COP4530 – Project 5

Hash Tables and Its Application

Submission:

1. Create a tar file with the required files.
2. Make an electronic submission of your tarball file on Canvas.

Due Date: November 27, 2019, 11:50pm for all students
Late Penalty: maximum one day at the penalty of 15%.

1. Overview

In this project, you shall implement a generic hash table ADT and other supporting user interfaces. The objectives of this project include the following:

- Develop a simple password server program.
- Understand and get familiar with the hash table data structure, along with its application in managing user accounts.

2. Description of Provided Source Files

A tar file `proj05src.tgz` is available on Canvas. To extract these files from the tar file, you may use the following command **on the `linprog` machines, where your code will be tested and graded**.

- `tar zxf proj05src.tgz`

In this tarball, the following partial code has been provided to you.

- `hashtable.h:` partial implementation
- `hashtable.hpp:` partial implementation.
- `pass_server.h:` partial implementation
- `pass_server.cpp:` partial implementation.
- `userpass.cpp:` a driver program, partial implementation.
- `userpass:` a sample executable for linprog.cs.fsu.edu
- `scrypt.cpp:` sample program to use crypt() to encrypt password.
- `Makefile:` an initial Makefile that compiles the script.cpp program.
- `test0.txt:` a sample test case (which contains the commands that a user will type). You can redirect it to userpass as "userpass < test0.txt". Results will save in the file "testoutput". During execution, test0.txt also specifies a file (sample_users) to load a collection of users and passwords.
- `sample_users:` a sample input file with a collection of users and passwords.
- `output.txt:` the file contains the output from running userpass with test0.txt as piped input. The beginning of this file has output from running the final Makefile and script.

3. Project Description

This project contains two parts. In the first part of the project, you need to implement a hash table class template named HashTable. In the second part of the project, you will develop a simple password server program using the hash table you developed.

Task 1: Requirements of HashTable Class Template

- Your implementation of HashTable must be in the namespace of cop4530.
- You must provide the template declaration and implementation in two different files hashtable.h (containing HashTable class template declaration) and hashtable.hpp (containing the implementation of member functions). You must include hashtable.hpp inside hashtable.h as we have done in the previous projects.
- You must implement hash table using the technique of chaining with separate lists (separate chaining). That is, the internal data structure of the hash table class template should be a vector of lists. Use the STL containers for the internal storage (instead of any containers you developed in the previous projects).
- You must at least implement all the interfaces specified below for the HashTable class template.

Public HashTable interface (K and V are template parameters (generic data types), which represent the key and value types for a table entry, respectively)

- HashTable(size_t size = 101): constructor. Create a hash table, where the size of the vector is set to prime_below(size) (where size is default to 101), where prime_below() is a private member function of the HashTable and provided to you.
- ~HashTable(): destructor. Delete all elements in hash table.
- bool contains(const K & x): check if key k is in the hash table.
- bool match(const std::pair<K, V> & kv) const: check if key-value pair is in the hash table.
- bool insert(const std::pair<K, V> & kv): add the key-value pair kv into the hash table. Don't add if kv or key is already in the hash table. ~~If the key is the hash table but with a different value, the value should be updated to the new one with kv.~~ Return true if kv is inserted ~~or the value is updated~~; return false otherwise (i.e., if kv is in the hash table).
- bool insert (std::pair<K, V> && kv): move version of insert.
- bool remove(const K & k): delete the key k and the corresponding value if it is in the hash table. Return true if k is deleted, return false otherwise (i.e., if key k is not in the hash table).
- void clear(): delete all elements in the hash table
- bool load(const char * filename): load the content of the file with name filename into the hash table. In the file, each line contains a single pair of key and value, separated by a white space.
- void dump(): display all entries in the hash table. If an entry contains multiple key-value pairs, separate them by a vertical bar (|) (see the provided executable for the exact output format).
- bool write_to_file(const char *filename): write all elements in the hash table into a file with name filename. Similar to the file format in the load function, each line contains a pair of key-value pair, separated by a white space.

Private helper functions

- void makeEmpty(): delete all elements in the hash table. The public interface clear() will call this function.
- void rehash(): Rehash function. Called when the number of elements in the hash table is greater than the size of the vector.
- size_t myhash(const K & x): return the index of the vector entry where k should be stored.

- unsigned long prime_below (unsigned long) and void setPrimes(vector<unsigned long>&): two helpful functions to determine the proper prime numbers used in setting up the vector size. Whenever you need to set hash table to a new size "sz", call prime_below(sz) to determine the new proper underlying vector size. These two functions have been provided in hashtable.h and hashtable.hpp.

Make sure to declare as const member functions any for which this is appropriate.

You need to write a simple test program to test various functions of hash table. More details are provided in a later part of this description.

Task 2: Requirements of the Password Server Class (PassServer)

- Name the password server class as PassServer. Its declaration and implementation should be provided in two files, pass_server.h and pass_server.cpp, respectively.
- PassServer should be implemented as an adaptor class, with the HashTable you developed as the adaptee class. The type for both K and V in HashTable should be string. The key and value will be the username and password, respectively.
- PassServer must store username and encrypted password pairs in the hash table.
- PassServer must at least support the following member functions (again, make sure to declare as const member functions any that are appropriate):

Public interface:

- PassServer(size_t size = 101): constructor, create a hash table of the specified size. You just need to pass this size parameter to the constructor of the HashTable. Therefore, the real hash table size could be different from the parameter size (because prime_below() will be called in the constructor of the HashTable).
- ~PassServer(): destructor. You need to decide what you should do based on your design of PassServer (how you develop the adaptor class based on the adaptee HashTable). In essence, we do not want to have memory leak.
- bool load(const char *filename): load a password file into the HashTable object. Each line contains a pair of username and encrypted password with a whitespace between the two.
- bool addUser(std::pair<string, string> & kv): add a new username and password. The password passed in is in plaintext, it should be encrypted before insertion.
- bool addUser(std::pair<string, string> && kv): move version of addUser.
- bool removeUser(const string & k): delete an existing user with username k.
- bool changePassword(const pair<string, string> &p, const string & newpassword): change an existing user's password. Note that both passwords passed in are in plaintext. They should be encrypted before you interact with the hash table. If the user is not in the hash table, return false. If p.second does not match the current password, return false. Also return false if the new password and the old password are the same (i.e., we cannot update the password).
- bool find(const string & user): check if a user exists (if user is in the hash table).
- void dump(): show the structure and contents of the HashTable object to the screen. Same format as the dump() function in the HashTable class template.
- size_t size(): return the size of the HashTable (the number of username/password pairs in the table).

- bool write_to_file(const char *filename): save the username and password combination into a file. Same format as the write_to_file() function in the HashTable class template.

Private helper function

- string encrypt(const string & str): encrypt the parameter str and return the encrypted string.

  For this project, we shall use the GNU C Library's crypt() method to encrypt the password. The algorithm for the crypt() method shall be MD5-based. The salt shall be the character stream "1########". The resulting encrypted character stream is the

  `"$1$########" + '$' + 22 characters` = 34 characters in total.

  A user password is the sub string containing the last 22 characters, located after the 3rd '$'.

  Note: A sample program to demonstrate the use of the crypt() method is also provided. In order compile a program calling crypt(), you will need to link with the crypt library when you compile. You can read more information on the manual page of crypt(). Example compile command (used to build the sample scrypt program):

```
g++ -std=c++11 scrypt.cpp -lcrypt -o scrypt
```

Driver Program: In addition to developing the HashTable class template and the PassServer class, you need to write a driver program to test your code. Name the driver program userpass.cpp.

- A partial implementation of userpass.cpp is provided to you, which contains a menu() function. You must use this function as the standard option menu for user to type input. You may not alter the menu function.
- The driver program must re-prompt the user for the next choice from the menu and exit the program only when the user selection the exit "x" option.
- Run the provided executable userpass on linprog to see the expected behavior of each of the menu options, and the expected order of inputs. Make sure to test with error cases too, so that you see the appropriate error messages that are printed

4. Programs and files you need to submit.

You need to submit a total of **six** files for the completion of this assignment:

1) hashtable.h
2) hashtable.hpp
3) pass_server.h
4) pass_server.cpp
5) userpass.cpp
6) Makefile

Create a tarball file from these files using the command below, where `fsuid` is your FSU ID. All characters should be on the same command line.

- `tar zcf <fsuid>_proj5.tgz hashtable.h hashtable.hpp pass_server.h pass_server.cpp userpass.cpp Makefile`

Your program must compile on linprog.cs.fsu.edu. If you program does not compile on linprog, the grader cannot test your submission. Your executable must be named as `userpass`.

The interaction and output (including error messages) of your client's executable(s) must behave in the same manner as the distributed `userpass` (on linprog.cs.fsu.edu). For example, one of the ways to test your program would be to run a "diff" command between the output file(s) created by your executable and the output file(s) created by the distributed executable.

5. General Requirements

- Document your code appropriately so that it is readable and easy to navigate
- Make sure your implementation compile and run on linprog.cs.fsu.edu with g++, using the C++11 standard compilation flag. Your Makefile should function in a similar manner to produce the driver programs, as shown in the file `output.txt`.

6. Breakdown of points

The grade breakdown of points for these assessments is provided as follows:

Points	Requirements
40	Have completed hashtable.h and hashtable.hpp in a reasonably correct manner as specified in the project description. The code can compile.
20	Have completed all functions for pass_server.h and pass_server.cpp in a reasonably correct manner as specified in the project description. The code can compile.
30	Have completed userpass.cpp for all 10 choices, including the invalid choice. The results should be right for a choice to be counted as correctly implemented. 3 points for each choice.
5	Error messages and output are well formatted, similar to the content in output.txt.
5	Have a correct Makefile that compiles both programs (userpass and scrypt) on linprog machines.

7. Miscellaneous

The first person to report any compilation or execution errors in the provided materials will be given 3% extra credit (maximum 15% per person). Automatic plagiarism detection software will be used on all submissions – any cases detected will result in a grade of 0 for those involved.