CMPT 201 – Winter 2017, Assignment 2

Due: Friday, March 17th at 11:00 PM
Weight: 10% of your final mark
Work type: Individual assignment

Objective

Your assignment is to implement a hash table module and use it to

- implement a program that can both check dictionary entries and modify the dictionary master file (details given below) ,
- implement a program that discovers anagrams using your hash table implementation

Milestones

This assignment has two milestones which are to be met:

- **Milestone 1: due at midnight on the day of your lab during the week of February 27th**. Submit on Blackboard a text file containing details on your plan for working on the assignment. This includes dates, algorithms, implementation ideas, and issues you anticipate facing. Name the text file ***your_macewan_user_name*_plan.txt.**
- **Milestone 2: due at the start of your lab during the week of March 6th**. You must have the implementation of the hash table module working and you have to demo it to your lab instructor.

Specification for the hash table module:

You must make and justify any important design decisions for your implementation, based on the following important requirements:

- **`hashP.h`** is given to supply your module with a common public interface. You must have these functions written to spec, using the structs called **`entry`** and **`hashTable`** to do them. This way we can link your module with our own test program. You may add whatever other functions you need to this module, but it must run under these conditions.
- We will test your hash table implementation under the assumption that it can handle any number of entries (as long as memory is available), and be able to grow during use.
- You must implement the hash table, test it appropriately, and then report your effort in a test report (see below). Note: the tasks given for you to do in **`dictH`** will not necessarily use all the functions in the interface, but you need to make sure they all work properly (hence testing is worth something here). Test your programs extensively, including automated rules for your tests (we want to run the same tests that you have!).
- Binary Search Trees are not allowed (not that they are bad). You must use some kind of a hash table as it is a very powerful tool for searching.

Description of Program:

You must use the C programming language.

You must submit at least four source files, named **`hashP.h`**, **`hashP.c`**, **`dictH.c`** and **`anagramsH.c`** (you may also submit additional testing source files or modules if you write them):

- **`hashP.h`** Contains the header file that defines the interface for your implementation. Remember, you must not alter our portion of the interface specification (aside from the hashTable struct, which you

will modify for your hash table design, but only internally); you can only add to the header file those declarations necessary for your implementation.

- **`hashP.c`** Contains your implementation of the hash table module.
- **`dictH.c`** Contains the main dictionary program, able to either add new dictionary entries to a master dictionary file, look up words from a file and report which ones are not in the dictionary, or look up words from a file and report their meanings if they are present. What mode it runs in is handled from the command line, this program does not take standard input. For sanity and practice, consider using **`getopt`** for the command line manipulation. The program invocations are ([] indicate a filename):

```
dict –a [file: entries to add to master] –d [file: master dictionary]
dict –l [file: entries to spell check against master] –d [file: master dictionary]
dict –m [file: entries to report meanings] –d [file: master dictionary]
```

The format of the input and output is very specific, make sure you produce the right materials!

Master Dictionary File Format: a text file with a number of dictionary entries. The format used here gives first the word or phrase (all in capital letters, followed by . and a space, but the word/term may have spaces in it), followed by the definition for the word/term in regular text. An example would be: "APPLE. Tasty fruit. Origin: trees.", where APPLE is the word/term, and the remainder of the text is the definition. Entries will be additionally separated by an empty line.

File Format for –a file: a text file with a number of dictionary entries (same format as Master Dictionary File). The format used here gives first the word or term (all in capital letters, followed by . and a space, but the word/term may have spaces in it), followed by the definition for the word/term in regular text. An example would be: "APPLE. Tasty fruit. Origin: trees.", where APPLE is the word/term, and "Tasty fruit. Origin: trees.", is the definition. Entries will be additionally separated by an empty line.

File Format for –l or -m file: a text file with a number of dictionary words/terms (remember that the terms can be multiple words, phrases can be defined in our situation). The format used here gives first the word or term all in capital letters, followed by . . Entries will be additionally separated by an empty line.

Output Format for –a file: on standard output, print a message stating the success or failure of the add. The actual addition of new terms will show in the updated master dictionary file.

Output Format for –l: on standard output, print the words/terms that are not present in the dictionary, 1 per line.

Output Format for –m: on standard output, print the word/term that is present in the dictionary, followed by a space and then the definition. Do this for all the terms in the look-up file that are in the dictionary (do no output for the ones that are not in the dictionary).

- **`anagramsH.c`** Contains the main program, and other functions for finding words that are anagrams of each other. For example: some of the anagrams of caret are: cater, crate, react, and trace.
  Its invocation is:
  `anagram`

Algorithm Hint:

1. Read the input one word at a time.
2. Translate each word into lower-case as it is read.
3. Store each word in the hash table, indexed by its signature.
   A signature is the letters of the word in alphabetical order.
   - Example: cater, crate, react, and trace ->acert
   - Example: anagram is aaagmnr

4. Before storing a word, check to see if a word with the same signature is already stored in the hash.
    - If so, print out the word already in the hash along with the current word, but do not store the new word.

Example:

```
% echo caret cater crate react trace | anagram
```

The command will output:

```
caret cater
caret crate
caret react
caret trace
```

Files

Your instructor's blackboard has several files of interest:

- **`hashP.h`** the specification of the hash table interface.
- **`masterDictionary`** a text file with a number of dictionary entries. The format used here gives first the word or term (all in capital letters, followed by . and a space, but the word/term may have spaces in it), followed by the definition for the word/term in regular text. An example would be: "APPLE. Tasty fruit. Origin: trees.", where "APPLE" is the word/term, and "Tasty fruit. Origin: trees.", is the definition. Entries will be additionally separated by an empty line.

Notes:

1. Please follow **Good Programming Style** document for guidelines on style. **You must not use global variables in this assignment.**
2. Function documentation should follow the example provided in the header file hashP.h.
3. Some hash table techniques will be covered in class. You are encouraged to seek out information about other collision resolution techniques and good hash functions, but the implementation must be your own (you must explain how the techniques you use work clearly in your design document, so make sure you understand it!). Make sure to reference any sources you use for other collision methods and/or hash functions (these will be short English descriptions or formulas for hash functions, do not use any actual hash table implementation (code) as a reference).
4. You should design your data structure carefully. A good possibility is some kind of expanding table, or the use of separate chaining, but you can use other techniques.
5. A user of your hash table implementation should be able to use multiple instances of the hash table in the same program (which means your implementation must not use any global variables).

Packaging and Deliverables

Your tar file should contain a ***your_macewan_user_name*`_as2`** directory which includes all files related to this assignment. In particular, the ***your_macewan_user_name*`_as2`** directory should contain:

- a file called **`README`** explaining what the contents of each file you have included in the directory
- Makefile, **`make all`** should produce all executables as you described in your user manual
- all source files: hashP.h, hashP.c, dictH.c, anagram.c. and any other source files you might have created.
- a design document, called **`design.txt`** (or **`design.pdf`** if you prefer a pdf file), that describes the design decisions you made. The design document is supposed to record the important decisions you made about what had to be done (analysis) and how to do it (design). In particular, it should contain the reasons for why you made your decisions. Part of your design document will be the issues list, which

identifies the issues that you encountered during the assignment, whether resolved or not, and how you resolved those that were resolved.

- a collection of your regression test files, **in a directory called `Tests`**, which also contains a testing report in a text file called **`testing.txt`** (or **`testing.pdf`** if you prefer a pdf file), describing what and how you tested. Don't forget, for this assignment, we want to be able to easily run the (highest level at least) tests that you ran, so be sure to automate their use in your makefile (this can and should include at least one main program that is only for testing your module).

Only include source files and documentation files. Do not include any executable, object or core files.

You are expected to meet normal coding standards.

1. **Packaging:**
   - Makefile must work perfectly, so that all executables are created when **`make all`** run with a checked out copy of your submission. You should test this yourself. **NO EXECUTABLE FILES SHOULD BE INCLUDED - ONLY SOURCES AND TEXT FILES OR MARKS WILL BE DEDUCTED - no executables, core dumps or compiled programs should be in there.**
   - Code must compile with no avoidable warnings under **`-Wall`**. We expect to see your Makefile using this flag when compiling. Marks will be deducted if it does not.
2. **Design**
   - Were design decisions documented?
   - Was a coherent design approach attempted and how well did it work?
3. **Coding:**
   - Produces reasonable output, even with unreasonable input (i.e., exits with error, warns about problems).
   - Prints out warnings upon detection of problems (i.e., **`"Couldn't open file 'blah'!"`**).
   - Marks will be deducted for significant coding errors, such as:
     - Not checking the return code where it should be checked.
     - Not matching the specifications pre/post conditions.
     - Failure to document a non-obvious loop. This documentation must convince the reader of the correctness of the loop.
4. **Testing Strategy**:
   - Any programs (source code) used in order to test programs must be included. We expect you to submit the regression tests you are using and explain why you believe them to be a satisfactory test of your program.
5. **Style**:
   - All source files must have a header comment box giving the usage information and purpose of the program.
   - All additional functions must have function information and purpose documented (put such documentation with the prototype of the function), as well as having the code documented internally.
   - "Bad" documentation (spurious, misleading or just plain wrong) will be penalized.

Submission: Your compressed tar file will be submitted to Blackboard.

Marking Scheme

The following marking scheme will be used for this assignment:

ITEM	MARKS
Milestone 1	5
Milestone 2	6
Packaging - module has correct structure(README, makefile, source files, directory Tests, no extra files) , tarfile extracts properly, ..etc.	6
compiles and runs	3
makefile: • make all • executable generated • testing targets	6
Hash table module passes our tests	25
Correct dictH program	13
Correct anagram program	10
Module testing strategy, testing.txt, test documentation	10
Module and program documentation, design.txt, coding style	8
Program design, good program structure, no global variables, use of multiple files, error checking (insufficient memory, files, ..etc.), etc	8
TOTAL	**100**