CMSC 216 Project #2 Fall 2010

Hash Table I **Due: October 8, 2010, 6PM**

In this project, you will implement a hash table with the open addressing scheme (with linear probing) that you learned about in lecture/discussion. To aid in learning to write your own tests, you will also write a testing program that will give you practice in reading from files.

1 Procedure

1.1 Obtain the project files

We have supplied three files for your use in this project: (a) `hashtable.h`, which provides prototypes for the functions your hash table must implement; (b) `.submit`, which is necessary to submit the project to the submit server; and a (c) `Makefile`, which will help you work on your project more efficiently. These files are contained in `~/216public/project2/project2.tgz`; you should copy this file to your home directory and extract the files in a similar manner to the procedures you followed in Project #1.

Your code for this project should be contained in your `~/216/project2` subdirectory, which is created by extracting files from the tarball you copied.

1.2 Implement the hash table

Create a file named `hashtable.c` to contain your implementation of the hash table. Your hash tables need to use an open addressing, linear probing approach to resolving hash collisions, as discussed in lecture. Both the keys and the values to be inserted into your hash tables are strings (`char` pointers). You *must* use the hash code algorithm defined in Section 2.2.2 to generate the hash codes for your keys, since we will be inspecting the state of your hash table, and using a different algorithm will generate incorrect results.

1.3 Write the testing program

You will also be writing a program that reads in operations from a file (or standard input), and then perform those operations on a hash table. The testing program is compiled against a hash table implementation and is used to test that implementation.

2 Specifications

2.1 Use of a `Makefile`

Makefiles will be covered more in-depth in class, but instead of issuing `gcc` commands every time you want to recompile your program, you can simply execute "`make`" from the command line. Make recompiles your `hashtester` program, as well as any public tests we give you.

2.2 Hash Table

The hash table you will implement is a fixed-size hash table using open addressing (with linear probing to handle collisions).

2.2.1 Linear probing

This is intended as a brief review of the concept of linear probing; please refer to CMSC 132 notes or other notes for details on hash tables.

Linear probing is a method to deal with the occurrence of collisions when hashing keys to buckets in a hash table. When an insertion of an element is attempted into a full bucket (located at an index i), the linear probing algorithm attempts to insert the element into the next bucket in the table (i.e., the bucket at index $i + 1$). The algorithm repeats the probing with subsequent buckets until it either finds an empty bucket that can hold the element, or finds that all buckets are full.

Because this algorithm maps elements to a bucket that is not necessarily at index $h \pmod n$ (where h is the

hash code of the key and n the total number of buckets), you need to take this possibility into account in performing search and delete operations on the hash table. Consider inserting two elements a and b that map to the same index, i. a is placed in the bucket at index i, but b goes into index $i+1$. But if a is subsequently deleted, b remains at index $i+1$; this displacement requires you to introduce a new bucket state, in addition to *empty* and *full*, called *dirty*, meaning that a bucket was once full, but no longer is. With the additional state, you can adapt the search and delete operations to find displaced elements in the table. Learning how to adapt those operations is part of your assignment for the project.

2.2.2 Hash code algorithm

As mentioned earlier, you are hashing strings in this project. You must write a function to generate a hash code for a string; this hash code (modulo the size of the table) provides the starting point for finding a position to insert or find a string. This is the algorithm you need to implement in the `hash_code()` function that is part of your `hashtable.c` implementation:

1. an empty string (`""`) has a hash code of 0.
2. non-empty strings have a hash code determined by finding the hash code of the string obtained by removing the last character, multiplying that code by 65599, then adding the ASCII value of the removed character.

For example, the hash code (HC) of `"ice"` is 451,845,518,507:

$$\begin{aligned}
HC(\text{"ice"}) &= HC(\text{"ic"}) \times 65599 + ASCII(\text{'e'}) \\
&= (HC(\text{"i"}) \times 65599 + ASCII(\text{'c'})) \times 65599 + ASCII(\text{'e'}) \\
&= [(HC(\text{""}) \times 65599 + ASCII(\text{'i'})) \times 65599 + ASCII(\text{'c'})] \times 65599 + ASCII(\text{'e'}) \\
&= [(0 \times 65599 + 105) \times 65599 + 99] \times 65599 + 101 \\
&= [105 \times 65599 + 99] \times 65599 + 101 \\
&= 6887994 \times 65599 + 101 \\
&= 451845518507
\end{aligned}$$

For long strings, this algorithm could result in a number that exceeds the maximum value of an `unsigned long`. Therefore, you should assume that all multiplications and additions are done modulo (`ULONG_MAX`[1] + 1); this can be done by just assigning the results of the multiplications and additions into an `unsigned long` and the hardware will perform the necessary conversions via truncation.

2.2.3 Hash table functions

You must implement the following functions:

1. `void init_table(Table *t)`

   Initialize the table to be empty. You can assume that the table has capacity `NUM_BUCKETS`, as defined in `hashtable.h`. Do nothing if `t` is `NULL`.

2. `void reset_table(Table *t)`

   Reset the table to an empty state, meaning that all buckets are in the empty state. Do nothing if `t` is `NULL`.

3. `int insert(Table *t, const char *key, const char *val)`

   Insert a key/value pair into the table. Insertion fails if: a. either `key` or `val` is `NULL`; b. either the key or value string is longer than the `MAX_STR_SIZE` defined in `hashtable.h`; c. `t` is `NULL`; or d. the table is full.

   If the key is already in the table, its corresponding value in the table should be overwritten with the new value. Note that in this case, even if the table is full the insert succeeds.

   Return 0 if insertion is successful, -1 otherwise.

[1] `ULONG_MAX` is a constant defined in `<limits.h>`, which is equal to the largest value an `unsigned long` variable can hold. As described above, you do not need to actually include `limits.h` in your program to perform operations modulo `ULONG_MAX`.

4. `int search(Table *t, const char *key, char *val)`

   Search for a key in the table. If the key is present and `val` is not `NULL`, the value that is paired with the key in the table should be copied into the buffer `val` points to (you may assume the buffer is large enough to hold the value). If either `t` or `key` is `NULL`, the search is defined as failing.

   Return 0 if the key is in the table, -1 if the search fails.

5. `int delete(Table *t, const char *key)`

   Attempt to remove the key from the table. If either `t` or `key` is `NULL`, or if the key does not exist in the table, the attempt fails.

   Return 0 if the deletion succeeds, -1 otherwise.

6. `int key_count(Table *t)`

   Return the number of keys present in the table, or -1 if `t` is `NULL`.

7. `int bucket_count(Table *t)`

   Return the number of buckets the table has; return -1 if `t` is `NULL`.

8. `unsigned long hash_code(const char *str)`

   Return the hash code of the string, as defined by the algorithm in Section 2.2.2. If `str` is `NULL`, return 0.

2.3 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.

2.3.1 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`[2].

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`.

2.3.2 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 (‘`#`’); or
3. a command, where the line is composed of one or more strings of non-whitespace characters.

For example, the following file contains all three forms of valid lines:

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

```
␣␣␣␣#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 2.3.3 below.

If your program encounters an invalid line of any kind, with any kind of input error, it should print out an appropriate error message to standard error and exit with the exit code `EX_DATAERR`.

2.3.3 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: DIRTY
```

The first bucket in this list is numbered 0.

2.3.4 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: DIRTY
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
```

3 Important Points and Hints

1. You must use the hash code algorithm described in Section 2.2.2. If you do not get the results described in the example above, or fail the public tests that check the hash algorithm, make sure you fix it before continuing.
2. You must use the constant `NUM_BUCKETS` whenever you need to use the number of buckets in the table in your code. Our tests will define this value and your table needs to behave correctly when the value is set.
3. Input to the hashtester program may not be properly formed; this means that you must check to ensure that input has the proper format (including meeting the restriction on line length to 1024 characters). We suggest using the `fgets()`, `strlen()`, and `sscanf()` functions to ensure that input is well-formed.
4. Some of the required functions in the hash table perform similar operations as part of fulfilling their specifications. Your style grade depends in part on your ability to avoid unnecessary code duplication, either by creating helper functions to perform common operations, or reusing the required functions in other functions.

4 Grading Criteria

Your project grade is determined as follows:

Results of public tests	20%
Results of secret tests	60%
Code style grading	20%
Extra credit	5%

The public tests will be made available shortly after the project is released. You can find out your results on those tests by checking the submit server a few minutes after submitting your project. Secret tests, and their results, will not be released until after the project's late deadline has passed.

4.1 Style grading

Style grading will take place after the late deadline; 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 (tab stops count as 8 spaces for the purposes of this rule). You can check your code's line lengths using the `linecheck` program in the `~/216public/bin` directory on Grace, which should be in your path. Just run "`linecheck filename.c`" and you will get a report listing any lines that are too long.
- Use reasonable and consistent indentation. If you like, you may use the `indent` command to help format your code, but we request you use at least the `-kr` switch to the program if you use it – the default indentation style is less than preferable.
- Use descriptive and meaningful variable names, using the naming convention described in lecture.
- No global variables are allowed.
- You are welcome to create helper functions, but each one you write must have a prototype placed at the beginning of your source code, and must be made `static` (i.e., private to that file).
- 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.
- 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.
- Use appropriate whitespace, especially horizontal whitespace between operators and operands.

4.2 Extra credit

Once again, we offer extra credit to students who make an early submission that passes all public tests. You will receive 5 points of extra credit if you make a submission by 6PM on Friday, October 1, that passes all public tests for the project.

5 Submission

5.1 Deliverables

The only files we will grade are (a) `hashtable.c`, your hash table implementation; and (b) `hashtester.c`, your hash table testing program. We will use our versions of all header files to build our tests, so do not make any changes to the header files.

5.2 Procedure

The submission procedure is the same as for the previous project (execute "`submit`" in the project directory). Refer to Project #1 for details on submission and checking public test results before submitting.

Note that you can see the results of a public test **much** more quickly by compiling and executing it on Grace than you can if you simply submit to the submit server and then wait several minutes for your submission to be tested there.

6 Other Notes

6.1 Academic Integrity

As described in the syllabus, any evidence of cheating will be referred to the Student Honor Council and may result in a grade of XF in this course. Submissions will be checked with an automated source code comparison tool to look for evidence of cheating. This tool has already proved itself to be very effective, so we advise you to remember that the course syllabus (§9) forbids cooperation between students on projects, including either copying or sharing source code. You are also further advised to refrain from copying code from external resources. Your projects are to be your own work, and only your work.

Your instructors are quite serious about this, and will not hesitate to refer cases to the Honor Council should they have sufficient evidence to do so.

6.2 Deadlines

Submission deadlines are strictly enforced by the submit server, and we will not extend them for things such as network outages. Extensions may be given on a case-by-case basis, but will likely only be granted in emergency situations. Therefore, you should start work on this project soon, as last-minute technical problems are not an excuse for missing either the on-time or late deadline.