ECE 2035 Project 2-1: Hash Table Implementation and Test

Objective: In this part of Project 2, you will build a hash table library for use in P2-2.

Building to a given specification (the Application Programmer Interface, or API) and verifying correctness of your implementation are important concerns in software engineering. APIs establish a sort of contract between the creators and users of a software library about its expected behavior and functionality. The interface also allows developers to separate the concerns of design and implementation of large projects. Early on, design can proceed without distraction from the details of implementation by instead producing a specification of required components and their interfaces. Later, the implementation of each module can proceed more independently from the design of the project as a whole.

We will explore this idea by creating and later using a C library based on a pre-specified API. In this project, you will create a Hash Table library that will later be used as a component of P2-2. Compliance to the specification is particularly important, so we'll take a look at automated testing as a tool for verifying implementation correctness.

Hash Table Library

The hash table is made of an array of buckets that can hold any arbitrary data from users. There will be a hash function provided by the user that maps a key to an index for a specific bucket. Each bucket can hold multiple entries of data and will be implemented as a singly linked list. An overview of the hash table implementation is:

Structs:

- `HashTable`
- `HashTableEntry`

Public Interface Functions:

- `createHashTable`
- `destroyHashTable`
- `insertItem`
- `getItem`
- `removeItem`
- `deleteItem`

Private Helper Functions: (only in hash_table.c)

- `createHashTableEntry`
- `findItem`
- (Any other useful helper functions)

**Note: The user has to provide a pointer to the custom hash function when the hash table is initialized.*

The detailed API for your hash table library can be found in `hash_table.h`. The functions in this module create and manipulate hash tables and hash table entries. (e.g., inserting/deleting entries, accessing entries, creating/destroying the table).

This document is the single source of information for how your implementation should behave! Any code that later uses the library will assume that it behaves according to this API.

The files `hash_table.h` and `hash_table_shell.c` provide a shell implementation of this API. Note that the function `createHashTable()` is provided for you and you need to complete the remaining functions.

Automated Testing

Up to this point, your testing of projects has likely been mostly ad hoc and manual, such as trying some different inputs by hand. For this project, we introduce more powerful tools for writing automated tests. By generating a comprehensive test suite that can run automatically, you can be confident that your implementation meets the API specification. Automated tests are also useful during development, since they let you evaluate changes quickly and alert you immediately if anything breaks.

We'll be using a combination of three tools for this part of the project:

1. Google Test Framework: a library for writing and automatically executing tests;
2. make: a build tool to ease compilation and run code, and
3. Address Sanitizer: a Linux-based tool for detecting memory errors.

GTest: A simple test suite has been provided in `ht_tests_shell.cpp`. This test suite has a few cases that exercise the hash table implementation, but they are provided mainly as examples of how to use the framework. As you implement your library, **you'll need to add more tests** to exercise all the edge cases for each function. This is the same framework we will use to grade your submissions, so it is to your advantage to test thoroughly. The tutorial gtest_hashtable.pdf describes unit testing and the GTest framework (linked from P2-1 Assignment on Canvas).

`make`: A Makefile is provided to facilitate using the testing framework. After navigating to the directory where the module is located, you can compile the module and the test bench by using the Unix/Linux command :

```
> make
```

> Note: if this gives you an error about missing commands, such as "`g++: command not found`" you may need to install a set of basic utilities by executing the command
> `> sudo apt install build-essential`

You can use the following command to remove all the files generated by make:

```
> make clean
```

Address Sanitizer: To check for memory errors, use the following command:

```
> make leak
```

This compiles and runs your code with Address Sanitizer to detect memory access errors. Detailed instructions and an example are provided in Mem-error-detection-w-Addr-Sanitizer.zip (linked from P2-1 Assignment on Canvas), including information on using the remote ECE Linux server to do this if it's not configured on your local machine.

Summary Process for P2-1: The unit testing process allows you to incrementally design and test your code in an iterative process of testing interleaved with revising/extending your code. The testing involves using GTest to check that the code satisfies specifications and using Address Sanitizer to detect memory errors. You should write small portions of your code during each of these design-and-test cycles. Do not write a large portion of your code before running tests. This

makes it much more difficult to find and repair errors. P2-1-incremental design and test.pdf gives a step-by-step guide for completing P2-1 using unit testing.

Project Submission

In order for your solution to be properly received and graded, there are a few requirements. **For P2-1**: Upload the following files to Canvas before the scheduled due date:

- `A zip file named ht.zip containing the following files:`
  - `hash_table.h`
  - `hash_table.c`
  - `ht_tests.cpp`

Your code will be evaluated based on:

- **correctness** (passes tests exercising the library functions, covering all edge cases)
- **no memory errors** (Address Sanitizer detects no errors in your code)
- **no compilation errors** (make compiles your code with no errors/warnings)

We will run GTest and Address Sanitizer using a set of tests we created to exercise and evaluate each student's code. The ht_tests.cpp file you hand in will provide additional information demonstrating your design and testing effort.

Supplementary Documentation: (linked from the P2-1 Assignment page on Canvas)

- gtest_hashtable.pdf describes unit testing and the GTest framework
- P2-1-incremental design and test.pdf gives a step-by-step guide for completing P2-1 using unit testing
- Mem-error-detection-w-Addr-Sanitizer.zip contains a guide to checking your code for memory errors using Address Sanitizer, with an accompanying example.

You should design, implement, and test your own code. There are many ways to code this project. Any submitted project containing code (other than the provided framework code) not fully created and debugged by the student constitutes academic misconduct.

Good luck!