CTEC2901 Data Structures and Algorithms Coursework Part 2

Tanuja Patel & David Smallwood

Spring 2014

Contents

1 Preamble

This is an **individual assignment**. The mark contributes one half of the coursework component for the module. (i.e. 25% of the module mark).

Date Set:	24 February 2014
Due In:	Monday 7th April 2014 (23:59:59 according to Blackboard)

Late submission of coursework policy: Late submissions will be processed in accordance with current University regulations which state:

> "The time period during which a student may submit a piece of work late without authorisation and have the work capped at 40% if passed is **14 calendar days**. Work submitted unauthorised more than 14 calendar days after the original submission date will receive a mark of 0%. These regulations apply to a

student's first attempt at coursework. Work submitted late without authorisation which constitutes reassessment of a previously failed piece of coursework will always receive a mark of 0%."

Academic Offences and Bad Academic Practice:

These include plagiarism, cheating, collusion, copying work and reuse of your own work, poor referencing or the passing off of somebody else's ideas as your own. If you are in any doubt about what constitutes an academic offence or bad academic practice you must check with your tutor. Further information is available at:

http://www.dmu.ac.uk/dmu-students/the-student-gateway/academic-support-office/academic-offences.aspx

http://www.dmu.ac.uk/dmu-students/the-student-gateway/academic-support-office/bad-academic-practice.aspx

The **learning outcome** assessed by this assignment is:

[1] Explain and implement a variety of classical data structures

[2] Apply classic sequential algorithms for searching and sorting

2 Coursework Specification

Consider the following text file:

```
computer Electronic device for processing data according to instructions
playground Area of outdoor play or recreation
plateau Land area with high, level surface
aardvark Burrowing, nocturnal mammal native to Africa
zebra African horse with black and white stripes
tractor Motor driven vehicle for pulling machinery
```

It represents a dictionary. The first word on each line is followed by its description. This simple text file has a straightforward syntax for each dictionary entry which can be described using the format string `"%s %[^\n]"` in which the word is matched by `%s` and the description by `%[^\n]`.

You should develop a C library, `dictionary.c`, that implements a *dictionary* specified by the following header file, `dictionary.h`:

```
#ifndef DICTIONARY_H
#define DICTIONARY_H

#define MAX_WORD_SIZE   40
#define MAX_DESC_SIZE  200

/**
 * A dictionary is a collection of mappings from WORDs to DESCRIPTIONs
 * A WORD is a sequence of characters up to MAX_WORD_SIZE in length
 * A DESCRIPTION is a sequence of characters (including spaces) up to
 *   MAX_DESC_SIZE in length
 */

/**
 * d_initialise: initialise the dictionary so that it contains no entries
 */
void d_initialise();

/**
 * d_read_from_file:
 *              Reads a textfile of words and their descriptions and
 *              adds these to the dictionary.  NB: If a word is
 *              encountered that is already in the dictionary then the
 *              new description replaces the existing one.  Reading
 *              stops when a line is read from the file that contains
 *              a single dot "." character in the first position.
 *
 * filename:    a string representing the filename for the dictionary
 *              entries.  The file is a text file with one entry per
 *              line.  An entry consists of a word followed by some
 *              whitespace followed by a description.  The data in the
 *              file are not necessarily ordered.  An example entry
 *              could be:
 *
 *              aardvark Burrowing, nocturnal mammal native to Africa
 *
 * Returns:     true (1) if the file was successfully imported into the
 *              dictionary;
 *              false (0) if the file was not successfully imported.
 */
int d_read_from_file(const char * filename);
```

```
/**
 * d_lookup:    Looks up a word in the dictionary and returns the
 *              description in the user-supplied character buffer.
 *
 * word:        the word the user is searching for in the dictionary.
 *
 * meaning:     the description is copied into this buffer which the
 *              user supplies and guarantees to be at least
 *              MAX_DESC_SIZE+1 characters.
 *
 * Returns:     true (1) if the word was found in the dictionary;
 *              false (0) if the word was not found in the dictionary.
 */
int d_lookup(const char * word, char * meaning);

#endif
```

2.1 Notes

1. Do not worry about *duplicate* meanings of words. In this dictionary each word is unique and has only one description. This is a deliberate simplification.

2. When searching for a word, ensure that the search is *case sensitive*. Thus, in the dictionary, we treat, e.g., 'apple' and 'Apple' as *different* words.[1]

3. You are not creating a `dictionary` data type (as for `stack`, `queue`, etc.), but rather your C file will *encapsulate* a single instance of your dictionary data structure.

4. How you choose to implement the dictionary data structure is up to you. Fundamentally the dictionary should store words and their definitions, and should be searchable. However, you may also wish to consider performance issues when you design your data structure.

5. When a text file of definitions is input using the function `d_read_from_file` the definitions in that text file are added to the dictionary. This means that multiple calls to this function can be used to build up a dictionary.

   Note further that when an entry is read in whose word already appears in the dictionary then the new meaning *replaces* the old meaning. (Remember that we do not store multiple meanings for words. In this implementation the most recently loaded definition is the one that is stored.)

[1]This actually simplifies the searching because words do not have to be normalised before being compared.

6. Do NOT change the header file and/or any of the function headers or defined constants. Feel free to use the supplied test program, `d_run.c`, but do NOT modify it in any way. (You may, of course, develop your own separate test program with a different name.) When marking the assignment we will use the given test program to test your implementation.

3 What you should produce

1. An implementation of the dictionary in a file called $\boxed{\texttt{dictionary.c}}$. This implementation should include the given header file `dictionary.h` and implement the three specified functions. Do NOT edit the header file in any way.

   The intention is that you demonstrate the ability to write the implementation in C. (We do not want you to download, e.g., a similar application library from the internet and simply write cover functions for that.) However, you are free to use or adapt any of the data structure libraries that have been presented as part of this module or, of course, to develop a completely bespoke solution.

2. A Unix $\boxed{\texttt{makefile}}$ that contains the instructions to build (at least) the following *targets*:

   clean:
   Remove all *derived files* from the directory – i.e. compiler output such as executables and `.o` files.

   d_run:
   Makes an executable program from the given test program `d_run.c`. The executable must be called `d_run`.

   zip:
   Creates a *zipped, tar file* containing the files: `dictionary.h`, `dictionary.c`, `d_run.c`, `README`, `makefile`, and any other source files that your implementation requires. The zipped tar file must be called `p12345678.tar.gz` where the "p-number" (`p...`) is *your actual p-number.*

   NB You can easily check if your `zip` target is working by (a) running `make zip`; (b) copying the `tar.gz` file to `/tmp`; (c) changing directory to `/tmp`; (d) decompressing the archive (`tar xvzf ...`); and finally, (e) building the application (`make all`). If the program builds and then runs successfully then you have created a portable archive of your application.

   all:
   Builds the library and `d_run` executable. Normally this will refer simply to the `d_run` target.

NB It *must* be possible to just type `make`[2] and for the application `d_run` to be generated successfully.

3. A $\boxed{\texttt{README}}$ file. This is a text file that contains a description of the components of your distribution. It should include

   Author and Purpose
   The author of the library and an outline of the application's functionality. (Only include your p-number as author if you wish to maintain anonymity for marking purposes.)

   A Manifest
   A list of all the files included in the distribution and the purpose of each one.

   Installation Instructions
   Where to place the various files and how to install the application.

   Usage Instructions
   Notes on how to run the application once it has been built.

   Known Bugs
   A list of known bugs with the software in its current state.

   Implementation Notes
   This is a very specific category for this assignement: you should include a short paragraph in which you *describe* briefly and *justify* your chosen implementation strategy for the dictionary.

4 What we are looking for

"gzipped" tarfile (5%)
Your dictionary implementation, including all components needed to build it as specified in these instructions, should be correctly "*tar'd*" and "*gzipped*", and named `p12345678.tar.gz` (where your actual p-number is used instead of 12345678). And this should be handed in via Blackboard by the due date. We will refer to this file in what follows as the "*distribution.*"

README (10%)
Your distribution should contain the `README` file with the contents as specified above.

makefile (10%)
Your distribution should contain the `makefile` with the functionality as specified above.

[2] This always defaults to `make all`.

Implementation strategy (20%)

We will award marks in this category according to the appropriateness and sophistication of the implementation strategy selected. We will not outline here which strategies will receive better marks because part of the assessment is that you should *choose* an appropriate strategy and *justify* your decision.

Note that we will not award marks simply for the *selection* of a particular implementation strategy – we must also see that a serious attempt has been made to implement that strategy.

Justification for strategy (5%)

The `README` file should contain a brief explanation and justification of the implementation strategy. Marks are awarded here for the subjective quality of that explanation and justification. (Clearly we must keep in mind the purpose of the application when assessing the appropriateness of the explanation and justification.)

Correctness (30%)

We will run your `makefile`, and then our test program `d_run` with input dictionary file(s) of our choice. This verification exercise will allow us to determine a grade for this criterion.

Maintainability (20%)

We will look at the source code in your `dictionary.c` file[3] and form a view of how it has been written. It should be well-laid out with good and appropriate use of indentation, variable names, comments, etc. It should not be *over-commented* as this detracts from the readability. We are looking for an elegant piece of code that is written in such a way that it would be highly maintainable if passed on to other software developers. This is a subjective criterion.

5 Marking Scheme

Each of the above criteria will be assessed and weighted according to the following multipliers:

1.0	'Faultless' and innovative solution
0.8	Minor areas for improvement
0.6	Some areas for improvement
0.4	Significant areas for improvement
0.0	Not present or invalid attempt

The percentage attached to each criterion is a multiple of five so the mark for each criterion will be a whole number.

[3]and any other files that we feel are necessary