P2-1 Hash Table Implementation and Test

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.

Through the end of the course, 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, a quest based video game. Compliance to the specification is particularly important, so we'll take a look at automated testing as a tool for verifying implementation correctness.

In this video game, the main character will be navigating a two-dimensional world with objects such as trees, rocks, traps, other characters and obstacles. To complete the quest or mission, the main character has to interact with these components in the world. As these game components are sparsely located (spread out) in the two-dimensional world, a hash table will provide a memory-efficient way to to store and access these component attributes.

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 truth for how you 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 two tools for this part of the project: Google Test Framework, a library for writing and automatically executing tests; and make, a build tool to ease compilation. Additional information on using the test framework will be provided by the TAs.

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 many 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.

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*

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

> *make clean*

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`

NOTE: follow the above instructions exactly.

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!

Project Grading: The project grade will be determined as follows:

part	*description*	*percent*
P2-1	Hash Table Implementation	35
P2-2	Baseline features	40
P2-2	Advanced features (up to max of 10 features, for max 50 points)	50
P2-2	TA demo/checkoff	
	Total	125/100