Assignment 1
Text Analytics

Aims

This assignment aims to

- give you practice with processing text data
- give you practice implementing binary search trees

Admin

Marks	contributes 15% towards your final mark
Submit	`give cs2521 ass1 Dict.c tw.c` or via Webcms3
Deadline	submit by 11:59pm on Sunday 5th July
Late Penalty	0.1 marks off the max mark for each hour late

Background

The field of Data Analytics is currently ***hot***. Text Analytics is an important subfield of Data Analytics. Data extracted from text documents is used in applications such as web retrieval, sentiment analysis, authorship determination, etc.

In this assignment, we aim to write a program which can extract one important text analytics "measure": frequency of occurrence of the most common individual words in the text.

The text documents we will use are drawn from the collection in Project Gutenberg, a long-running project aimed at digitizing out-of-copyright books in simple text format and making them available for free, for anyone to use. The books tend to be *classics* (such as "Moby Dick"), but are important works which Project Gutenberg aims to preserve in a simple, resilient format (ASCII text).

Project Gutenberg books contain the full text of the book, but this is surrounded by meta-data and legal requirements, and it is a condtion of use that these be left intact. Fortunately, the actual text can be easily delineated from the other text by the following markers

```
... meta-data, such as when uploaded, who provided the text, ...
*** START OF THIS PROJECT GUTENBERG EBOOK title of book ***
... actual text of book ...
*** END OF THIS PROJECT GUTENBERG EBOOK title of book ***
... tons of text giving licensing/legal details ...
```

Text analysis is not as simple as carving a document into word tokens, and then using those tokens. Some additional processing is needed on the tokens before they are used in determining analytics. Three forms of processing are typically applied:

tokenising/normalising

English text consists of words and punctuation. We are interested primarily in the words, so we need to extract individual words from a document. We define a *word* as any sequence of characters that includes only alphabetics (upper and lower case), numbers, single-quote and hyphen. Once we have extracted a token, we "normalise" it by reducing to all lower-case.

This simple approach to word extraction occasionally leads to strange "words" like "`'''`" or "`--`" or "`-'-`". Since these kind of words occur infrequently, we allow them, and don't try any further restrictions such as requiring at least one alphabetic char. However, we do ignore any "words" containing just a signle character.

stopword removal

Some words are very common and make little contribution to distinguishing documents, or contributing to defining the semantics of a given document, e.g. "`an`", "`the`", "`you`", "`your`", "`since`", etc. Such words are called "stop words" and are typically skipped (ignored) in text analysis. We have supplied a stopword list for use in this task.

stemming

Words occur in different forms, e.g. "love", "loves", "lovely", "dog", "dogs", "doggy". We do not wish to distinguish such variations, and so text analysis typically reduces words to their stem, e.g. "dogs" reduces to "dog", and the forms of "love" might all reduce to "lov".

We have supplied a stemming module for use in this task. The supplied stemmer is an implementation of the classic Porter stemming algorithm. It sometimes produces "unexpected" results, e.g reducing "`prince`" and "`princes`" to "`princ`". This is ok; take what the stemmer produces as The Answer.

The following example shows how a small piece of text would be reduced to a sequence of word stems for use in later analytic tasks:

```
There are many lights in the city of Sydney.
These lights shine all night long, but they
aren't the best lights.
```

which is converted to

```
light city sydney light shine night long best light
```

The Porter stemmer for this assignment actually produces

```
light citi sydnei light shine night long best light
```

Whether the word is reduced to "`city`" or "`citi`" doesn't matter for most text analytic purposes.

Setting Up

To keep your files manageable, it's worth doing each assignment in a separate directory (folder). I'd suggest creating a subdirectory in your home directory called "cs2521", and then creating a subdirectory under that called "assignments", and then subdirectories "ass1", "ass2". Let's assume that the directory you set up for this lab is *Ass1Dir*.

Change into your *Ass1Dir* directory and run the following command:

```
$ unzip /web/cs2521/20T2/ass/ass1/code.zip
```

If you're working at home, download `code.zip` by right-clicking on the above link and then run **`unzip`** on your local machine.

After running the `unzip` command, you should have the following files in your *Ass1Dir* directory:

`Makefile`	manages compilation
`stopwords`	a list of english stopwords
`Dict.h`	interface to `Dict` ADT
`Dict.c`	partial implementation of `Dict` ADT
`WFreq.h`	type definition for (word,freq) pairs
`stemmer.h`	interface for the stemming function
`stemmer.c`	code for a stemmer, using the Porter algorithm
`tw.c`	main program to compute word frequencies

For working at home, you'll also need to download the data files to your local machine.

The data files are available in the file

```
/web/cs2521/20T2/ass/ass1/data.zip
```

It is probably best not to put this data under your home directory on the CSE servers, since it's quite large and will consume a large amount of disk space.

The data files are accessible under the directory

```
/web/cs2521/20T2/ass/ass1/data
```

You can get almost the same effect as loading the files into your home directory on the CSE servers by going into the *Ass1Dir* directory and running the command

```
$ ln  -s  /web/cs2521/20T2/ass/ass1/data/  ./data
```

As supplied, the code will compile but does nothing, except check commandline arguments. You can check this by running commands like:

```
$ ./tw  data/0011.txt
$ ./tw  20  data/0011.txt
```

The above assumes that the data files are accessible in a directory or symlink called `data` in the current directory.

Exercise

You are to complete a program `tw` (short for **T**op **W**ords) that

- takes one or two command line arguments (`./tw Nwords File`)
    - the first optional argument gives the number of words to be output
    - the second argument gives the name of a text file

  If the `Nwords` argument is not given, the default value of 10 is used
- reads text from the file, and computes word (stem) frequencies
- prints a list of the top *N* most frequent words, from most frequent to least frequent

The behaviour of this program can be described by the following pseudo-code:

```
process command-line args to obtain Nwords and File
build dictionary of stopwords
open File for reading
skip to line containing "*** START OF"
go to next line
while not EOF and not reached line containing "*** END OF" do
   for each Word on the current line do
      ensure Word is all lower-case
      if not a stop word then
         apply stemmer to Word
         if not in dictionary then
            add Word to dictionary
         end if
         increment counter for Word
      end if
   end for
end while
print most frequent top Nwords, most frequent to least frequent
```

The above describes the behaviour of the main program. The references to the dictionary need to be handled by calls to the appropriate functions in `Dict.c`.

Your Task:

The code files `tw.c` and `Dict.c` are incomplete. Your task is to complete them so that the final `tw` shows the top *N* most frequent words (more accurately, word stems), and computes the results efficiently.

You need to modify `tw.c` so that it implements the above algorithm. You need to modify `Dict.c` so that it implements an efficient dictionary data type based on a *binary search tree*.

Note that you can implement any style of binary tree that you want, and can add any extra private functions you like to `Dict.c` to support your tree implementation (e.g. rotations). Similarly, you are free to change the internal data structures in `Dict.c` to aid in representing the tree style you use.

None of the data files should take more than **1** second to be processed (and should be way under 1 second if you use a sensible data structure). Most data files should be processed in less than 1 or 2 seconds. If your program is taking longer than this, you need to rethink your dictionary data structures. We will be applying a **1**-second time limit on each execution of `./tw` when auto-testing.

Extra Notes:

Your C code does not have to follow the above logic exactly; it is sufficient to produce the same list of words as the sample output with the same frequency counts.

Apply the transformations to the words in the following order: convert to lower-case, check for stopword, stem the word.

Note that above allows for a missing "`*** END OF`". If you reach the end of the file before seeing one, then treat the end of file as end-of-book, and produce the top N most frequet words as usual. If the line containing "`*** START OF`" is missing, this is treated as an error (see below).

Don't argue with our choice of stopwords; just use the list as given. You can discuss improvements in the forum, if you want, but do not change the `stopwords` file. We are using this file, as supplied, in auto-marking your assignments.

Similarly, don't question the stemmer. This is a classic algorithm, used in many text analysis systems, although there are more recent and possibly better stemmers. Read the code, if you want, although it is often impenetrable. Feel free to comment in the forum, and write a better one, but *this* is the stemmer we're using in auto-marking your assignments.

Do *not* change the files: `stopwords`, `Dict.h`, `stemmer.c`, `stemmer.h`, `WFreq.h`.

You may change the `Makefile` to suit your local conditions and your debugging needs, but when we test it we will use the supplied version of the `Makefile`.

Error Handling

You must handle all of the following errors, and produce the error message exactly as given here:

Incorrect number of command line arguments

```
fprintf(stderr,"Usage: %s [Nwords] File\n", argv[0]);
```

No stopwords file

```
fprintf(stderr, "Can't open stopwords\n");
```

File name on command-line is non-existent/unreadable

```
fprintf(stderr, "Can't open %s\n",fileName);
```

Can't find the "`*** START OF`" line

```
fprintf(stderr, "Not a Project Gutenberg book\n");
```

The first error condition is given in the skeleton `tw.c`.

Testing

To assist with testing, we provide an extra collection of files, which you can extract via:

```
$ unzip /web/cs2521/20T2/ass/ass1/testing.zip
```

Unzip this into your *Ass1Dir* directory. You will need to overwrite the existing `Makefile`. This adds the following files:

`Makefile`
: augmented `Makefile`, which also compiles `./stem`

`stem.c`
: reads words, one per line, from stdin and writes stemmed version

`stop.sed`
: script to remove stop words from a stream of words

`check.sh`
: a shell script to check whether `./tw` is working

The `check.sh` does the following:

```
for each data file F.txt do
   generate sample output for F.txt using a giant Unix pipeline
      // only takes the first 100 most frequent words
   run the ./tw program on F.txt
      // only takes the first 100 most frequent words
   compare the output of the pipeline with that of ./tw
end for
```

The `check.sh` script requires that you have the `data` directory. It also requires a compiled version of `stem.c`, so you'll need to run the command:

```
$ make stem
```

Your are not required to understand what all of the commands in `check.sh` are doing, but they might just pique your curiosity to go and have a look at the Unix Programmers Manual (the `man` comand).

At this point, you might ask "If we can do this with a pipeline of Unix tools, why are we bothering to write our own program?". One reason: it'll be *much* faster. Another reason: to give you practice with implementing trees.

Assessment

We will test your program using a script like `check.sh`, but which applies a 1-second time limit to the execution of `./tw`. We will also re-run the tests with a longer time limit (3 secs) if your program fails the original time limit. A correct, but too-slow, program is worth 2/3 of the marks.

Marks will be awarded largely on the performance of these tests; if the program produces the correct results in the required time, each of the components below, except for style, receives full marks. If the program doesn't pass the tests, then we examine the code in more detail and award marks based on how close you were to completing each of the following

`main()` function in `tw.c` (4 marks)
: How much of the pseudo-code from above was implemented

`newDict()` function (1 mark)
: Does this function produce an initially empty dictionary.

`DictInsert()` function (3 marks)
: Does the function insert new values correctly into the dictionary. The mark for this includes all of the auxiliary functions that this function relies on.

`DictFind()` function (2 marks)
: Does the function insert find values correctly in the dictionary. The mark for this includes all of the auxiliary functions that this function relies on.

`findTopN()` function (3 marks)
: Does the function correctly produce the top N most frequent words from the dictionary.

Programming style (2 marks)
: Consistent indentation, good naming, sensible partitioning of work between main functions and their helpers, etc.

Submission

You need to submit the files `Dict.c` and `tw.c`. You can submit these via the command line using `give` or you can submit them from within WebCMS.

Have fun, *jas*