CMSC 216 Project #3 Spring 2016

Hash Tester **Due: Thursday March 10, 2016, 8PM**

In this project, you will implement a testing program that will allow you practice reading from files. For this project you need to copy the hashtable.c and hashtable.h files from the previous project to the project3 directory. Do not copy the file Makefile (we are providing one for this project).

1 Overview

1.1 Obtain the project files

To obtain the project files copy the folder project3 available in the 216 public directory to your 216 directory. Keep in mind that the Makefile for this project is different from the one used in project2. By executing "make" on the command line, you will be able to build all the public tests.

1.2 Fixing problems with your hash table code

After the late deadline for project2, you will be able to see results for secret tests in the submit server. A TA during office hours (and only during office hours) will be able to show you any test and why the test failed (if that is the case). You are responsible for fixing your code before submitting this project. Keep in mind that if you passed all the project2 tests that does not mean you don't have bugs. In this project we will be testing your hash table again, so it is in your best interest to test your code thoroughly.

1.3 Hash table testing program

The testing program you must write, called `hashtester`, reads commands one line at a time, parses them, and then executes certain operations on a hash table. Upon starting execution, your program should initialize a single hash table, and then perform operations on that table as instructed by the commands the program reads. Make sure you name the file with your program hashtester.c. This program will include hashtable.h.

1.3.1 Method of operation

A user calls your program in one of two ways:

```
hashtester
hashtester filename
```

In other words, the program should have zero or one arguments on the command line; if there are more, the program prints out an appropriate usage message to standard error, and then exits with exit code `EX_USAGE`[1].

If no filename is specified on the command line, the program should read its data from standard input. If a file is named, however, the program reads its data from that file. In case of an error opening the file, you should print an appropriate error message to standard error, and exit with the exit code `EX_OSERR`.

1.3.2 File format

An input file (or input coming from standard input) contains multiple lines with commands, and the commands are executed in the order they are encountered. No valid line can be more than 1024 characters (including the newline character).

A valid line takes one of three forms:

1. an empty line, where the only character in the line is the newline
2. a comment, where the first non-whitespace character is a hash symbol ('`#`')
3. a command, where the line is composed of one or more strings of non-whitespace characters

[1] This and the other exit codes beginning with `EX_` mentioned here are all obtained by including `<sysexits.h>` in your C program file.

4. a line with only whitespace characters

For example, the following file contains valid lines:

```
␣␣␣␣#Comment␣line.

␣insert␣␣␣key␣␣value
delete␣key
```

(The ␣ symbol is used to show where a space exists.) The first line is an example of a comment; the second, an empty line; and the third and fourth, commands.

Valid commands must follow one of the formats specified in Section 1.3.3 below.

If your program encounters an invalid line, it should print out an appropriate error message to standard error and exit with the exit code `EX_DATAERR`. For this project, you can ignore additional information provided in an input line that starts with a valid command (no error will be generated in this case). For example, "insert mary cake bla bla" will add mary/cake to the table (the rest of the line will be ignored and no error message will be generated). Additional examples are provided below.

```
% hashtester
insert mary cake bla bla
Insertion of mary => cake succeeded.
search mary morebla
Search for mary succeeded (cake).
display key_count bla bla
Key count: 1
display table bla bla
Bucket 0: FULL (mary => cake)
Bucket 1: EMPTY
Bucket 2: EMPTY
Bucket 3: EMPTY
Bucket 4: EMPTY
delete mary bla bla
Deletion of mary succeeded.
reset bla bla bla
Table reset.
 %
```

1.3.3 Commands

A valid file can contain the following commands. After execution of any command, your program must print a report of the results of executing the command to standard output; each command has the format of those results specified below.

1. `insert key_string value_string`

   The `insert` command has two arguments. After an attempt at insertion, `hashtester` should print one of the following lines as appropriate:

   ```
   Insertion of key_string => value_string succeeded.
   Insertion of key_string => value_string failed.
   ```

   where `key_string` and `value_string` are replaced by the actual key and value supplied by the data file (this convention is used by the rest of the commands in this list).

2. `search key_string`

   The `search` command has only one argument. After the search, one of these lines is printed as appropriate:

   ```
   Search for key_string succeeded (value_string).
   Search for key_string failed.
   ```

3. `delete key_string`

   The `delete` command also has only one argument. After the attempt at deletion, one of these lines is printed as appropriate:

   ```
   Deletion of key_string succeeded.
   Deletion of key_string failed.
   ```

4. `reset`

   The `reset` command takes no arguments, and deletes all contents of the table. The following line is printed after execution:

   ```
   Table reset.
   ```

5. `display item`

   The `display` command takes one argument, which is one of the following: a. `key_count`; or b. `table`. These are the only valid arguments to `display`.

   If the argument is `key_count`, a line is printed with the appropriate key count used:

   ```
   Key count: 10
   ```

   If the argument is `table`, then the following procedure is followed: starting at the first bucket, and running through the table to the last bucket, a line is printed out detailing the state of that bucket, according to the formats shown below:

   ```
   Bucket 5: EMPTY
   Bucket 12: FULL (key_string => value_string)
   Bucket 17: DELETED
   ```

   The first bucket in this list is numbered 0.

1.3.4 Example command file

Here is an example of a command file for your hash table testing program:

```
# insert a few area codes
insert 202 Washington_DC
insert 240 W_Maryland
insert 443 E_Maryland

# delete one
delete 202

# is 299 here?
search 299
```

```
# other stuff
display key_count
display table
reset
display table
```

Assuming your hash table has five buckets, your testing program should print the following as output:

```
Insertion of 202 => Washington_DC succeeded.
Insertion of 240 => W_Maryland succeeded.
Insertion of 443 => E_Maryland succeeded.
Deletion of 202 succeeded.
Search for 299 failed.
Key count: 2
Bucket 0: EMPTY
Bucket 1: FULL (240 => W_Maryland)
Bucket 2: DELETED
Bucket 3: FULL (443 => E_Maryland)
Bucket 4: EMPTY
Table reset.
Bucket 0: EMPTY
Bucket 1: EMPTY
Bucket 2: EMPTY
Bucket 3: EMPTY
Bucket 4: EMPTY
```

1.4 Important Points and Hints

1. Data should only be allocated statically. You may not use `malloc()` etc.
2. Do not use perror to generate error messages. Instead use fprintf and stderr.
3. Make sure you use diff (rather than visual inspection) in order to check the output of your project against the expected output. If you think you are passing a test in grace, but it does not work in the submit server, probably the problem is a small typo (e.g., missing a period in the output).
4. If you pass a test in grace, but not in the submit server you may have a problem with your variable declarations. For example, you are not declaring an array that is large enough, or you are passing to a function a pointer to a character, instead of a pointer to an array of characters.

2 Grading Criteria

Your project grade will be determined with the following weights:

Results of public tests	35%
Results of secret tests	50%
Code style grading	15%

2.1 Style grading

For this project, your code is expected to conform to the following style guidelines:

- Your code must have a comment at the beginning with your name, university ID number, and UMD Directory ID (i.e., your username on Grace).

- No lines longer than 80 columns are allowed. You can check your code's line lengths using the `linecheck` program in grace. Just run "`linecheck filename.c`" and it will report any lines that are too long.
- Use reasonable and consistent indentation.
- Use descriptive and meaningful identifiers.
- Do not use global variables.
- Feel free to use helper functions for this project. Just make sure to define them as static.
- Each function must have, at a minimum, a comment describing its purpose and operation. If you use a complicated algorithm to implement a function, you definitely need an extra comment explaining the complicated steps of your algorithm.
- You should `#define` symbolic constants to use in place of numeric literals. As a general rule, a number that is not 0 or 1 should be replaced with a symbolic constant.
- Follow the C style guidelines available at:

  http://www.cs.umd.edu/class/spring2016/cmsc216/content/resources/coding-style.html

3 Submission

3.1 Procedure

You can submit your project by executing in your project directory the **submit** command.

3.2 Possible problem with submit command

If you try to submit your project in grace, and you get the error:

"Exception in thread 'main' java.lang.OutOfMemoryError: unable to create new native thread"

then close all terminals windows except one, and try to submit again.

4 Academic Integrity

Please see the syllabus for project rules and academic integrity information. All programming assignments in this course are to be written individually (unless explicitly indicated otherwise in a written project handout). Cooperation between students is a violation of the Code of Academic Integrity.