	is	it
no	not	of	on	or
s	such	t	that	the
their	then	there	these	they
this	to	was	will	with