CMSC 216 Project #3 Fall 2010

Hash Table II **Due: October 25, 2010, 6:00PM**

In Project #2, you created a hash table with a fixed size. In this project, you will use the dynamic memory allocation techniques you've learned in lecture to both add linked list "chains" to your hash table, as well as the ability to resize the table to improve table performance. In addition, you will use a modified version of the hash table testing program you wrote to write test cases for your hash table implementations.

1 Procedure

1.1 Obtain the project files

We have supplied four files for your use in this project: (a) `hashtable.h`, which provides prototypes for the functions your hash table must implement; and also provides structure definitions, typedefs and symbolic constants to be used when implementing the new hash table. Note that this file is **different** from the one from Project #2; (b) a `hashtester.o` object file that is somewhat different from the `hashtester` program you wrote for Project #2; (c) a program named `htester` that is used to compare output from tester programs with the correct output; and (d) a `.submit` file, so you can submit to the submit server.

The files are located in `~/216public/project3/project3.tgz`. You should copy the tarfile file to your home directory and extract the files in a similar manner to the procedures you followed in previous projects.

The use of the `hashtester.o` and `htester` files is described in detail in Section 2.3.

Your code for this project should be contained in your `~/216/project3` subdirectory, which was created by extracting the files from the tarfile.

1.2 Create a `Makefile`

As we did for you for Project #2, you should create a `Makefile` that we will use to build the C public tests against your hash table implementation. Section 2.1 lists the targets you are required to implement, as well as any other requirements for your `Makefile`.

1.3 Implement the hash table

Create a file named `hashtable.c` to hold your implementation of the hash table. Your hash tables will need to use the closed addressing, chaining approach described in detail in Section 2.2.1. As in Project #2, the keys and values are strings, but the structures we've provided for you do not have buffers allocated to hold the strings, so you will need to allocate and free memory for them as necessary.

1.4 Write your tests

The `htester` program accepts input files that have both commands and expected output. Your task for this project is to write a test suite of input files for the `htester` program, which should detect errors in an incorrect implementation. To work with our submit server testing setup, the input files you want to have graded **must** be named with the format `testfileXX`, where `XX` is a two digit number between 00 and 29.

2 Specifications

2.1 `Makefile`

Your `Makefile` should be set up so that all programs are built using separate compilation (i.e., source files are turned into object files, which are then linked together in a separate step). All object files should be built using the `-ansi`, `-Wall`, `-Werror` and `-pedantic-errors` flags to `gcc`.

You will need to have the following targets present in your `Makefile`:

1. `all`: make all executables

2. `clean`: delete all object files and executables
3. `public*.o`: because you are required to use separate compilation, for each public test that is a C program you will need to make an object file for it, and use that file to make the corresponding executable.
4. `hashtable.o`: the object file for `hashtable.c`.
5. `public*`: you will need a rule to build each public test executable – note that not all of our tests will be programs, as we may write some tests for the testing program as well, which would just be text files.
6. `hashtester`: the testing executable, made by linking the object file we provide with your hash table object file.

While you are welcome to add other targets to your `Makefile`, the ones listed above must be present.

Note that the rule for the `clean` target should be careful not to delete the `hashtester.o` object file that is given to you. One easy way to ensure that is to rename the file before deleting object files, then renaming it back after the deletion command.

We will use the `Makefile` you provide to build the public test executables on the submit server; if there is no `Makefile` or the `Makefile` does not operate, your whole project will fail on the submit server. For that reason, we strongly suggest that you submit early to the submit server, to test that your code builds correctly with your `Makefile`.

2.2 Hash Table

The hash table you will implement is a fixed-size hash table using closed addressing (with linked list chaining to handle collisions).

2.2.1 Chaining and table management

Linked list chaining works by maintaining a linked list for each bucket in the hash table. As an element to be inserted is hashed to a bucket, it is added to that bucket's list; to delete an element, the element is deleted from the appropriate list. This allows a nearly infinite (bounded by the amount of memory available) number of elements to be inserted into the hash table without resizing. However, long linked lists present a performance problem, so you must keep track of the average list length, and double the size of the table when the average list length is more than 5 elements.

2.2.2 Hash code algorithm

Please refer to the Project #2 specification for the hash code algorithm.

2.2.3 Hash table functions

Since many operations in this project require dynamic memory allocation, and memory allocation calls can fail, you will need to check the results of every memory allocation call to ensure your program is executing correctly. Should any memory allocation fail while executing your code, your program should print out an appropriate error message to standard error, and then exit with the `EX_OSERR` exit code from `<sysexits.h>`.

You need to implement the following functions:

1. `Table *create_table()`

   Allocate and return a properly initialized hash table (with a bucket count of `INIT_BUCKETS` as specified in `hashtable.h`).

2. `void reset_table(Table *t)`

   Reset the table to an empty state. Do nothing if `t` is `NULL`.

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

   Attempts to insert a key/value pair into the table. Insertion fails if `key`, `val` or `t` is `NULL`.

   If the key is already in the table, its corresponding value in the table should be overritten by the value parameter given.

Return 0 if the insert is successful, -1 otherwise.

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, and `*val` should be set to point to that copy. If either `t` or `key` is `NULL`, the search is defined to fail.

   Return 0 if the key is found in the table, -1 otherwise.

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 currently 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.

9. `char **get_keys(Table *t)`

   Return an array of all keys present in the table. The keys should be in ascending sorted order. The array, and all its contents, should be dynamically allocated, with the elements of the array being copies of the keys in the table. The caller is responsible for later freeing the memory in the array. If `t` is `NULL`, the function should return `NULL`.

10. `char **get_values(Table *t)`

    Return an array of all values present in the table. The values should be sorted by the keys associated with them (so if the pairs `"apple"` ⇒ `"zebra"` and `"banana"` ⇒ `"koala"` are in the table, the value array would be `["zebra", "koala"]`, because `"apple"` is lexicographically less than `"banana"`). Once again, the array and all contents should be dynamically allocated, and the caller must take care of freeing the array. If `t` is `NULL`, this function will also return `NULL`.

11. `void destroy_table(Table *t)`

    Frees all memory associated with the given table, including keys and values. If `t` is `NULL`, the function should do nothing.

2.3 Hash Table Testing

The `hashtester.o` object file we provide is a program that should be linked against a hash table implementation that defines all the functions in (this project's) `hashtable.h`. You may create different executables by linking the object file against different implementations; for example, if you have two hash table implementations, `good_table.c` and `bad_table.c`, you could build two executables like this:

```
gcc -g -Wall -Werror -pedantic-errors -ansi -c good_table.c
gcc -g -Wall -Werror -pedantic-errors -ansi -c bad_table.c
gcc good_table.o hashtester.o -o good_table
gcc bad_table.o hashtester.o -o bad_table
```

Running the program `good_table` would then test the `good_table.c` implementation, and the program `bad_table` would test the `bad_table.c` implementation. These programs are referred to here as `hashtester` programs.

`hashtester` programs are very similar to the `hashtester` program you were asked to implement for Project #2, but they have differences that are explained in Section 2.3.1, especially regarding the format of the input files.

The `htester` program we also provide accepts these new input files, which also may optionally have expected output in them. If there exists expected output (as explained in Section 2.3.1), `htester` will compare the output generated by a `hashtester` program with the expected output listed in the input file. In this way, you can create test cases for `hashtester` programs that are contained in a single file.

`htester` should be run like this:

```
htester good_table testfile
```

where `good_table` is a `hashtester` program, and `testfile` is an input file. `htester` will run the `good_table` program with an argument of `testfile`, and then report either (a) the output of `good_table`, if `testfile` does not contain expected output; or (b) whether or not `good_table` "passes" the test provided, by comparing the output generated against the expected output, if expected output is provided.

2.3.1 Differences from Project #2 for the `hashtester` program

The format for the input files is similar to Project #2, but there are some important changes to the format, as well as the operation of the program, as described here:

1. The `display` command's `item` argument may now be one of a. `key_count`, which operates in the same manner as for Project #2; b. `keys`, which prints out all keys in the array returned from `get_keys();` and c. `values`, which prints out all values in the array returned from `get_values()`. Note that `display table` is not a valid command for this project, as it was in Project #2.

   The `keys` and `values` options have output in this format:

   ```
   Key 1: apple
   Key 2: banana
   ...
   Value 1: zebra
   Value 2: koala
   ```

2. If a line containing only the string `__END__` is present in the input file, all lines following that line are considered to be the expected output of the `hashtester` program, given the preceding lines as input.
3. Once a `hashtester` program reaches the end of commands (either at EOF or `__END__`), the hash table is destroyed. Then the `hashtester` program checks for memory leaks, and prints a message stating whether or not dynamically allocated memory is still not freed. Memory leak checking is done after all other operations have occurred. In addition, heap integrity checks are performed, and may cause the program to crash (which would cause `htester` to classify the hash table as an incorrect implementation).
4. The `htester` program does not read input data from standard input; instead, it only reads from the file given on the shell command line when the program is run.

3 Important Points

1. You must use the constant `INIT_BUCKETS` for the initial number of buckets in the table. Our tests may change this value and your table needs to behave appropriately when the value changes.
2. As for Project #2, some of the required functions for the hash table will perform similar operations as part of fulfilling their specifications. Your style grade will depend in part on your ability to minimize unnecessary code duplication, either by creating helper functions to perform common operations, or reusing the required functions in other functions.
3. As with all programs that use dynamic memory allocation, you must avoid memory leaks in your code, in all circumstances.

4 Grading Criteria

Your project grade will be determined by the following:

Results of public tests	20%
Results of secret tests	65%
Code style grading	15%
Extra credit	5%

The public tests will be made available shortly after the project specification is released. You can find out your results on these 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.

The style grading will take place after the late deadline, and will follow the standard course code style guidelines (as described in the Project #2 specification).

Secret tests will consist of both C programs and trials where we will run your test files against several of our implementations of the hash table. In these trials, you will receive points based on whether your test suite correctly distinguishes incorrect and correct hash table implementations (according to the specification of **this** project, not Project #2). A test suite is considered to call an implementation correct if the expected output matches the actual output in all test cases; if any mismatch is triggered by any test case, the implementation is labeled incorrect by the test suite. To make testing reasonable, you are limited to at most 30 test files; any `testfileXX` files where `XX` is greater than 29 will be ignored. Similarly, each test file is limited to no more than 100 commands (commands, not lines); the `hashtester` program will enforce this limit. As a guide, you may assume that the only files we will consider part of your test suite are those files displayed when executing the command "`ls testfile[0-2][0-9]`" in your project directory.

4.1 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 Monday, October 18, that passes all public tests for the project.

5 Submission

5.1 Deliverables

The only files we will grade are (a) your `Makefile`; (b) `hashtable.c`, your hash table implementation; and (c) `testfileXX`, the test files you have submitted. We will use our versions of the header file `hashtable.h` to build our tests, so do not make any changes to that file.

5.2 Procedure

The submission procedure is the same as for previous projects (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.

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 cases. 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.

We would also like to remind you that this project is not trivial, and so we recommend you start as soon as possible.