Project 2-1 Testbench

For this project, you'll be required to implement a hash table that meets specified interface requirements. A common technique to make this process easier is known as “Test Driven Development.” This paradigm takes its inspiration from double booking in accounting – if the two sides of your balance sheet don't match, you know you've made a mistake. Similarly, test driven development prescribes that you write two separate programs: the code you wanted in the first place – in our case a hash table - and a test program that validates it. If the test program fails, you know something is wrong.

Test driven follows a cycle for adding features:

1. Red: Write failing tests for the new feature
2. Green: Make the failing tests pass
3. Refactor: Clean up the code to prepare for the next feature

In the first step, you write failing tests that describe the desired behavior of your hash table. This code will use the hash table implementation directly to perform some action, and then will verify that the ending state of the table is as expected. For this assignment, we'll be using Google Test Framework for implementing test cases. More details on the framework are given below.

It's important to note that you should always strive to see your tests fail the first time you run them. By seeing the tests fail, you know that they're actually validating something. If you write a test for a new feature that passes before you've implemented that feature, it's likely not testing what you would expect! Be sure in this step to write tests for all your edge cases. Each test should be as specific as possible, so that when you see it fail you can pinpoint what part of your code caused the failure. Most features will require more than one test to cover all the edge cases: for example, when retrieving values from the table, what should happen if the table is empty? not empty?

Once your failing tests are in place, then move on to implement the new feature. During this process, make small changes to your hash table code, and recompile and run your tests often to measure your progress. You'll sometimes find that you've broken something that was working previously – this is called a *regression*, and it's a normal part of the software development cycle. With unit tests, you can find regressions early during development and figure out the cause before you forget what changed.

Finally, once all your tests are passing, you have the opportunity to clean up your code. Again, make small changes and run your test suite often to avoid any regressions. You can be confident that any changes you make while cleaning up didn't break your implementation, since the test suite will immediately notify you of any failures.

Unit Testing with Google Test Framework (gtest)

Google Test Framework is a common and powerful software package for unit testing. It gives you easy-to-use macros for defining tests, and includes code for automatically detecting your tests and building an executable to run them. This is all packaged for you in the project zip folder.

You can find the function prototypes and the expected behaviors of the functions for hash table documented in `hash_table.h`. Based on that documentation, you should write your hash table implementation in `hash_table.c`. Your unit tests should be in `ht_tests.cpp`.

The example below shows an example of a unit test for overwriting an existing entry in the hash table by calling `insertItem` function:

```
TEST(InsertTest, InsertAsOverwrite)
{
	HashTable* ht = createHashTable(hash, BUCKET_NUM);

	// Create list of items to be added to the hash table.
	size_t num_items = 2;
	HTItem* m[num_items];
	make_items(m, num_items);

	// Only insert one item with key=0 into the hash table.
	insertItem(ht, 0, m[0]);

	// When we are inserting a different value with the same key=0, the hash table
	// entry should hold the new value instead. In the test case, the hash table
	// entry corresponding to key=0 will hold m[1] and return m[0] as the return
	 // value.
	EXPECT_EQ(m[0], insertItem(ht, 0, m[1]));

	// Now check if the new value m[1] has indeed been stored in hash table with
	// key=0.
	EXPECT_EQ(m[1], getItem(ht,0));

	destroyHashTable(ht);
	free(m[0]);    // don't forget to free item 0
}
```

Let's take this apart line by line.

```
TEST(InsertTest, InsertAsOverwrite)
{
	…
}
```

This is the general format for declaring a unit test. The first argument to the TEST macro is a group name. The second argument is the test name. Group names can be shared among multiple tests – in this example, I've put this test in with my test group for insert functions. Each test name must be unique within the group.

```
// Initialize items for your test
HashTable* ht = createHashTable(hash, BUCKET_NUM);

// Create list of items to be added to the hash table.
size_t num_items = 2;
HTItem* m[num_items];
make_items(m, num_items);
```

This initializes dummy table items in an array `m[]` that contains two items to use in this test case. The `make_items` function is provided for you in `ht_tests_shell.cpp`. You can change `num_items` to the number of items you need for your test. To avoid weird behavior in your test code, make sure you create exactly as many items as you use in the test case, and that you insert each item you create into the table once.

```
// Only insert one item with key=0 into the hash table.
insertItem(ht, 0, m[0]);
```

This is where you set up your table for this test case. For our example, we're simply inserting one item into the hash table with key 0. `m[0]` is the dummy item we created earlier.

```
// When we are inserting a different value with the same key=0, the hash table
// entry should hold the new value instead. In the test case, the hash table
entry
// corresponding to key=0 will hold m[1] and return m[0] as the return value.
EXPECT_EQ(m[0], insertItem(ht, 0, m[1]));

// Now check if the new value m[1] has indeed been stored in hash table with
// key=0.
EXPECT_EQ(m[1], getItem(ht,0));
```

This is the guts of the test. Now that we have set up our table as desired for the test, we need to make sure that it behaves as we expect it to. The `EXPECT_EQ` macro is one of many ways to tell the test framework if the test passed or failed. For this test, we are doing two different checks. First, the `insertItem` function should return what is in the original bucket when a new item is inserted with the same key. Therefore, when we insert a different item, namely `m[1]`, into the hash table with the same key 0, we expect to get `m[0]` from the hash table. In the second check, we ensure that the new item `m[1]` has been successfully inserted into the hash table. Therefore, when we call `getItem` function with key 0, we should retrieve `m[1]` back.

The meaning of the `EXPECT_EQ` macro arguments is as follows:

```
EXPECT_EQ( <expected value>, <actual value> );
```

If either one of the two `EXPECT_EQ` calls fails, the test case will fail and we'll see that failure in the output of the test executable.

```
// Delete the table
destroytable(table);
free(m[0]);    // don't forget to free item 0
```

Finally, we have to clean up the table. According to the specification, this deletes all the items you inserted in the table, so we don't need to worry about freeing memory for the `m[1]`. However, since `m[0]` is no longer in the hash table, we have to manually free the memory for it. Otherwise, memory leak occurs. Note that unit tests for memory leaks are not explicitly supported in Google Test. Therefore, you could use Valgrind, a programming tool for memory debugging, memory leak detection, and profiling. More information about Valgrind can be found here: http://valgrind.org/

Below are several other validation macros that you might find useful in writing your tests:

```
EXPECT_EQ(val1, val2); // val1 == val2
EXPECT_NE(val1, val2); // val1 != val2
EXPECT_LT(val1, val2); // val1 <  val2
EXPECT_LE(val1, val2); // val1 <= val2
EXPECT_GT(val1, val2); // val1 >  val2
EXPECT_GE(val1, val2); // val1 >= val2
```

These `EXPECT_*` macros are known as *non-fatal assertions*. This means that the test will keep running, even if the expected condition is not met. If the condition is wrong, the test will still fail, but the remaining code in the test case will still have the chance to execute. There is an identical set of *fatal assertion*s: `ASSERT_*`. Fatal assertions will cause a test case to stop running immediately if the condition is not met.

More information on Google Test Framework can be found here:
https://github.com/google/googletest/blob/master/googletest/docs/Primer.md

The Build System

Until now, your C programs have consisted of just a single source file. For this project, you'll be directly editing two (2) source files – `hash_table.c` and `ht_tests.cpp` – and making use of a few more that are the source code for the test framework. In order to coordinate building all these files, we'll be using a tool called make.

This document is not going to talk about what make is or how to write Makefiles. I'll instead focus on how to use make for this project. However, make is ubiquitous in open source software, particularly in C/C++-based projects, so you're highly encouraged to study the Makefile we provide and to learn about it on your own.

To use the build system for this project, you'll first need to ensure you have `make` installed. On Ubuntu, run this command in the terminal:

```
sudo apt-get install make
```

Once you have make installed, navigate to the project folder and run the command

```
make
```

This will build and run the test executable, and you should see the output of running all your tests. If everything succeeds, it will look something like this:

```
[==========] Running 1 test from 1 test case.
[----------] Global test environment set-up.
[----------] 1 test from InsertTest
[ RUN      ] InsertTest.InsertAsOverwrite
[       OK ] InsertTest.InsertAsOverwrite (0 ms)
[----------] 1 test from InsertTest (0 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test case ran. (0 ms total)
[  PASSED  ] 1 test.
```

On failure, you should expect something like this:

```
[==========] Running 1 test from 1 test case.
[----------] Global test environment set-up.
[----------] 1 test from InsertTest
[ RUN      ] InsertTest.InsertAsOverwrite
ht_tests.cpp:180: Failure
      Expected: m[0]
      Which is: 0x7ffca0402ae0
To be equal to: insertItem(ht, 0, m[1])
      Which is: 0x7fff5fb66f70
ht_tests.cpp:184: Failure
```

```
      Expected: m[1]
      Which is: 0x7ffca0402ad0
To be equal to: getItem(ht,0)
      Which is: NULL
[  FAILED  ] InsertTest.InsertAsOverwrite (0 ms)
[----------] 1 test from InsertTest (0 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test case ran. (0 ms total)
[  PASSED  ] 0 tests.
[  FAILED  ] 1 test, listed below:
[  FAILED  ] InsertTest.InsertAsOverwrite

 1 FAILED TEST
make: *** [test] Error 1
```

The framework will tell you at the bottom of the output if you failed any tests and which tests failed. It also gives you detailed feedback for why the assertion failed and the line number of the assertion that generated the failure. This output makes it easy to focus your debugging efforts on particular problematic sections of code and sets the stage for using tools like GDB breakpoints to observe the behavior of your failing test cases.

The Makefile is set up for a few other functions too. Here's the reference:

`make [test]` - builds everything, and runs the tests
`make build` - builds only, don't run tests
`make clean` - removes all files generated by make