CMSC 216 Project #4 Spring 2016

Hash Table II **Thursday March 31, 8:00PM**

1 Overview

In this project you will use dynamic memory allocation techniques in order to implement a variable-size hash table using closed addressing (with linked list chaining to handle collisions). This hash table implementation allows keys (strings) of any length, and a key can be associated with any kind of object.

- Mon Mar 28, 8:00PM - Your code must pass the first two public tests (public01, public02). That is the only requirement for this deadline. We will not grade the code for style. This first part is worth .5% of your course grade (NOT .5% of this project grade). Notice you can still submit late for this part.
- Thu Mar 31, 8:00PM - Final deadline for the project. Notice you can still submit late (as usual).

Remember that every class project has a good faith attempt. Requirements and deadlines can be found at

`http://www.cs.umd.edu/class/spring2016/cmsc216/goodfaithattempt.shtml`

2 Project Files

The files for this project can be found in the project4 directory of the public class directory. Make sure you copy that folder to the 216 folder in your home directory. The files associated with the distribution are:

1. `htable.h` - This file provides prototypes for the functions your hash table must implement. It also provides structure definitions, typedefs and symbolic constants to be used when implementing the hash table. **Keep in mind that you may not modify this file.**
2. htable.c - This file represents your hash table implementation. Notice we have provided a hash function for you.
3. .h and .c files associated with the memory checker tool.
4. Makefile - This is the file where you will define the project's makefile. The name of the makefile must start with capital M.
5. public01.c, public02.c, public03.c - The public tests for this project.
6. student_tests.c - You will write student tests in this file.
7. `.submit` file - This file allows you to submit your project.

3 Specifications

3.1 `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 following options:

-ansi -Wall -g -O0 -Wwrite-strings -Wshadow -pedantic-errors -fstack-protector-all

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

1. `all`: make all executables
2. `public01, public02, public03`: each one building a public test

3. `student_tests`: one of your executables.
4. `htable.o`:
5. `my_memory_checker_216.o`:
6. `clean`: delete all object files and executables

While you are welcome to add other targets to your `Makefile`, the ones listed above must be present. Make sure you add all the necessary targets that avoid any unnecessary compilation (hint: use the touch command to check your makefile). **You must implement the Makefile using explicit rules**; using implicit rules is not allowed.

3.2 Hash Table

In the file named `htable.c` provide an implementation of the hash table. The hash table you will implement is a variable-size hash table using closed addressing (with linked list chaining to handle collisions).

3.2.1 Chaining and table management

Linked list chaining works by maintaining a linked list for each bucket in the hash table. When 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. **Note: when adding an element to the list, add it to the beginning of the list.**

3.2.2 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. Functions where memory allocation fails should return FAILURE (value defined in htable.h).

The functions you need to implement rely on the following structures:

```
typedef struct bucket {
   char *key;
   void *value;
   struct bucket *next;
} Bucket;

typedef struct {
   int key_count;
   int table_size;
   void (*free_value)(void *);
   Bucket **buckets;
} Table;
```

1. `int create_table(Table **table, int table_size, void (*free_value)(void *))` - This function allocates a Table structure and initializes the buckets field with an array (of size table_size) of pointers to Bucket structures. Each array entry must be initialized to NULL. The function will initialize other table fields based on parameter values.

   This function returns SUCCESS (value defined in htable.h) if the table can be created. The function will fail and return FAILURE if table is NULL or table_size is 0.

   The free_value parameter represents a function that will be used when removing entries from the table. If the user provides a free_value function, your code must use that function in order to free the memory associated with the value (not the key) of a hash table entry. If a free_value function is not provided (i.e., a value of NULL is provided), when deleting a table entry you only need to free the

memory associated with the key. Otherwise the user of the hash table is responsible for deallocating the memory associated with the value. We show an example of using a free_value function in the file example_free_value_function.c.

2. `int destroy_table(Table *table)`

   Frees ALL memory associated with the given table, including all nodes of the hash chains. All remaining values should be freed by invoking the free_value function on each.

   This function returns SUCCESS if the table can be destroyed successfully. The function will fail and return FAILURE if table is NULL.

3. `int put(Table *table, const char *key, void *value);`

   Attempts to insert a key/value pair into the table. The function will make a copy of the key. The value is "kept" by the table. The put() function should not attempt to copy value, since it does not know how much memory the value uses.

   If the key does not exist, it will be added to the table by making a copy of the key. If the key is already in the table, no additional copy must be made, and the corresponding value in the table should be updated with the value parameter. The previous value should be freed by free_value, if both are not NULL.

   This function returns SUCCESS if the value was added to the table. The function will fail and return FAILURE if table is NULL or key is NULL. Value is allowed to be NULL.

4. `int get_value(const Table *table, const char *key, void **value);`

   This function will copy the value pointer into the value out parameter if the key is found. If the key is not found in the table, *value will be set to NULL and the function will return FAILURE.

   This function returns SUCCESS if the key was found in the table. The function will fail and return FAILURE if table is NULL or key is NULL.

5. `int get_key_count(const Table *table);`

   This function returns the number of keys in the table. The function will fail and return FAILURE if table is NULL.

6. `int remove_entry(Table *table, const char *key);`

   This function will remove the key/value pair from the table if the key is present in the table. It should invoke free_value on the associated value if the key is present in the table and the associated value is not NULL.

   This function returns SUCCESS if the value was deleted from the table. The function will return FAILURE if table is NULL, key is NULL or if the the key is not found.

7. `int clear_table(Table *table)`

   This function removes the list associated with each bucket. After removing the lists, the buckets will be set to NULL. Notice that you must not remove the array of bucket pointers. The key count will be set to 0 as part of the clearing process. All values must be freed using free_value();

   This function returns SUCCESS if the table is cleared and FAILURE if the table is NULL.

8. `int is_empty(const Table *table)`

   This function returns SUCCESS if there are no keys in the table or table is NULL, and 0 otherwise.

9. `int get_table_size(const Table *table)`

   This function returns the table size. The function returns FAILURE if the table is NULL.

3.3 Hash Table Testing

You must define tests for your hash table implementation as described in the file student_tests.c. You will be graded on the quality of these tests. The main function in student_tests should run all tests, but exit with the EXIT_FAILURE code if any of your tests fails, and EXIT_SUCCESS otherwise. The student_tests.c file in the distribution performs this task, though you are welcome to rewrite it as you add tests.

3.4 Dynamic Memory Allocation

1. As with all programs that use dynamic memory allocation, you must avoid memory leaks in your code, in all circumstances.
2. Keep in mind that when you run valgrind on code that uses the memory tool we have provided, the tool may detect leaks for memory that valgrind allocates. To test your code using valgrind, disable our memory tool (by commenting out the function calls associated with the tool) and run valgrind.
3. If a function requires multiple dynamic-memory allocations and one of them fails (malloc/callocs returns NULL), you are not responsible for freeing the memory of those allocations that were successful. For example:

```
t1 = malloc(...) /* This one is successful */
t2 = malloc(...) /* This one failed, just return FAILURE and don't worry about t1. */
```

4. The best way to check you are using dynamic-memory allocation correctly is to test often. **You need to write your student tests as you develop your code**. You will need to show your student tests to TAs, if you expect help from them.

4 Grading Criteria

Your project grade will be determined by the following:

Results of public tests	10 pts
Results of release tests	4 pts
Results of secret tests	50 pts
Student tests	12 pts
Code Style	14 pts
Makefile	10 pts

4.1 Style grading

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

1. Do not use global variables.
2. Follow the C style guidelines available at:

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

   Graders will go over each of the items in this description. Make sure you go over this list and verify you are satisfying all the style requirements, otherwise you will lose credit.

4.2 Other

Do not cast hash_code() to int as this will not allow you to generate correct results. For example,

`index = hash_code(k) % t->table_size;` is correct

whereas `index = (int) hash_code(key) % table->table_size`

is incorrect.

5 Submission

5.1 Deliverables

The only files we will grade are the htable.c, student_tests.c and the Makefile. We will use our versions of the header file `htable.h` to build our tests, so if you make changes to that file your code *will not compile*.

5.2 Procedure

The submission procedure is the same as for previous projects (execute "`submit`" in the project directory).

5.3 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.

6 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.