CS271: Data Structures

Instructor: Dr. Stacey Truex

Project #3

This project is meant to be completed in groups. You should work in your Unit 2 groups. Implementation solutions should be written in `C++` and analysis should be written in LaTeX. Only one submission (the last submission uploaded to canvas) will be graded per group. Submissions should be a compressed file following the naming convention: `NAMES_cs271_project3.zip` where `NAMES` is replaced by the first initial and last name of each group member. For example, if Dr. Truex and Dr. Currin were in a group they would submit one file titled `STruexFCurrin_cs271_project3.zip`. **You will lose points if you do not follow the course naming convention**. Your `.zip` file should contain a *minimum* of 7 files:

1. `makefile`
2. `hash_table.cpp`
3. `test_hash_table.cpp`
4. `usecase.cpp`
5. `main.cpp`
6. `analysis.pdf`
7. `commits.pdf`: a commit history for your GitHub project

Additional files such as a `hash_table.h` header file or `README.md` are welcome. The above merely represent the minimum files required for project completion. Your code is expected to implement both a `HashTable` and an `Element` class. Details for each part of the project are as follows.

Specifications

Element

Implement an `Element` template class that, at a minimum, has the following operations:

- get_key(): `e.get_key()` should return the element's **numeric** key value. For example:

```
Element<string> e("string data", 5);
cout << e.get_key() << endl;
```

  should result in the printing of the numeric value 5

- get_data(): `e.get_data()` should return the element's **template** data. For example, using `e` from above:

```
cout << e.get_data() << endl;
```

  should result in the printing of the string "string data"

You may also choose to implement methods other than `get_key`. This choice along with other parts of the class design are left to each pair.

Hash Table

Implement a `HashTable` template class that implements a hash table storing `Element` objects. Your hash table should have, at a minimum, support for the following three dictionary operations.

- insert(`data`, `key`): `ht.insert(d, k)` should insert an element with data `d` and key `k` into the hash table `ht`. For example:

```
HashTable<string> ht(5);
ht.insert("example", 8);
```

  should result in a hash table with 5 slots and an element with data `"example"` and key `8` in slot $h(8) \in \{0, 1, 2, 3, 4\}$ for a set (by you) hash function h. Collisions should be handled via chaining using a doubly linked list. New elements to a doubly linked list should be inserted at the head.

- remove(`k`): `ht.remove(k)` should delete the `Element` with key `k` from the hash table `ht`. For example, using `ht` from above:

```
ht.remove(8);
```

  should result in an empty hash table. You may assume that keys are distinct.

- member(`d`, `k`): `ht.member(d, k)` should indicate whether the hash table `ht` contains an `Element` with data `d` and a key `k`. For example, using the table `ht` from above:

```
ht.insert("test", 28);
if(ht.member("test", 28)){
    cout << "(test, 28) is a member of ht" << endl;
}
if(ht.member("other", 28)){
    cout << "(other, 28) is a member of ht" << endl;
}
```

  should **only** result in the printing of the statement `"(test, 28) is a member of ht"`.

Unit Testing

For unit testing, you should set your hash function to $h(k) = k\%m$ where m is the size of the hash table.

Additionally, you are also expected to implement a `to_string()` method which returns a string with the elements in each doubly linked list *separated by a single space* and displayed as `(data, key)`. Each slot in the hash table should be separated by a new line. For example, using the table `ht` generated in the specifications:

```
cout << ht.to_string() << endl;
```

should result in the printing of the string

```
0:
1:
2:
3: (test,28)
4:
```

Your `to_string()` method will be required for your class to pass testing.

For each `HashTable` method included in your `HashTable` class and `Element` method included in your `Element` class, write a unit test method in a separate unit test file that *thoroughly* tests that method. Think, in addition to common cases: what are my boundary cases? edge cases? disallowed input? Each method should have *its own* test method.

An example test file `test_hashtable_example.cpp` has been provided and demonstrates (1) a general outline of what is expected in a test file and (2) a guide on how your projects will be tested after submission. The tests included in `test_hashtable_example.cpp` are not exhaustive. The unit testing in your `test_hashtable.cpp` file should be much more complete. Additionally, for grading purposes, your code will be put through significantly more thorough testing than what is represented by `test_hashtable_example.cpp`. Passing the tests in this example file should be viewed as a lower bound.

Documentation

The expectation of all coding assignments is that they are well-documented. This means that logic is documented with line comments and method pre- and post- conditions are properly documented immediately after the method's parameter list.

Pre-conditions and post-conditions are used to specify precisely what a method does. However, a pre-condition/post-condition specification does not indicate how that method accomplishes its task (if such commenting is necessary it should be done through line level comments). Instead, pre-conditions indicate what must be true before the method is called while the post-condition indicates what will be true when the method is finished.

Use Case

Finally, use your `HashTable` to solve the following problem: You are given a csv file in which each line represents a username, password pair. Using a hash table, you should then support the following interaction with the program user:

- Ask the user to enter their username and password.
- If a correct password is entered, notify the user that access has been granted.
- If an inaccurate password is entered, notify the user that access has been denied.

An example csv file has been provided. Note that **passwords should not be stored** and can be any numeric value $\in \mathbb{Z}_{10^{10}}$ while user names can be any random string.

Your solution should be implemented in `usecase.cpp` using the following two functions:

```
HashTable<T>*create_table(string fname, intm)
bool login(HashTable<T>*ht, T username, string password)
```

where `fname` is the name of the csv file containing the username, password pairs and `m` is the intended size of the hash table. Your generated hash table should then be used with the `login` function where `ht` is the table from the `create_table` function, `username` is the entered username, and `password` is the entered password. Your function should return `true` if and only if the `username` and `password` match one of the known pairs (from the csv file). Note that your use case will only be tested when the template is set to `string`.

In your `main.cpp` file, your `main` function should include at least one example test case demonstrating the accuracy of your solution which allows for user input from the terminal.

Analysis

Finally, experiment with both of the following two hash functions:

1. Most Significant Bits Method: let $h(k)$ be the p most significant bits of k.
2. Cormen's Multiplication Method: let x be the factional part of $k * A$, then $h(k)$ should be the floor of $m * x$. Let A be as suggested by Knuth, $A = (\sqrt{5} - 1)/2$ and m be some power of 2.

Use LaTeXto create the document `analysis.pdf` which compares the performance of these two hash functions. Indicate which you believe is a *better* has function and *why*. **Use experimental data to support your position**.

Makefile

With each project you should be submitting a corresponding makefile. Once unpacking your `.zip` file, the single command `make` should create a `test` executable **and** a `usecase` executable. The command `./test` should then run all the unit tests in your `test_hash_table.cpp` file evaluating your `HashTable` and `Element` classes. The command `./usecase` should run the example test case in your `main.cpp` file demonstrating the accuracy of your login solution.

Rubric

Note that any coding projects that do not compile with the provided `test_hashtable_example.cpp` file will be given a 0. All projects that are able to be successfully compiled will be graded using the following rubric.

C++ Implementation	does not compile: 0/40		
	40 Total Points		
	Code	**Completeness** met submission requirements	10 pts
		Correctness passes unit testing	15 pts
		Validation implementation deductions ex: compile error with string template	——
	Usecase	**Correctness** passes unit testing	4 pts
		Validation implementation deductions ex: login is not a template function	——
		Analysis	3 pts
		incomplete, barebones, or missing	0/3
		incorrect conclusion	1/3
		conclusion lacks supporting evidence	2/3
		good, detailed, accurate analysis	3/3
	Efficiency	**Time Test**	2 pts
		encountered error - could not complete time test	0/2
		takes over 2x fastest submission	1/2
		within 2x fastest submission	2/2
		fastest submission	3/2
	Documentation	**Documentation**	3 pts
		extremely sparse documentation	0/3
		missing comments or pre- and post-conditions	1/3
		documentation lacks detail in areas	2/3
		detailed comments & pre- and post-conditions	3/3
	Testing	**Unit tests**	3 pts
		does not expand on example test file	0/3
		not all functions tested *or* testing not implemented as unit testing *or* no variation in templates	1/3
		caught some of the bugs in classmates' code	2/3
		caught most bugs in classmates' code	3/3